Making Win32® Applications Mobile

Porting to Windows® CE

Nancy Nicolaisen
Making Win32® Applications Mobile
Porting to Windows® CE
With love, to Denis, Amy, Chris, and Virginia
Gearhead Press Books in Print

(For complete information about current and upcoming titles, go to www.wiley.com/compbooks/)

Books in the Gearhead Press Point to Point Series
Migrating to Microsoft Exchange 2000 by Stan Reimer

Installing and Configuring Web Servers Using Apache by Melanie Hoag

VoiceXML: 10 Projects to Voice Enable Your Website by Mark Miller

Books in the Gearhead Press In the Trenches Series
Windows 2000 Automated Deployment by Ted Malone and Rolly Perraux

Robust Linux: Assuring High Availability by Iain Campbell
ISBN: 0-471-07040-8

Programming Directory Services for Windows 2000 by Donis Marshall
ISBN: 0-471-15216-1

Programming ADO.NET by Richard Hundhausen and Steven Borg
ISBN: 0-471-20187-1

Designing .NET Web Services Using ADO.NET and XML by Richard Hundhausen and Steven Borg

Making Win32 Applications Mobile: Porting to Windows CE by Nancy Nicolaisen
ISBN: 0-471-21618-6

Programming Windows CE Wireless Applications by Barry Shilmover and Derek Ball
ISBN: 0-471-21469-8

Mastering SQL Server 2002 Security by Mike Young and Curtis Young

Microsoft.NET Security Programming by Donis Marshall
Chapter 2  A Better Approach to Forms  67
Command Bands: The Win CE Forms
   Solution  68
Steps to a Form Using CommandBands  81
Really, Really, Really Long Forms  90
I Want It All Now  106
Looking Ahead  131

Chapter 3  Handling Graphical Input and Output  133
The Linker, Your New Best Friend  133
Drawing Lines  137
Dissecting the EtchASketch Example  143
Using Bitmaps  152
Device-Independent vs. Device-Dependent Bitmaps  159
Making Bitmap Backgrounds Transparent  160
When to Use Device-Independent Bitmaps  167
Color  168
Pens, Brushes, and Fonts  168
Looking Ahead  170

Chapter 4  Handling Stylus Input  171
The Stylus Behaves as a One-Button Mouse  172
Using the CE Ink Control  180
First Steps with the RichInk Control  186
Take-Home Lessons  191
Looking Ahead  191

Chapter 5  The Windows CE Shell  193
The Job of the CE Shell  194
Where the Files Are  194
Examining the FindDirs Example  201
Finding Special Folder Paths on the HPC and HPC Pro  204
The Shell’s User Interface Elements  206
Adding Shortcuts to the Start Menu  213
Adding and Deleting Shortcuts Without
   Using the Start Menu  214
Adding Application Icons to the Taskbar  214
Adding Documents to the Most Recently Used List  216
Shell Behavior for the Palmtop CE Platforms  217
Looking at the FullScreenDlg Example  225
Taking Over the Entire Screen  226
Looking Ahead  228

Part II  Translating Win32 Application
Behaviors to Windows CE  231

Chapter 6  Writing Memory-Efficient CE Applications  233
Something New Under the Sun  234
Anatomy 101  235
Writing a book like this one is an enormous project, involving the effort, knowledge, and skill of many people. When it is completed and there is time to reflect, one can’t help but feel gratitude for all of the help and encouragement that made it possible. My husband, Denis Hall, has learned more than he ever wanted to know about childcare, cooking, and chauffeuring in these last few months. Always there to smooth out the rough spots for the rest of us, he has soldiered along, keeping me going when I didn’t think I had another page left in me. In this partnership I am truly blessed. Thank you so much, Denis. Likewise, our kids, Amy, Chris, and Virginia, have been especially self-reliant, stepping in and doing the jobs that keep a family going, looking after themselves and us. Best of friends to one another and greatest of treasures to your parents, we are so proud of you all.

In some ways, tackling this subject has been the culmination of twenty-five years in the technology field. I’ve had to reach down into my bag of tricks and incorporate the entire spectrum of things I’ve learned over my career—everything from video game design to satellite ground station engineering plays into Windows CE technology. It is humbling to realize just how many people have given me the training and education it took to make a volume like this one possible. I’d like to take this opportunity to express appreciation for my teachers, and for all teachers. In particular, I’d like to thank Dr. Bruce Bartleson, Professor Emeritus, Western State College of Colorado for teaching me that giving your best and having a great time are actually the same thing; Dr. S.E. Phast for sharing his research on low friction transport systems; and Stephanie Niemi and Cathy Sporcich, principals of the truly wonderful public school my kids attend, Crested Butte Community School.
In closing, I would also like to acknowledge two particularly wonderful editors. My first editor, and also mentor and friend, was Karen Offermann. More than anyone else, Karen was responsible for my transition from research to writing. Bright, clever, energetic, and tenacious, she gave me a wonderful opportunity and certainly changed the course of my life. Many thanks, old friend. Donis Marshall, the editor of this volume, has been a true colleague throughout this project. Knowledgeable and erudite, his direction has been invaluable. *Mange Takk*, Donis.

Nancy Nicolaisen  
7 March 2002  
Crested Butte, Colorado USA
One winter day in 1986, drinking a cup of coffee and reading the latest issue of Dr. Dobbs, I learned of a technology that seemed to hold out the promise of solving a lot of problems for me and my colleagues. According to Dr. Dobbs editor Jon Erickson, Microsoft’s Windows 1.0 was going to do great things for IBM PC users. “Hock the Volvo” he admonished, “and buy the Windows SDK right now!” I took him at his word, and plunged ahead, though I was fortunate enough to hang on to the family station wagon. Inside six months I was totally immersed in Windows programming, and have been ever since. In the intervening years, I have written code for every version of Windows, written about writing Windows code for most of the serious programming journals (including Dr. Dobbs ), and, counting this one, published three books on Windows programming. I can honestly say I never met a Windows I didn’t like, and Windows CE is far and away my favorite so far. Here’s why:

The Windows CE Opportunity

The Windows revolution of the early 1980s did two things. First, it created a consistent, predictable, intuitive user interface. Over time this attracted a broad community of users, and by extension, a public body of knowledge about how to use personal computers and software. Second, Windows interposed a layer of hardware abstraction that made peripheral devices independent of specific operating systems and thus dramatically stimulated the PC aftermarket. If you weren’t active in personal computer programming before the advent of Windows, this may not mean much, but it was very important. It meant that a nontechnical person could buy a computer, printer, and mouse,
take them home, connect them—and they worked! At the time, this was remarkable. The wealth and variety of inexpensive, interoperable peripherals we have today is a direct result of the maturation of the Windows Plug and Play concept.

Windows CE brings us to the brink of a similar revolution for handheld and palmtop computers. We are on the threshold of changes that will have as much or more impact on the way we live as did the personal computer revolution. Win CE has inherent technical momentum, because it leverages the huge base of Win 32 knowledge, skill, and code. With a little savvy effort, Win 32 code moves quickly to Win CE. By porting existing code rather than starting from scratch, you’ll incur less risk, deliver applications more quickly, and enjoy the reassuring prospect of a stable API in the future.

Finally, the best thing about Windows CE is that it’s the perfect medium for individual creativity and innovation by software developers. Because Win CE applications must be small and lightweight, they lend themselves to the kind of projects a single person or a very small team can undertake. The devices themselves are fairly inexpensive, the SDK is an add on to tools you probably already have and know well, and probably at least some of your code will move across without any changes at all. In short, this time around, you can give it a try without hocking the Volvo.

What’s in This Book and How It Is Organized

The goal of this book is to help you get your Win 32 code across the often minor, but occasionally intimidating gulf between Win 32 and Windows CE. This implies a couple of important points. First, to get the most out of this book, you must have a reasonable degree of experience with Win 32 programming. You can learn the basics of that topic in any of several good references, but, unfortunately, this isn’t one of them. Also, this is not an exhaustive examination of native Win CE programming. Instead, I’ll focus on presenting reliable alternatives and strategies for getting existing Win32 code up and working, in the spirit of the Win CE environment, as quickly as possible.

The other point I’d like to highlight here has to do with the organization of this book. My porting strategy is of the time-tested divide-and-conquer school. For that reason, this book is divided into three sections. The first section, “Adapting Application Appearance to Windows CE” covers the issues that deal with the appearance of your application. Starting by moving the user interface offers the advantage of working with something you can see. Also, conveniently, your desktop Windows 32 application provides you with a very useful roadmap for this portion of the work, in the form of your application’s resource file. The second section “Translating Win32 Application Behaviors to
Windows CE” explores issues related to an application’s use of system resources: memory, files, native databases, graphics, the registry, and threads. Finally, a third section, “Power Tools” examines issues that arise from the mobility inherent in a Windows CE application. In this section we look at Win CE’s Remote API, manipulating CE databases from the desktop, and the HTML viewer control.

**Tips for Setting Up the Development Environment**

**Choosing Win CE Target Platforms**

The Windows CE SDK provides Windows NT-hosted emulation for both the HPC and the PPC devices. Unfortunately, the emulation environment is not entirely faithful to any real device, and it has several serious inherent limitations. For this reason it’s not practical to use it exclusively. You’ll need access to at least one real device. If you are targeting the PPC principally, try to pick an ARM or Strong ARM based device, as these seem to be emerging as “standard” architectures. For the HPC, MIPs processors seem to hold a current position of dominance.

**Connecting the Win CE Device to the Development Host**

Your Win CE device will include hardware and software that allow you to connect it serially, under the control of Windows CE Services. This is adequate for exchanging and updating small application oriented data files, but is much less satisfactory if you are developing software. A PC to Win CE Ethernet link is a better, faster, more stable way to download code and run remote tools. To set up the Ethernet connection you’ll need a standard Ethernet card in your desktop computer, a null modem Ethernet 10/100BaseT cable, and a couple of specialized components for the handheld side of the connection. For my HPC, I use a Xircom Compact Card Ethernet 10 card. The Ethernet component is a compact flash, and the card slot in my HPC is a PC card format, so I insert the compact flash into a card caddy that is compatible with my slot. The Xircom Ethernet10 card comes with an adapter cable that accepts the 10/100BaseT cable from my computer. A Xircom software CD provides tools that set up the drivers and parameters on both sides of the connection. For my PPC, I use the same setup, but remove the Ethernet Compact Card from the card caddy, and insert it directly into the slot on my PPC.
Adapting Application Appearance to Windows CE

Size matters.

—ROSEANNE BARR

Every single thing we do in moving a Win 32 application’s user interface to Windows CE is motivated by the fact that CE (an acronym that stands for Compact Edition) is small. It has an abbreviated API, is designed with low-power devices in mind, has fierce memory and storage limitations, mandates tiny screens, assumes few to no peripheral devices, and offers relatively little in the way of second chances to recover from errant code. In short, the job of porting Win 32 code to Windows CE could be summed up in the bumper sticker one sees so frequently:

REDUCE, REUSE, RECYCLE

In this first section, I am focusing exclusively on the tasks that will get a user interface up and running for your application in the least time. In many cases this involves trimming back the lush, repetitive feature set we have come to expect on the desktop. This may go against the grain of your prior experience.
What you have to remember is that the physical limitations of Windows CE are real. You can run out of memory, and you can run out of power.

This one rule is the key to making a transition from the desktop user interface to a CE style interface: In any trade-off situation, we always choose the design solution that makes the application smaller.
Our object in this chapter is to start with a Win32 application and get a visible Windows CE facsimile of it up and running as quickly as possible. Notice that I said *visible facsimile*. Here’s what I mean by that. As a first step, we are going to isolate the parts of the Win32 application that have to do with its user interface and include only those in our new Windows CE project. Right now we don’t want to even consider the parts of the application that encode its specific behaviors and functionality.

We do this for several reasons. First, we divide the porting problem into the smallest, simplest elements possible, because this is the fastest path to stable code. If you fully port one aspect of an application at a time, test it, stabilize it, and document it, then you’ll consistently move forward, building on this incremental success. Second, a typical Win32 app, even a fairly lightweight one, will contain tens if not hundreds of unsupported function calls, library references, and other baggage that would take too long to successfully prune away if we started with all of it. Third, and most important of all, as you move the user interface of your porting subject to the highly constrained Win CE environment, you will undoubtedly shed a great deal of the desktop application’s feature set. Shedding features isn’t bad. It’s good, and it’s necessary. When you have the user interface finalized on the CE side, you will be in a
much better position to decide what functionality it makes sense to port. Our objective here is a ported app that reflects the spirit of the Win CE environment and its Win32 origins: It should be a small, lightweight, intelligently distilled essence of the Win32 app from which it evolved.

## The Resource File Is a Road Map

If a Windows application has a user interface at all, most of its significant details are recorded in the resource file. Figure 1.1 is a screenshot of a simple Win32 application that includes icon, accelerator, dialog, menu, and string table resources. These are all defined in the application’s resource file, which today is typically generated by the graphical development tools of the Visual C++ IDE. While this is certainly convenient, there is no dark druid magic involved. The resource file is simply a text file that is processed by the resource compiler, which in turn assembles a database containing the cursors, icons, strings, and fonts your application uses to build its user interface.

Let’s dissect the resource file for this application and assess the relative portability of its user interface.

```plaintext
resource_file.rc
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
undef APSTUDIO_HIDDEN_SYMBOLS
```
#include "resource.h"

////////////////////////////////////////////////////////////////////////
/////
#undef APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////
/////
// English (U.S.) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32
////////////////////////////////////////////////////////////////////////
/////
// Icon
// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDIRESOURCEFILE ICON DISCARDABLE "resource_file.ICO"
IDI_SMALL ICON DISCARDABLE "SMALL.ICO"
////////////////////////////////////////////////////////////////////////
/////
// Menu
// IDC_RESOURCE_FILE MENU DISCARDABLE
BEGIN
POPUP "&File"
BEGIN
  MENUITEM "Open", ID_FILE_OPEN
  MENUITEM "Close", ID_FILE_CLOSE
  MENUITEM "Save", ID_FILE_SAVE
  MENUITEM "SaveAs", ID_FILE_SAVEAS
  MENUITEM "Print", ID_FILE_PRINT
  MENUITEM "Print Setup", ID_FILE_PRINTSETUP
  MENUITEM SEPARATOR
  MENUITEM "E&xit", IDM_EXIT
  MENUITEM SEPARATOR
  MENUITEM "Recent Files", ID_FILE_RECENTFILES
END
POPUP "More Menus"
BEGIN
  POPUP "Submenu 1"
  BEGIN
    MENUITEM "Item1", ID_SUBMENU1_ITEM1
    MENUITEM "Item2", ID_SUBMENU1_ITEM2
  END
END
Chapter 1

POPUP "&Help"
BEGIN

MENUITEM "&About ...", IDM_ABOUT

END

END

///////////

///

// Accelerator
//
IDC_RESOURCE_FILE ACCELERATORS MOVEABLE PURE
BEGIN

"?", IDM_ABOUT, ASCII, ALT

END

///////////

///

// Dialog
//
IDD_ABOUTBOX DIALOG DISCARDABLE 22, 17, 230, 75
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "About"
FONT 8, "System"
BEGIN

ICON IDI_RESOURCE_FILE,IDC_MYICON,14,9,16,16
LTEXT "resource_file Version 1.0",IDC_STATIC,49,10,119,8, SS_NOPREFIX
LTEXT "Copyright (C) 2001",IDC_STATIC,49,20,119,8
DEFPUSHBUTTON "OK",IDOK,195,6,30,11,WS_GROUP

END

IDD_DIALOG1 DIALOG DISCARDABLE 0, 0, 242, 201
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "ResourceFile Example"
FONT 8, "MS Sans Serif"
BEGIN

DEFPUSHBUTTON "OK",IDOK,185,5,50,14
PUSHBUTTON "Cancel",IDCANCEL,185,25,50,14
EDITTEXT IDC_EDIT1,25,25,135,45,ES_AUTOHSCROLL
LISTBOX IDC_LIST1,25,95,95,50,LBS_SORT | LBS_NOINTEGRALHEIGHT |

GROUPBOX "Radio Button Group",IDC_STATIC,150,90,75,55
LTEXT "Scrolling Listbox",IDC_STATIC,25,80,100,10
LTEXT "Edit Control",IDC_STATIC,25,10,125,10

END

#endif APSTUDIO_INVOKED

///////////
// 2 TEXTINCLUDE DISCARDABLE
BEGIN
   
   
END
3 TEXTINCLUDE DISCARDABLE
BEGIN
   
END
1 TEXTINCLUDE DISCARDABLE
BEGIN
   "resource.h"
END
#endif    // APSTUDIO_INVOKED

GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
   IDD_DIALOG1, DIALOG
BEGIN
   LEFTMARGIN, 7
   RIGHTMARGIN, 235
   TOPMARGIN, 7
   BOTTOMMARGIN, 194
END
END
#endif    // APSTUDIO_INVOKED

// String Table

STRINGTABLE DISCARDABLE
BEGIN
   IDS_APP_TITLE           "resource_file"
   IDS_HELLO               "Hello World!"
   IDC_RESOURCE_FILE       "RESOURCE_FILE"
END
#endif    // English (U.S.) resources

}
Porting Icons and Cursors

Icons

Here’s the fragment of the resource file that describes the application’s icons:

```plaintext
// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_RESOURCE_FILE ICON DISCARDABLE "resource_file ICO"
IDI_SMALL ICON DISCARDABLE "SMALL ICO"
```

The first icon shown, IDI_RESOURCE_FILE, is the application’s main icon. That is, it is the one that is displayed when the user creates a shortcut to the application. Win32 applications’ main icons are 32 by 32 pixels and may use the full range of system colors. By contrast, Windows CE icons must be no larger than 16 by 16 pixels, and must be explicitly loaded by the application. If the icon is too large, it won’t load. The second icon in the resource file, IDI_SMALL, is used to represent the application in list views (for example, Windows Explorer). This icon is superfluous on the CE side and should be eliminated.

Task 1:

Modify the application’s main icon so that its dimensions are exactly 16 by 16 pixels. Since many CE devices are limited in their color depth, check to make sure your icon still reads correctly in black and white. (Most paint programs have a function that lets you change a color image to grayscale or black and white.) Delete the small icon’s file and references from the project.

Cursors

The resource_file application includes no special cursor resources. However, this simply means that the app doesn’t store any images specifically intended
to be displayed as cursors in its resource file. Win32 applications would typically use the API’s LoadCursor, LoadCursorFromFile, or CreateCursor to get or create a special cursor.

In most cases, using special-purpose cursors is a bad practice under Win CE, both because it is unnecessarily a waste of memory and because the concept of a cursor is permeated by the assumption that the user has a keyboard and a mouse as well. By contrast, Windows CE devices are heavily biased toward stylus input. Since the user taps the screen to choose the current position (which is really just the location of the input focus), a cursor is superfluous. Most Windows CE devices provide implicit support for the wait cursor, however. Standard practice is to display the wait cursor whenever you undertake an operation that will make the system unresponsive for a noticeable period of time. Here’s how to load the native wait cursor:

```c
//LoadCursor loads the cursor image and returns the handle
//SetCursor sets the new cursor and returns the handle to the old cursor
hOldCursor = SetCursor(LoadCursor( NULL, IDC_WAIT));
```

Here’s one more thing to note about custom cursor resources. The file that encodes a cursor resource includes information for a variety of display devices. On the desktop, this means the cursor is drawn correctly for all resolution and aspect ratios, so the actual cursor file may be several times as large as the bitmap used for the cursor. Also, Windows CE doesn’t support color cursors, and attempting to load one may produce a program crash.

**Task 2:**

Identify code that manipulates the cursor. Except for code that displays the wait cursor during lengthy operations, eliminate calls to cursor-handling functions. Where the cursor changes signal change of mode or application state, devise ways to inform users. For example, use message boxes or modify the application’s caption on the taskbar icon.

**Accelerators**

Here are the lines from the resource file that encode the default accelerator for the resource_file application:

```c
IDCRESOURCEFILE ACCELERATORS MOVEABLE PURE
BEGIN
   "?", IDM_ABOUT, ASCII, ALT
END
```

This entry means a user can type Alt-? to open the application’s About... box. Much of what was said about cursors applies to accelerators. They are not
as wasteful of memory or as risky if misused, but they lose their meaning in an environment where the user has options such as stylus and voice input.

**Task 3:**

Eliminate code that defines or manipulates accelerators.

**Strings**

Here’s the application’s string table resource code:

```
STRINGTABLE DISCARDABLE
BEGIN
  IDS_APP_TITLE           "resource_file"
  IDS_HELLO               "Hello World!"
  IDC_RESOURCE_FILE       "RESOURCE_FILE"
END
```

So far, we’ve mostly been tossing unnecessary things out. Not so with the string table, which is a key tool for several reasons. First, Windows CE has aggressively embraced the world market for handheld and palmtop computers. This is the motivation behind CE’s standardization on Unicode. Putting all the application’s static text in the string table means that an application can be fully translated by manipulating only the resource file and the help files. It doesn’t even have to be recompiled, only linked with a new language version of the resource file. More important, however, is the string table’s potential to conserve the runtime memory. In a nutshell, here’s the memory advantage of using the string table.

Every Windows CE application occupies a region of read-only memory in which the program image and read-only static data are stored. This region is separate and distinct from the read/write memory used by an executing application. String resources are stored in read-only memory and never used en masse, but instead are loaded on demand at run time by an application. This dramatically reduces the application’s footprint in the memory region used for the application heap and stack.

**Task 4:**

Locate the literal text in your application and move it to the string table. Replace the literals with calls to LoadString.

**Menus**

Here is the menu section for the resource_file application:

```
// Menu
```
With menus, we come to a serious porting effort. First, we’ll look at what we can get rid of and then we’ll examine the changed nature of menu handling under Windows CE.

In the File menu, you can eliminate Exit because it duplicates a standard element on the Windows CE menu bar. You can also eliminate File.Print Setup. Printing is supported on CE devices only through the serial or infrared ports, which essentially amounts to transferring a file to a desktop computer and delegating to its printer, so you can’t effect changes in printer setup from the handheld. For this reason you may want to eliminate File.Print, since it’s basically a subterfuge for moving a file to the desktop and doing the file handling there.

Skipping over the “More Menus” popup for now, look at the “Help” menu. It’s only purpose in this case is to display an item that shows the application’s About...box. “Help” should go because it is a waste of valuable screen real estate. (CE Applications usually include a very spare HTML-based help file. See Appendix B: Implementing Help for more information.) The About...box should go because it’s a waste of valuable memory resources. If you feel strongly about displaying copyright and version information, use MessageBox when the program opens to do this. The dialog template and code that implants the MessageBox function is already present, so using it to display
a splash screen with copyright, owner, and version number information incurs very little overhead.

Now let’s look more closely at the “More Menus” item. This is our first real instance of making a choice to keep or eliminate a feature based on its congruence with the spirit of the Windows CE environment. “More Menus” resource description is shown below:

```
POPUP "More Menus"
BEGIN
  POPUP "Submenu 1"
  BEGIN
  MENUITEM "Item1",       ID_SUBMENU1_ITEM1
  MENUITEM "Item2",       ID_SUBMENU1_ITEM2
  END
END
```

Floating menus and popup submenus are a key thing to be on the lookout for when deciding whether to move existing menu code or to undertake redesign. If you have nested POPUP menus, you need to consider changing the menu design. POPUPS on the desktop are used to help organize groups of relationships among menu items. On a CE device, scarcities of memory and screen real estate make this approach cumbersome and wasteful.

**Windows CE Menu Bars**

From a realistic point of view, users aren’t likely to be doing (or attempting to do) a very large variety of things at one time on a CE device. Because of their small screen size and relatively slow processing speed, it’s reasonable to assume task-oriented patterns of user behavior. Probably for this reason, as well as the physical limits of the device, the designers of Windows CE have emphasized a more flexible, dynamic method of displaying menus than the one on which we mostly relied on the desktop. CE’s use of menu bars provides a highly efficient way of displaying compact menus that are tailored to the task in which the user is engaged.

Instead of creating one menu for an application that presents every possible action or mode, you dynamically create and display a variety of task-specific menu bars. When the user signals a change in task activity, you destroy the old menu bar and create a new one that supports the new task.

Next, let’s take a look at an application that demonstrates the versatility of menu bars. Before we do so, I’ll say a word about the source files of the example apps. I use Visual C++ to build and test my Windows CE applications, both for the purposes of this book and in my business. I have found that in order to get things to build and link smoothly (most particularly, in order to avoid problems with unresolved function references at link time), I start new projects using the procedure shown in Figure 1.2.
Figure 1.2 Create a Windows CE project.

Open the dialog box shown above from the File.New.Project menu (see Figure 1.3). Choose WCE Application from the left pane, and check every processor you wish to target in the Platforms pane. This last bit is important, because it’s difficult to add platforms later.

Choose a typical “Hello World” application from the list in this dialog box. Though it’s tempting to start with an empty project and import some of your own files into a clean space, it hasn’t worked well for me. Using the “Hello World” files sets up all the linkage parameters and is much faster and less error prone.

Figure 1.3 Choosing initial project files.
Build and run the Hello World app before you go on. In particular, check its list of target platforms to make sure you got all of the ones you intended. There is a pair of toolbar dropdowns that you can check if you use Visual C++. The first is the WCE Configuration bar, which shows the available Windows CE configurations. It should include entries for the Handheld PC and the Palmtop PC, plus the version numbers of the SDK for each. Another control, the Build toolbar, lists all of the targets for which you can generate code in this project. The list should include all the processor types you specified in the Platforms pane of the New Project window, with Release and Debug build versions for each.

**The MenuBar Example**

Now we’ll begin to examine the alternatives for menus that are determined to be inappropriate for direct porting from Win32. Here is a listing of the files for the MenuBar Project, an example that demonstrates how to implement the functionality of complex or deeply nested menus using Command Bars.

```
// Resource.h
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by MenuBar.rc
//
```
#define IDS_APP_TITLE 1
#define IDS_HELLO 2
#define IDC_MENUBAR 3
#define IDC_COMBO_BUTTON 4
#define IDI_MENUBAR 101
#define IDM_MENU 102
#define IDD_ABOUTBOX 103
#define IDB_ONE_TWO 105
#define IDR_DROP_BTN_MENU 109
#define IDM_FILE_EXIT 40002
#define IDM_HELP_ABOUT 40003
#define IDM_MENU_BTNS 40004
#define IDM_ONE 40005
#define IDM_COMBO 40006
#define IDM_TWO 40007
#define IDM_DROP_BTN 40008
#define IDM_UNDO 40009
#define IDM_CHECK_BTN 40010
#define IDM_REDO 40011
#define IDM_RADIO_BTNS 40012
#define IDM_DROP1 40013
#define IDM_DROP_ITEM2 40014

// Next default values for new objects
/
#endif APSTUDIO_INVOKED
#endif APSTUDIO_READONLY_SYMBOLS
 seaborn

MenuBar.rc
//Microsoft Developer Studio generated resource script.
/
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS

//(------------------------------)
#define _APS_NEXT_RESOURCE_VALUE 110
#define _APS_NEXT_COMMAND_VALUE 40014
#define _APS_NEXT_CONTROL_VALUE 1001
#define _APS_NEXT_SYMED_VALUE 101
#endif

// English (U.S.) resources
#include "afxres.h"
//(------------------------------)
#define AFXRESOURCE_DLL || defined(afx_TARG_ENU)


#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////////////////
///
// Icon
///
// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_MENUBAR ICON DISCARDABLE "MenuBar.ICO"
#endif    // APSTUDIO_INVOKED

////////////////////////////////////////////////////////////////////////
///
// TEXTINCLUDE
///
1 TEXTINCLUDE DISCARDABLE
BEGIN
 "resource.h\0"
 END
2 TEXTINCLUDE DISCARDABLE
BEGIN
 "\0"
 END
3 TEXTINCLUDE DISCARDABLE
BEGIN
 "\0"
 END
#endif    // APSTUDIO_INVOKED

////////////////////////////////////////////////////////////////////////
///
// Menu
///
IDM_MENU MENU DISCARDABLE
BEGIN
 POPUP "&File"
 BEGIN
  MENUITEM "E&xit", IDM_FILE_EXIT
 END
 POPUP "View"
 BEGIN
  MENUITEM "Add Menu Buttons", IDM_MENU_BTNS
  MENUITEM "Add Combo", IDM_COMBO
  MENUITEM "Add Dropdown Button", IDM_DROP BTN
  MENUITEM "Add Check Button", IDM_CHECK BTN
  MENUITEM "Add Radio Buttons", IDM_RADIO BTN
 END

16 Chapter 1
POPUP "&Help"
BEGIN
  MENUITEM "&About", IDM_HELP_ABOUT
END
END

DROPMENU MENU DISCARDABLE
BEGIN
  POPUP "DropBtn"
  BEGIN
    MENUITEM "Drop Item 1", IDM_DROP1
    MENUITEM "Drop Item 2", IDM_DROP_ITEM2
  END
END

////////////////////////////////////////////////////////////////////////
/////
// Dialog
////
IDD_ABOUTBOX DIALOG DISCARDABLE 0, 0, 110, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
EXSTYLE WS_EX_CAPTIONOKBTN
CAPTION "About MenuBar"
FONT 8, "Tahoma"
BEGIN
  ICON IDI_MENUBAR,IDC_STATIC,11,17,20,20
  LTEXT "MenuBar Version 1.0",IDC_STATIC,40,10,119,8,SS_NOPREFIX
  LTEXT "Copyright (C) 2001",IDC_STATIC,40,25,119,8
END

////////////////////////////////////////////////////////////////////////
/////
// DESIGNINFO
////
#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
  IDD_ABOUTBOX, DIALOG
  BEGIN
    LEFTMARGIN, 7
    RIGHTMARGIN, 56
    TOPMARGIN, 7
  END
END
#endif    // APSTUDIO_INVOKED

 Accelerator

 IDC_MENUBAR ACCELERATORS DISCARDABLE
BEGIN
// Bitmap
//
IDB_ONE_TWO BITMAP DISCARDABLE "one_two.bmp"
IDB_NEXTPREV BITMAP DISCARDABLE "next_pre.bmp"
IDB_PREVNEXT BITMAP DISCARDABLE "prev_nex.bmp"

// String Table
//
STRINGTABLE DISCARDABLE
BEGIN
  IDS_APP_TITLE "MenuBar"
  IDS_HELLO "Hello World!"
  IDC_MENUBAR "MENUBAR"
END

#endif  // English (U.S.) resources

#endif  // not APSTUDIO_INVOKED

MenuBar.cpp

// MenuBar.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "MenuBar.h"
#include "resource.h"
#include <commctrl.h>
#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst;  // The current instance
HWND hwndCB;      // The command bar handle

// Command band button initialization structure
TBBUTTON tbCmdButtons[] = {
  // BitmapIndex  Command  State  Style   UserData  String
  {0,            IDM_ONE,  TBSTATE_ENABLED,                     
   TBSTYLE_BUTTON, 0, 0},
  {1,            IDM_TWO,  TBSTATE_ENABLED,                     
   TBSTYLE_BUTTON, 0, 0},
  {STD_UNDO+3,    IDM_UNDO,  TBSTATE_ENABLED,                   
   TBSTYLE_BUTTON, 0, 0},

  // BitmapIndex  Command  State  Style   UserData  String
  {2,            IDM_HELP_ABOUT, ASCII, ALT, NOINVERT
   TBSTATE_BUTTON, 0, 0},
  {3,            VK_F4,    IDM_FILE_EXIT, VIRTKEY, ALT, NOINVERT
   TBSTATE_BUTTON, 0, 0},

  {4,            IDM_ABOUT, NULL,       NULL,  NULL, 0}
};
TBSTYLE_BUTTON, 0, 0},
{STD_REDO, IDM_REDO, TBSTATE_ENABLED,
 TBSTYLE_BUTTON, 0, 0},
);
// Command band button initialization structure
TBBUTTON tbRadioButtons[] = {
// BitmapIndex Command State Style UserData String
{0, IDM_ONE, TBSTATE_ENABLED,
 TBSTYLE_GROUP | TBSTYLE_CHECKGROUP,
0, 0},
{1, IDM_TWO, TBSTATE_ENABLED,
 TBSTYLE_CHECKGROUP,
0, 0},
};
// Command band button initialization structure
TBBUTTON tbCheckButton[] = {
// BitmapIndex Command State Style UserData String
{0, IDM_ONE, TBSTATE_ENABLED,
 TBSTYLE_CHECK, 0, 0},
};
// Command band button initialization structure
TBBUTTON tbDropButton[] = {
// BitmapIndex Command State Style UserData String
{0, IDM_ONE, TBSTATE_ENABLED,
 TBSTYLE_DROPDOWN,
0, 0},
};
// Forward declarations of functions included in this code module:
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance(HINSTANCE, int);
HWND AddCmdButtons(HWND);
HWND AddDropDownButton(HWND);
HWND AddCombo(HWND);
HWND AddRadioButtons(HWND);
HWND AddCheckButton(HWND);
long HandleDropButton(HWND, LPNMHDR, LPNMTOOLBAR);

int WINAPI WinMain(HINSTANCE hInstance,
 HINSTANCE hPrevInstance,
 LPTSTR    lpCmdLine,
 int       nCmdShow)
{

MSG msg;
HACCEL hAccelTable;
INITCOMMONCONTROLSEX icex; //this struct is used to init the
 //common controls dll
//init the structure
memset(&icex, 0x0, sizeof(INITCOMMONCONTROLSEX));
//windows ce requires explicit init of the common controls
icex.dwSize = sizeof(INITCOMMONCONTROLSEX);
//just load what we need
icex.dwICC = ICC_COOL_CLASSES;
InitCommonControlsEx(&icex);
// Perform application initialization:
if (!InitInstance(hInstance, nCmdShow))
{
    return FALSE;
}

hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_MENUBAR);
// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
return msg.wParam;

//
// FUNCTION: MyRegisterClass()
//
// PURPOSE: Registers the window class.
//
// COMMENTS:
//
This function and its usage is only necessary if you want this code
//to be compatible with Win32 systems prior to the 'RegisterClassEx'
//function that was added to Windows 95. It is important to call this
//function so that the application will get 'well formed' small icons
//associated with it.
//
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS    wc;
    wc.style     = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra  = 0;
    wc.cbWndExtra  = 0;
    wc.hInstance  = hInstance;
    wc.hIcon      = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_MENUBAR));
    wc.hCursor    = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = 0;
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);
}

//
// FUNCTION: InitInstance(HANDLE, int)
//
// PURPOSE: Saves instance handle and creates main window.
//
// COMMENTS:
// In this function, we save the instance handle in a global variable
// and create and display the main program window.

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND    hWnd;
    TCHAR    szTitle[MAX_LOADSTRING];       // The title bar text
    TCHAR    szWindowClass[MAX_LOADSTRING]; // The window class name
    hInst = hInstance;  // Store instance handle in our global variable
    // Initialize global strings
    LoadString(hInstance, IDC_MENUBAR, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
                        0, 0, CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL);
    // if we didn't get a Window, fail and bail
    if (!hWnd)
    {
        return FALSE;
    }
    // now show the window and make sure it gets painted
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    // if we have a command bar, show it now
    if (hwndCB)
    {
        CommandBar_Show(hwndCB, TRUE);
        return TRUE;
    }
}

// FUNCTION: WndProc(HWND, unsigned, WORD, LONG)

// PURPOSE: Processes messages for the main window.
// WM_COMMAND - process the application menu
// WM_NOTIFY - get information about menu items we have to handle
// WM_PAINT - Paint the main window
// WM_DESTROY - post a quit message and return

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    int wmId, wmEvent, idBitmap, rc;
    PAINTSTRUCT ps;
    TCHAR szHello[MAX_LOADSTRING];
    LPNMHDR lpnmHeader;
    LPNMTOOLBAR lpnmDropButton;
    switch (message)
    {
        case WM_COMMAND:
            wmId = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            break;
    }
}
// Parse the menu selections:
switch (wmId)
{
    case IDM_ONE:
        MessageBox(hWnd, TEXT("1 Pressed"),
            TEXT("Menu Demo"), MB_OK );
        break;
    case IDM_TWO:
        MessageBox(hWnd, TEXT("2 Pressed"),
            TEXT("Menu Demo"), MB_OK );
        break;
    case IDM_UNDO:
        MessageBox(hWnd, TEXT("Std Undo Pressed"),
            TEXT("Menu Demo"), MB_OK );
        break;
    case IDM_REDO:
        MessageBox(hWnd, TEXT("Std Redo Pressed"),
            TEXT("Menu Demo"), MB_OK );
        break;
    case IDM_MENU_BTNS:
        hwndCB = AddCmdButtons(hWnd);
        break;
    case IDM_DROP_BTN:
        hwndCB = AddDropDownButton(hWnd);
        break;
    case IDM_COMBO:
        hwndCB = AddCombo(hWnd);
        break;
    case IDM_RADIO_BTNS:
        hwndCB = AddRadioButtons(hWnd);
        break;
    case IDM_CHECK_BTN:
        hwndCB = AddCheckButton(hWnd);
        break;
    case IDM_HELP_ABOUT:
        DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX,
            hWnd, (DLGPROC)About);
        break;
    case IDM_FILE_EXIT:
        DestroyWindow(hWnd);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    break;
    case WM_CREATE:
        hwndCB = CommandBar_Create(hInst, hWnd, 1);
        CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
        CommandBar_AddAdornments(hwndCB, 0, 0);
        break;
    case WM_NOTIFY:
        // Notify messages have a generic header.
        //check it to see what sent the notify message.
lpnmHeader = (LPNMHDR)lParam;
if (lpnmHeader->code == TBN_DROPDOWN)
{
   //if the notify message came from our button,
   //recast lParam to a toolbar notify pointer
   //and call the handler for the drop button
   return HandleDropButton(hWnd, (LPNMHDR)lParam,
                           (LPNMTOOLBAR)lParam);
}
break;
case WM_PAINT:
   RECT rt;
   hdc = BeginPaint(hWnd, &ps);
   GetClientRect(hWnd, &rt);
   LoadString(hInst, IDS_HELLO, szHello, MAX_LOADSTRING);
   DrawText(hdc, szHello, _tcslen(szHello), &rt,
            DT_SINGLELINE | DT_VCENTER | DT_CENTER);
   EndPaint(hWnd, &ps);
   break;
case WM_DESTROY:
   CommandBar_Destroy(hwndCB);
   PostQuitMessage(0);
   break;
default:
   return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;

// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message,
                        WPARAM wParam, LPARAM lParam)
{
   RECT rt, rt1;
   int DlgWidth, DlgHeight; //dialog width and height
   //in pixel units
   int NewPosX, NewPosY;
   switch (message)
   {
   case WM_INITDIALOG:
      // trying to center the About dialog
      if (GetWindowRect(hDlg, &rt1)) {
         GetClientRect(GetParent(hDlg), &rt);
         DlgWidth = rt1.right - rt1.left;
         DlgHeight = rt1.bottom - rt1.top;
         NewPosX = (rt.right - rt.left - DlgWidth)/2;
         NewPosY = (rt.bottom - rt.top - DlgHeight)/2;

         // if the About box is larger than the physical screen
         if (NewPosX < 0) NewPosX = 0;
         if (NewPosY < 0) NewPosY = 0;
         SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                      0, 0, SWP_NOZORDER | SWP_NOSIZE);
      }
   }
return TRUE;
case WM_COMMAND:
if ((LOWORD(wParam) == IDOK) || (LOWORD(wParam) == IDCANCEL))
{
    EndDialog(hDlg, LOWORD(wParam));
    return TRUE;
} 
break;
return FALSE;

HWND    AddCmdButtons( HWND hWnd )
{
    //we have to destroy the old one before expanding
    CommandBar_Destroy( hwndCB );
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    // Insert the menu band
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    //add custom bitmaps for buttons
    CommandBar_AddBitmap(hwndCB, hInst, IDB_ONE_TWO, 2, 0, 0);
    // Add standard bitmaps for buttons.
    CommandBar_AddBitmap (hwndCB, HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
                          0, 0, 0);
    //Add the buttons
    CommandBar_AddButtons(hwndCB, 4, &tbCmdButtons);
    // Add exit button to command band control.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    CommandBar_Show(hwndCB, TRUE);
    return hwndCB;
}

HWND    AddDropDownButton( HWND hWnd )
{
    //we have to destroy the old one before expanding
    CommandBar_Destroy( hwndCB );
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    // Insert the menu band
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    //add bitmaps for buttons
    CommandBar_AddBitmap(hwndCB, hInst, IDB_ONE_TWO, 2, 0, 0);
    //Add the buttons
    CommandBar_AddButtons(hwndCB, 1, &tbDropButton);
    // Add exit button to command band control.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    CommandBar_Show(hwndCB, TRUE);
    return hwndCB;
}

HWND    AddCombo( HWND hWnd )
{
    int i;
    TCHAR tszBuff[24];
    //we have to destroy the old one before expanding
    CommandBar_Destroy( hwndCB );
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
// Insert the menu band
CommandBar_InsertMenuBar(hwndCB, hInst, IDM_MENU, 0);

// Add a combo box between the view icons and the sort icons.
CommandBar_InsertComboBox(hwndCB, hInst,
GetSystemMetrics(SM_CXSCREEN) / 4,
    CBS_DROPDOWNLIST | WS_VSCROLL,
    IDC_COMBO_BUTTON, 1);
// Fill in combo box.
for (i = 0; i < 5; i++)
{
    wsprintf(tszBuff, TEXT("Combo List %d"), i);
    SendDlgItemMessage(hwndCB, IDC_COMBO_BUTTON, CB_INSERTSTRING, -1, (LPARAM)tszBuff);
}
// select list item 0
SendDlgItemMessage(hwndCB, IDC_COMBO_BUTTON, CB_SETCURSEL, 0, 0);

// AddAdornments puts an exit button on the command band control
// It replaces the "Exit" menu item
CommandBar_AddAdornments(hwndCB, 0, 0);
CommandBar_Show(hwndCB, TRUE);
return hwndCB;

HWND AddRadioButtons(HWND hWnd)
{
    // we have to destroy the old one before expanding
    CommandBar_Destroy(hwndCB);
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    // Insert the menu band
    CommandBar_InsertMenuBar(hwndCB, hInst, IDM_MENU, 0);
    // add bitmaps for buttons
    CommandBar_AddBitmap(hwndCB, hInst, IDB_ONE_TWO, 2, 0, 0);
    // Add the buttons - the button structs define the button
    // radio behaviors
    CommandBar_AddButtons(hwndCB, 2, tbRadioButtons);
    // Add exit button to command band control.
    CommandBar_AddAdornments(hwndCB, 0, 0);
    CommandBar_Show(hwndCB, TRUE);
    return hwndCB;
}

HWND AddCheckButton(HWND hWnd)
{
    // we have to destroy the old one before expanding
    CommandBar_Destroy(hwndCB);
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    // Insert the menu band
    CommandBar_InsertMenuBar(hwndCB, hInst, IDM_MENU, 0);
    // add bitmaps for buttons
    CommandBar_AddBitmap(hwndCB, hInst, IDB_ONE_TWO, 2, 0, 0);
    // Insert buttons.
    CommandBar_AddBitmap(hwndCB, HINST_COMMCTRL, IDB_STD_SMALL_COLOR, 0, 0, 0);
Add the buttons - the flags in the button structs
// specify the check box behaviors
CommandBar_AddButtons(hwndCB, 1, tbCheckButton);
// Add exit button to command band control.
CommandBar_AddAdornments(hwndCB, 0, 0);
CommandBar_Show(hwndCB, TRUE);
return hwndCB;

long HandleDropButton(HWND hWnd, LPNMHDR lpnmHeader,
                   LPNMTOOLBAR lpnmDropButton)
{
    RECT rect;
    TPMPARAMS tpm;
    HMENU hMenu;
    if (lpnmDropButton->iItem == IDM_ONE)
    {
        // Get the rectangle of the dropdown button.
        SendMessage(lpnmHeader->hwndFrom, TB_GETRECT,
                    lpnmDropButton->iItem, (LPARAM)&rect);
        // Convert rect into screen coordinates. The rect is
        // considered here to be an array of 2 POINT structures.
        MapWindowPoints(lpnmHeader->hwndFrom, HWND_DESKTOP,
                        (LPPOINT)&rect, 2);
        // Prevent the menu from covering the button.
        tpm.cbSize = sizeof(tpm);
        CopyRect(&tpm.rcExclude, &rect);
        hMenu = LoadMenu(hInst, TEXT("DROPMENU"));
        hMenu = GetSubMenu(hMenu, 0);
        TrackPopupMenuEx(hMenu, TPM_LEFTALIGN | TPM_VERTICAL,
                          rect.left, rect.bottom, hWnd, &tpm);
    }
    return 0;
}

MenuBar’s WinMain()

A lot of the startup code for a Windows CE application is identical to that of
Win32 (see Tables 1.1 and 1.2). Here’s one notable exception, depicted in this
code excerpt from WinMain():

    INITCOMMONCONTROLSEX icex; // this struct is used to init the
    // common controls dll
    memset(&icex, 0x0, sizeof(INITCOMMONCONTROLSEX));
    // windows ce requires explicit init of the common controls
    icex.dwSize = sizeof(INITCOMMONCONTROLSEX);
    // just load what we need
    icex.dwICC = ICC_COOL_CLASSES;
    InitCommonControlsEx(&icex);
You must explicitly register common control classes before your program can use them. The InitCommonControlsEx() function takes the structure INITCOMMONCONTROLSEX as its only parameter. The value of icex.dwICC determines which group of controls is registered in this call. If you make more than one call to InitCommonControlsEx(), control registration is cumulative. You can OR the flags to register more than one group at a time (for example ICC_BAR_CLASSES | ICC_COOL_CLASSES).

One more thing, before we go on. Notice that I used memset() to initialize the structure contents to 0x0 and sizeof(INITCOMMONCONTROLSEX) to set the size of the structure in the icex.dwsize member. Even where the documentation says it isn’t necessary, I have found that initializing and setting the size member of structures passed as parameters to be a very important defense against erratic behavior.

<table>
<thead>
<tr>
<th>CONTROL TYPE FLAGS</th>
<th>CONTROLS REGISTERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC_BAR_CLASSES</td>
<td>Loads toolbar, status bar, trackbar, and command bar classes.</td>
</tr>
<tr>
<td>ICC_COOL_CLASSES</td>
<td>Loads rebar control class.</td>
</tr>
<tr>
<td>ICC_DATE_CLASSES</td>
<td>Loads date and time-picker control class.</td>
</tr>
<tr>
<td>ICC_LISTVIEW_CLASSES</td>
<td>Loads list view and header control classes.</td>
</tr>
<tr>
<td>ICC_PROGRESS_CLASS</td>
<td>Loads progress bar control class.</td>
</tr>
<tr>
<td>ICC_TAB_CLASSES</td>
<td>Loads tab control classes.</td>
</tr>
<tr>
<td>ICC_TREEVIEW_CLASSES</td>
<td>Loads tree view control classes.</td>
</tr>
<tr>
<td>ICC_UPDOWN_CLASS</td>
<td>Loads Up-Down control class.</td>
</tr>
</tbody>
</table>

You must explicitly register common control classes before your program can use them. The InitCommonControlsEx() function takes the structure INITCOMMONCONTROLSEX as its only parameter. The value of icex.dwICC determines which group of controls is registered in this call. If you make more than one call to InitCommonControlsEx(), control registration is cumulative. You can OR the flags to register more than one group at a time (for example ICC_BAR_CLASSES | ICC_COOL_CLASSES).

One more thing, before we go on. Notice that I used memset() to initialize the structure contents to 0x0 and sizeof(INITCOMMONCONTROLSEX) to set the size of the structure in the icex.dwsize member. Even where the documentation says it isn’t necessary, I have found that initializing and setting the size member of structures passed as parameters to be a very important defense against erratic behavior.

<table>
<thead>
<tr>
<th>NON PORTABLE WIN32 FLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC_ANIMATE_CLASS</td>
</tr>
<tr>
<td>ICC_HOTKEY_CLASS</td>
</tr>
<tr>
<td>ICC_INTERNET_CLASSES</td>
</tr>
<tr>
<td>ICC_USEREX_CLASSES</td>
</tr>
</tbody>
</table>
WinMain() calls MyRegisterClass() and InitInstance(), both of which do routine and familiar tasks. There are a few differences to note that have to do with menu creation, however. Notice this line in MyRegisterClass(), where the members of the WNDCLASS structure are being set.

```
//set the pointer to the menu resource name = 0
wc.lpszMenuName = 0;
```

We set this window class parameter to 0 because we are going to take over the construction and display of the menu by creating a CommandBar control and positioning it where the menu would normally appear.

Also, note these lines at the end of InitInstance():

```
//now show the window and make sure it gets painted
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
//if we have a command bar, show it now
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
```

The reason we have to explicitly tell the command bar to show itself is that Windows doesn’t automatically handle its update and display. In contrast to a Win32 style menu, a command bar has no blueprint in the resource file that tells Windows how to build it and where to put it. Also, it has to be painted onto the main window after that window becomes visible.

### Creating the Command Bar

The first time we create the command bar–based menu is in response to the WM_CREATE message, which we process in the message switch of WndProc().

```
    case WM_CREATE:
        hwndCB = CommandBar_Create(hInst, hWnd, 1);
        CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
        CommandBar_AddAdornments(hwndCB, 0, 0);
        break;
```

Here’s something to keep in mind with command bars (and also with their cousins, command bands, which we cover in depth in the next chapter). Creating a command bar reserves some real estate and sets up a container in which you can place other controls and interface elements. Most of the work of your program will entail dealing with the objects you’ve placed in the command bar. You can manipulate those objects only by using the command bar’s access functions.
hwndCB = CommandBar_Create(hInst, hWnd, 1);

This line creates an empty command bar. The parameters, in the order shown, are the instance handle, the main window’s handle, and the command bar’s ID, which we set to 1. The return value is the handle to the command bar.

CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);

This line sets a menu resource into the command bar. The parameters, in the order shown, are the handle to the command bar, the instance handle, the resource ID of the menu to load, and the zero-based index of a button on the command bar. The menu will be loaded to the left of the button specified by the index. It is possible to load more than one menubar in a command bar, but this contravenes the Windows CE design guidelines, so it’s not a good practice. According to the guidelines, there should be one menubar per command bar, and it should be the leftmost element.

CommandBar_AddAdornments(hwndCB, 0, 0);

This bit of code is a real powerhouse, and it has some implications of which it’s worth being aware. “Adornments” are the optional controls that appear at the right-hand side of a menu based on a command bar. Most command bars will include the Close button, and that is the one we are adding here. You may optionally include the OK and the Help buttons as well. There is no flag for the Close button, because it is the default adornment. Table 1.3 lists CommandBar adornment flags.

The Close, OK, and Help buttons exist not only to provide the main window with messages, but can also replace the same buttons in a dialog box. This means that your dialogs can be smaller. Command bar elements each occupy a specific region on the command bar and this location is important to the internal management of the command bar’s controls. Always add the command bar adornments last, that is to say, after all the rest of the command bar’s control elements have been added. This will ensure they have sole claim to the real estate at the rightmost edge of the command bar.

<table>
<thead>
<tr>
<th>ADORNMENT FLAG</th>
<th>WHAT IT ADDS</th>
<th>COMMAND MESSAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMDBAR_HELP</td>
<td>Help button</td>
<td>WM_HELP</td>
</tr>
<tr>
<td>CMDBAR_OK</td>
<td>OK button</td>
<td>WM_COMMAND (with IDOK as the message identifier)</td>
</tr>
</tbody>
</table>
Dynamically Modifying the Command Bar–Based Menu

Because of the small amount of screen real estate available, you may want to make menu construction more dynamic and responsive to user actions. Let’s take a look at how to implement this sort of behavior.

When we initially created the command bar menu, we loaded the menu resource IDM_MENU. This resource file fragment shows the content of the View menu item.

```
POPUP "View"
BEGIN
  MENUITEM "Add Menu Buttons", IDM_MENU_BTNS
  MENUITEM "Add Combo", IDM_COMBO
  MENUITEM "Add Dropdown Button", IDM_DROP_BTN
  MENUITEM "Add Check Button", IDM_CHECK_BTN
  MENUITEM "Add Radio Buttons", IDM_RADIO_BTNS
END
```

The MenuBar example lets you modify the command bar menu on the fly, adding the different kinds of elements shown in menu choices. In fact, what we have to do is destroy the command bar and create a new one each time we get one of these requests from the user. First, we’ll go through the code that creates and displays the new kinds of command bar elements, and then we’ll look at how the menu items communicate with the main window.

Adding Command Bar Buttons

Here’s some great news about this whole command bar–based menu idea. The code for adding any kind or number of buttons to the command bar is always the same. It looks like this:

```c
// Add the buttons
CommandBar_AddButtons(hwndCB, 4, &tbCmdButtons);
```

The parameters, in the order shown, are the handle to the command bar, the number of buttons to add, and the address of an array of TBBUTTON structures. As you may see, the TBBUTTON structure is the key to the process. Here’s the declaration of the TBBUTTON structure.

```c
typedef struct _TBBUTTON {
  int iBitmap;  // Resource ID of button’s bitmap image
  int idCommand; // Command Id for this button
  BYTE fsState;  // Initial state of the button
  BYTE fsStyle;  // Style flags
} TBBUTTON, *LPBUTTON;
```

88 Chapter 1
encode button behaviors

DWORD dwData; //point to optional user data
int iString;
//pointer to optional caption string
} TBBUTTON

Managing the insertion of buttons into a command bar is mostly a matter of loading the images that are displayed on the buttons and correctly initializing an array of these structures.

**Loading the Button Images**

There are two basic kinds of images you can use on buttons: the ones you create and the ones you borrow. Let’s look at the way to use images of your own first.

// Bitmap
//
IDB_ONE_TWO BITMAP DISCARDABLE "one_two.bmp"

This fragment of the menubar.rc file associates the constant IDB_ONE_TWO with a bitmap file named one_two.bmp. IDB_ONE_TWO contains two contiguous bitmap images, each 16 by 16. Here are two important things to know about bitmap button images. They must be exactly 16 by 16 pixels or they won’t load properly. Unlike in other versions of Windows, you don’t get any breaks when you try to load images that aren’t correctly sized. Also, you should be very conservative about the use of color in these bitmaps. There are many black and white CE devices in the hands of users. Loading unsupported color bitmaps into a menu bar may well be catastrophic. Choosing bitmap colors from a 2-bit grayscale palette will always work, everywhere.

We make these images available to the command bar like this:

//add bitmaps for buttons
CommandBar_AddBitmap(hwndCB, hInst, IDB_ONE_TWO, 2, 0, 0);

The parameters, in the order shown, are the handle to the command bar, the instance handle, the resource ID of the bitmap, the number of bitmap images aggregated in the bitmap file, and two reserved parameters which must be set to 0.

Adding bitmaps to a command bar makes each individual bitmap part of a list. You reference individual bitmaps by their zero-based index in the list. If you call CommandBar_AddBitmap() more than once, these indexes are cumulative. By this I mean that if we followed the function call above with one like this:

//add 2 more bitmaps for buttons
CommandBar_AddBitmap(hwndCB, hInst, IDB_THREE_FOUR, 2, 0, 0);
After the second call returned, we’d have four bitmaps in the command bar’s list. The index of the first bitmap in IDB_ONE_TWO would be 0, and the index of the first bitmap in IDB_THREE_FOUR would be 2.

It’s very important to make sure you add a number of bitmaps at least equal the number of buttons specified in CommandBar_AddButtons().

Using Standard Button Images

You can also get bitmap images by borrowing them from the Windows CE small color bitmap lists. This strategy has a couple of big advantages. First, the bitmaps are already there, so you don’t have to use any memory to get them. Also, you are assured of their color safety and proper size. Here’s how you make the standard bitmap list available.

```c
// Add standard bitmaps for buttons.
CommandBar_AddBitmap (hwndCB, HINST_COMMCTRL, IDB_STD_SMALL_COLOR, 0, 0, 0);
```

The parameters, in the order shown, are the handle to the command bar, a constant representing the instance handle to the common control dll, the resource ID of a system bitmap list, with the remaining three parameters that are always set to zero. The resource identifier IDB_STD_SMALL_COLOR references a bitmap list that contains the standard bitmaps. If you use the resource identifier IDB_VIEW_SMALL_COLOR, you get access to the “view” bitmap list. See Tables 1.4 and 1.5.

| Table 1.4 Individual Bitmap Resource Identifiers for Standard Bitmaps |
|-----------------------|-------------------|-----------------------|
| STD_COPY              | STD_CUT           | STD_DELETE            |
| STD_FIND              | STD_FILENEW       | STD_FILEOPEN          |
| STD_FILESAVE          | STD_HELP          | STD_PROPERTIES        |
| STD_REDOW             | STD_REPLACE       | STD_PASTE             |
Like your own bitmaps, adding the system’s bitmaps to the command bar list is cumulative. When you want to reference a specific bitmap, you add its individual constant to the zero-based offset of the last bitmap loaded before it.

### Putting It All Together

Here are the three steps for adding buttons to a command bar:

- Initialize an array of TBBUTTON structures. There must be one for each button.
- Add button bitmap images to the command bar.
- Add the buttons.

This code adds buttons with custom bitmap images:

```c
// Command band button initialization structure, custom bitmaps
TBBUTTON tbCmdButtons[] = {
    // BitmapIndex   Command  State     Style    UserData  String
    {0,           IDM_ONE,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0,    0},
    // Other buttons...
};
```
HWND AddCustomButtons(HWND hWnd)
{
    // We have to destroy the old one before expanding
    CommandBar_Destroy(hwndCB);
    hwndCB = CommandBar_Create(hInst, hWnd, 800);
    // Insert the menu band
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    // Add bitmaps for buttons
    CommandBar_AddBitmap(hwndCB, hInst, IDB_ONE_TWO, 2, 0, 0);
    // Add the buttons
    CommandBar_AddButtons(hwndCB, 2, tbCustomButtons);
    // Add exit button to command band control.
    CommandBar_AddAdornments(hwndCB, 0, 0);
    CommandBar_Show(hwndCB, TRUE);
    return hwndCB;
}

This code adds buttons using system bitmaps:

// Command band button initialization structure, std bitmaps
TBBUTTON tbCustomButtons[] = {
    {STD_UNDO,  IDM_UNDO, TBSTATE_ENABLED,
        TBSTYLE_BUTTON, 0, 0},
    {STD_REDO,  IDM_REDO, TBSTATE_ENABLED,
        TBSTYLE_BUTTON, 0, 0},
};
HWND AddStdCmdButtons(HWND hWnd)
{
    // We have to destroy the old one before expanding
    CommandBar_Destroy(hwndCB);
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    // Insert the menu band
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    // Insert buttons.
    CommandBar_AddBitmap(hwndCB, HINST_COMMCTRL,
        IDB_STD_SMALL_COLOR, 0, 0, 0);
    // Add the buttons
    CommandBar_AddButtons(hwndCB, 2, tbCustomButtons);
    // Add exit button to command band control.
    CommandBar_AddAdornments(hwndCB, 0, 0);
    CommandBar_Show(hwndCB, TRUE);
    return hwndCB;
}

This code adds buttons with automatic radio button behavior. Radio buttons are a group of buttons where one and only one button may be selected at a time.
// Command band button initialization

structureTBBUTTON tbRadioButtons[] = {
// BitmapIndex Command State Style UserData String
0, IDM_ONE, TBSTATE_ENABLED,
TBSTYLE_GROUP | TBSTYLE_CHECKGROUP,
0, 0,
1, IDM_TWO, TBSTATE_ENABLED,
TBSTYLE_CHECKGROUP,
0, 0,
}

HWND AddRadioButtons(HWND hWnd)
{
//we have to destroy the old one before expanding
CommandBar_Destroy(hwndCB);
hwndCB = CommandBar_Create(hInst, hWnd, 1);
// Insert the menu band
CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
//add bitmaps for buttons
CommandBar_AddBitmap(hwndCB, hInst, IDB_ONE_TWO, 2, 0, 0);
//Add the buttons - the button structs define the button
//radio behaviors
CommandBar_AddButtons(hwndCB, 2, tbRadioButtons);
// Add exit button to command band control.
CommandBar_AddAdornments(hwndCB, 0, 0);
CommandBar_Show(hwndCB, TRUE);
return hwndCB;
}

This code adds buttons with check box behavior. A check box button may assume one of two states, either checked or unchecked. Its state is not inherently contingent on the state of other buttons.

// Command band button initialization structure

TBBUTTON tbCheckButton[] = {
// BitmapIndex Command State Style UserData String
0, IDM_ONE, TBSTATE_ENABLED,
TBSTYLE_CHECK, 0, 0,
};

//Command band button initialization structure- check box behavior
TBBUTTON tbDropButton[] = {
// BitmapIndex Command State Style UserData String
0, IDM_ONE, TBSTATE_ENABLED,
TBSTYLE_DROPDOWN,
0, 0,
};

HWND AddCheckButton(HWND hWnd)
{
//we have to destroy the old one before expanding
CommandBar_Destroy(hwndCB);
hwndCB = CommandBar_Create(hInst, hWnd, 1);
// Insert the menu band
Adding Combo Boxes

Compared to buttons, adding a combo box to a command bar menu is a snap. There is really only one sticky point, and it’s something that is siunously woven throughout Windows CE: Unicode. More than any other single thing, Unicode will likely be the memorable porting gotcha of moving your code from Win32 to Win CE.

Here are a few important Unicode rules to keep in mind when porting the user interface of your Win32 application to Windows CE:

- Always use the TEXT( ) macro when passing string literals as function arguments.
- Never assume a reference to a Unicode string is the same as the starting address for the string. When you need an address, use the & operator.
- Don’t assume that a count of characters in a string is the same as the size of the buffer needed to hold the characters. Always multiply the character count (plus a terminal null, if one is necessary) by either sizeof(TCHAR) or sizeof(wchar_t) to calculate storage space.
- Always use the string handling functions that are matched to the declared type of the strings you are manipulating.

Here’s the code to add a combo box to a combo bar:

```c
HWND AddCombo(HWND hWnd)
{
    int i;
    TCHAR tszBuff[24];
    // we have to destroy the old one before expanding
    CommandBar_Destroy(hWndCB);
    hWndCB = CommandBar_Create(hInst, hWnd, 1);
    // Insert the menu band
    CommandBar_InsertMenubar(hWndCB, hInst, IDM_MENU, 0);
    // add bitmaps for buttons
    CommandBar_AddBitmap(hWndCB, hInst, IDB_ONE_TWO, 2, 0, 0);
    // Insert buttons.
    CommandBar_AddBitmap(hWndCB, HINST_COMMCTRL, IDB_STD_SMALL_COLOR, 0, 0, 0);
    // Add the buttons - the flags in the button structs
    // specify the check box behaviors
    CommandBar_AddButtons(hWndCB, 1, tbCheckButton);
    // Add exit button to command band control.
    CommandBar_AddAdornments(hWndCB, 0, 0);
    CommandBar_Show(hWndCB, TRUE);
    return hWndCB;
}
```
// Add a combo box between the view icons and the sort icons.
CommandBar_InsertComboBox (hwndCB, hInst,
    GetSystemMetrics(SM_CXSCREEN) / 4,
    CBS_DROPDOWNLIST | WS_VSCROLL,
    IDC_COMBO_BUTTON, 1);

// Fill in combo box.
for (i = 0; i < 5; i++)
{
    wsprintf (tszBuff, TEXT("Combo List %d"), i);
    SendMessage (hwndCB, IDC_COMBO_BUTTON,
        CB_INSERTSTRING, -1, (LPARAM)tszBuff);
}

// select list item 0
SendMessage (hwndCB, IDC_COMBO_BUTTON, CB_SETCURSEL, 0, 0);

// AddAdornments puts an exit button on the command band control
// It replaces the "Exit" menu item
CommandBar_AddAdornments (hwndCB, 0, 0);
CommandBar_Show(hwndCB, TRUE);
return hwndCB;

Let’s look at the combo box code in a bit more detail.

The CommandBar_InsertComboBox() parameters, in the order shown, are
the command bar handle, the instance handle, the width of the box in device
units (pixels), the style flags, the combo box’s control ID, and the 0-based index
of the button to its right.

Notice that the width of the combo box is calculated using a call to GetSys-

The CommandBar_InsertComboBox() parameters, in the order shown, are
the command bar handle, the instance handle, the width of the box in device
units (pixels), the style flags, the combo box’s control ID, and the 0-based index
of the button to its right.

Notice that the width of the combo box is calculated using a call to GetSys-

The next parameter is the style flags. Here we begin to see some of the
shrinkage of the Win32 API. Windows CE supports only two combo box styles:
CBS_DROPDOWN or CBS_DROPDOWNLIST. The familiar CBS_SIMPLE
style, where the list box is always visible and the current selection is displayed
in the edit control, is not supported under Windows CE. This style was elimi-
nated because it used too much screen real estate.

The CBS_DROPDOWN and CBS_DROPDOWNLIST styles show something
that looks like a single line text edit control, with a dropdown arrow icon dis-
played next to it. The list box is displayed when the user taps the dropdown
arrow control. The CBS_DROPDOWNLIST style has a static text field that
always displays the current selection and only allows users to select from a
predefined list of items. CBS_DROPDOWN has an edit control and allows users to select from the list or provide their own input.

You can combine the combo style with other window style flags as we have done in the preceding call, but again, under Windows CE there is a limited subset of these style flags. A few particularly useful flags are given in Table 1.6.

Table 1.6  My Favorite Combo Control Style Flags

<table>
<thead>
<tr>
<th>STYLE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBS_AUTOHSCROLL</td>
<td>Horizontally scrolls text as a user types. Without this flag, a user can enter only as much text as the control width allows.</td>
</tr>
<tr>
<td>CBS_DROPDOWN</td>
<td>Displays only an edit control unless the user taps the dropdown arrow next to the edit control. Users can provide their own input or choose from the list.</td>
</tr>
<tr>
<td>CBS_DROPDOWNLIST</td>
<td>Displays only the current list selection unless the user taps the dropdown arrow next to the edit control.</td>
</tr>
<tr>
<td>CBS_LOWERCASE</td>
<td>Converts all input to lowercase.</td>
</tr>
<tr>
<td>CBS_UPPERCASE</td>
<td>Converts all input to uppercase.</td>
</tr>
</tbody>
</table>

Finally, we insert strings into the combo box like this:

```c
TCHAR tszBuff[24];

for (i = 0; i < 5; i++)
{
    wsprintf (tszBuff, TEXT ("Combo List %d"), i);
    SendDlgItemMessage (hwndCB, IDC_COMBO_BUTTON,
        CB_INSERTSTRING, -1, (LPARAM)tszBuff);
}
```

Notice the function `wsprintf()`, which looks a lot like the familiar `sprintf()`, but there are a couple of important differences. Note that the first parameter, `tszBuff`, is of type `TCHAR`.

`TCHAR` resolves to a WCHAR type if Unicode is defined in the Project build settings; otherwise, it resolves to CHAR.

I make a special point of highlighting this because, as I mentioned earlier, Unicode is one of the most difficult and ubiquitous porting issues you will encounter. Using the `TCHAR` type is about the best insurance you’ll find against memory overwrite errors when copying strings. The second parameter is also a key tool for working with Unicode strings. TEXT ("Combo List %d")
converts what is between the double quotes to a Unicode string. Like with ASCII character literals, you use the backslash to precede escape characters like tabs and newlines, and a double backslash to create a string that includes a backslash.

To create Unicode literal strings suitable for list box text, use the TEXT() macro, with the string literal enclosed in double quotes as a parameter.

```
TEXT("This is how to make a Unicode string literal")
```

### The Dropdown Button

The dropdown button is definitely more work than anything we’ve seen so far. It has one great redeeming advantage, though. Visually, the dropdown button has the impact and space requirements of a button with a drop arrow icon next to it. When the user taps it, it displays a dropdown menu. For this reason, it’s an important design option if you have floating menus or submenus in the original Win32 resource file.

If you can keep popup submenus and floating menus logically intact, then you won’t have nearly as much work to do in porting the message handling behind the menu items.

There are three things we have to do to support the dropdown button. First, we add the button to the command bar, which is the same as for other types, except that we specify the TBSTYLE_DROPDOWN in the TBBUTTON structure.

```c
// Command band button initialization structure
TBBUTTON tbDropButton[] = {
    // BitmapIndex Command  State     Style    UserData  String
    {0,         IDM_ONE, TBSTATE_ENABLED, TBSTYLE_DROPDOWN, 0,    0},
    // Add the buttons
    CommandBar_AddButtons(hwndCB, 1, tbDropButton);

HWND    AddDropDownButton( HWND hWnd )
{
    //we have to destroy the old one before expanding
    CommandBar_Destroy( hwndCB );
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    // Insert the menu band
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    //add bitmaps for buttons
    CommandBar_AddBitmap(hwndCB, hInst, IDB_ONE_TWO, 2, 0, 0);
    //Add the buttons
    CommandBar_AddButtons(hwndCB, 1, tbDropButton);
    // Add exit button to command band control.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    CommandBar_Show(hwndCB, TRUE);
    return hwndCB;
}
```
Unlike other button types, however, the dropdown button doesn’t do much to manage its own appearance. When the user taps a dropdown button, the button will change its appearance to look pressed, and send the application a WM_NOTIFY message. Since many control elements can generate these messages, we must check to make sure it’s from our dropdown button.

Here’s how to do this. The lParam of the WM_NOTIFY message is a pointer to a generic notification message header. To determine if the notification is coming from the button, we cast the lParam to a pointer of LPNMHDR type, and then use the pointer to get the value of the lpnHeader->code. The code tells us what kind of notification structure is being provided. In this case, we check for a toolbar notification from a dropdown button by checking the WM_NOTIFY lParam’s data. If that’s what we got, then we call our handler.

```c
        case WM_NOTIFY:
            // Notify messages have a generic header.
            // check it to see what sent the notify message
            lpnHeader = (LPNMHDR)lParam;
            if (lpnHeader->code == TBN_DROPDOWN)
            {
                // if the notify message came from our button,
                // recast lParam to a toolbar notify pointer
                // and call the handler for the drop button
                return HandleDropButton( hWnd, (LPNMHDR)lParam,
                                      (LPNMTOOLBAR)lParam);
            }
```

Here’s where the rubber meets the road, so to speak. We have to display the dropdown menu directly underneath the dropdown button, making sure we don’t cover up the button itself.

```c
long HandleDropButton( HWND hWnd, LPNMHDR lpnHeader,
                       LPNMTOOLBAR lpnDropButton )
{
    RECT rect;
    TPMPARAMS tpm;
    HMENU hMenu;
    if (lpnDropButton->iItem == IDM_ONE)
    {
        // Get the rectangle of the dropdown button.
        SendMessage (lpnHeader->hwndFrom, TB_GETRECT,
                     lpnDropButton->iItem, (LPARAM)&rect);
        // Convert rect into screen coordinates. The rect is
        // considered here to be an array of 2 POINT structures.
        MapWindowPoints (lpnHeader->hwndFrom, HWND_DESKTOP,
                         (LPPOINT)&rect, 2);
        // Prevent the menu from covering the button.
        tpm.cbSize = sizeof (tpm);
```
We get the button’s screen rectangle with a call to SendMessage(). The parameters, in the order shown, are the handle to the button window, which we get from the notify message header, the Windows message constant TB_GETRECT, the control ID of the toolbar button, and the address of a RECT structure which will contain the rectangle coordinates when the function returns.

// Get the rectangle of the dropdown button.
SendMessage (lpnmHeader->hwndFrom, TB_GETRECT,
lpmnDropButton->iItem, (LPARAM)&rect);

The rectangle of the button is reported in parent window coordinates, but we need screen coordinates to draw the menu properly. MapWindowPoints() translates the button rectangle to the screen coordinate systems. Notice that we treat the RECT structure as an array of two POINT structures in this call.

// Convert rect into screen coordinates. The rect is // considered here to be an array of 2 POINT structures.
MapWindowPoints (lpnmHeader->hwndFrom, HWND_DESKTOP,
(LPPOINT)&rect, 2);

We get a bit of help in putting up the menu. We initialize a popup menu tracking structure with the screen coordinates of the button’s rectangle so that it can be excluded when the menu is painted, load the menu resource, and get a handle to it. TrackPopupMenuEx() does the rest.

// Prevent the menu from covering the button.
memset (tpm, 0, sizeof (tpm));
CopyRect (&tpm.rcExclude, &rect);
hMenu = LoadMenu (hInst, TEXT("DROPMENU"));
hMenu = GetSubMenu ( hMenu, 0);
TrackPopupMenuEx (hMenu, TPM_LEFTALIGN | TPM_VERTICAL,
    rect.left, rect.bottom, hWnd, &tpm);
} return 0; }

User Interface 41
I’d like you to notice two things in particular. First, LoadMenu() uses the text macro to create the literal string with the menu name. Filenames, resource names, pathnames, and control text are always Unicode under Windows CE. Second, notice that the alignment, positioning, and upper-left-hand corner of the popup are set in the second, third, and fourth parameters of TrackPopupMenuEx.

This brings us to the point where we can articulate the rules for menu porting decisions.

**Task 5:**

Examine menu resources and eliminate menu items that are superfluous under Windows CE. If there are four or fewer dropdowns, you may be able to port with no changes. If there are popup submenus or floating menus, consider converting to command bar–style menus. If you choose command bar menus, subdivide menu functionality across several command bar menus, creating them as needed. Eliminate menu items that duplicate the command bar adornments.

**Porting Dialogs**

Dialogs are a less superficial porting issue than most of the other items you find in the resource file, for a couple of reasons. First, dialogs invariably include a lot of text items, all of which must be handled as Unicode strings under Windows CE. Second, many desktop application dialogs will be too large to display properly on a CE-size screen. We’ve looked a bit at handling Unicode strings, and we’ll expand that discussion in the chapters ahead. Now let’s look at what to do with dialogs that are too big for the Windows CE screen.

**Task 6:**

Examine the resource file and find dialogs that are too big to display on CE screens. Subdivide their controls into groups based on function, and redesign the groups as pages of a tabbed dialog.

For purposes of illustration, we are going to the user interface of the dialog-based application pictured in Figure 1.5.
Figure 1.5 DataDlg is a Win32 application that uses this dialog box as its main user interface.

Listing 1.1 Win32 source files for the DataDlg example. (continues)
#define IDC_CUTE 1019
#define IDC_SPOCK 1020
#define IDC_THINKER 1021
#define IDC_WHOOPS 1022
#define IDS_WHERE1 61204
#define IDS_WHERE2 61205
#define IDS_WHERE3 61206
#define IDS_PHYS_EVIDENCE1 61207
#define IDS_PHYS_EVIDENCE2 61208
#define IDS_PHYS_EVIDENCE3 61209
#define IDS_OFFICE1 61210
#define IDS_OFFICE2 61211
#define IDS_OFFICE3 61212
#define IDS_ABDUCTIONS1 61213
#define IDS_ABDUCTIONS2 61214
#define IDS_ABDUCTIONS3 61215

// DataDlg.cpp : Defines the entry point for the application.
//
#include <windows.h>
#include <stdlib.h>
#include <stdafx.h>
#include "resource.h"
//Function Prototypes
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK ModelessDlgProc (HWND, UINT, WPARAM, LPARAM);
void InitUFODialog(HWND);

// Global variables
static char szAppName[] = "DataDlg";  //Name string for this app
static HWND ghDlgModeless;           //Handle to modeless dialog
static HINSTANCE ghInst;             //instance handle for this module

//dim macro calculates # of struct elements
#define dim(x) (sizeof(x) / sizeof(x[0]))

// Structure associates list ctrls with string resources
struct CtrlInit {
    UINT CtrlID;
    UINT StringResBaseID;
    UINT uiMessage;
    UINT uiSel;
};

const struct CtrlInit CtrlStringsStruct[] = {
    //ctrl id          base string id         ctrl
    IDC_WHERE_SEEN,   IDS_WHERE1,     CB_INSERTSTRING,    CB_SETCURSEL,
    IDC_PHYS_EVIDENCE,IDS_PHYS_EVIDENCE1,
};

Listing 1.1 Win32 source files for the DataDlg example.
IDC_OFFICE,       IDS_OFFICE1,   LB_INSERTSTRING,   LB_SETCURSEL,
IDC_ABDUCTIONS,  IDS_ABDUCTIONS1, CB_INSERTSTRING,  CB_SETCURSEL,
};

int WINAPI WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR     lpCmdLine,
                    int       nCmdShow)
{
    HWND       hwnd ;
    MSG        msg ;
    WNDCLASSEX wndclass ;

    //register the main window class
    wndclass.cbSize        = sizeof (wndclass) ;
    wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc   = WndProc ;
    wndclass.cbClsExtra    = 0 ;
    wndclass.cbWndExtra    = 0 ;
    wndclass.hInstance     = hInstance ;
    wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
    wndclass.hCursor       = LoadCursor (hInstance, szAppName) ;
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName  = NULL ;
    wndclass.lpszClassName = szAppName ;
    wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;
    RegisterClassEx (&wndclass) ;

    //save the instance handle in a global variable
    ghInst = hInstance ;

    //create & show the main frame window
    hwnd = CreateWindow (szAppName, "Data Entry Dialog",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL) ;
    ShowWindow (hwnd, nCmdShow) ;
    UpdateWindow (hwnd) ;
    // Use CreateDialog to create modeless dialog box.
    ghDlgModeless =
        CreateDialog (ghInst,"UfoDlgBox", hwnd, ModelessDlgProc);
    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{

Listing 1.1 Win32 source files for the DataDlg example. (continues)
// this procedure is hollow, because we are only
// dealing with the user interface of the dialog right now
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
// Modeless ClearList dialog box procedure.
//
BOOL CALLBACK ModelessDlgProc (HWND hWnd, UINT wMsg,
WPARAM wParam,
LPARAM lParam)
{

HWND hWndCtrl;
long lCurrScrollPos = 0;
switch (wMsg)
{
    case WM_INITDIALOG:
        InitUFODialog(hWnd );
        break;
    case WM_HSCROLL:
        // lParam is the handle to the scrollbar ctrl window
        hWndCtrl = (HWND) lParam;
        // we store the current scroll
        // pos in the window's user data
        // words -- under ce this buffer size
        // must be a multiple of 4 bytes
        lCurrScrollPos = GetWindowLong( hWndCtrl,
            GWL_USERDATA );
        switch( LOWORD( wParam ) )
        {
            // now recalculate the scroll bar
            // position and redraw
            case SB_LINELEFT:
            case SB_PAGELEFT:
                lCurrScrollPos =
                    ( lCurrScrollPos - 1 > 0 ) ?
                    lCurrScrollPos - 1 : 1 ;
                SetScrollPos(hWndCtrl, SB_CTL,
                    (int)lCurrScrollPos, TRUE );
                break;
            case SB_LEFT:
                lCurrScrollPos = 1;
                SetScrollPos(hWndCtrl, SB_CTL,
                    (int)lCurrScrollPos, TRUE );
                break;
            case SB_RIGHT:
                lCurrScrollPos = 5;
                SetScrollPos(hWndCtrl, SB_CTL,
                    (int)lCurrScrollPos, TRUE );
                break;
        }
}

Listing 1.1 Win32 source files for the DataDlg example.
case SB_LINERIGHT:
case SB_PAGERIGHT:
    lCurrScrollPos =
        ( lCurrScrollPos + 1 < 6 ) ?
        lCurrScrollPos + 1 : 5 ;
    SetScrollPos(hWndCtrl, SB_CTL,
        (int)lCurrScrollPos, TRUE );
    break;
default:
    break;
}//end scroll bar message switch
//write the new scroll pos value to the window bytes
SetWindowLong( hWndCtrl, GWL_USERDATA,
    lCurrScrollPos );
break;
case WM_COMMAND:
    switch (LOWORD (wParam))
    {
        //we're just tourists at this point
        case IDOK:
            //Fall thru!!
        case IDC_CLEARCTRLS:
            //Fall thru!!
        case IDCANCEL:
            //send WM_CLOSE message to the main
            // window and quit the app
            SendMessage( (GetParent(hWnd)),
                WM_CLOSE, 0,0 );
            // Modeless dialog boxes can't
            // use EndDialog.
            DestroyWindow( hWnd);
            return TRUE;
        default:
            break;
    }
    break;
}
return FALSE;

Listing 1.1 Win32 source files for the DataDlg example. (continues)
return TRUE;
    
    break;
  
  return FALSE;

//Init the Dialog
void InitUFODialog(HWND hWndDlg )
{
    int i, j, iStringID;
    HWND hWndCtl;
    char szBuffer[128];
    for( j = 0; j < dim( CtrlStringsStruct ); j++ )
    {
        hWndCtl = GetDlgItem( hWndDlg, CtrlStringsStruct[j].CtrlID );
        for( i = 0, iStringID = CtrlStringsStruct[j].StringResBaseID;
            i < 3; i++, iStringID++ )
        {
            LoadString( ghInst, iStringID,
                        szBuffer, sizeof(szBuffer) );
            //add the string to the end of the list
            SendMessage( hWndCtl, CtrlStringsStruct[j].uiMessage,
                         -1, (LPARAM)(LPSTR)szBuffer );
        } //end for i...
        SendMessage( hWndCtl, CtrlStringsStruct[j].uiSel,0, 0 );
    } //end for j...

    //set ranges for the scoll bars
    hWndCtl = GetDlgItem( hWndDlg, IDC_FRIENDLY );
    SetScrollRange( hWndCtl, SB_CTL, 1, 5, TRUE );
    SetScrollPos( hWndCtl, SB_CTL, 1, TRUE );
    //store the current scroll bar pos in the ctrl's window's
    //user data bytes
    SetWindowLong( hWndCtl, GWL_USERDATA, 1 );

    hWndCtl = GetDlgItem( hWndDlg, IDC_TALKATIVE );
    SetScrollRange( hWndCtl, SB_CTL, 1, 5, TRUE );
    SetScrollPos( hWndCtl, SB_CTL, 1, TRUE );
    SetWindowLong( hWndCtl, GWL_USERDATA, 1 );
}

Listing 1.1 Win32 source files for the DataDlg example.

All we really need to observe about the code for the UFO dialog is that it’s
modeless, so we can’t use EndDialog() to dismiss it, and that we’ve conven-
iently omitted all the code that processes and stores dialog input. From the
screenshot in Figure 1.1, we can easily determine that this dialog is much too
large to display on a CE screen. You may not have the luxury of screen shots of
all your application’s dialogs, so a good place to look in order to assess the dimensions of a dialog that’s a candidate for porting is (you knew it) the resource file.

Here’s an excerpt from DataDlg.rc that gives the dimensions of the UFO dialog.

UFODLGBOX DIALOGEX 0, 0, 422, 237
STYLE WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "UFO Data Management Form X"
FONT 8, "MS Sans Serif"
BEGIN

LTEXT           "&Name", IDC_STATIC, 5, 5, 20, 8
EDITTEXT        IDC_NAME, 35, 5, 70, 15, ES_AUTOHSCROLL
LTEXT           "&Badge Number", IDC_STATIC, 5, 30, 48, 8
EDITTEXT        IDC_BADGE, 65, 30, 40, 15, ES_AUTOHSCROLL
LTEXT           "&Description of Encounter", IDC_STATIC, 5, 55, 90, 8
EDITTEXT        IDC_DESC, 30, 65, 75, 40, ES_MULTILINE | ES_AUTOVSCROLL
GROUPBOX        "&Category", IDC_STATIC, 25, 115, 80, 50
CONTROL         "Sighting", IDC_CAT, "Button",
                  BS_AUTORADIOBUTTON | WS_GROUP |
                  WS_TABSTOP, 30, 130, 65, 10
CONTROL         "Contact", IDC_RADIO2, "Button", BS_AUTORADIOBUTTON |
                  WS_TABSTOP, 30, 145, 70, 10
GROUPBOX        "&Mode of Travel", IDC_STATIC, 25, 165, 80, 65, WS_GROUP
CONTROL         "Space Ship", IDC_SPACESHIP, "Button", BS_AUTOCHECKBOX
                  WS_TABSTOP, 30, 180, 70, 10
CONTROL         "Levitation", IDC_LEVITATION, "Button", BS_AUTOCHECKBOX
                  WS_TABSTOP, 30, 195, 65, 10
CONTROL         "Bicycle", IDC_BICYCLE, "Button", BS_AUTOCHECKBOX |
                  WS_TABSTOP, 30, 210, 70, 10
LTEXT                "Where Seen (simple combo)", IDC_STATIC, 140, 5, 90, 8
COMBOBOX        IDC_WHERE_SEEN, 140, 20, 90, 45, CBS_SIMPLE | CBS_SORT | CBS_VSCROLL |
                  WS_TABSTOP
LTEXT                "&Physical Evidence", IDC_STATIC, 145, 75, 59, 8
LTEXT                "(dropdown combo)", IDC_STATIC, 145, 85, 65, 8
COMBOBOX        IDC_PHYS_EVIDENCE, 140, 100, 85, 50,
                  CBS_DROPDOWN | CBS_SORT |
                  WS_VSCROLL | WS_TABSTOP
SCROLLBAR       IDC_FRIENDLY, 120, 145, 105, 11
LTEXT                "Friendliness", IDC_STATIC, 160, 160, 38, 8
SCROLLBAR       IDC_TALKATIVE, 120, 185, 105, 11
LTEXT                "Willingness to Communicate", IDC_STATIC, 130, 200, 89, 8
LTEXT                "&Home Office (list box)", IDC_STATIC, 255, 5, 68, 8, 0, 0,
                  HIDC_STATIC
LISTBOX         IDC_OFFICE, 245, 20, 85, 40, LBS_SORT |
                  LBS_NOINTEGRALHEIGHT |
                  WS_VSCROLL | WS_TABSTOP
Here’s what we learn from this excerpt. First, the dialog size is 422 by 237 screen units. It might fit on the screen of a PC Pro Windows CE device, but on anything else it would be too large. Windows CE supports the same set of pre-defined controls as Win32 (buttons, listboxes, and so on), so we really have only one concern about the individual controls at this point: their size. We’ll make the edit controls smaller, but since they can be set to scroll both horizontally and vertically, shrinking them has no impact on their functionality.

**Using Windows CE Tabbed Dialogs**

The easiest and most obvious way to restructure the UFO dialog is to subdivide its functional groups of controls and organize them as the pages of a tabbed dialog. This style of dialog is called a property sheet. It has a lot of advantages. Porting is quick and requires little change in the dialog’s message handling logic. You can reuse a lot of the resource and header files from the original application. The existing message processing logic must be subdivided into functions that handle each of the pages, but in most other ways, dialog handling code goes straight across. And finally, existing help files and user documentation won’t need drastic modification, because the application works in the same general way as before.

If you are familiar with the format of the text-based resource file, you can save a lot of time by copying strings and manifest constants from the Win32 application into the CE side files. If this approach has no appeal, here’s how to
use the Visual C++ resource editor tools to copy controls from one dialog to another.

- In the Resource Editor, open the dialog template from which you want to copy controls or strings.
- Select what you want to copy by clicking on it. An outline box with handles will appear around it at the corners and midpoints of the enclosing rectangle.
- Press Ctrl-C or use Edit.Copy from the pull down menu.
- Again in the resource editor, open or insert the dialog you want to paste the control into. (Right click over the Dialog folder of the Resource workspace tab to open the menu that lets you insert a new dialog.)
- Use Ctrl-V or Edit.Paste to copy the control to the new dialog template.

In this fashion you can work between the Win32 dialog and the Property Sheet page templates using the clipboard to copy and paste controls and static text. Though this approach can become a bit tedious, it copies the ID’s of the controls along with their size and appearance. This saves time when connecting the controls to their processing logic because there is less chance of making errors like assigning different symbolic constants the same value.

From a user’s point of view, a property sheet behaves like a dialog with tabbed pages. When they tap on a tab, the corresponding page is displayed. From a programmer’s point of view, it works a little differently. The individual pages are treated as discreet dialogs, each of which has its own dialog procedure. The tabs across the top of the property sheet behave as another control, one that sends your application WM_NOTIFY messages. These messages tell when a page is losing or gaining focus, and give the individual dialog procedures a chance to acquire and validate input. A dialog page that is about to lose the focus also has the opportunity to prevent this from happening. This is useful if you want to require a given page’s controls be completely filled in or when you can only accept validated control input.

Here’s the source code for the tabbed dialog. Header and resource files are omitted here, but can be found on the accompanying CD.

```cpp
// TabbedDlg.cpp : Defines the entry point for the application.
/
#include "stdafx.h"
#include "TabbedDlg.h"
#include "resource.h"
#include <commctrl.h>
#include <commdlg.h>                 // Common dialog box includes
#include <prsht.h>                   // Property sheet includes
#define MAX_LOADSTRING 100
#define dim(x) (sizeof(x) / sizeof(x[0]))
// Global Variables:
```
HINSTANCE hInst;                      // The current instance
HWND hwndCB;                          // The command bar handle

// Forward declarations of functions included in this code module:
ATOM MyRegisterClass (HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);
BOOL CreatePropertySheet(HWND);

// Property Sheet Dialog Procs
BOOL CALLBACK AgentDlgProc (HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam);
BOOL CALLBACK EncounterDlgProc (HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam);
BOOL CALLBACK ActionsDlgProc (HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam);
BOOL CALLBACK AppearanceDlgProc (HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam);
BOOL CALLBACK SceneDlgProc (HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam);
BOOL CALLBACK DescriptionDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam);
void InitComboCtrls (HWND, int, int);

int WINAPI WinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPTSTR    lpCmdLine,
int       nCmdShow)
{
MSG msg;
HACCEL hAccelTable;
// Perform application initialization:
if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}

hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_TABBEDDLG);
// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
return msg.wParam;
}

//
// FUNCTION: MyRegisterClass()
//
// PURPOSE: Registers the window class.
//
// COMMENTS:
//
// This function and its usage is only necessary if you want this code to be compatible with Win32 systems prior to the 'RegisterClassEx' function that was added to Windows 95. It is important to call this function so that the application will get 'well formed' small icons associated with it.

ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_TABBEDDLG));
    wc.hCursor = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = 0; // don't load a menu, we'll use a command bar
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);
}

//
// FUNCTION: InitInstance(HANDLE, int)
//
// PURPOSE: Saves instance handle and creates main window
//
// COMMENTS:
//
// In this function, we save the instance handle in a global variable and create and display the main program window.

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    TCHAR szTitle[MAX_LOADSTRING];                   // The title bar text
    TCHAR szWindowClass[MAX_LOADSTRING];             // The window class

// Store instance handle in our global variable
hInst = hInstance;
// Initialize global strings
LoadString(hInstance, IDC_TABBEDDLG, szWindowClass, MAX_LOADSTRING);
MyRegisterClass(hInstance, szWindowClass);
LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
 hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE, 0, 0,
 CW_USEDEFAULT, NULL, NULL, hInstance,
 NULL);
// fail and bail if we didn't get a window
if (!hWnd)
{
    return FALSE;
}
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
return TRUE;

// FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
// PURPOSE:  Processes messages for the main window.
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    switch (message)
    {
    case WM_COMMAND:
        wmId = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_SHOW_TABBED_DLG:
            CreatePropertySheet(hWnd);
            break;
        case IDM_HELP_ABOUT:
            DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX,
                hWnd, (DLGPROC)About);
            break;
        case IDM_FILE_EXIT:
            DestroyWindow(hWnd);
            break;
        }
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
break;
case WM_CREATE:
    // create and show a command bar menu
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    CommandBar_AddAdornments(hwndCB, 0, 0);
    break;
case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    RECT rt, rt1;
    int DlgWidth, DlgHeight;    // dialog width and height in pixel units
    int NewPosX, NewPosY;
    switch (message)
    {
        case WM_INITDIALOG:
            // trying to center the About dialog
            if (GetWindowRect(hDlg, &rt1)) {
                GetClientRect(GetParent(hDlg), &rt);
                DlgWidth = rt1.right - rt1.left;
                DlgHeight = rt1.bottom - rt1.top;
                NewPosX = (rt.right - rt.left -
                          DlgWidth)/2;
                NewPosY = (rt.bottom - rt.top -
                          DlgHeight)/2;
                NewPosX = (rt.right - rt.left -
                          DlgWidth)/2;
                NewPosY = (rt.bottom - rt.top -
                          DlgHeight)/2;
            }
            // if the About box is larger than the physical screen
            if (NewPosX < 0) NewPosX = 0;
            if (NewPosY < 0) NewPosY = 0;
            SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                         0, 0, SWP_NOZORDER  | SWP_NOSIZE);
        }
        return TRUE;
        case WM_COMMAND:
            if ((LOWORD(wParam) == IDOK)  ||  (LOWORD(wParam) ==
                IDCANCEL))
            {
                // User Interface 55
EndDialog(hWnd, LOWORD(wParam));
    return TRUE;
  }
  break;
}
return FALSE;
}
//----------------------------------------------------------------------
// DoMainCommandShowProp - Process show property sheet command.
//
BOOL CreatePropertySheet(HWND hWnd)
{
  PROPSHEETPAGE pspPropPage[6];
  PROPSHEETHEADER pshPropSheet;
  int i;
  // Zero all the property page structures.
  memset(&pspPropPage, 0, sizeof(pspPropPage));
  // Fill in default values in property page structures.
  for (i = 0; i < dim(pspPropPage); i++)
  {
    pspPropPage[i].dwSize = sizeof(PROPSHEETPAGE);
    pspPropPage[i].dwFlags = PSP_DEFAULT;
    pspPropPage[i].hInstance = hInst;
    pspPropPage[i].lParam = (LPARAM)hWnd;
  }
  // Set the dialog box templates for each page.
  pspPropPage[0].pszTemplate = MAKEINTRESOURCE(IDD_AGENT_DIALOG);
  pspPropPage[1].pszTemplate = MAKEINTRESOURCE(IDD_ENCOUNTER_DIALOG);
  pspPropPage[2].pszTemplate = MAKEINTRESOURCE(IDD_SCENE_DIALOG);
  pspPropPage[3].pszTemplate = MAKEINTRESOURCE(IDD_ACTIONS_DIALOG);
  pspPropPage[4].pszTemplate = MAKEINTRESOURCE(IDD_APPEARANCE_DIALOG);
  pspPropPage[5].pszTemplate = MAKEINTRESOURCE(IDD_DESCRIPTION_DIALOG);
  // Set the dialog box procedures for each page.
  pspPropPage[0].pfnDlgProc = AgentDlgProc;
  pspPropPage[1].pfnDlgProc = EncounterDlgProc;
  pspPropPage[2].pfnDlgProc = SceneDlgProc;
  pspPropPage[5].pfnDlgProc = DescriptionDlgProc;
  // Initialize property sheet structure.
  pshPropSheet.dwSize = sizeof(PROPSHEETHEADER);
  pshPropSheet.dwFlags = PSH_PROPSHEETPAGE;
  pshPropSheet hwndParent = hWnd;
  pshPropSheet.hInstance = hInst;
  pshPropSheet.pszCaption = TEXT("UFO Dialog");
  pshPropSheet.nPages = dim(pspPropPage);
  pshPropSheet.nStartPage = 0;
  pshPropSheet.ppsp = pspPropPage;
  pshPropSheet.pfnCallback = 0;
  // Create and display property sheet.
BOOL CALLBACK AgentDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    HWND hWndCtl;
    switch (message)
    {
    case WM_INITDIALOG:
        // init the Home Office Combo
        hWndCtl = GetDlgItem( hDlg, IDC_OFFICE );
        InitComboCtrls(hWndCtl, IDS_OFFICE1, 3);
        return TRUE;
    case default:
    break;
    }
    return FALSE;
}

BOOL CALLBACK EncounterDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    HWND hWndCtl;
    switch (message)
    {
    case WM_INITDIALOG:
        // init the Category Combo
        hWndCtl = GetDlgItem( hDlg, IDC_CAT );
        InitComboCtrls(hWndCtl, IDS_CAT1, 2);
        // init the Mode of Travel Combo
        hWndCtl = GetDlgItem( hDlg, IDC_TRAVEL );
        InitComboCtrls(hWndCtl, IDS_TRAVEL1, 3);
        return TRUE;
    case WM_INITDIALOG:
    break;
    }
    return FALSE;
}

BOOL CALLBACK ActionsDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    HWND hWndCtrl;
    switch (message)
    {
    case WM_INITDIALOG:
        // init the Abductions Combo
        hWndCtrl = GetDlgItem( hDlg, IDC_ABDUCTIONS );
        InitComboCtrls(hWndCtrl, IDS_ABDUCTIONS, 3);
        return TRUE;
    case WM_INITDIALOG:
    case default:
    break;
    }
    return FALSE;
}
InitComboCtrls( hWndCtrl, IDS_ABDUCTIONS1, 3 );
//set ranges for the scroll bars
hWndCtrl = GetDlgItem( hDlg, IDC_FRIENDLY );
SetScrollRange( hWndCtrl, SB_CTL, 1, 5, TRUE );
SetScrollPos( hWndCtrl, SB_CTL, 1, TRUE );
//store the current scroll bar pos in the ctrl's
//window's
//user data bytes
SetWindowLong( hWndCtrl, GWL_USERDATA, 1 );
//now do the same init for the 2nd scroll bar
hWndCtrl = GetDlgItem( hDlg, IDC_TALKATIVE );
SetScrollRange( hWndCtrl, SB_CTL, 1, 5, TRUE );
SetScrollPos( hWndCtrl, SB_CTL, 1, TRUE );
SetWindowLong( hWndCtrl, GWL_USERDATA, 1 );
return TRUE;
case WM_HSCROLL:
    //lparam is the handle to the scrollbar ctrl window
    hWndCtrl = (HWND) lParam;
    //we store the current scroll pos in the window's user
    //data
    //words -- under ce this buffer size must be a
    //multiple of 4 bytes
    lCurrScrollPos = GetWindowLong( hWndCtrl, GWL_USERDATA );
    switch( LOWORD( wParam ) )
    {
        //now recalculate the scroll bar position and
        //redraw
        case SB_LINELEFT:
            case SB_PAGELIST:
                lCurrScrollPos =
                    ( lCurrScrollPos - 1 > 0 ) ?
                    lCurrScrollPos - 1 : 1 ;
                SetScrollPos(hWndCtrl, SB_CTL,
                    (int)lCurrScrollPos, TRUE );
                break;
        case SB_LEFT:
            lCurrScrollPos = 1;
            SetScrollPos(hWndCtrl, SB_CTL,
                (int)lCurrScrollPos, TRUE );
            break;
        case SB_RIGHT:
            lCurrScrollPos = 5;
            SetScrollPos(hWndCtrl, SB_CTL,
                (int)lCurrScrollPos, TRUE );
            break;
        case SB_LINERIGHT:
            case SB_PAGERIGHT:
                lCurrScrollPos =
                    ( lCurrScrollPos + 1 < 6 ) ?
                    lCurrScrollPos + 1 : 5 ;
                SetScrollPos(hWndCtrl, SB_CTL,
                    (int)lCurrScrollPos, TRUE );
...break;
default:
    break;
} // end scroll bar message switch
// write the new scroll pos value to the window bytes
SetWindowLong(hWndCtrl, GWL_USERDATA, lCurrScrollPos);

break;
case default:
    break;

default:
    break;
}
return FALSE;
}
BOOL CALLBACK AppearanceDlgProc(HWND hDlg, UINT message,
WPARAM wParam, LPARAM lParam)
{
    return FALSE;
}
BOOL CALLBACK SceneDlgProc(HWND hDlg, UINT message,
WPARAM wParam, LPARAM lParam)
{
    HWND hWndCtl;
    switch (message)
    {
    case WM_INITDIALOG:
        hWndCtl = GetDlgItem(hDlg, IDC_WHERE_SEEN);
        InitComboCtrls(hWndCtl, IDS_WHERE1, 3);
        hWndCtl = GetDlgItem(hDlg, IDC_PHYS_EVIDENCE);
        InitComboCtrls(hWndCtl, IDS_PHYS_EVIDENCE1, 3);
        return TRUE;
    case default:
        break;
    }
    return FALSE;
}
BOOL CALLBACK DescriptionDlgProc(HWND hDlg, UINT message,
WPARAM wParam, LPARAM lParam)
{
    return FALSE;
}
// init the combos
void InitComboCtrls(HWND hwndCtrl, int iBaseStringId, int
iNumberOfStrings)
{
    TCHAR szBuffer[48];
int i;
LONG lOk;
for(i = 0; i < iNumberStrings; i++, iBaseStringId++ )
{
  LoadString( hInst, iBaseStringId, szBuffer, sizeof(szBuffer) );
  //add the string to the end of the list
  lOk = SendMessage( hwndCtrl, CB_INSERTSTRING, -1, (LPARAM)(LPSTR)szBuffer );
} //end for i...

lOk = SendMessage( hwndCtrl, CB_SETCURSEL, 0, 0 );

**Setting Up a Property Sheet**

In the TabbedDlg example, we launch the property sheet in response to a drop-down menu choice:

```c
case IDM_SHOW_TABBED_DLG:
    CreatePropertySheet(hWnd);
    break;
```

After we split the UFO dialog into a set of smaller templates, there are three more things we have to do to turn them into a property sheet: Initialize an array of property sheet structures, call the `PropertySheet()` function to create the control, and subdivide the original dialog message handling code into units that match the controls in each of the new dialog templates for the property pages.

Here’s the declaration of the PROPSHEETPAGE structure, more or less as found in the Windows CE prsht.h header file. (I eliminated some of the conditionals to make it more readable and added the comments about the uses of the members.)

```c
typedef struct _PROPSHEETPAGE {
    DWORD           dwSize;        //size of this structure
    DWORD           dwFlags;       //flags set behaviors and
    HINSTANCE   hInstance;     //instance handle of this app
    union
    {
        LPCSTR          pszTemplate; // name of dialog template
        LPCDLGTEMPLATE  pResource;  //pointer to a memory
        int i;
    }
}';
As you can see from the structure declaration, the PROPSHEETPAGE effectively defines the appearance and behavior of a page, tying it to a dialog procedure and a dialog template. Here’s how we set up a group of pages for the UFO Property Sheet:

```c
BOOL CreatePropertySheet(HWND hWnd)
{
    PROPSHEETPAGE pspPropPage[6];
    PROPSHEETHEADER pshPropSheet;
    int i;
    // Zero all the property page structures.
    memset (&pspPropPage, 0x0, sizeof (pspPropPage));

    // Fill in default values in property page structures.
    for (i = 0; i < dim(pspPropPage); i++)
    {
        pspPropPage[i].dwSize = sizeof (PROPSHEETPAGE);
        pspPropPage[i].dwFlags = PSP_DEFAULT;
        pspPropPage[i].hInstance = hInst;
        pspPropPage[i].pszTitle = NULL; // null terminated title string
        pspPropPage[i].pszIcon = NULL; // must be set to NULL
        pspPropPage[i].pszTemplate = NULL; // must be set to NULL
        pspPropPage[i].pfnDlgProc = NULL; // ptr to the DlgProc for the page
        pspPropPage[i].lParam = (LPARAM)hWnd; // optional user data passed in
        pspPropPage[i].pcRefParent = NULL; // optional ptr to reference count.
    }

    return TRUE;
}
```
Next, we loop through the array and set the members that are the same for all of the structures.

```
// Set the dialog box templates for each page.
pspPropPage[0].pszTemplate = MAKEINTRESOURCE(IDD_AGENT_DIALOG);
pspPropPage[1].pszTemplate = MAKEINTRESOURCE(IDD_ENCOUNTER_DIALOG);
pspPropPage[2].pszTemplate = MAKEINTRESOURCE(IDD_SCENE_DIALOG);
pspPropPage[3].pszTemplate = MAKEINTRESOURCE(IDD_ACTIONS_DIALOG);
pspPropPage[4].pszTemplate = MAKEINTRESOURCE(IDD_APPEARANCE_DIALOG);
pspPropPage[5].pszTemplate = MAKEINTRESOURCE(IDD_DESCRIPTION_DIALOG);
```

Now we’ve associated the dialog template for each of the pages, identifying the templates with their manifest constants.

```
// Set the dialog box procedures for each page.
pspPropPage[0].pfnDlgProc = AgentDlgProc;
pspPropPage[1].pfnDlgProc = EncounterDlgProc;
pspPropPage[2].pfnDlgProc = SceneDlgProc;
pspPropPage[5].pfnDlgProc = DescriptionDlgProc;
```

Finally, we tie each of the property pages to the dialog procedure that processes it’s control messages.

We’ve defined a group of pages for the property sheet. Now all that’s left to do is initialize the structure that collects the pages in a property sheet control.

```
// Initialize property sheet structure.
pshPropSheet.dwSize = sizeof (PROPSHEETHEADER);
pshPropSheet.dwFlags = PSH_PROPSHEETPAGE;
pshPropSheet.hwndParent = hWnd;
pshPropSheet.hInstance = hInst;
pshPropSheet.pszCaption = TEXT("UFO Dialog");
pshPropSheet.nPages = dim(pspPropPage);
pshPropSheet.nStartPage = 0;
pshPropSheet.ppsp = pspPropPage;
pshPropSheet.pfnCallback = 0;
```

Finally, we create the control:

```
// Create and display property sheet.
PropertySheet (&pshPropSheet);
return 0;
}
Here is the code for one of the dialog procedures that handles the Encounter-Dlg property page:

```c
BOOL CALLBACK EncounterDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    HWND hWndCtl;
    switch (message)
    {
    case WM_INITDIALOG:
        //init the Category Combo
        hWndCtl = GetDlgItem( hDlg, IDC_CAT );
        InitComboCtrls( hWndCtl, IDS_CAT1, 2 );
        //init the Mode of Travel Combo
        hWndCtl = GetDlgItem( hDlg, IDC_TRAVEL );
        InitComboCtrls( hWndCtl, IDS_TRAVEL1, 3 );
        return TRUE;
    case default:
        break;
    }
    return FALSE;
}
```

This, like the other property page functions in this example, is really just a placeholder. We wanted to see the page without becoming involved in exchanging or validating data just yet. There are, however, a couple of significant things about property page functions we should touch on before continuing.

Property sheets communicate with their pages by means of WM_NOTIFY messages and the notification message header structure, NMHDR. We used both of these in the MenuBar example to aid in properly displaying the drop button menu.

Notice that, in the previous example code, when we called the handler for the menu bar drop button, the lParam of the WM_NOTIFY is used as pointer to the NMHDR structure and to the LPNMTOOLBAR structure.

```c
//if the notify message came from our button,
//recast lParam to a toolbar notify pointer
//and call the handler for the drop button
return HandleDropButton( hWnd, (LPNMHDR)lParam,
                        (LPNMTOOLBAR)lParam);
```

All notification messages have at least an NMHDR, pointed to by the lParam of the message. The structure is defined in winuser.h and looks like this:

```c
typedef struct tagNMHDR
{
    HWND hwndFrom;  //handle of the window the message is from
```
The NMHDR was designed to be an extensible message format, so that controls could send complex and detailed messages to their parent window. If there is more information than fits in an NMHDR, the header is reallocated, appended with the extra information, and the code field is updated with a value that identifies the type and amount of additional information intended for the target of the notify message.

Context specific notification messages are how property sheets allow individual pages to do specialized initialization processing, clean up before the property sheet is destroyed, and manage the transfer of focus. Table 1.7 lists some of the more important property sheet notification messages and their meanings.

The time to validate a page’s data is in response to the PSN_KILLACTIVE notification. If data passes validation, a property page does two things to tell the property sheet that it’s ready to lose the input focus.

The property page sets the return value of its window structure to PSNRET_NOERROR with code like this:

```
SetWindowLong( hwndThisPage, DWL_MSGRESULT,
              PSNRET_NOERROR);
```

<table>
<thead>
<tr>
<th>MESSAGE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSN_APPLY</td>
<td>The user tapped the OK button and wants to apply changes</td>
</tr>
<tr>
<td>PSN_HELP</td>
<td>User tapped the Help button</td>
</tr>
<tr>
<td>PSN_KILLACTIVE</td>
<td>The page is losing focus, either because another page is gaining the focus or because the user tapped OK</td>
</tr>
<tr>
<td>PSN_QUERYCANCEL</td>
<td>User tapped Cancel</td>
</tr>
<tr>
<td>PSN_RESET</td>
<td>User tapped Cancel and the page is about to be destroyed</td>
</tr>
<tr>
<td>PSN_SETACTIVE</td>
<td>The page is about to be activated</td>
</tr>
</tbody>
</table>
The property page returns TRUE. To keep the page from losing focus, set the window structure return value to PSNRET_INVALID_NOCHANGEPAGE:

```c
SetWindowLong( hwndThisPage, DWL_MSGRESULT, PSNRET_INVALID_NOCHANGEPAGE);
```

On a PSN_APPLY notification, a page should get and save control input. You can respond to a PSN_QUERYCANCEL notification either by returning FALSE to allow the closure of the property sheet, or returning TRUE, which will prevent the sheet from closing.

**Looking Ahead**

The dialog has assumed a natural role on the desktop as a common front end to data collection applications. Limited screen size makes dialogs much less effective for this purpose on Windows CE devices. We can break a dialog into pages and stack them in a property sheet, but this approach has some drawbacks. Users may overlook pages of the property sheet or forget which ones they have visited. If you want to validate input, you must do so before a user leaves a page of the property sheet, and prevent that page from losing focus until the user provides valid input. This has the potential to be very frustrating and confusing to the user (Table 1.8).

A large share of these problems arise from the fact that simply making a large Win32 dialog into a stack of smaller CE compliant dialogs really isn’t in the spirit of Windows CE. If collecting data from users is our objective, then we need to create a user interface that uses screen real estate effectively, but is also relatively self-documenting and easy for the user to navigate. In the next chapter, we’ll learn how to use command bands to create a better data entry interface.

**Table 1.8  Interface Troubleshooting Tips**

<table>
<thead>
<tr>
<th>SYMPTOMS</th>
<th>POSSIBLE CAUSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Icon isn’t displayed</td>
<td>Check icon bitmap dimensions to ensure the image is no more than 16 by 16.</td>
</tr>
<tr>
<td>Toolbar bitmaps aren’t displayed</td>
<td>Bitmaps are the wrong size or contain unsupported colors. Check for contiguous 16-bit images. Use a 2-bit grayscale color palette.</td>
</tr>
<tr>
<td>Some tool bar buttons are blank</td>
<td>Make sure you added as many bitmaps as buttons. Also make sure the bitmap dimensions of individual images in the bitmap file are correct.</td>
</tr>
</tbody>
</table>

(continues)
<table>
<thead>
<tr>
<th>SYMPTOMS</th>
<th>POSSIBLE CAUSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combo/List Box strings are garbled or absent</td>
<td>Make sure the strings are in Unicode before adding them to the control. If they are being converted from ASCII, make sure to translate them using mbstowcs() or similar functions. Check to ensure translation buffers are correctly sized and typed. The destination buffer should be typed TCHAR or WCHAR.</td>
</tr>
<tr>
<td>Combo boxes or menu bars don’t appear or are truncated in a command bar</td>
<td>Check physical sizes to make sure the elements aren’t too large for the screen real estate they are supposed to occupy. For combo boxes, use the return from GetSystemMetrics(SM_CXSCREEN) to calculate the control’s width.</td>
</tr>
<tr>
<td>Combo box won’t allow entry of text wider than its window</td>
<td>Specify the CBS_AUTOHSCROLL style in the CommandBar_InsertComboBox() styles parameter.</td>
</tr>
<tr>
<td>Menu or other named resource won’t load</td>
<td>Use the TEXT() macro to create string literals that specify resource names, filenames or path names.</td>
</tr>
</tbody>
</table>
In Chapter 1 we saw how to rapidly and simply move a user interface from Win32 to Windows CE. But the fastest way isn’t always the best way, or even an adequate way. From my point of view, the place this becomes most glaringly apparent is in the case of form-based data entry applications.

On the desktop, dialog box–based forms are a good, self-documenting way to collect data from users. For Windows CE devices, because of screen size limitations, most nontrivial dialog-based forms need to be reimplemented as a set of tabbed dialog pages. In terms of the user’s prior experience, reorganizing one dialog-based form into a series of dialogs representing the same form sets up a subtle contradiction. Tabbed dialogs on the desktop usually imply a collection of related but independent information. Using a tabbed dialog to represent a form violates the user’s intuition and prior experience about what is expected of her. Should all of the tabs be visited? In order? Is there an obvious way to tell when input is required and when input has been validated? The tabbed dialog has real drawbacks, because in most cases, the information in a form is related and dependent. A first name is meaningless without a last name, a shipping address is not useful without a billing address, and so forth.
Command Bands: The Win CE Forms

Solution

Windows CE CommandBands provide the tool for creating intuitive forms that accommodate the limitations of the HPC and PPC, wringing meaning out of every last pixel on the tiny Windows CE screen. The CommandBands approach to form design allows you to retain the visual metaphor of a single form, but in a way that efficiently uses screen resources.

CommandBands controls are related to the CommandBar control we used to replace an application’s resource-based menu in Chapter 1’s MenuBar example. In the MenuBar example, we saw that CommandBars act as containers for controls, managing painting and visual changes in the control states, and message routing from the controls to the window procedure. CommandBands build on and extend this concept by providing a container for CommandBars. This powerful construct lets us create a form composed of an aggregation of optionally moveable, resizable CommandBars. Each CommandBar contains one control. To put it another way, the CommandBand represents the form as a whole, and the CommandBars represent the individual fields of the form.

This approach overcomes the limitations of the tabbed dialog as a form-based data entry interface. First, it allows you to put all of the fields of the form on the screen at once, because the user can resize CommandBands. For example, you can show a few fields at full size, and the rest as grippers the user can drag to open the field. Second, it simplifies and rationalizes data validation, because the fields in a form enjoy the unifying relationship established by containment in the CommandBands. And finally, it improves your options if you choose not to let users move and resize the bands containing the fields. In this case, you can create pages of bands that users navigate using next/previous controls. This kind of navigation behavior is less prone to confusing the user than the tabbed dialog approach, and it is more amenable to application-specific specialization.

The first example shows how to create and use CommandBands (see Listing 2-1). Creating and using CommandBands entails these steps:

1. Create the CommandBands control, setting the overall band style, the base band ID (the lowest individual band ID in the control), and optionally, the controls image list.
2. Allocate an array of REBARBANDINFO structures, one per band.
3. Initialize each of the band structures.
4. Add the bands.
5. Loop through the bands, inserting a control in the CommandBar contained in each band.

6. Initialize controls as necessary.

```c
DataBands.rc
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////
/////
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////
/////
#define APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////
/////
// English (U.S.) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif _WIN32

////////////////////////////////////////////////////////////////////////
/////
// Icon
// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_DATABANDS ICON DISCARDABLE "DataBands.ICO"
#endif APSTUDIO_INVOKED

////////////////////////////////////////////////////////////////////////
/////
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
 "resource.h\0"
END
2 TEXTINCLUDE DISCARDABLE
BEGIN
 "afxres.h"
END
```

**Listing 2.1** Source code for DataBands. (continues)
Listing 2.1  Source code for DataBands.
Listing 2.1  Source code for DataBands. (continues)
IDS_WHERE2                       "Moonlit Seascape"
IDS_WHERE3                       "Hard to describe"
IDS_PHYS_EVIDENCE1               "Burnt ring of grass"
IDS_PHYS_EVIDENCE2               "Distressed livestock"
IDS_PHYS_EVIDENCE3               "Alien landing party"
IDS_OFFICE1                      "Sedona, Arizona"
IDS_OFFICE2                      "Crestone, Colorado"
IDS_OFFICE3                      "Roswell, New Mexico"
IDS_ABDUCTIONS1                  "Took family pets"
IDS_ABDUCTIONS2                  "Took Cell Phone sales person"

END
STRINGTABLE DISCARDABLE
BEGIN
IDS_ABDUCTIONS3                  "Returned chemistry teacher"
IDS_APPEARANCE1                  "Scary"
IDS_APPEARANCE2                  "Cute"
IDS_APPEARANCE3                  "Dressed Like L. Nemoy"
IDS_APPEARANCE4                  "Not Dressed"
IDS_APPEARANCE5                  "Can't Recall"
IDS_FRIENDLY1                    "Really Friendly"
IDS_FRIENDLY2                    "So So"
IDS_FRIENDLY3                    "Standoffish"
IDS_TALKATIVE1                   "Pleasantly chatty"
IDS_TALKATIVE2                   "Shy but sensitive"
IDS_TALKATIVE3                   "Taciturn"
END
#endif    // English (U.S.) resources

Listing 2.1  Source code for DataBands.
#endif //
!defined(AFX_DATABANDS_H__CFBABE68_8CDA_11D5_8B0A_00A0D2168AEA__INCLUDED _)
#define IDC_BAND_BASE_ID       99            //BAND 0 IS THE MENU BAND
#define IDC_NAME               100
#define IDC_BADGE              101
#define IDC_DESC               102
#define IDC_CAT                103
#define IDC_MODE_TRAVEL        104
#define IDC_WHERE_SEEN         105
#define IDC_PHYS_EVIDENCE      106
#define IDC_OFFICE             107
#define IDC_ABDUCTIONS         108
#define IDC_FRIENDLY           109
#define IDC_TALKATIVE          110
#define IDC_APPEARANCE         111
#define IDC_COMBO_BASE_ID      200

DataBands.cpp
// DataBands.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "DataBands.h"
#include <commctrl.h>
#define MAX_LOADSTRING 100
#define MIN_BAND_WIDTH  50
//this is an easy & flexible way to set a loop index
//for iterating an array of structures
#define dim(x) (sizeof(x) / sizeof(x[0]))
struct CmdBarParms {
    UINT    uiID;
    UINT    uiCaption;
    long    lStyles;
    int     iStringIndexBase;
    int     iNumStrings;
};
// Setup information for the controls that implement
//the fields of the forms
const struct CmdBarParms structCmdBarInit[] = {
    //Ctrl ID, //Label text, //Ctrl Styles, //String Index, //#strings,
    IDC_NAME, IDS_NAME, ES_AUTOHSCROLL, NULL, 0,
    IDC_BADGE, IDS_BADGE, ES_AUTOHSCROLL, NULL, 0,
    IDC_DESC, IDS_DESC, ES_MULTILINE | ES_AUTOVSCROLL, NULL, 0,
    IDC_CAT, IDS_CAT, CBS_DROPDOWNLIST, IDS_CAT1, 2,
    IDC_MODE_TRAVEL, IDS_MODE_TRAVEL, CBS_DROPDOWNLIST, IDS_MODE_TRAVEL1, 3,
    IDC_WHERE_SEEN, IDS_WHERE_SEEN, CBS_DROPDOWNLIST, IDS_WHERE1, 3,
    IDC_PHYS_EVIDENCE, IDS_PHYS_EVIDENCE, CBS_DROPDOWNLIST, IDS_PHYS_EVIDENCE1, 3,
};

Listing 2.1  Source code for DataBands. (continues)
Listing 2.1  Source code for DataBands.
FUNCTION: MyRegisterClass()

PURPOSE: Registers the window class.

COMMENTS:

This function and its usage is only necessary if you want this code to be compatible with Win32 systems prior to the 'RegisterClassEx' function that was added to Windows 95. It is important to call this function so that the application will get 'well formed' small icons associated with it.

ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(hInstance,
        MAKEINTRESOURCE(IDI_DATABANDS));
    wc.hCursor = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = 0;
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);
}

FUNCTION: InitInstance(HANDLE, int)

PURPOSE: Saves instance handle and creates main window

COMMENTS:

In this function, we save the instance handle in a global variable and create and display the main program window.

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    TCHAR szTitle[MAX_LOADSTRING]; // The title bar text
    TCHAR szWindowClass[MAX_LOADSTRING]; // The window class name
    // Store instance handle in our global variable

Listing 2.1 Source code for DataBands. (continues)
hInst = hInstance;
// Initialize global strings
LoadString(hInstance, IDC_DATABANDS,
    szWindowClass, MAX_LOADSTRING);
MyRegisterClass(hInstance, szWindowClass);
LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
hWnd = CreateWindow(szWindowClass, szTitle,
    WS_VISIBLE,
    0, 0,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL, NULL, hInstance,
    NULL);

// if no window, fail and bail
if (!hWnd)
{
    return FALSE;
}
// it's all good.  Show the window and command bar menu
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
return TRUE;
}

// FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
// PURPOSE: Processes messages for the main window.
// WM_COMMAND - process the application menu
// WM_PAINT - Paint the main window
// WM_DESTROY - post a quit message and return
//
// LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
//  WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    switch (message)
    {
        case WM_COMMAND:
            wmId    = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // process the menu selections:
            switch (wmId)
            {
                case IDM_HELP_ABOUT:

Listing 2.1  Source code for DataBands.
Listing 2.1  Source code for DataBands. (continues)
if (NewPosY < 0) NewPosY = 0;
SetWindowPos(hDlg, 0, NewPosX, NewPosY,
0, 0, SWP_NOZORDER | SWP_NOSIZE);
}
return TRUE;
case WM_COMMAND:
if ((LOWORD(wParam) == IDOK) ||
(LOWORD(wParam) == IDCANCEL))
{
    EndDialog(hDlg, LOWORD(wParam));
    return TRUE;
}
break;
}
return FALSE;
}

LPARAM CreateDataBands(HWND hWnd, UINT message,
WPARAM wParam,
LPARAM lParam)
{
    HBITMAP      hBmp;
    LPREBARBANDINFO prbi;
    HWND hwndBand, hwndCombo;
    int i, j, rc = 0, nBandsPerRow;
    TCHAR tszCaptionBuff[34];
    TCHAR tszPlatType[256];
    //initialize nBands = # ctrls + the main menu band
    nBands = dim( structCmdBarInit ) + 1;
    // Create a command band; save the handle in a global variable. We need to preserve it to access the CommandBars control
    hwndCB = CommandBands_Create(hInst, hWnd,
        IDC_BAND_BASE_ID,
        RBS_BANDBORDERS |
        RBS_AUTOSIZE |
        RBS_FIXEDORDER, NULL);
    // Load bitmap used as background for command bar.
    hBmp = LoadBitmap (hInst, TEXT("CmdBarBack"));
    // Allocate space for the REBARBANDINFO array
    // Need one entry per ctl band ( nBands ) plus the menu
    prbi = (LPREBARBANDINFO)LocalAlloc(LPTR,
        (sizeof (REBARBANDINFO) * nBands) );
    //always test returns if you attempt to allocate memory
    if (!prbi) {
        MessageBox( hWnd, TEXT("LocalAlloc Failed"),
            TEXT("Oops!")), MB_OK);
        return 0;
    }
// Initialize common REBARBANDINFO structure fields.
for (i = 0; i < nBands; i++) {
    prbi[i].cbSize = sizeof (REBARBANDINFO);
    prbi[i].fMask = RBBIM_ID |
                    RBBIM_SIZE |
                    RBBIM_BACKGROUND |
                    RBBIM_STYLE |
                    RBBIM_IDEALSIZE ;
    prbi[i].wID    = IDC_BAND_BASE_ID+i;
    prbi[i].hbmBack = hBmp;
}

// Initialize REBARBANDINFO structure for Menu band
prbi[0].cx = GetSystemMetrics(SM_CXSCREEN ) / 4;
prbi[0].fStyle = prbi[1].fStyle = RBBS_NOGRIPPER |
                 RBBS_BREAK;

// Set bands per row based on platform type
memset( &tszPlatType, 0x0, sizeof(tszPlatType));
rc = SystemParametersInfo( SPI_GETPLATFORMTYPE, 
                          sizeof(tszPlatType), &tszPlatType, 0);
if( lstrcmp( tszPlatType, TEXT("Jupiter") ) == 0 )
    { nBandsPerRow = 4;}
else if( lstrcmp( tszPlatType, TEXT("HPC") ) == 0 )
    { nBandsPerRow = 3;}
else if( lstrcmp( tszPlatType, TEXT("Palm PC") ) == 0 )
    { nBandsPerRow = 1;}

// Initialize data entry band attributes
for (i = 1; i < nBands; i++)
{
    // Common Combobox ctrl band attributes
    prbi[i].fMask |= RBBIM_IDEALSIZE;
    prbi[i].cx = (GetSystemMetrics(SM_CXSCREEN ) - 20 ) 
                 / nBandsPerRow ;
    prbi[i].cxIdeal = (GetSystemMetrics(SM_CXSCREEN ) - 20 ) 
                     / nBandsPerRow ;
    // Set style for line break at the end of a row
    if(((i - 1 ) % nBandsPerRow ) == 0 )
    {
        prbi[i].fStyle |= RBBS_BREAK | RBBS_NOGRIPPER;
    }
    else
    {
        prbi[i].fStyle |= RBBS_NOGRIPPER;
    }
}

// Add the bands-- we've reserved real estate
// for our dlg controls.
CommandBands_AddBands (hwndCB, hInst, nBands, prbi);
    //Move past the menu bar and set the control behaviors
    for (i = 1; i < nBands; i++)
    {
        hwndBand = CommandBands_GetCommandBar (hwndCB, i);
        CommandBar_InsertComboBox (hwndBand, hInst, 
            prbi[i].cx - 6, 
            structCmdBarInit[i].lStyles,
            IDC_COMBO_BASE_ID + (i - 1),
            0);

        //get a handle to the combo in this band
        hwndCombo = GetDlgItem(hwndBand, IDC_COMBO_BASE_ID +
            (i - 1));
        //get the caption string
        LoadString(hInst, structCmdBarInit[i - 1].uiCaption,
            (LPTSTR)&tszCaptionBuff,sizeof(tszCaptionBuff));
        //insert the label string
        SendMessage (hwndCombo, CB_INSERTSTRING, 0,
            (long)&tszCaptionBuff);
        //highlight the caption
        SendMessage (hwndCombo, CB_SETCURSEL, 0, 0);
        //does this combo have a string list?
        if(structCmdBarInit[i - 1].iNumStrings)
        {
            //if so, insert them in the combo
            for( j = 0;
                j < structCmdBarInit[i - 1].iNumStrings;
                j++ )
            {
                LoadString(hInst, 
                    structCmdBarInit[i - 1].iStringIndexBase + j,
                    (LPTSTR)&tszCaptionBuff,sizeof(tszCaptionBuff));
                //add strings to the end of the list
                SendMessage (hwndCombo, CB_INSERTSTRING, -1,
                    (long)&tszCaptionBuff);
            } //end for(iNumStrings)
        } //end if(iNumStrings)
    }

    // Add menu to band 0 --wahoo
    hwndBand = CommandBands_GetCommandBar (hwndCB, 0);
    CommandBar_InsertMenubar (hwndBand, hInst, IDM_MENU, 0);
    // Add exit button to command band control.
    CommandBands_AddAdornments (hwndCB, hInst, 0, 0);
    //we're done with this now
    LocalFree(prbi);
    return 0;
}
Steps to a Form Using CommandBands

Let’s assume that the sole purpose of our application is to collect data using the UFO CommandBands-based form. We’ll call CreateDataBands(), the function that creates the form when we process the WM_CREATE message for the application. This has the same functional effect as if we created a modeless dialog. In other words, the form will persist until we destroy it or until the user closes the application. Here’s the code:

```c
case WM_CREATE:
    // build the bands that replace the 
    // Win32 dialog box form
    CreateDataBands(hWnd, message, wParam, lParam);
    break;
```

The job of building the bands is handled in the function CreateDataBands(). Notice the parameters to this function are simply copies of the parameters to WindProc that came with the WM_CREATE message. Now let’s look more closely at the CreateDataBands() function. First, notice this declaration:

```c
LPREBARBANDINFO prbi;
```

This datum is a long pointer to a REBARBANDINFO structure. This structure contains the information used to size, position, and specify the behavior and appearance of each band. This is a subtlety that I can’t emphasize enough. The band is the real estate where the CommandBar and its control live. If you want to access the bar and, by extension, its contained control, you have to do so through its band.

**PORTING TIP** As you are most probably aware, all of these controls are implemented as windows, and they have ownership relationships among themselves. I mention this because if you are porting a real application, it’s very tempting to try clever solutions that will preserve your existing code, and a lot of Win32 forms code does interact directly with the windows that underlie dialog elements. There are very legitimate reasons for doing this (data validation and input masking are two good examples). I have found that trying to treat any of the CommandBands constituents (bands, bars, or controls) as individual windows (as opposed to bands that contain bars that contain controls) produces highly unpredictable behavior.

Here is the declaration of the REBARBANDINFO structure, along with the meanings of its members:
typedef struct tagREBARBANDINFO
{
    UINT cbSize;        // size of this structure
    UINT fMask;         // flags that tell either what
                        // info is being passed
                        // or what info is being requested
    UINT fStyle;        // flags for band styles
    COLORREF clrFore;   // band foreground color
    COLORREF clrBack;   // band background color
    LPSTR lpText;       // ptr to band label text
    UINT cch;           // size of text in the band label in bytes
    int iImage;         // 0 based index of image list image
                        // to be displayed in this band
    HWND hwndChild;     // handle to child window contained in band
    UINT cxMinChild;    // min child window width in pixels
    UINT cyMinChild;    // min child window height in pixels
    UINT cx;            // band width, pixels
    HBITMAP hbmBack;    // handle to a bitmap for band background
                        // if this member is filled,
                        // clrFore and clrBack are ignored
    UINT wID;           // ID the CommandBand control uses to
                        // notify this band
    UINT cyChild;       // initial height of child window
    UINT cyMaxChild;    // max child height - not used unless
                        // style RBBS_VARIABLEHEIGHT is set
    UINT cyIntegral;    // step value for band height increase
    UINT cxIdeal;       // ideal band width in pixels
    LPARAM lParam;      // application specific data
} REBARBANDINFO

Whew. It’s a truckload of structure. In most cases, you’ll only need to fill in
a few of these members to create a CommandBands control. One structure is
required for each band in the control. We use this structure to initialize an
existing CommandBands control.

Here’s how to create the control:

    // Create a command band; save the handle in a global
    // variable. We need to preserve it to access the CommandBands
    // control
    hwndCB = CommandBands_Create(hInst, hWnd,
        IDC_BAND_BASE_ID,
        RBS_BANDBORDERS |
        RBS_AUTOSIZE |
        RBS_FIXEDORDER, NULL);

Notice that we don’t specify any details about the individual bands, not
even their number. The parameters to CommandBands_Create(), in the order
shown, are the instance handle of this app, the main window’s handle, the ID
of the first band in the control (which is the zeroth band), and the style flags that define the appearance and behavior of the CommandBands control. These styles apply to all the individual bands. RBS_BANDBORDERS means the bands will be separated by thin lines that give the control a gridded appearance. RBS_AUTOSIZE means the bands will be sized to fit within the control if the position of the control changes. RBS_FIXEDORDER means the bands will maintain their initial ordering, even if the user moves a band using its gripper. The last parameter is the optional handle to an image list. We aren’t using any images in this control, so it’s NULL here. If successful, CommandBands_Create() returns the handle to the control.

Next, we allocate an array of REBARBANDINFO structures.

    // Allocate space for the REBARBANDINFO array
    // Need one entry per ctl band (nBands) plus the menu
    prbi = (LPREBARBANDINFO)LocalAlloc(LPTR,
        (sizeof(REBARBANDINFO) * nBands) );

    //always test returns if you attempt to allocate memory
    if (!prbi) {
        MessageBox( hWnd, TEXT("LocalAlloc Failed"),
            TEXT("Oops!"), MB_OK);
        return 0;
    }

These structures are quite large, and it’s a definite possibility that your allocation request could fail. Always check the return from an allocation request. Also, notice that we called LocalAlloc() with the LPTR flag as its first parameter. This flag allocates fixed memory that is zero initialized. It’s important to initialize the structures before you use them because many of the fields and their interpretations are interdependent. For example, if there is a value in the hbmBack member, the members for foreground and background colors are ignored. The fmask flags define your intentions about what structure members are valid and how they should be used, but if you zero initialize the structures, you won’t be as vulnerable to subtleties of the mask flags.

There are several REBARBANDINFO structure fields that are the same for every band. We initialize these first.

    // Initialize common REBARBANDINFO structure fields.
    for (i = 0; i < nBands; i++) {
        prbi[i].cbSize = sizeof(REBARBANDINFO);
        prbi[i].fMask = RBBIM_ID |
            RBBIM_SIZE |
            RBBIM_BACKGROUND |
            RBBIM_STYLE |
            RBBIM_IDEALSIZEx;
        prbi[i].wID    = IDC_BAND_BASE_ID+i;
The structure members initialized here, in the order they appear, are the size field, which gives the size of the REBARBANDINFO structure; the mask flags; the ID of this band, calculated using the band base ID and the loop index; and the handle to the bitmap that serves as the background for this band.

It may seem superfluous to pass the size of the structure, but notice that when we call the function that adds the bands using this structure, we pass its address. Since the very first member of the structure contains the size, the called function can accurately determine the size of the structure by dereferencing only the DWORD at the address in the pointer. Supplying the structure size makes for more reliable allocation behavior in the called function.

**PORTING TIP** Always initialize the size member of the structure if it has one.

The fmask member of the structure tells which other members of the REBARBANDINFO structure are valid. Some of the masks define more than one member to be valid or specifically exclude others from being valid. The ones specified in the call above validate the following members of the structure:

- wID
- cx
- hbmBack
- fStyle
- cxIdeal

Recall that in the MenuBar example in Chapter 1, the CommandBar menu replaced the resource-based menu. When you use a CommandBands control, the CommandBar in the 0th band serves as the menu. For this reason, the 0th REBARBANDINFO is initialized differently than the bands that hold the controls that make up the form.

```c
// Initialize REBARBANDINFO structure for Menu band
prbi[0].cx = GetSystemMetrics(SM_CXSCREEN) / 4;
prbi[0].fStyle = prbi[1].fStyle = RBBS_NOGRIPPER | RBBS_BREAK;
```

We limit the width of the menu bar to one-quarter of the screen dimension, but it could be larger, because no other bands are inserted between it and the Adornments. We set its style flags and the style flags of the immediately following band to RBBS_NOGRIPPER | RBBS_BREAK. This is an important
point because the RBBS_NOGRIPPER style means that the user can’t move the
band to a new location by dragging it with the gripper control. We want the
menu to stay at the top of the screen, so we explicitly eliminate the gripper,
which is a default without this style.

We also specify the RBBS_BREAK, which means the band begins on a new
line. (Of course, the 0th band always begins on a new line.) To make sure the 0th
band occupies the full screen width, the band following it must have the
RBBS_BREAK style as well.

Next, we prepare to initialize the structures for the bands that contain the
controls. This is where we get an initial glimpse of the effect of variability in
Windows CE device sizes. Obviously, the bigger devices can legibly display
more bands in a single row of screen real estate that the smaller devices. We
choose the number of bands per row based on runtime information about the
platform type of the host.

```c
//Set bands per row based on platform type
memset( &tszPlatType, 0x0, sizeof(tszPlatType));
rc = SystemParametersInfo( SPI_GETPLATFORMTYPE,
                         sizeof(tszPlatType),
                         &tszPlatType, 0);

if( lstrcmp( tszPlatType, TEXT("Jupiter") ) == 0 )
{ nBandsPerRow = 4;}
else if( lstrcmp( tszPlatType, TEXT("HPC") ) == 0 )
{ nBandsPerRow = 3;}
else if( lstrcmp( tszPlatType, TEXT("Palm PC") ) == 0 )
{ nBandsPerRow = 1;}
```

Notice that to distinguish between the different form factors, we use the Sys-
temParametersInfo() function. The flag SPI_GETPLATFORMTYPE requests a
string that describes the class of the Windows CE device on which we are
running, rather than an OS version. Here’s why this is the most reliable way to
distinguish between hosts at runtime.

The Windows CE Platform Developers’ Kit is designed in a highly compo-
nentized fashion, essentially allowing device vendors to ”roll their own” version
of the OS to implement on their devices. This offers them a great competitive
advantage and the opportunity to innovate. Not surprisingly, they do. For this
reason, Windows CE version numbers don’t provide the same kind of definite
information as Windows version numbers on the desktop, because hardware
developers have great latitude in choosing what parts of CE they implement
and how they do it. Put another way, you can’t safely or effectively use OS ver-
sion numbers for decision making at runtime.

The real business of creating the form is accomplished when we initialize
the structures for the bands that contain the form’s fields.
Initialize data entry band attributes
for (i = 1; i < nBands; i++)
{
    // Common ComboBox ctrl band attributes
    prbi[i].fMask |= RBBIM_IDEALSIZE;
    prbi[i].cx = (GetSystemMetrics(SM_CXSCREEN) - 20) / nBandsPerRow;
    prbi[i].cxIdeal = (GetSystemMetrics(SM_CXSCREEN) - 20) / nBandsPerRow;
    // Set style for line break at the end of a row
    if (((i - 1) % nBandsPerRow) == 0)
    {
        prbi[i].fStyle |= RBBS_BREAK | RBBS_NOGRIPPER;
    }
    else
    {
        prbi[i].fStyle |= RBBS_NOGRIPPER;
    }
}

Notice that we OR the RBBIM_IDEALSIZE flag to the fmask member. Next we set the initial size, cx, and the ideal band size, cxIdeal, to a fraction of the screen width that is determined by the platform type. Finally, we use a conditional to assign styles that will assure that we have the correct number of bands in each row. The structure array is fully initialized, and we are ready to add the bands to the CommandBands control.

Add the bands-- we've reserved real estate
// for our dlg controls.
CommandBands_AddBands(hwndCB, hInst, nBands, prbi);

The parameters to the CommandBands_AddBands() function, in the order shown, are the handle to the CommandBands control to which the bands are being added, the instance handle, the number of bands being added (this is the count of the bands, not the highest zero based index), and the address of the array of REBARBANDINFO structures.

We have a CommandBands control, the control is initialized with bands, and now all that remains is to insert the form’s controls and set their behaviors. A word about the strategy behind the form building steps is in order here. We conserve screen real estate in a couple of different ways using command bands. First, and most obvious, is that the CommandBands control itself is fairly condensed. The second is that we eliminate the need for control captions by making all of the controls in the form list boxes or combo boxes.

If the field really is a list (that is, users must pick one or more items but can’t add items of their own), then the first item in the list is the string that would
normally be the caption text for the control if we were using a dialog-based form. If, on the other hand, the user can type input to the control (the combo box style), then the first item in the combo box list is the caption, and user input to the edit control is always inserted at the end of the list.

Initially, all the controls are displayed with the caption string selected, which has the effect of labeling the controls, but without using additional screen real estate to do so. If the user visits a control and provides input, we change the current list selection from the field’s caption string to the user’s typed input or list selection. This has the dual advantages of making it easy for the user to see where she has entered data and easy for the programmer to test for and validate input.

Now let’s set up the controls for the form. Here’s the initialization array where we store the initialization information for each control.

```c
// Setup information for the controls that implement //the fields of the forms
const struct CmdBarParms structCmdBarInit[] = {
    //Ctrl ID, //Label text, //Ctrl Styles, //String Index, //=strings
    IDC_NAME, IDS_NAME, ES_AUTOHSCROLL, NULL, 0,
    IDC_BADGE, IDS_BADGE, ES_AUTOHSCROLL, NULL, 0,
    IDC_DESC, IDS_DESC, ES_MULTILINE | ES_AUTOVSCROLL, NULL, 0,
    IDC_CAT, IDS_CAT, CBS_DROPDOWNLIST, IDS_CAT1, 2,
    IDC_MODE_TRAVEL, IDS_MODE_TRAVEL, CBS_DROPDOWNLIST, IDS_MODE_TRAVEL1, 3,
    IDC_WHERE_SEEN, IDS_WHERE_SEEN, CBS_DROPDOWNLIST, IDS_WHERE1, 3,
    IDC_PHYS_EVIDENCE, IDS_PHYS_EVIDENCE, CBS_DROPDOWNLIST, IDS_PHYS_EVIDENCE1, 3,
    IDC_OFFICE, IDS_OFFICE, CBS_DROPDOWNLIST, IDS_OFFICE1, 3,
    IDC_ABDUCTIONS, IDS_ABDUCTIONS, CBS_DROPDOWNLIST, IDS_ABDUCTIONS1, 3,
    IDC_FRIENDLY, IDS_FRIENDLY, CBS_DROPDOWNLIST, IDS_FRIENDLY1, 3,
    IDC_TALKATIVE, IDS_TALKATIVE, CBS_DROPDOWNLIST, IDS_TALKATIVE1, 3,
    IDC_APPEARANCE, IDS_APPEARANCE, CBS_DROPDOWN, IDS_APPEARANCE1, 5
};
```

To set up the controls, we loop the bands of the CommandBand control and the control initialization array, inserting controls and setting their attributes using the control ID, label text string, and control styles specified in the structCmdBarInit array.

```c
//Move past the menu bar and set the control behaviors
for (i = 1; i < nBands; i++) {
    hwndBand = CommandBands_GetCommandBar (hwndCB,i);
    CommandBar_InsertComboBox (hwndBand, hInst,
        prbi[i].cx - 6,
        structCmdBarInit[i].lStyles,
        IDC_COMBO_BASE_ID + (i - 1),
        0);
}```
Next, we initialize each control with its list of strings. If the control is intended to behave as an edit control, it has only one string: its caption. If the control is a list, its array entry specifies its caption, a manifest constant that gives the index of its first string resource in the string table, and a count of the strings.

To initialize the control, you must get its handle from the band that contains it.

```c
hwndCombo = GetDlgItem(hwndBand, IDC_COMBO_BASE_ID + (i - 1));
```

To recap, the steps necessary to gain access to an individual control are:

- Get a handle to a band by calling CommandBands_GetCommandBar()
- Get a handle to the dialog control GetDlgItem() for the hwnd returned by the previous step

Next, load and select the string for the caption. (Every control needs this initialization.)

```c
//get the caption string
LoadString(hInst, structCmdBarInit[i - 1].uiCaption, (LPTSTR)&tszCaptionBuff, sizeof(tszCaptionBuff));
//insert the label string
SendMessage(hwndCombo, CB_INSERTSTRING, 0, (long)&tszCaptionBuff);
//highlight the caption
SendMessage(hwndCombo, CB_SETCURSEL, 0, 0);
```

Test the number of list strings. If this control needs a list of selection items, we’ll insert the list strings now.

```c
//does this combo have a string list?
if(structCmdBarInit[i - 1].iNumStrings)
{
```

The strings are stored in the resource file in the order that they appear in the list, and so they have consecutive resource IDs. We load the strings by incrementing the based string ID and limit the loop iteration using the count of strings, both of which are stored in the structCmdBarInit structure.

```c
//if so, insert them in the combo
for( j = 0;
    j < structCmdBarInit[i - 1].iNumStrings;
    j ++ )
{
```
LoadString(hInst, 
    structCmdBarInit[i - 1].iStringIndexBase + j, 
    (LPTSTR)&tszCaptionBuff, 
    sizeof(tszCaptionBuff));
    //add strings to the end of the list
    SendMessage (hwndCombo, CB_INSERTSTRING, -1, 
    (long)&tszCaptionBuff);
    }
    }
    
There you have it. We created a form using controls of fixed size and order, 
all of which are fully displayed when the form is created. To query the controls 
for input, you use the same sort of looping construction that we saw in the initial-
ization process, followed with a call to GetDlgItemText() to retrieve user 
input. Here’s a code fragment that demonstrates how to retrieve input from a 
control in a CommandBand:

    //a stack based buffer for the control's input
    TCHAR szSelString[48];
    //get a handle to the bar contained by the band at index i
    hwndBand = CommandBands_GetCommandBar (hwndCB,i);
    //get a handle to the combo in this band
    hwndCombo = GetDlgItem(hwndBand,IDC_COMBO_BASE_ID +
    (i - 1));
    //ask the list what is selected
    j = SendMessage(hWndCombo, CB_GETCURSEL , 0, 0 );
    //copy the string at index j to the buffer
    SendMessage(hWndCombo, CB_GETLBTEXT , j,
    (long)&szSelString );

Here are a couple of things to note about this code fragment. First, notice 
that the type of the buffer szSelString is TCHAR. Control strings, filenames, 
and the like are Unicode under CE. Consistently using TCHAR types when 
dealing with controls saves you from having to think about the physical size of 
strings you are manipulating, and thus greatly reduces the likelihood of mem-
ory overwrites. Second, this code works for single-selection list strings, but not 
for multiple-selection strings. Multiple-selection lists don’t have a current 
selection. For multiple selection lists, you must first query the control for num-
ber of selections, allocate a buffer to hold an array of selection item indices, get 
the sel indices, and then loop through the list or combo control to get the 
strings. Here’s a code fragment that demonstrates this strategy:

    //get the count of selections
    j = SendMessage(hWndCombo, LB_GETSELCOUNT , 0, 0 );
    //allocate an integer array to hold the selection indices

piSelIndices = (LPINT)LocalAlloc( LPTR, sizeof(int) * j;
//ALWAYS check the return of an allocation call
if( ! piSelIndices )
{
//no alloc, fail & bail
return FALSE;
}
//get the array of sel indices
SendMessage(hWndCombo, LB_GETSELITEMS, j, (long)piSelIndices);
//loop thru the array and get the strings at the selection
indices
for( k = 0; k < j; k++, piSelIndices++ )
{
//copy the string at index j to the buffer
SendMessage(hWndCombo, LB_GETLBTEXT, j,
(long)&szSelString );

//do something with the string
}

---

**Really, Really, Really Long Forms**

The preceeding example works quite well on HPC and PC Pro devices, but on a PPC device (or in a really large form), we’d run out of screen real estate before we got all of the controls displayed. You may be wondering what happens if you create and initialize more controls than fit on the screen. It depends. Sometimes nothing really bad happens, other than the fact that you can’t see all of the controls in the bands. You can’t scroll them into view, though, and to be useful, bands and their controls must be visible. Listing 2.2 shows one way you can handle forms that have more than a single screen’s worth of bands. This form in the PagedBands example organizes controls into sets of pages, through which the programmer controls navigation.

---

**Listing 2.2** Source code for PagedBands.
Listing 2.2  Source code for PagedBands. (continues)
Listing 2.2  Source code for PagedBands.
IDC_PAGEDBANDS ACCELERATORS DISCARDABLE
BEGIN
"/", IDM_HELP_ABOUT, ASCII, ALT, NOINVERT
VK_F4, IDM_FILE_EXIT, VIRTKEY, ALT, NOINVERT
END

////////////////////////////////////////////////////////////////////////
//////
// Bitmap
////
IDB_PREV_NEXT BITMAP DISCARDABLE "prev_nex.bmp"
////////////////////////////////////////////////////////////////////////
//////
// String Table
//
STRINGTABLE DISCARDABLE
BEGIN
IDS_APP_TITLE "PagedBands"
IDC_PAGEDBANDS "PAGEDBANDS"
IDS_NAME "Name"
IDS_Badge "Badge"
IDS_DESC "Description"
IDS_CAT "Category"
IDS_MODE_TRAVEL "Travel by..."
IDS_WHERE_SEEN "Where Seen"
IDS_PHYS_EVIDENCE "Physical Evidence"
IDS_OFFICE "Home Office"
IDS_ABDUCTIONS "Abductions"
IDS_FRIENDLY "Friendly"
IDS_TALKATIVE "Communicates"
IDS_APPEARANCE "Appearance"
END
STRINGTABLE DISCARDABLE
BEGIN
IDS_CAT1 "Sighting"
IDS_CAT2 "Contact"
IDS_MODE_TRAVEL1 "Space Ship"
IDS_MODE_TRAVEL2 "Levitation"
IDS_MODE_TRAVEL3 "Bicycle"
IDS_WHERE1 "Moonlit Desert"
IDS_WHERE2 "Moonlit Seascape"
IDS_WHERE3 "Hard to describe"
IDS_PHYS_EVIDENCE1 "Burnt ring of grass"
IDS_PHYS_EVIDENCE2 "Distressed livestock"
IDS_PHYS_EVIDENCE3 "Alien landing party"
IDS_OFFICE1 "Sedona, Arizona"
IDS_OFFICE2 "Crestone, Colorado"

Listing 2.2  Source code for PagedBands. (continues)
IDS_OFFICE3 "Roswell, New Mexico"
IDS_ABDUCTIONS1 "Took family pets"
IDS_ABDUCTIONS2 "Took Cell Phone sales person"

END
STRINGTABLE DISCARDABLE
BEGIN
IDS_ABDUCTIONS3 "Returned chemistry teacher"
IDS_APPEARANCE1 "Scary"
IDS_APPEARANCE2 "Cute"
IDS_APPEARANCE3 "Dressed Like L. Nemoy"
IDS_APPEARANCE4 "Not Dressed"
IDS_APPEARANCE5 "Can't Recall"
IDS_FRIENDLY1 "Really Friendly"
IDS_FRIENDLY2 "So So"
IDS_FRIENDLY3 "Standoffish"
IDS_TALKATIVE1 "Pleasantly chatty"
IDS_TALKATIVE2 "Shy but sensitive"
IDS_TALKATIVE3 "Taciturn"
END
#endif  // English (U.S.) resources

Listing 2.2  Source code for PagedBands.
#define IDC_MODE_TRAVEL 104
#define IDC_WHERE_SEEN 105
#define IDC_PHYS_EVIDENCE 106
#define IDC_OFFICE 107
#define IDC_ABDUCTIONS 108
#define IDC_FRIENDLY 109
#define IDC_TALKATIVE 110
#define IDC_APPEARANCE 111
#define IDC_COMBO_BASE_ID 200

PagedBands.cpp

// DataBands.cpp : Defines the entry point for the application.

#include "stdafx.h"
#include "PagedBands.h"
#include <commctrl.h>
#define MAX_LOADSTRING 100
#define MIN_BAND_WIDTH 50
#define FIVE_BANDS 5
#define NUM_CTRL_BANDS 4

//this is an easy & flexible way to set a loop index
//for iterating an array of structures
#define dim(x) (sizeof(x) / sizeof(x[0]))

struct CmdBarParms {
    UINT uiID;
    UINT uiCaption;
    long lStyles;
    int iStringIndexBase;
    int iNumStrings;
};

// Setup information for the controls that implement
//the fields of the forms
const struct CmdBarParms structCmdBarInit[] = {
    //Ctrl ID, //Label text, //Ctrl Styles, //String Index, //#strings
    IDC_NAME, IDS_NAME, ES_AUTOHSCROLL, NULL, 0,
    IDC_BADGE, IDS_BADGE, ES_AUTOHSCROLL, NULL, 0,
    IDC_DESC, IDS_DESC, ES_MULTILINE | ES_AUTOVSCROLL, NULL, 0,
    IDC_CAT, IDS_CAT, CBS_DROPDOWNLIST, IDS_CAT1, 2,
    IDC_MODE_TRAVEL, IDS_MODE_TRAVEL, CBS_DROPDOWNLIST, IDS_MODE_TRAVEL1, 3,
    IDC_WHERE_SEEN, IDS_WHERE_SEEN, CBS_DROPDOWNLIST, IDS_WHERE1, 3,
    IDC_PHYS_EVIDENCE, IDS_PHYS_EVIDENCE, CBS_DROPDOWNLIST, IDS_PHYS_EVIDENCE1, 3,
    IDC_OFFICE, IDS_OFFICE, CBS_DROPDOWNLIST, IDS_OFFICE1, 3,
    IDC_ABDUCTIONS, IDS_ABDUCTIONS, CBS_DROPDOWNLIST, IDS_ABDUCTIONS1, 3,
    IDC_FRIENDLY, IDS_FRIENDLY, CBS_DROPDOWNLIST, IDS_FRIENDLY1, 3,
    IDC_TALKATIVE, IDS_TALKATIVE, CBS_DROPDOWNLIST, IDS_TALKATIVE1, 3,
    IDC_APPEARANCE, IDS_APPEARANCE, CBS_DROPDOWN, IDS_APPEARANCE1, 5
};

// Command band button initialization structure

Listing 2.2  Source code for PagedBands. (continues)
TBBUTTON tbCmdButtons[] = {
    // BitmapIndex  Button ID     Command  State       Style
    UserData  String
    {0,                      IDM_VIEWPREVPAGE,
     TBSTATE_ENABLED,
     TBSTYLE_BUTTON, 0,    0},
    {1,                        IDM_VIEWNEXTPAGE,
     TBSTATE_ENABLED,
     TBSTYLE_BUTTON, 0,    0},
    TBSTYLE_BUTTON, 0,    0},

}; // Global Variables:
HINSTANCE       hInst;        //The current instance
int          nBands = 4;   //# ctrls we insert in the CommandBands ctrl
int          iCurTopBand = 0;
//index to the 1st visible structCmdBarInit
int          iNumPages;       //# pages needed to display all of our ctrls
// Forward declarations of functions included in this code module:
ATOM MyRegisterClass (HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);
HWND CreateDataBands (HWND);
void ChangePageBands();

int WINAPI WinMain(HINSTANCE hInstance,

Listing 2.2  Source code for PagedBands.
(LPCTSTR)IDC_PAGEDBANDS);

// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
return msg.wParam;

// FUNCTION: MyRegisterClass()
// PURPOSE: Registers the window class.

ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon (hInstance, MAKEINTRESOURCE(IDI_PAGEDBANDS));
    wc.hCursor = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = 0;
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);
}

// FUNCTION: InitInstance(HANDLE, int)
// PURPOSE: Saves instance handle and creates main window

Listing 2.2  Source code for PagedBands. (continues)
98

Chapter 2

//In this function, we save the instance handle in
//a global variable and
//create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
HWND
hWnd;
HWND
hwndCB; // The command bar handle
TCHAR
szTitle[MAX_LOADSTRING]; // The title bar text
TCHAR
szWindowClass[MAX_LOADSTRING];
// The window class name
TCHAR
szAppName[24]; // The window caption
// Store instance handle in our global variable
hInst = hInstance;
// Initialize global strings
LoadString(hInstance, IDC_PAGEDBANDS,
szWindowClass, MAX_LOADSTRING);
MyRegisterClass(hInstance, szWindowClass);
//build a window title that includes page# / total pages for form
LoadString(hInstance, IDS_APP_TITLE, szAppName,
sizeof(szAppName));
wsprintf( szTitle, TEXT(“%s%s%i”),
szAppName, TEXT(“-1/”),
dim(structCmdBarInit) / NUM_CTRL_BANDS );
hWnd = CreateWindow(szWindowClass, szTitle,
WS_VISIBLE,
0, 0,
CW_USEDEFAULT,
CW_USEDEFAULT,
NULL, NULL, hInstance,
NULL);
//if no hwnd, fail & bail
if (!hWnd)
{
return FALSE;
}
//create the form
hwndCB = CreateDataBands(hWnd);
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
//if we got a page of bands, show it now
if (hwndCB)
{
CommandBands_Show(hwndCB, TRUE);
}
else

Listing 2.2 Source code for PagedBands.


Listing 2.2  Source code for PagedBands. (continues)


Listing 2.2  Source code for PagedBands.
DlgHeight = rt1.bottom - rt1.top;
NewPosX  = (rt.right - rt.left - DlgWidth)/2;
NewPosY  = (rt.bottom - rt.top - DlgHeight)/2;

// if the About box is larger than
// the physical screen
if (NewPosX < 0) NewPosX = 0;
if (NewPosY < 0) NewPosY = 0;
SetWindowPos(hDlg, 0, NewPosX, NewPosY,
0, 0, SWP_NOZORDER | SWP_NOSIZE);

Listing 2.2  Source code for PagedBands. (continues)
// Initialize common REBARBANDINFO structure fields.
for (i = 0; i < FIVE_BANDS; i++) {
    prbi[i].cbSize = sizeof (REBARBANDINFO);
    prbi[i].fMask = RBBIM_ID | RBBIM_SIZE |
                    RBBIM_BACKGROUND | RBBIM_STYLE | RBBIM_IDEALSIZE ;
    prbi[i].wID = IDC_BAND_BASE_ID+i;
    prbi[i].hbmBack = hBmp;
    //put each band on its own line and don't
    //let the user move it
    prbi[i].fStyle = RBBS_NOGRIPPER | RBBS_BREAK;
}

// Initialize REBARBANDINFO structure for Menu band
prbi[0].cx = GetSystemMetrics(SM_CXSCREEN ) / 2;

//Initialize data entry band attributes
for (i = 1; i <= NUM_CTRL_BANDS; i++) {
    // Common Combobox ctrl band attributes
    prbi[i].fMask |= RBBIM_ID | RBBIM_SIZE;
    prbi[i].cx = (GetSystemMetrics(SM_CXSCREEN ) - 20 );
    prbi[i].cxIdeal = (GetSystemMetrics(SM_CXSCREEN ) - 20 );
}

// Add the bands-- now we've reserved real estate
// for our dlg controls.
CommandBands_AddBands (hwndCB, hInst, FIVE_BANDS, prbi);

//Move past the menu bar and set the control behaviors
for (i = 1; i <= NUM_CTRL_BANDS; i++) {
    hwndBand = CommandBands_GetCommandBar (hwndCB,i);
    CommandBar_InsertComboBox (hwndBand, hInst,
                    prbi[i].cx - 6,
                    structCmdBarInit[( i - 1 ) + iCurTopBand].lStyles,
                    IDC_COMBO_BASE_ID + (i - 1), 0);
    //get a handle to the combo in this band
    hwndCombo = GetDlgItem(hwndBand,
                    IDC_COMBO_BASE_ID + (i - 1));
    //get the caption string
    LoadString(hInst,
                    structCmdBarInit[( i - 1 ) + iCurTopBand].uiCaption,
                    (LPTSTR)&tszCaptionBuff,sizeof(tszCaptionBuff));
    //insert the label string
    SendMessage (hwndCombo, CB_INSERTSTRING, 0,
                    (long)&tszCaptionBuff);
    //highlight the caption
    SendMessage (hwndCombo, CB_SETCURSEL, 0, 0);
    //does this combo have a string list?
    if(structCmdBarInit[( i - 1 ) + iCurTopBand].iNumStrings)
{  
    // if so, insert them in the combo  
    for( j = 0;
        j < structCmdBarInit[(i - 1) + iCurTopBand].iNumStrings;
        j++ )
    {
        LoadString(hInst,
                   structCmdBarInit[(i - 1) + iCurTopBand].iStringIndexBase + j,
                   (LPTSTR)&tszCaptionBuff,
                   sizeof(tszCaptionBuff));

        // add strings to the end of the list
        SendMessage(hwndCombo, CB_INSERTSTRING, -1,
                    (long)&tszCaptionBuff);
    }  // end for(iNumStrings)
}  // end if(iNumStrings)

// Add menu to band 0 --wahoo
hwndBand = CommandBands_GetCommandBar(hwndCB, 0);
CommandBar_InsertMenubar(hwndBand, hInst, IDM_MENU, 0);
    idBitmap = (int)LoadBitmap(hInst,
                MAKEINTRESOURCE(IDB_PREV_NEXT));
    CommandBar_AddBitmap(hwndBand, NULL, idBitmap, 2, 0, 0);
    // If there's only one page, here's how to
    // hide the next/prev buttons
    iNumPages = dim(structCmdBarInit) / NUM_CTRL_BANDS;
    if( dim(structCmdBarInit) % NUM_CTRL_BANDS)
    {
        iNumPages++;
    }
    if( iNumPages == 1 )
    {
        tbCmdButtons[0].fsState = TBSTATE_HIDDEN;
        tbCmdButtons[1].fsState = TBSTATE_HIDDEN;
    }
    // Add the buttons
    i = CommandBar_AddButtons(hwndBand,
                              sizeof(tbCmdButtons)/sizeof(TBBUTTON),
                              tbCmdButtons);
    // Add exit button to command band control.
    CommandBands_AddAdornments(hwndCB, hInst, 0, 0);
    // change the window title to show what page we are on
    LoadString(hInst, 
               IDS_APP_TITLE, (LPTSTR)&tszCaptionBuff,
               sizeof(tszCaptionBuff));

Listing 2.2  Source code for PagedBands. (continues)
wsprintf( (LPTSTR)&tszNewWinTitle,
    TEXT("\%s\%s\%s\%i"), (LPTSTR)&tszCaptionBuff,
    TEXT("-"), (iCurTopBand + 4) / 4,
    TEXT("/"),
    dim(structCmdBarInit) / NUM_CTRL_BANDS );
SetWindowText( hWnd, (LPCTSTR)&tszNewWinTitle );
//we’re done with this now
LocalFree(prbi);
return hwndCB;
}

Listing 2.2  Source code for PagedBands. (continued)

As you can see, PagedBands builds on the DataBands example. For this reason, we won’t dissect the entire application. Instead, we’ll focus on the significant changes. First, let’s look at how we change the window title to let the user know which page of a multipage form they are viewing. Initially, we set the window title to show that we are on page 1.

//load the app title from the string table
LoadString(hInstance, IDS_APP_TITLE, szAppName,
    sizeof(szAppName));
//append page#/ of pages
wsprintf( szTitle, TEXT("\%s\%s\%i"), szAppName, TEXT("-1/"),
    dim(structCmdBarInit) / NUM_CTRL_BANDS );
//create the window using this title string
hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
    0, 0, CW_USEDEFAULT, CW_USEDEFAULT,
    NULL, NULL, hInstance, NULL);

In the PagedBands example, the total number of bands is an even multiple of the total number of pages. If this weren’t the case, we’d need to take it into account when calculating the number of pages. Here’s a code fragment that calculates the number of pages in a more general way.

iPartialPage =
    ( (dim(structCmdBarInit) % NUM_CTRL_BANDS) > 0 )? 1 : 0;
iNumPages =
    dim(structCmdBarInit) / NUM_CTRL_BANDS + iPartialPage;

This fragment uses the modulus operator to see if there is a non-zero remainder when you divide the total number of bands by the number of bands per screen. If there is, we add one page.
Though it appears to the user that they are flipping through the pages of the form, in reality, the previous CommandBands control must be destroyed and a new one created whenever there is a page change. This requires a few modifications to the CreateDataBands() function. Specifically, we have to add command bar buttons to navigate the form pages and we have to reset the window title to reflect the current page.

```
// Add menu to band 0 --wahoo
hwndBand = CommandBands_GetCommandBar (hwndCB, 0);
CommandBar_InsertMenuBar (hwndBand, hInst, IDM_MENU, 0);
    idBitmap = (int)LoadBitmap(hInst,
        MAKEINTRESOURCE(IDB_PREV_NEXT));
    CommandBar_AddBitmap(hwndBand, NULL, idBitmap, 2, 0, 0);
//If there's only one page, here's how to
// hide the next/prev buttons
iNumPages = dim(structCmdBarInit) / NUM_CTRL_BANDS;
if( dim(structCmdBarInit) % NUM_CTRL_BANDS)
    {
        iNumPages++;
    }
if( iNumPages == 1 )
    {
        tbCmdButtons[0].fsState = TBSTATE_HIDDEN;
        tbCmdButtons[1].fsState = TBSTATE_HIDDEN;
    }
//Add the buttons
i = CommandBar_AddButtons(hwndBand,
    sizeof(tbCmdButtons)/sizeof(TBBUTTON),
    tbCmdButtons);
```

First, we load the images for the Next and Previous buttons, and we add them to the CommandBar. We check to see if there is more than one page worth of fields for this form. If not, we set the navigation button's initial state to TBSTATE_HIDDEN, which keeps them from becoming visible. Otherwise, they retain the TBSTATE_ENABLED, which is specified in the global structure, tbCmdButtons. This step is not necessary to the case at hand, but it is included in the example because it could be very useful for dynamically composed forms. Finally, we add the buttons to the menu band of the CommandBands control.

When the user taps one of the navigation buttons, we get a WM_COMMAND message. This is an excerpt from the message switch in WndProc() that handles the button messages:

```
// switch the menu selections:
switch (wmId)
{
    //move backward one band page and reinit
    case IDM_VIEWPREVPAGE:
```
iCurTopBand -=4;
if(iCurTopBand < 0)
{
    iCurTopBand =
dim(structCmdBarInit) -
    NUM_CTRL_BANDS;
}
hwndCB = GetDlgItem( hWnd,
    IDC_BAND_BASE_ID );
DestroyWindow( hwndCB );
CreateDataBands(hWnd);
break;

//move forward one band page and reinit
case IDM_VIEWNEXTPAGE:
iCurTopBand +=4;
if(iCurTopBand >=
    dim(structCmdBarInit))
{
    iCurTopBand -=
    dim(structCmdBarInit);
}
hwndCB = GetDlgItem( hWnd,
    IDC_BAND_BASE_ID );
DestroyWindow( hwndCB );
CreateDataBands(hWnd);
break;

The real work of the message handling code is to calculate the new value of iCurTopBand. This global variable is used as the index to the structure array that contains initialization information for the fields. When we get a page turning command from the user, we either add or subtract the number of fields in a page from iCurTopBand. If we go beyond the beginning or end of the page range when we do this, then we add or subtract the total number of fields. This has the effect of allowing continuous rollover of pages in either direction.

I Want It All Now

Our last example demonstrates a forms solution that allows you to put a very large number of fields on a single screen, without using paging. Listing 2.3 shows how to make a form using a CommandBands control with moveable, resizeable bands.
MoveAndSize.rc
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////
/////
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"
/////////////////////////////////////////////////////////////////////
/////
#undef APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////
/////
// English (U.S.) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32
/////////////////////////////////////////////////////////////////////
/////
// Icon

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_MOVEANDSIZE ICON DISCARDABLE "MoveAndSize.ICO"
#endif APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////
/////
// TEXTINCLUDE

1 TEXTINCLUDE DISCARDABLE
BEGIN
"resource.h\0"
END
2 TEXTINCLUDE DISCARDABLE
BEGIN

Listing 2.3  Source code for MoveAndSize. (continues)
Listing 2.3  Source code for MoveAndSize.
Listing 2.3  Source code for MoveAndSize. (continues)
<table>
<thead>
<tr>
<th>IDS_CAT</th>
<th>&quot;Category&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDS_MODE_TRAVEL</td>
<td>&quot;Travel by...&quot;</td>
</tr>
<tr>
<td>IDS_WHERE_SEEN</td>
<td>&quot;Where Seen&quot;</td>
</tr>
<tr>
<td>IDS_PHYS_EVIDENCE</td>
<td>&quot;Physical Evidence&quot;</td>
</tr>
<tr>
<td>IDS_OFFICE</td>
<td>&quot;Home Office&quot;</td>
</tr>
<tr>
<td>IDS_ABDUCTIONS</td>
<td>&quot;Abductions&quot;</td>
</tr>
<tr>
<td>IDS_FRIENDLY</td>
<td>&quot;Friendly&quot;</td>
</tr>
<tr>
<td>IDS_TALKATIVE</td>
<td>&quot;Communicates&quot;</td>
</tr>
<tr>
<td>IDS_APPEARANCE</td>
<td>&quot;Appearance&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDS_CAT1</th>
<th>&quot;Sighting&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDS_CAT2</td>
<td>&quot;Contact&quot;</td>
</tr>
<tr>
<td>IDS_MODE_TRAVEL1</td>
<td>&quot;Space Ship&quot;</td>
</tr>
<tr>
<td>IDS_MODE_TRAVEL2</td>
<td>&quot;Levitation&quot;</td>
</tr>
<tr>
<td>IDS_MODE_TRAVEL3</td>
<td>&quot;Bicycle&quot;</td>
</tr>
<tr>
<td>IDS_WHERE1</td>
<td>&quot;Moonlit Desert&quot;</td>
</tr>
<tr>
<td>IDS_WHERE2</td>
<td>&quot;Moonlit Seascape&quot;</td>
</tr>
<tr>
<td>IDS_WHERE3</td>
<td>&quot;Hard to describe&quot;</td>
</tr>
<tr>
<td>IDS_PHYS_EVIDENCE1</td>
<td>&quot;Burnt ring of grass&quot;</td>
</tr>
<tr>
<td>IDS_PHYS_EVIDENCE2</td>
<td>&quot;Distressed livestock&quot;</td>
</tr>
<tr>
<td>IDS_PHYS_EVIDENCE3</td>
<td>&quot;Alien landing party&quot;</td>
</tr>
<tr>
<td>IDS_OFFICE1</td>
<td>&quot;Sedona, Arizona&quot;</td>
</tr>
<tr>
<td>IDS_OFFICE2</td>
<td>&quot;Crestone, Colorado&quot;</td>
</tr>
<tr>
<td>IDS_OFFICE3</td>
<td>&quot;Roswell, New Mexico&quot;</td>
</tr>
<tr>
<td>IDS_ABDUCTIONS1</td>
<td>&quot;Took family pets&quot;</td>
</tr>
<tr>
<td>IDS_ABDUCTIONS2</td>
<td>&quot;Took Cell Phone sales person&quot;</td>
</tr>
</tbody>
</table>

| IDS_ABDUCTIONS3 | "Returned chemistry teacher" |
| IDS_APPEARANCE1 | "Scary" |
| IDS_APPEARANCE2 | "Cute" |
| IDS_APPEARANCE3 | "Dressed Like L. Nemoy" |
| IDS_APPEARANCE4 | "Not Dressed" |
| IDS_APPEARANCE5 | "Can't Recall" |
| IDS_FRIENDLY1 | "Really Friendly" |
| IDS_FRIENDLY2 | "So So" |
| IDS_FRIENDLY3 | "Standoffish" |
| IDS_TALKATIVE1 | "Pleasantly chatty" |
| IDS_TALKATIVE2 | "Shy but sensitive" |
| IDS_TALKATIVE3 | "Taciturn" |

Listing 2.3  Source code for MoveAndSize.
A Better Approach to Forms 111

Listing 2.3  Source code for MoveAndSize. (continues)
```c
long lStyles;
int iStringIndexBase;
int iNumStrings;

// Message dispatch table for MainWindowProc
const struct CmdBarParms structCmdBarInit[] = {
    // Ctrl ID, // Label text, // Ctrl Styles, // String Index, // #strings
    IDC_NAME, IDS_NAME, ES_AUTOHSCROLL, NULL, 0,
    IDC_BADGE, IDS_BADGE, ES_AUTOHSCROLL, NULL, 0,
    IDC_DESC, IDS_DESC, ES_AUTOHSCROLL, NULL, 0,
    IDC_CAT, IDS_CAT, CBS_DROPDOWNLIST, IDS_CAT1, 2,
    IDC_MODE_TRAVEL, IDS_MODE_TRAVEL, CBS_DROPDOWNLIST, IDS_MODE_TRAVEL1, 3,
    IDC_WHERE_SEEN, IDS_WHERE_SEEN, CBS_DROPDOWNLIST, IDS_WHERE1, 3,
    IDC_PHYS_EVIDENCE, IDS_PHYS_EVIDENCE, CBS_DROPDOWNLIST, IDS_PHYS_EVIDENCE1, 3,
    IDC_OFFICE, IDS_OFFICE, CBS_DROPDOWNLIST, IDS_OFFICE1, 3,
    IDC_ABDUCTIONS, IDS_ABDUCTIONS, CBS_DROPDOWNLIST, IDS_ABDUCTIONS1, 3,
    IDC_FRIENDLY, IDS_FRIENDLY, CBS_DROPDOWNLIST, IDS_FRIENDLY1, 3,
    IDC_TALKATIVE, IDS_TALKATIVE, CBS_DROPDOWNLIST, IDS_TALKATIVE1, 3,
    IDC_APPEARANCE, IDS_APPEARANCE, CBS_DROPDOWNLIST, IDS_APPEARANCE1, 5
};

// Global Variables:
HINSTANCE hInst;        // The current instance
int nBands;       // #bands in the CommandBands ctrl
BOOL bRestore = FALSE;
int iCtrlInitOrderArray[ dim(structCmdBarInit)];
// when we hide the bands, we store
// their current order in this array
COMMANDBANDSRESTOREINFO cbrRestoreArray[ dim(structCmdBarInit) + 1];
// this struct saves band size and
// position for the bands

// Forward declarations of functions included in this code module:
ATOM MyRegisterClass (HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
HWND CreateDataBands (HWND);
BOOL CALLBACK ShowRecDlgProc (HWND, UINT, WPARAM, LPARAM);
void HideRecord (HWND);
HWND RestoreDataBands(HWND);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(      HINSTANCE hInstance,
                         HINSTANCE hPrevInstance,
                         LPTSTR    lpCmdLine,
                         int       nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    INITCOMMONCONTROLSEX icex; // this struct is used to init the
```
//common controls dll
//windows ce requires explicit init of the common controls
icex.dwSize = sizeof (INITCOMMONCONTROLSEX);
//just load what we need
icex.dwICC = ICC_COOL_CLASSES;
InitCommonControlsEx (&icex);
// Perform application initialization:
if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}
hAccelTable = LoadAccelerators(hInstance,
    (LPCTSTR)IDC_MOVEANDSIZE);
// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (! TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
return msg.wParam;
}

// FUNCTION: MyRegisterClass()
// PURPOSE: Registers the window class.
// COMMENTS:
//This function and its usage is only necessary if you want this
// code to be compatible with Win32 systems prior to the
// 'RegisterClassEx function that was added to Windows 95.
// It is important to call this function
// so that the application will get 'well formed' small icons
// associated with it.
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(hInstance,
        MAKEINTRESOURCE(IDI_MOVEANDSIZE));
Listing 2.3 Source code for MoveAndSize. (continues)
Listing 2.3  Source code for MoveAndSize.
Listing 2.3  Source code for MoveAndSize. (continues)
// dialog Box
CreateDataBands(hWnd);
break;
case WM_DESTROY:
    DestroyWindow (hwndCB);
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc(hWnd, message,
                          wParam, lParam);
}
return 0;
}
// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    RECT rt, rt1;
    int DlgWidth, DlgHeight;
    int NewPosX, NewPosY;
    switch (message)
    {
    case WM_INITDIALOG:
        // trying to center the About dialog
        if (GetWindowRect(hDlg, &rt1)) {
            GetClientRect(GetParent(hDlg), &rt);
            DlgWidth      = rt1.right - rt1.left;
            DlgHeight     = rt1.bottom - rt1.top;
            NewPosX       = (rt.right - rt.left - DlgWidth)/2;
            NewPosY       = (rt.bottom - rt.top - DlgHeight)/2;
            if (NewPosX < 0) NewPosX = 0;
            if (NewPosY < 0) NewPosY = 0;
            SetWindowPos(hDlg, 0, NewPosX, NewPosY, 0, 0, SWP_NOZORDER | SWP_NOSIZE);
        }
        return TRUE;
        case WM_COMMAND:
        if ((LOWORD(wParam) == IDOK) ||
            (LOWORD(wParam) == IDCANCEL)) {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
        break;
    }
Listing 2.3 Source code for MoveAndSize.
return FALSE;

HWND CreateDataBands(HWND hWnd)
{
    HBITMAP hBmp;
    LPREBARBANDINFO prbi;
    HWND hwndBand, hwndCombo, hwndCB;
    int i, j, nBandsPerRow;
    TCHAR tszCaptionBuff[34];
    TCHAR tszPlatType[256];
    //initialize nBands = # ctrls + the main menu band
    nBands = dim(structCmdBarInit) + 1;
    // Create a command band.
    hwndCB = CommandBands_Create(hInst, hWnd, IDC_BAND_BASE_ID,
        RBS_BANDBORDERS | RBS_AUTOSIZE | RBS_FIXEDORDER, NULL);
    // Load bitmap used as background for command bar.
    hBmp = LoadBitmap(hInst, TEXT("CmdBarBack"));
    // Allocate space for the REBARBANDINFO array
    // Need one entry per ctl band ( nBands ) plus the menu
    prbi = (LPREBARBANDINFO)LocalAlloc(LPTR, (sizeof(REBARBANDINFO) * nBands));
    if (!prbi) {
        MessageBox(hWnd, TEXT("LocalAlloc Failed"),
            TEXT("Oops!")), MB_OK);
        return 0;
    }
    // Initialize common REBARBANDINFO structure fields.
    for (i = 0; i < nBands; i++) {
        prbi[i].cbSize = sizeof(REBARBANDINFO);
        prbi[i].fMask = RBBIM_ID | RBBIM_IMAGE | RBBIM_SIZE |
            RBBIM_BACKGROUND | RBBIM_STYLE | RBBIM_IDEALSIZE;
        prbi[i].wID = IDC_BAND_BASE_ID + i;
        prbi[i].hbmBack = hBmp;
    }
    // Initialize REBARBANDINFO structure for Menu band
    prbi[0].cx = GetSystemMetrics(SM_CXSCREEN) / 4;
    prbi[0].iImage = 0;
    prbi[0].fStyle = prbi[1].fStyle = RBBS_NOGRIPPER | RBBS_BREAK;
    //Set bands per row based on platform type
    memset(&tszPlatType, 0x0, sizeof(tszPlatType));
    SystemParametersInfo(SPI_GETPLATFORMTYPE,
        sizeof(tszPlatType), &tszPlatType, 0);
    if ( lstrcmp(tszPlatType, TEXT("Jupiter")) == 0 )
        nBandsPerRow = 4;
}

Listing 2.3  Source code for MoveAndSize. (continues)
else if( lstrcmp( tszPlatType, TEXT("HPC") ) == 0 )
{ nBandsPerRow = 3; }
else if( lstrcmp( tszPlatType, TEXT("Palm PC") ) == 0 )
{ nBandsPerRow = 1; }

//Initialize data entry band attributes
for (i = 1; i < nBands; i++)
{
    // Common Combobox ctrl band attributes
    prbi[i].fMask |= RBBIM_ID | RBBIM_IMAGE | RBBIM_SIZE | RBBIM_CHILDSIZE;
    prbi[i].iImage = 1;
    prbi[i].cx = (GetSystemMetrics(SM_CXSCREEN ) / 2 ) ;
    prbi[i].cxMinChild = (GetSystemMetrics(SM_CXSCREEN ) / 6 ) ;
    prbi[i].cxIdeal = (GetSystemMetrics(SM_CXSCREEN ) / 2 ) ;
    //Set style for line break at the end of a row
    if(((i - 1 ) % nBandsPerRow ) == 0 )
    {
        prbi[i].fStyle |= RBBS_BREAK | RBBS_GRIPPERALWAYS;
    }
    else
    {
        prbi[i].fStyle |= RBBS_GRIPPERALWAYS;
    }
}

// Add the bands-- now we've reserved real estate
// for our dlg controls.
CommandBands_AddBands (hwndCB, hInst, nBands, prbi);

//Move past the menu bar & close box and
//set the control behaviors
for (i = 1; i < nBands; i++)
{
    hwndBand = CommandBands_GetCommandBar (hwndCB, i);
    SetWindowLong(hwndBand, GWL_USERDATA, i - 1 );
    CommandBar_InsertComboBox (hwndBand, hInst, prbi[i].cx - 6,
    structCmdBarInit[i - 1].lStyles,
    IDC_COMBO_BASE_ID + (i - 1 ), 0);
    //get a handle to the combo in this band
    hwndCombo = GetDlgItem(hwndBand,
    IDC_COMBO_BASE_ID + (i - 1 ));
    //get the caption string
    LoadString(hInst, structCmdBarInit[i - 1].uiCaption,
    (LPTSTR)&tszCaptionBuff,sizeof(tszCaptionBuff));
//insert the label string
SendMessage (hwndCombo, CB_INSERTSTRING, 0,
   (long)&tszCaptionBuff);

//highlight the caption
SendMessage (hwndCombo, CB_SETCURSEL, 0, 0);

//does this combo have a string list?
if(structCmdBarInit[i - 1].iNumStrings)
{
   //if so, insert them in the combo
   for( j = 0;
       j < structCmdBarInit[i - 1].iNumStrings;
       j ++ )
   {
      LoadString(hInst,
       structCmdBarInit[i - 1].iStringIndexBase + j,
       (LPTSTR)&tszCaptionBuff,sizeof(tszCaptionBuff));
       //add strings to the end of the list
       SendMessage (hwndCombo, CB_INSERTSTRING, -1,
       (long)&tszCaptionBuff);
   } //end for(iNumStrings)
} //end if(iNumStrings)

// Add menu to band 0 --wahoo
hwndBand = CommandBands_GetCommandBar (hwndCB, 0);
   CommandBar_InsertMenubar (hwndBand, hInst, IDM_MENU, 0);
   // Add exit button to command band control.
   CommandBands_AddAdornments (hwndCB, hInst, 0, 0);
   //we're done with this now
   LocalFree(prbi);
   return 0;
}

// Message handler for the About box.
BOOL CALLBACK ShowRecDlgProc(HWND hDlg, UINT wMsg,
   WPARAM wParam, LPARAM lParam)
{
   HWND hwndList, hwndCB, hwndBand, hwndCombo;
   int iIndex, iInputCharCnt, i;
   TCHAR tszInput[34];
   TCHAR tszCaptionCompare[64];
   switch (wMsg)
   {
   //show the input in the dialog
   case WM_INITDIALOG:
       //get the handle to the command bands control
       hwndCB = GetDlgItem( (HWND)lParam,
       IDC_BAND_BASE_ID );
       break;
   //more cases...
   }
   return 0;

Listing 2.3  Source code for MoveAndSize. (continues)
for( i = 1; i < nBands; i++ )
{
    //since the combos may have
    //been moved by the user,
    //we can count on their ordering
    //only if we loop
    //them using their combo index
    iIndex = SendMessage (hwndCB, RB_IDTOINDEX,
        IDC_BAND_BASE_ID + i, 0);
    hwndBand =
        CommandBands_GetCommandBar (hwndCB, iIndex);
    hwndCombo =
        GetDlgItem(hwndBand, IDC_COMBO_BASE_ID + i-1 );
    iInputCharCnt =
        SendMessage( hwndCombo, WM_GETTEXTLENGTH, 0, 0);
    //Get ctrl contents
    SendMessage( hwndCombo, WM_GETTEXT, iInputCharCnt + 1,
        (LPARAM)(LPSTR)&tszInput);
    //compare to caption
    LoadString( hInst, IDS_NAME + i-1,
        (LPTSTR)&tszCaptionCompare,
        sizeof(tszCaptionCompare));
    //if no input
    if( !wcscmp( tszCaptionCompare, tszInput ) )
    {
        //cat ":
        wcscat(tszCaptionCompare, TEXT(":");
    }
    else
    {
        //cat ":
        wcscat(tszCaptionCompare, TEXT(":");
        //cat input to caption
        wcscat(tszCaptionCompare, tszInput);
    }
    //insert in the listbox
    hwndList = GetDlgItem(hDlg,IDC_RECORD_LIST);
    SendMessage( hwndList, LB_INSERTSTRING, -1,
        (long)&tszCaptionCompare );
}
return TRUE;
case WM_COMMAND:
    switch (LOWORD (wParam))
    {
    case IDCANCEL:

Listing 2.3  Source code for MoveAndSize.
EndDialog (hDlg, 1);
    return TRUE;

    default:
        break;
    }
break;
    default:
        break;
    }//end switch (wMsg)
return FALSE;
}
void HideRecord( HWND hWnd )
{
    HWND hwndCB, hwndBand;
    int i, iIndex;
    HBITMAP hBmp;
    LPREBARBANDINFO prbi;
    hwndCB = GetDlgItem (hWnd, IDC_BAND_BASE_ID);
    for (i = 1; i < nBands; i++)
    {
        // Get band index from ID value.
        iIndex = SendMessage (hwndCB, RB_IDTOINDEX, 
                        IDC_BAND_BASE_ID + i , 0);
        // Get the restore information.
        cbrRestoreArray[i].cbSize = sizeof (COMMANDBANDSRESTOREINFO);
        // get restore info for the band
        CommandBands_GetRestoreInformation (hwndCB, iIndex,
                        &cbrRestoreArray[i]);
        // now get restore info for the ctrl
        // get the handle to this band
        hwndBand = CommandBands_GetCommandBar (hwndCB, iIndex);
        // get it's window long value -- this indexes
        // the control init struct array
        iCtrlInitOrderArray[i - 1] =
            (int)GetWindowLong(hwndBand, GWL_USERDATA );
    } 
    DestroyWindow (hwndCB);
    // set flag that allows us to restore the old bands 
    bRestore = TRUE;
    // Allocate space for the REBARBANDINFO array 
    // Need one entry per ctl band ( nBands ) plus the menu 
    prbi = (LPREBARBANDINFO)LocalAlloc(LPTR, sizeof (REBARBANDINFO));

Listing 2.3  Source code for MoveAndSize. (continues)
if (!prbi) {
    MessageBox(hWnd, TEXT("LocalAlloc Failed"),
              TEXT("Oops!"), MB_OK);
    return;
}

// Load bitmap used as background for command bar.
if (hBmp = LoadBitmap(hInst, TEXT("CmdBarBack"))) {
    // Initialize REBARBANDINFO structure fields.
    prbi->cbSize = sizeof(REBARBANDINFO);
    prbi->fMask = RBBIM_ID | RBBIM_IMAGE | RBBIM_SIZE |
                  RBBIM_BACKGROUND | RBBIM_STYLE |
                  RBBIM_IDEALSIZE;
    prbi->wID = IDC_BAND_BASE_ID;
    prbi->hbmBack = hBmp;
    prbi->cx = GetSystemMetrics(SM_CXSCREEN);
    prbi->iImage = 0;
    prbi->fStyle = RBBS_NOGRIPPER | RBBS_BREAK;
    // Create a command band control.
    hwndCB = CommandBands_Create(hInst, hWnd, IDC_BAND_BASE_ID,
                                 RBS_BANDBORDERS | RBS_AUTOSIZE | RBS_FIXEDORDER,
                                 NULL);
    //add a band for the menu control
    CommandBands_AddBands(hwndCB, hInst, 1, prbi);
    // Add menu to band 0 --wahoo
    hwndBand = CommandBands_GetCommandBar(hwndCB, 0);
    CommandBar_InsertMenubar(hwndBand, hInst, IDM_MENU, 0);
    // Add exit button to command band control.
    CommandBands_AddAdornments(hwndCB, hInst, 0, 0);
    //we're done with this now
    LocalFree(prbi);
    return;
}

HWND RestoreDataBands(HWND hWnd)
{
    HBITMAP hBmp;
    LPREBARBANDINFO prbi;
    HWND hwndBand, hwndCombo, hwndCB;
    int i, j;
    TCHAR tszCaptionBuff[34];
    int iCtrlInitIndex;
    //bail if there is nothing to restore
    if (!bRestore )
        { return 0;}
    else
        { bRestore = FALSE; }
    //destroy the menu band so we can replace it

Listing 2.3  Source code for MoveAndSize.
hwndCB = GetDlgItem(hWnd, IDC_BAND_BASE_ID);
DestroyWindow(hwndCB);
// Create a command band.
hwndCB = CommandBands_Create(hInst, hWnd, IDC_BAND_BASE_ID,
    RBS_BANDBORDERS | RBS_AUTOSIZE | RBS_FIXEDORDER,
    NULL);
// Load bitmap used as background for command bar.
hBmp = LoadBitmap(hInst, TEXT("CmdBarBack"));
// Allocate space for the REBARBANDINFO array
// Need one entry per ctl band (nBands) plus the menu
prbi = (LPREBARBANDINFO)LocalAlloc(LPTR,
    (sizeof(REBARBANDINFO) * nBands));
if (!prbi) {
    MessageBox(hWnd, TEXT("LocalAlloc Failed"), TEXT("Oops!"),
        MB_OK);
    return 0;
}
// Initialize common REBARBANDINFO structure fields.
for (i = 0; i < nBands; i++) {
    prbi[i].cbSize = sizeof(REBARBANDINFO);
    prbi[i].fMask = RBBIM_ID | RBBIM_SIZE |
        RBBIM_BACKGROUND | RBBIM_STYLE | RBBIM_IDEALSIZE;
    prbi[i].wID = IDC_BAND_BASE_ID+i;
    prbi[i].hbmBack = hBmp;
}
// Initialize REBARBANDINFO structure for Menu band
prbi[0].cx = GetSystemMetrics(SM_CXSCREEN) / 4;
prbi[0].iImage = 0;
prbi[0].fStyle = prbi[1].fStyle = RBBS_NOGRIPPER | RBBS_BREAK;

//Initialize data entry band attributes
for (i = 1; i < nBands; i++)
{
    // Common Combo box ctrl band attributes
    prbi[i].fMask |= RBBIM_ID | RBBIM_SIZE| RBBIM_CHILDSIZE;
    prbi[i].cxMinChild = (GetSystemMetrics(SM_CXSCREEN) / 6);
    prbi[i].cxIdeal = (GetSystemMetrics(SM_CXSCREEN) / 2);
    prbi[i].cx = cbrRestoreArray[i].cxRestored;
    prbi[i].fStyle = cbrRestoreArray[i].fStyle;
}
// Add the bands-- now we've reserved real estate
// for our dlg controls.
CommandBands_AddBands(hwndCB, hInst, nBands, prbi);
//now we restore the bands that were maximized
for( i = 0; i < dim(cbrRestoreArray); i++)
{
}
if( cbrRestoreArray[i].fMaximized)
{
    hwndBand = CommandBands_GetCommandBar (hwndCB, i);
    SendMessage(hwndBand, RB_MAXIMIZEBAND, i, prbi[i].cxIdeal);
} // end if max'ed
} // end for cbr RestoreArray

//loop the user bands
for (i = 1; i < nBands; i++)
{
    hwndBand = CommandBands_GetCommandBar (hwndCB, i);
    //get the index of the init struct for this band
    iCtrlInitIndex = iCtrlInitOrderArray[i - 1];
    //set its ctrl array index in the window long value
    SetWindowLong( hwndBand, GWL_USERDATA, iCtrlInitIndex );
    hwndCombo =
        CommandBar_InsertComboBox (hwndBand,
        hInst, prbi[i].cx - 6,
        structCmdBarInit[iCtrlInitIndex].lStyles,
        IDC_COMBO_BASE_ID + (i - 1), 0);
    //get the caption string
    LoadString(hInst, structCmdBarInit[iCtrlInitIndex].uiCaption,
        (LPTSTR)&tszCaptionBuff,sizeof(tszCaptionBuff));
    //insert the label string
    SendMessage (hwndCombo, CB_INSERTSTRING, 0,
        (long)&tszCaptionBuff);
    //highlight the caption
    SendMessage (hwndCombo, CB_SETCURSEL, 0, 0);
    //does this combo have a string list?
    if(structCmdBarInit[iCtrlInitIndex].iNumStrings)
    {
        //if so, insert them in the combo
        for( j = 0; 
            j < structCmdBarInit[iCtrlInitIndex].iNumStrings; 
            j ++ )
        {
            LoadString(hInst,
                structCmdBarInit[iCtrlInitIndex].iStringIndexBase + j,
                (LPTSTR)&tszCaptionBuff,sizeof(tszCaptionBuff));
            //add strings to the end of the list
            SendMessage (hwndCombo, CB_INSERTSTRING, -1,
                (long)&tszCaptionBuff);
        } //end for(iNumStrings)
    } //end if(iNumStrings)
} //end for(iNumStrings)

Listing 2.3  Source code for MoveAndSize.
Like the preceding example, this one builds on the material presented earlier in this chapter, so we’ll focus on the differences between this example and the other CommandBands-based forms.

One assumption that we’ve made up to this point is that we could safely iterate through the CommandBands control using band indices, grab the contained command bar by its handle, and proceed to party on the embedded control. This was safe in the earlier examples because we specified the CommandBands style flag that forces individual bands to retain their order. Here’s the place where we made this decision in the two earlier examples:

```c
//NOT WHAT HAPPENS IN THE MOVEANDSIZE EXAMPLE
// Create a command band.
hwndCB = CommandBands_Create(hInst, hWnd, IDC_BAND_BASE_ID,
                   RBS_BANDBORDERS | RBS_AUTOSIZE | RBS_FIXEDORDER, NULL);
```

This time we have a subtle but very important difference. Here’s how we create the MoveAndSize CommandBands control:

```c
// Create a command band with moveable bands.
hwndCB = CommandBands_Create(hInst, hWnd, IDC_BAND_BASE_ID,
                   RBS_BANDBORDERS | RBS_AUTOSIZE, NULL);
```

Recall that one of the first points I made about CommandBands controls is that the control itself is mostly a screen real estate manager and a container of CommandBars. If we omit the RBS_FIXEDORDER style, we are allowing users to move CommandBars between bands. When they do that, it changes the
containment relationships between individual bands and bars, but it doesn’t change the control’s view of band indices. To the CommandBands control, band 0 is the topmost, leftmost band, regardless of which bar happens to occupy that spot at any given point in time.

This means that we can’t simply iterate through the band indices when we want to access the controls contained in the command bars if its possible that a band has been moved by the user. To get access to specific moveable controls, we have to translate their band index to a control ID. Getting the control index by using the RB_IDTOINDEX parameter to SendMessage() will let you iterate the controls in the order you created them, no matter where the user has put them. Here’s how to do this:

```c
//get the handle to the command bands control
hwndCB = GetDlgItem( (HWND)lParam, IDC_BAND_BASE_ID );
for( i = 1; i < nBands; i++ )
{
    //since the combos may have been moved by the user,
    //we can count on their ordering only if we loop
    //them using their combo index
    iIndex = SendMessage (hwndCB, RB_IDTOINDEX, IDC_BAND_BASE_ID + i, 0);
    hwndBand = CommandBands_GetCommandBar (hwndCB, iIndex);
    hwndCombo = GetDlgItem(hwndBand, IDC_COMBO_BASE_ID + i-1);
    //get the character count of the input
    iInputCharCnt = SendMessage( hwndCombo, WM_GETTEXTLENGTH, 0, 0);
    .
    .
    .
    //do some kind of control access
}
```

The excerpt above comes from ShowRecDlgProc(), a dialog box that shows user input to controls. The caption and user input for each field is displayed in the dialog in the order that the fields appear in the initial form layout, even if the bands have been rearranged. Here’s one more thing to note before moving on. Notice the return from the SendMessage() call that uses WM_GETTEXTLENGTH. This return is a count, not a size, and it excludes the terminal NULL. If you were to allocate memory for the control text based on the reported length, you’d do it like this:
PORTING TIP  Look carefully at all dialog code that copies text from controls. If the text is bound for stack-based buffers, make the buffers type TCHAR. If you allocate space for control text, use SendMessage() with WM_GETTEXTLENGTH, and calculate the buffer space like this:

\[(iInputCharCnt + 1) \times \text{sizeof(TCHAR)}\]

Here’s another issue that arises from the fact that the bands are moveable: What happens if the user moves the bands, the control gets destroyed, and then the bands need to be shown again in the order the user last saw them? In practice, this could happen if the application allows data entry to be an interruptible operation (a very likely prospect in mobile computing settings). We accommodate this need by collecting restoration information from the control, and saving it in a special purpose structure called COMMANDBANDS-RESTOREINFO. When we next build a control, some of the information in the REBARBANDINFO structure is initialized with saved data. In the MoveAndSize example program, we use two arrays of structures to save CommandBand restore information.

```c
//when we hide the bands, we store //their current order in this array int iCtrlInitOrderArray[ dim(structCmdBarInit)]; //this struct saves band size and //position for the bands COMMANDBANDSRESTOREINFO cbrRestoreArray[ dim(structCmdBarInit) + 1];
```

As mentioned earlier, the controls contained in the individual bands have sort of a dual nature: They are constituent parts of the CommandBands, but they are also inherently controls, and therefore windows. While it’s not a good idea to treat the controls as independent windows in any context where they have a containment relationship, we can take advantage of the window-ness of them by using their individual Window Long values to store state information or other important per-control details. In MoveAndSize, we demonstrate the use of the window long value by storing the original index of the band there. You can also use the window long store pointers to more elaborate aggregations of data. This is an excerpt from CreateDataBands. It is the top of the band initialization loop. (It occurs just after the call to CommandBands_AddBands().)
To set a value that stays with its window, use `SetWindowLong()`. The parameters, in the order shown, are the handle to the window whose value is being set, the constant `GWL_USERDATA`, which means that you are setting an application specific datum, and the value you wish to set. The last parameter is of type `long`, so don’t try to copy anything larger than that. (If you need more space than a `long` provides, allocate it and store its address.)

We hide and restore the CommandBands form in response to menu items in the MoveAndSize example. Here is the excerpt from the `WndProc()` message switch where these menu commands are handled:

```c
case WM_COMMAND:
    wmId = LOWORD(wParam);
    wmEvent = HIWORD(wParam);
    // handle the menu selections:
    switch (wmId)
    {
      case IDM_HIDE_RECORD:
        //destroy the record's command band control
        //and replace it w/ a single band
        //containing the menu
        HideRecord( hWnd );
        break;
      case IDM_RESTORE_RECORD:
        //use the restore information to restore the bands
        //as we last saw them
        RestoreDataBands(hWnd);
        break;
    }

In `HideRecord()`, we save band restoration information, destroy the CommandBands control that implements the form, and replace it with a minimal CommandBands control, which contains only the application’s menu. Here’s the portion of `HideRecord()` that saves the restore information:
//get the handle to the command bands control
hwndCB = GetDlgItem(hWnd, IDC_BAND_BASE_ID);
for (i = 1; i < nBands; i++)
{
    // Get band index from ID value.
    iIndex = SendMessage(hwndCB, RB_IDTOINDEX,
                        IDC_BAND_BASE_ID + i , 0);
    // ALWAYS initialize a struct's cbSize member if it has one.
    cbrRestoreArray[i].cbSize = sizeof(COMMANDBANDSRESTOREINFO);
    // get restore info for the band
    CommandBands_GetRestoreInformation (hwndCB, iIndex,
                                        &cbrRestoreArray[i]);

    // now get restore info for the ctrl
    // get the handle to this band
    hwndBand = CommandBands_GetCommandBar (hwndCB, iIndex);

    // get it's window long value -- this indexes
    // the control init struct array
    iCtrlInitOrderArray[i - 1] =
        (int)GetWindowLong(hwndBand, GWL_USERDATA );
}

When we restore the bands to their previous configuration, we simply use the
saved information to fill in the array of REBARBANDINFO structures. Here are
the places where the restored initialization info is used in CreateDataBands().
First, all the bands get their style and width from the saved data:

// Initialize data entry band attributes
for (i = 1; i < nBands; i++)
{
    // Common Combobox ctrl band attributes
    prbi[i].fMask |= RBBIM_ID | RBBIM_SIZE| RBBIM_CHILDSIZE;
    prbi[i].cxMinChild = (GetSystemMetrics(SM_CXSCREEN ) / 6 ) ;
    prbi[i].cxIdeal = (GetSystemMetrics(SM_CXSCREEN ) / 2 ) ;
    prbi[i].cx = cbrRestoreArray[i].cxRestored;
    prbi[i].fStyle = cbrRestoreArray[i].fStyle;
}

Next, any bands that were maximized are restored to that state:

// now we restore the bands that were maximized
for( i = 0; i < dim( cbrRestoreArray ); i++ )
{
if( cbrRestoreArray[i].fMaximized)
{
    hwndBand = CommandBands_GetCommandBar (hwndCB,i);
    SendMessage(hwndBand, RB_MAXIMIZEBAND, i, prbi[i].cxIdeal);
} // end if max'ed
//end for cbrRestoreArray

Finally, when the controls are reinitialized, the elements in the array where band order was saved, iCtrlInitOrderArray is used to index structCmdBarInit (the control initialization array).

//loop the user bands
for (i = 1; i < nBands; i++)
{
    hwndBand = CommandBands_GetCommandBar (hwndCB, i);
    //get the index of the init struct for this band
    iCtrlInitIndex = iCtrlInitOrderArray[i - 1];
    //set its ctrl array index in the window long value
    SetWindowLong( hwndBand, GWL_USERDATA, iCtrlInitIndex );
    hwndCombo = CommandBar_InsertComboBox (hwndBand,
                                      hInst, prbi[i].cx - 6,
                                      structCmdBarInit[iCtrlInitIndex].lStyles,
                                      IDC_COMBO_BASE_ID + (i - 1), 0);
    //get the caption string
    LoadString(hInst, structCmdBarInit[iCtrlInitIndex].uiCaption,
               (LPTSTR)&tszCaptionBuff,sizeof(tszCaptionBuff));
    //insert the label string
    SendMessage (hwndCombo, CB_INSERTSTRING, 0,
                 (long)&tszCaptionBuff);
    //highlight the caption
    SendMessage (hwndCombo, CB_SETCURSEL, 0, 0);
    //does this combo have a string list?
    if(structCmdBarInit[iCtrlInitIndex].iNumStrings)
    {
        //if so, insert them in the combo
        for( j = 0;
            j < structCmdBarInit[iCtrlInitIndex].iNumStrings;
                j ++ )
        {
            LoadString(hInst,
                        structCmdBarInit[iCtrlInitIndex].iStringIndexBase + j,
                        (LPTSTR)&tszCaptionBuff,sizeof(tszCaptionBuff));
            //add strings to the end of the list
            SendMessage (hwndCombo, CB_INSERTSTRING, -1,
                         (long)&tszCaptionBuff);
Forms. You’ve gotta love ‘em.

**Looking Ahead**

We’ve explored some of what you need to know in order to take a Win32 application’s controls to Windows CE. Now we’re going to explore two more important aspects of CE’s user interface: graphics and user input.
The Linker, Your New Best Friend

Graphics is a good place to start porting application behaviors, because you can see how your effort is going. Here’s our initial strategy for getting graphics behaviors up on the Windows CE side:

1. Choose the subset of graphics behaviors essential to the Win CE version of your application.
2. Copy the code for the behaviors into a new Win CE project.
3. Run a build to identify unsupported APIs.

It’s possible that these steps will yield fully functional Windows CE graphics code. If a clean compile and link occurs, you should try running your code on a CE device to test it. Don’t be too disappointed if this clean build doesn’t work properly. Some of the graphics APIs behave differently under CE and many support only subsets of the Win32 flags used as function parameters.

The more likely result is that the linker will report some unresolved external references. Listing 3.1 shows a fragment of code from a project called Unresolved. This application is an automatically generated “Hello World“
program, with the sole addition of the two lines shown in bold and their associated parameter.

```c
case WM_PAINT:
    RECT rt;
    static POINT ptCurrent;
    hdc = BeginPaint(hWnd, &ps);
    GetClientRect(hWnd, &rt);
    LoadString(hInst, IDS_HELLO, szHello, MAX_LOADSTRING);
    MoveToEx( hdc, 0, 0, &ptCurrent );
    LineTo( hdc, rt.right. rt.bottom );
    DrawText(hdc, szHello, _tcslen(szHello), &rt,
              DT_SINGLELINE | DT_VCENTER | DT_CENTER);
    EndPaint(hWnd, &ps);
    break;
```

**Listing 3.1** Adding two unsupported graphics APIs to “Hello World.”

When the project is built, Listing 3.2 shows the text shown in the output window.

```
--------------------Configuration: Unresolved - Win32 (WCE MIPS) Debug----------------
Compiling...
Unresolved.cpp
C \Unresolved\Unresolved.cpp(164) : error C2065: 'MoveToEx' : undeclared identifier
C:\Unresolved\Unresolved.cpp(165) : error C2065: 'LineTo' : undeclared identifier
Error executing clmips.exe.
Unresolved.exe - 2 error(s), 0 warning(s)
```

**Listing 3.2** Linker reporting unresolved external references.

This is incredibly helpful information to the porting process, but there are a couple of caveats to observe in putting the linker in the harness as your porting workhorse.

First, to get started, generate a Windows CE project and carefully add your own code to it. In particular, you have to pay attention to the system settings for the path to header files. If you use a generated “Hello World” application targeting all of the Windows CE platforms, this is automatic. If you create an empty project and add your files, then getting the proper header and library paths set is much more time consuming and difficult. Remember, all of the different CE targets have processor-specific libraries and linkage.
PORTING TIP When porting graphics and other behaviors, start with an automatically generated “Hello World” project and copy your code into the appropriate places in the generated files.

The second thing to be careful of is copying code files from your own projects that include hard coded paths to Win32 header files. Here is how a hard-coded include path looks:

```
#include "c:\ Unresolved\Unresolved.h"
```

The quotes around the filename tell the compiler to look in very specific places for the header file Unresolved.h. In this case, the search begins in the current directory (the one where the project workspace lives) and then proceeds through the directories specified in the INCLUDE environment variable. The compiler will use the first occurrence of Unresolved.h it comes upon in its search. While it is not likely that a file named Unresolved.h will turn up unexpectedly, the header filenames for various Windows programming components are exactly the same and occur in a variety of directories. Hard-coded Windows API header filenames that resolve incorrectly will send the build process wildly awry.

PORTING TIP Search for the string #include in your own code and double check hard-coded include file paths. Change or eliminate any that point to Win32 header files. Remember that header files often contain #include statements.

Most all your Windows include files will be specified in statements like this:

```
#include <commctrl.h>
```

The angle brackets around the include filename mean “look in standard places.” The standard places for your Windows CE project are always generated correctly if you start with a “Hello World” application.

Third, the linker report may include unresolved externals that don’t actually occur in the source code you wrote. Here’s what to do if you get an unresolved external that looks something like this:

```
C:\Unresolved\Unresolved.cpp(165) : error C2065: '_gcvt' : undeclared identifier
```

I’ve seen linker errors like this one in areas of my code that had to do with pointer arithmetic. Before you do anything, search your code files and make sure the reference is not just a typo. If the string isn’t found in your code files, choose the Project.Settings menu item, and go to the C++ tab. Select the category Listing Files, and under Listing File Type, choose Assembly with Source (see Figure 3.1).
Press OK, which will dismiss the dialog, and then rebuild. This build generates the file you need to examine in order to find the unresolved external "_gcvt". The file will have the suffix "._asm". For example, if we did this for the Unresolved project, we'd want to check the file Unresolved.asm.

When you search this file for the unresolved external "_gcvt", you'll find it somewhere in between two lines of your own source code. The one that precedes it is the source line that generated the unresolved external. Experiment with rearranging code or eliminating the statement. I've never had any really extreme difficulty getting past these, but I have found that sometimes it helps not to do too many operations in one statement. For example, if you dereference a pointer and increment it in one statement, you are doing multiple operations, and the success may depend on rules of precedence. It might be a better approach to break the operations up and have more explicit control of the order in which they happen.

The fourth and final caveat about letting the linker help you with the rough cut is that many of the Windows API functions implemented on CE are limited in the parameters and flag values they support. For example, the CE API CreateDIBSection() requires that the last two of its six parameters be set to 0. These parameters have legitimate uses under Win32, and ported code may well include nonzero values for them. A clean build tells you that no unsupported functions remain in your code, but it doesn't assure proper content in the function parameter lists.
Drawing Lines

The simplest graphic operation is drawing lines, and that is where we will start. You already know how to find unsupported APIs, as you saw in the build output of Unresolved MoveToEx() and LineTo(). Now let’s look more deeply into other limitations that have to do with line drawing. Figure 3.2 shows the code for an example program called EtchASketch (see Listing 3.3). (The resource file is omitted for the sake of brevity, but can be found on the CD.) EtchASketch is a line drawing program that functions in more or less the same way as the toy. Each time you choose one of the file menu items, a unit line segment is drawn in that direction.

Figure 3.2 The EtchASketch program.

Listing 3.3 EtchASketch example code. (continues)
HWND hwndCB; // The command bar handle
POINT ptLineSeg[2]; // POINT array used by Polyline
int iSegLen; // length of the line segment to draw
BOOL bFirstTime = TRUE;
BOOL bDrawingLines = FALSE;

// Forward declarations of functions included in this code module:
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPTSTR    lpCmdLine,
                   int       nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    // Perform application initialization:
    if (!InitInstance(hInstance, nCmdShow))
    {
        return FALSE;
    }
    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_ETCHASKETCH);
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        return msg.wParam;
    }
    // FUNCTION: MyRegisterClass()
    // PURPOSE: Registers the window class.
    // COMMENTS:
    // This function and its usage is only necessary if you want this code
to be compatible with Win32 systems prior to the 'RegisterClassEx'
function that was added to Windows 95. It is important to call
this function
so that the application will get 'well formed' small icons
associated
with it.

Listing 3.3 EtchASketch example code.
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS    wc;
    wc.style                      = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc        = (WNDPROC) WndProc;
    wc.cbClsExtra            = 0;
    wc.cbWndExtra          = 0;
    wc.hInstance               = hInstance;
    wc.hIcon                     = LoadIcon(hInstance,
MAKEINTRESOURCE(IDI_ETCHASKETCH));
    wc.hCursor                  = 0;
    wc.hbrBackground      = GetStockObject( WHITE_BRUSH );
    wc.lpszMenuName      = 0;
    wc.lpszClassName       = szWindowClass;
    return RegisterClass(&wc);
}

//  FUNCTION: InitInstance(HANDLE, int)
//  PURPOSE: Saves instance handle and creates main window
//  COMMENTS:
//    In this function, we save the instance handle in a global variable
//    and
//    create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND    hWnd;
    TCHAR    szTitle[MAX_LOADSTRING];  // The title bar text
    TCHAR    szWindowClass[MAX_LOADSTRING];  // The window class name
    // Store instance handle in our global variable
    hInst = hInstance;

    // Initialize global strings
    LoadString(hInstance, IDC_ETCHASKETCH,
                 szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle,
                     WS_VISIBLE, 0, 0,
                     CW_USEDEFAULT,
                     CW_USEDEFAULT,
                     NULL, NULL, hInstance,
                     NULL);
    if (!hWnd)
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
//initialize the starting point of the
//ptLineSeg array to the center pt of the screen
ptLineSeg[0].x = ptLineSeg[1].x = GetSystemMetrics(SM_CXSCREEN) / 2;
ptLineSeg[0].y = ptLineSeg[1].y = GetSystemMetrics(SM_CYSCREEN) / 2;
//init length of the line segment we draw in response
//to menu commands
iSegLen = GetSystemMetrics(SM_CXSCREEN) / 25;
return TRUE;
}

// FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
// PURPOSE: Processes messages for the main window.
// WM_COMMAND    - process the application menu
// WM_PAINT    - Paint the main window
// WM_DESTROY    - post a quit message and return
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    RECT rcClientPoints, rcInvalid;
    switch (message)
    {
    case WM_COMMAND:
        wmId = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_HELP_ABOUT:
            DialogBox(hInst,
                    (LPCTSTR)IDD_ABOUTBOX, hWnd,
                    (DLGPROC)About);
            break;
        case IDM_FILE_LEFT:
Listing 3.3  EtchASketch example code.
bDrawingLines = TRUE;
//set the endpoint of the line segment
ptLineSeg[1].x = ptLineSeg[0].x - iSegLen;
ptLineSeg[1].y = ptLineSeg[0].y;
//Invalidate the line seg area &
//send a paint message
CopyRect( &rcInvalid, (LPRECT)&ptLineSeg );
InflateRect( &rcInvalid, 1, 1 );
InvalidateRect( hWnd, &rcInvalid, FALSE);
UpdateWindow( hWnd );
//update the starting point of the next
//line segment
ptLineSeg[0].x = ptLineSeg[1].x;
ptLineSeg[0].y = ptLineSeg[1].y;
break;

case IDM_FILE_RIGHT:
    bDrawingLines = TRUE;
    //set the endpoint of the line segment
    ptLineSeg[1].x = ptLineSeg[0].x + iSegLen;
    ptLineSeg[1].y = ptLineSeg[0].y;
    //Invalidate the line seg area &
    //send a paint message
    CopyRect( &rcInvalid, (LPRECT)&ptLineSeg );
    InflateRect( &rcInvalid, 1, 1 );
    InvalidateRect( hWnd, &rcInvalid, FALSE);
    UpdateWindow( hWnd );
    //update the starting point of the next
    //line segment
    ptLineSeg[0].x = ptLineSeg[1].x;
    ptLineSeg[0].y = ptLineSeg[1].y;
    break;

case IDM_FILE_UP:
    bDrawingLines = TRUE;
    //set the endpoint of the line segment
    ptLineSeg[1].x = ptLineSeg[0].x;
    ptLineSeg[1].y = ptLineSeg[0].y - iSegLen;
    //Invalidate the line seg area &
    //send a paint message
    CopyRect( &rcInvalid, (LPRECT)&ptLineSeg );
    InflateRect( &rcInvalid, 1, 1 );
    InvalidateRect( hWnd, &rcInvalid, FALSE);
    UpdateWindow( hWnd );
    //update the starting point of the next
    //line segment
    ptLineSeg[0].x = ptLineSeg[1].x;
    ptLineSeg[0].y = ptLineSeg[1].y;
    break;

case IDM_FILE_DOWN:
    bDrawingLines = TRUE;
    //set the endpoint of the line segment
    ptLineSeg[1].x = ptLineSeg[0].x;
    ptLineSeg[1].y = ptLineSeg[0].y;
    //Invalidate the line seg area &
    //send a paint message
    CopyRect( &rcInvalid, (LPRECT)&ptLineSeg );
    InflateRect( &rcInvalid, 1, 1 );
    InvalidateRect( hWnd, &rcInvalid, FALSE);
    UpdateWindow( hWnd );
    //update the starting point of the next
    //line segment
    ptLineSeg[0].x = ptLineSeg[1].x;
    ptLineSeg[0].y = ptLineSeg[1].y;
    break;

Listing 3.3  EtchASketch example code. (continues)
Listing 3.3 EtchASketch example code.

//set the endpoint of the line segment
ptLineSeg[1].x = ptLineSeg[0].x;
p(LineSeg[1].y = ptLineSeg[0].y + iSegLen;
//invalidate the line seg area &
//send a paint message
CopyRect( &rcInvalid, (LPRECT)&ptLineSeg );
InflateRect( &rcInvalid, 1, 1 );
InvalidateRect( hWnd, &rcInvalid, FALSE);
UpdateWindow( hWnd );
//update the starting point of the next
//line segment
ptLineSeg[0].x = ptLineSeg[1].x;
p(LineSeg[0].y = ptLineSeg[1].y;
break;
case IDM_FILE_EXIT:
DestroyWindow(hWnd);
break;

default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
break;
case WM_CREATE:
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    CommandBar_AddAdornments(hwndCB, 0, 0);
    break;
case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    if( bDrawingLines )
    {
        //make a copy of the points
        CopyRect( &rcClientPoints, (LPRECT)&ptLineSeg );
        //convert to client area coordinates
        MapWindowPoints(HWND_DESKTOP,
                        hWnd,
                        (LPPOINT)&rcClientPoints, 2 );
        Polyline(hdc, (LPPOINT)&rcClientPoints, 2 );
        bDrawingLines = FALSE;
    }
    ValidateRect( hWnd, NULL );
    EndPaint(hWnd, &ps);
    break;
case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    RECT rt, rt1;
    int DlgWidth, DlgHeight;   // dialog width and height in pixel units
    int NewPosX, NewPosY;
    switch (message)
    {
    case WM_INITDIALOG:
        // trying to center the About dialog
        if (GetWindowRect(hDlg, &rt1)) {
            GetClientRect(GetParent(hDlg), &rt);
            DlgWidth = rt1.right - rt1.left;
            DlgHeight = rt1.bottom - rt1.top;
            NewPosX   = (rt.right - rt.left - DlgWidth)/2;
            NewPosY   = (rt.bottom - rt.top - DlgHeight)/2;

            // if the About box is larger than the physical screen
            if (NewPosX < 0) NewPosX = 0;
            if (NewPosY < 0) NewPosY = 0;
            SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                         0, 0, SWP_NOZORDER | SWP_NOSIZE);
        }
        return TRUE;
    case WM_COMMAND:
        if ((LOWORD(wParam) == IDOK) || (LOWORD(wParam) == IDCANCEL))
        {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
        break;
    }
    return FALSE;
}

Listing 3-3  EtchASketch example code. (continued)

Dissecting the EtchASketch Example

As you saw earlier, two of the casualties of Win CE API shrinkage are MoveToEx() and LineTo(). Now there is a single line drawing function called Polyline(). Here is the declaration of Polyline:
BOOL Polyline(HDC hdc, const POINT * lppt, int cPoints);

The parameters, in the order shown, are the handle to the display context, which receives the lines, a long pointer to an array of POINT structures, and a count of the number of points. The line segments are drawn sequentially, starting at the first point in the lppt array. CE doesn’t use the concept of a current point, so you may need to modify drawing data as well as code when you move from Win32 to Win CE.

**PORTING TIP** Windows CE doesn’t keep track of the current point when drawing lines, so the array of points passed to Polyline() must explicitly include the first point of the first line segment.

In our InitInstance function, here is the code that initializes the point structures for line drawing. We also set the length of the line segment here.

```c
//initialize the starting point of the
//ptLineSeg array to the center pt of the screen
ptLineSeg[0].x = ptLineSeg[1].x = GetSystemMetrics(SM_CXSCREEN) / 2;
ptLineSeg[0].y = ptLineSeg[1].y = GetSystemMetrics(SM_CYSCREEN) / 2;
//init length of the line segment we draw in response
//to menu commands
iSegLen = GetSystemMetrics(SM_CXSCREEN) / 25;
```

Now let’s look at what happens when the user makes a Left menu choice:

```c
case IDM_FILE_LEFT:
    bDrawingLines = TRUE;
    //set the endpoint of the line segment
    ptLineSeg[1].x = ptLineSeg[0].x - iSegLen;
    ptLineSeg[1].y = ptLineSeg[0].y;
    //Invalidate the line seg area &
    // send a paint message
    CopyRect( &rcInvalid, (LPRECT)&ptLineSeg );
    InflateRect( &rcInvalid, 1, 1 );
    InvalidateRect( hWnd, &rcInvalid, FALSE);
    UpdateWindow( hWnd );
    //update the starting point of the next
    //line segment
    ptLineSeg[0].x = ptLineSeg[1].x;
    ptLineSeg[0].y = ptLineSeg[1].y;
    break;
```

First, we set the Boolean bDrawingLines to TRUE, and update the endpoint of the line segment so that the line is drawn from the first point in the array of structures, one segment to the left. Notice that we’re doing simple arithmetic, using screen metrics information to calculate the location of the line segment’s end point. Windows CE supports only the MM_TEXT mapping mode; that is,
x coordinates increase to the right, y coordinates increase going down, and the
unit of measurement is pixels.

**PORTING TIP** Windows CE supports only the MM_TEXT mapping mode. To
draw visible lines, use only coordinates that fall within screen or client area
bounds.

The next bit of code has to with CE’s quirky erasure behavior, which I’ll say
more about when we look at the WM_PAINT message processing code:

```c
// Invalidate the line seg area &
// send a paint message
CopyRect( &rcInvalid, (LPRECT)&ptLineSeg );
InflateRect( &rcInvalid, 1, 1 );
InvalidateRect( hWnd, &rcInvalid, FALSE);
UpdateWindow( hWnd );
```

First we make a copy of the line segments points, treating it as a rectangle of
0 height. We inflate the rectangle by one pixel in each direction, and we use this
as a parameter to InvalidateRect(). Notice the final parameter of Invali-
dateRect() is FALSE, which means “don’t erase anything except that rectangle
we just invalidated.” Next, we call UpdateWindow(), which generates a
WM_PAINT message.

Finally, here is the WM_PAINT message processing code:

```c
case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    if( bDrawingLines )
    {
        // make a copy of the points
        CopyRect( &rcClientPoints, (LPRECT)&ptLineSeg );
        // convert to client area coordinates
        MapWindowPoints(HWND_DESKTOP,
                        hWnd,
                        (LPPOINT)&rcClientPoints, 2 );
        Polyline(hdc, (LPPOINT)&rcClientPoints, 2 );
        bDrawingLines = FALSE;
    }
    ValidateRect( hWnd, NULL );
    EndPaint(hWnd, &ps);
    break;
```

We open with BeginPaint() which returns a handle to the display context and
does an automatic accumulation of the invalidated regions of the client area,
erasing the sum of them. Next, we test to see if this paint message was generated
because we are drawing a line segment. If so, first we adjust the coordinates of
the points. We calculated the line segment length and location using screen
coordinates, so before we draw we convert them to client area coordinates, using MapWindowPoints.

The parameters, in the order shown, are the constant HWND_DESKTOP, the handle to our main application window, the address of the array of points to convert, and a count of points in the array. The constant HWND_DESKTOP used as the first parameter signifies that we are passing screen coordinates to this function and we want them translated to the coordinate space that belongs to the window identified by the second parameter. If you reverse the order of the first two parameters, then MapWindowPoints() converts application window coordinates to screen coordinates.

Finally, we call Polyline and draw the line, update bDrawingLines, validate the rectangle in which we drew the line segment, and clean up with a call to EndPaint().

All this brings us to the point at which we can examine why there was so much fuss about valid and invalid rectangles. If you use the BeginPaint() /EndPaint() couplet to open and close WM_PAINT message processing, it’s very difficult to control the erasure phase of painting.

On the desktop, the following bit of code usually brings you to the redrawing phase of paint processing with the client area of your window intact:

```
//Works fine on the desktop
InvalidateRect ( hWnd, NULL, FALSE );
```

You can’t count on this same behavior on CE, apparently because of the way invalid screen areas are accumulated. For this reason, with BeginPaint() / EndPaint() you must specifically invalidate the rectangle in which you want to draw, or the entire client area is erased.

**PORTING TIP** Use BeginPaint() and EndPaint() to open and close WM_PAINT processing only if you wish to redraw the entire client area in response to a paint message. To leave most of the client area unchanged and redraw only selected portions, use GetDC() and ReleaseDC().

Listing 3.4 shows the code for an improved version of EtchASketch, called Etch2. Notice that the lines drawn by Etch2 are solid rather than dashed like those in EtchASketch. The dashed lines weren’t caused by the pen—they were artifacts of inflating the invalid rectangle. (Once again, we omit the resource file code in the interest of brevity. This, and the rest of the project files, can be found on the accompanying CD.)

```
// Etch2.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "Etch2.h"
```

**Listing 3.4** Etch2 example code.
Listing 3.4  Etch2 example code. (continues)
so that the application will get 'well formed' small icons
associated with it.

ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS    wc;
    wc.style            = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc        = (WNDPROC) WndProc;
    wc.cbClsExtra        = 0;
    wc.cbWndExtra        = 0;
    wc.hInstance        = hInstance;
    wc.hIcon            = LoadIcon(hInstance,
                                  MAKEINTRESOURCE(IDI_ETCH2));
    wc.hCursor            = 0;
    wc.hbrBackground    = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName        = 0;
    wc.lpszClassName    = szWindowClass;
    return RegisterClass(&wc);
}

//  FUNCTION: InitInstance(HANDLE, int)
//  PURPOSE: Saves instance handle and creates main window
//  COMMENTS:
//    In this function, we save the instance
//    handle in a global variable and
//    create and display the main program window.
//  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND    hWnd;
    TCHAR    szTitle[MAX_LOADSTRING];            // The title bar text
    TCHAR    szWindowClass[MAX_LOADSTRING];   // The window class name
    hInst = hInstance;        // Store instance handle in global
    // Initialize global strings
    LoadString(hInstance, IDC_ETCH2, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle,
                         WS_VISIBLE,
                         0, 0,
                         CW_USEDEFAULT,
                         CW_USEDEFAULT,
                         NULL, NULL,
                         hInstance, NULL);

    if (!hWnd)
        
Listing 3.4  Etch2 example code.
return FALSE;
}
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);

//initialize the starting point of the
//ptLineSeg array to the center pt of the screen
ptLineSeg[0].x = ptLineSeg[1].x = GetSystemMetrics(SM_CXSCREEN) / 2;
ptLineSeg[0].y = ptLineSeg[1].y = GetSystemMetrics(SM_CYSCREEN) / 2;

//init length of the line segment we draw in response
//to menu commands
iSegLen = GetSystemMetrics(SM_CXSCREEN) / 25;
return TRUE;
}

Listing 3.4  Etch2 example code. (continues)
ptLineSeg[1].y = ptLineSeg[0].y;
DrawLines( hWnd );
break;
case IDM_FILE_UP:
  //set the endpoint of the line segment
  ptLineSeg[1].x = ptLineSeg[0].x;
  ptLineSeg[1].y = ptLineSeg[0].y - iSegLen;
  DrawLines( hWnd );
  break;
case IDM_FILE_DOWN:
  //set the endpoint of the line segment
  ptLineSeg[1].x = ptLineSeg[0].x;
  ptLineSeg[1].y = ptLineSeg[0].y + iSegLen;
  DrawLines( hWnd );
  break;
case IDM_FILE_EXIT:
  DestroyWindow(hWnd);
  break;
default:
  return DefWindowProc(hWnd, message, wParam, lParam);
}
break;
case WM_CREATE:
  hwndCB = CommandBar_Create(hInst, hWnd, 1);
  CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
  CommandBar_AddAdornments(hwndCB, 0, 0);
  break;
case WM_DESTROY:
  CommandBar_Destroy(hwndCB);
  PostQuitMessage(0);
  break;
default:
  return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
  RECT rt, rt1;
  int DlgWidth, DlgHeight;       // dialog width and height
  int NewPosX, NewPosY;         // in pixel units
  switch (message)
  {
    case WM_INITDIALOG:
      // trying to center the About dialog
      if (GetWindowRect(hDlg, &rt1)) {

Listing 3.4  Etch2 example code.
GetClientRect(GetParent(hDlg), &rt);
DlgWidth    = rt1.right - rt1.left;
DlgHeight   = rt1.bottom - rt1.top;
NewPosX     = (rt.right - rt.left - DlgWidth)/2;
NewPosY     = (rt.bottom - rt.top - DlgHeight)/2;

    // if the About box is larger than the physical screen
    if (NewPosX < 0) NewPosX = 0;
    if (NewPosY < 0) NewPosY = 0;
    SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                 0, 0, SWP_NOZORDER | SWP_NOSIZE);
}
return TRUE;
case WM_COMMAND:
    if ((LOWORD(wParam) == IDOK) ||
        (LOWORD(wParam) == IDCANCEL))
    {
        EndDialog(hDlg, LOWORD(wParam));
        return TRUE;
    }
    break;
}
return FALSE;

void DrawLines( HWND hWnd )
{
    HDC       hdc;
    RECT      rcClientPoints;
    hdc = GetDC(hWnd);
    //make a copy of the points
    CopyRect( &rcClientPoints, (LPRECT)&ptLineSeg );
    //convert to client area coordinates
    MapWindowPoints( hWnd, HWND_DESKTOP,(LPPOINT)&rcClientPoints, 2 );
    Polyline(hdc, (LPPOINT)&rcClientPoints, 2 );
    ReleaseDC(hWnd, hdc);

    //update the starting point of the next
    //line segment
    ptLineSeg[0].x = ptLineSeg[1].x;
    ptLineSeg[0].y = ptLineSeg[1].y;
}

Listing 3.4  Etch2 example code. (continued)

There are two main differences between EtchASketch and Etch2. First, Etch2
omits processing of the WM_PAINT message. This may seem strange, but it
has a couple of important advantages. First, it ensures that when a drop menu
or dialog control has covered part of the client area, the affected screen real
estate will get default invalidation and repainting. Second, it is completely reli-
able and adds nothing to your application’s code footprint.
For our application-specific drawing, we don’t use the WM_PAINT message at all, and therefore don’t have to invalidate anything in order to draw or to keep an existing drawing in the client area from being erased.

**PORTING TIP** To draw incrementally, without the client area being erased, write an application-specific drawing (or painting) function that uses GetDC() and ReleaseDC.

There are just a couple of points to make about line drawing before moving on. You may have noticed that I didn’t mention using CreateDC(), SaveDC(), or RestoreDC(), all of which are supported by Win CE. Here’s why. CE maintains a small pool of DCs, and you get access to one of these when you call GetDC(). When you call ReleaseDC(), it’s returned to the pool and available to any other process. CreateDC() allocates memory and builds you a DC. Unless you have a compelling reason for creating your own DC, this is wasteful. Likewise, SaveDC() pushes your DC onto a stack, where it unproductively takes up space until you need it again. Again, this is probably wasteful.

Last but not least, Polygon() and Polyline() are essentially the same, except that Polygon() automatically draws a closing line segment between the last point on the last line and the first point on the first line. You won’t have to add closing line segments to existing Polygon() coordinates.

**Using Bitmaps**

Bitmaps raise all sorts of porting issues: color depth, bitmap organization, size translations, and CE support for ROP codes. We are going to start with a look at basic bitmap raster operations under CE, using the functions BitBlt() and StretchBlt().

![Figure 3.3 The Dinner example.](image)
Here Listing 3.5 shows the main source code file for the Dinner application, Dinner.cpp, and Figure 3.3 shows a dinner example screen.

```cpp
// Dinner.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "Dinner.h"
#include <commctrl.h>
#include <commctrl.h>
#define MAX_LOADSTRING 100
#define NORMAL         0
#define ENLARGE        1
#define SHRINK         -1

// Global Variables:
HINSTANCE   hInst;                  // The current instance
HWND        hwndCB;                 // The command bar handle
int         iROPCode = SRCCOPY;     // ROP to use in the BitBlt
int         iZoomFactor = 0;        // 0 = physical size
          // 1 = enlarge
          // -1 = shrink
// Foward declarations of functions included in this code module:
ATOM MyRegisterClass (HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK  WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK  About (HWND, UINT, WPARAM, LPARAM);
void MyPaint (HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPTSTR    lpCmdLine,
                  int       nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }
    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_DINNER);
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return msg.wParam;
}
```

Listing 3.5  Dinner.cpp (continues)
FUNCTION: MyRegisterClass()

PURPOSE: Registers the window class.

COMMENTS:

This function and its usage is only necessary if you want this code to be compatible with Win32 systems prior to the 'RegisterClassEx' function that was added to Windows 95. It is important to call this function so that the application will get 'well formed' small icons associated with it.

ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS    wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_DINNER));
    wc.hCursor = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = 0;
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);
}

FUNCTION: InitInstance(HANDLE, int)

PURPOSE: Saves instance handle and creates main window

COMMENTS:

In this function, we save the instance handle in a global variable and create and display the main program window.

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND    hWnd;
    TCHAR    szTitle[MAX_LOADSTRING]; // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING]; // The window class name
hInst = hInstance; // Store instance handle in our global

// Initialize global strings
LoadString(hInstance, IDC_DINNER, szWindowClass, MAX_LOADSTRING);
MyRegisterClass(hInstance, szWindowClass);
LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
hWnd = CreateWindow(szWindowClass, szTitle,
                      WS_VISIBLE,
                      0, 0,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      NULL, NULL,
                      hInstance, NULL);

if (!hWnd)
{
    return FALSE;
}
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
return TRUE;

Listing 3.5 Dinner.cpp (continues)
switch (wmId)
{
    case IDM_ROPCODES_SRCAND:
        iROPCode = SRCAND;
        break;
    case IDM_ROPCODES_SRCCOPY:
        iROPCode = SRCCOPY;
        break;
    case IDM_ROPCODES_SRCCLEAR:
        iROPCode = SRCERASE;
        break;
    case IDM_ROPCODES_SRCPAINT:
        iROPCode = SRCPAINT;
        break;
    case IDM_ZOOM_NORMAL:
        iZoomFactor = NORMAL;
        break;
    case IDM_ZOOM_LARGE:
        iZoomFactor = ENLARGE;
        break;
    case IDM_ZOOM_SMALL:
        iZoomFactor = SHRINK;
        break;
    case IDM_HELP_ABOUT:
        DialogBox(hInst,
            (LPCTSTR)IDD_ABOUTBOX,
            hWnd, (DLGPROC)About);
        break;
    case IDM_FILE_EXIT:
        DestroyWindow(hWnd);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
}
InvalidateRect(hWnd, NULL, TRUE);
UpdateWindow(hWnd);
break;
case WM_CREATE:
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    CommandBar_AddAdornments(hwndCB, 0, 0);
    break;
case WM_PAINT:
    MyPaint(hWnd, message, wParam, lParam);

Listing 3.5  Dinner.cpp
break;

case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}

// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    RECT rt, rt1;
    int DlgWidth, DlgHeight;    // dialog width and height
    int NewPosX, NewPosY;
    switch (message)
    {
    case WM_INITDIALOG:
        // trying to center the About dialog
        if (GetWindowRect(hDlg, &rt1)) {
            GetClientRect(GetParent(hDlg), &rt);
            DlgWidth   = rt1.right - rt1.left;
            DlgHeight  = rt1.bottom - rt1.top;
            NewPosX    = (rt.right - rt.left - DlgWidth)/2;
            NewPosY    = (rt.bottom - rt.top - DlgHeight)/2;

            // if the About box is larger than the physical screen
            if (NewPosX < 0) NewPosX = 0;
            if (NewPosY < 0) NewPosY = 0;
            SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                         0, 0, SWP_NOZORDER | SWP_NOSIZE);
        }
        return TRUE;
    case WM_COMMAND:
        if ((LOWORD(wParam) == IDOK) ||
            (LOWORD(wParam) == IDCANCEL))
        {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
        break;
    }
    return FALSE;
}

void MyPaint(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
Listing 3.5  Dinner.cpp (continued)

```cpp
HDC hdc, hdcMemory;

PAINTSTRUCT ps;
RECT rt;
HBITMAP hBmp;
BITMAP bmp;

hdc = BeginPaint(hWnd, &ps);
GetClientRect(hWnd, &rt);
FillRect(hdc, &rt, (HBRUSH)GetStockObject(GRAY_BRUSH));
hdcMemory = CreateCompatibleDC(hdc);
hBmp = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_DINNER));
SelectObject(hdcMemory, hBmp);
GetObject(hBmp, sizeof(BITMAP), &bmp);
if (iZoomFactor == NORMAL)
{
    BitBlt(hdc, (GetSystemMetrics(SM_CXSCREEN)/10),
           (GetSystemMetrics(SM_CYSCREEN)/6),
           bmp.bmWidth, bmp.bmHeight,
           hdcMemory,
           0, 0, iROPCode);
}
if (iZoomFactor == ENLARGE)
{
    StretchBlt(hdc, (GetSystemMetrics(SM_CXSCREEN)/10),
               (GetSystemMetrics(SM_CYSCREEN)/6),
               bmp.bmWidth*2, bmp.bmHeight*2,
               hdcMemory,
               0, 0,
               bmp.bmWidth, bmp.bmHeight,
               iROPCode);
}
if (iZoomFactor == SHRINK)
{
    StretchBlt(hdc, (GetSystemMetrics(SM_CXSCREEN)/10),
               (GetSystemMetrics(SM_CYSCREEN)/6),
               bmp.bmWidth/2, bmp.bmHeight/2,
               hdcMemory,
               0, 0,
               bmp.bmWidth, bmp.bmHeight,
               iROPCode);
}
DeleteDC(hdcMemory);
EndPaint(hWnd, &ps);
```
Device-Independent vs. Device-Dependent Bitmaps

The real business of the Dinner example happens in the function MyPaint(). First, we fill the client area with gray, because this makes the actions of the various ROP codes more dramatic. Next, we create a compatible memory DC, load a bitmap image, select it into the memory DC, and finally, use either BitBlt() or StretchBlt() to paint it on the screen. This drawing code is 100 percent identical on Win32 and Win CE, but it relies on one pivotal assumption, that the CE device is capable of displaying the device-dependent bitmap identified by IDB_DINNER. As it turns out, this is a safe assumption, and we see, *voilà*, dinner.

This example introduces some of the larger issues of porting graphics to CE, to wit, that getting a graphics application up on CE may well involve modifying data as well as code.

**PORTING TIP** Porting graphics applications may require more modification of graphics data than code.

We saw in the EtchASketch example that in order to draw a line, you must explicitly supply its initial point. Now with bitmaps, we are confronted with another such consideration: possible differences in color depth between a bitmap and a display device. On the one hand, device-dependent bitmaps are compact, ubiquitous, and require very little coding to manipulate. On the other hand, when using device-dependent bitmaps you run a risk that a device can’t handle them—possibly with catastrophic results. There are two alternatives:

- Use DDBs carefully and intelligently, limiting the risks involved.
- Use DIBs, which are universally compatible, at the cost of a great deal more code and data.

You probably already know which alternative I favor. If you can live with 16-color or grayscale bitmaps, you will vastly reduce the effort of porting bitmap code. Simply opening the bitmaps in a graphics application and saving them as 16-color or 4-bit grayscale format may allow your device-dependent bitmap code to port to Win32 unchanged.

Your safety net on the CE side is provided by the GetDeviceCaps() function:

```c
int GetDeviceCaps(HDC hdc, int nIndex);
```

The parameters, in the order shown, are the handle to the display context and a constant identifying the device property you are querying. To find out how many colors a device supports, use GetDeviceCaps() like this:
int iNumColors;
//Get the number of colors the device can display
iNumColors = GetDeviceCaps(hdc, NUMCOLORS);
if ( (iNumColors >= 16) || (iNumColors == -1) )
{
    //safely use grayscale or 16 color DDBs
}

GetDeviceCaps() returns the number of colors the device can display if color depth is 8 bits or less. If more than 8 bits are used to identify a color, then the function returns -1.

Making Bitmap Backgrounds Transparent

Another really useful device-dependent bitmap function is TransparentImage(). This API draws a bitmap on the screen, treating all the bitmap pixels of a specified color as transparent. In other words, the existing window background shows through the transparent pixels. The DuckStamps example (see Figure 3.4) shows how to use TransparentImage().

![Figure 3.4 Using TransparentImage().](image)

Listing 3.6 DuckStamps example code.
Listing 3.6  DuckStamps example code. (continues)
{  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
}
return msg.wParam;
}

//  FUNCTION: MyRegisterClass()
//  PURPOSE: Registers the window class.
//  COMMENTS:
//  This function and its usage is only necessary if you want this code to be compatible with Win32 systems prior to the 'RegisterClassEx' function that was added to Windows 95. It is important to call this function so that the application will get 'well formed' small icons associated with it.

ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
  WNDCLASS    wc;
  wc.style     = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc = (WNDPROC) WndProc;
  wc.cbClsExtra  = 0;
  wc.cbWndExtra  = 0;
  wc.hInstance   = hInstance;
  wc.hIcon       = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_DUCKSTAMPS));
  wc.hCursor     = 0;
  wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
  wc.lpszMenuName = 0;
  wc.lpszClassName = szWindowClass;
  return RegisterClass(&wc);
}

//  FUNCTION: InitInstance(HANDLE, int)
//  PURPOSE: Saves instance handle and creates main window
//  COMMENTS:

Listing 3.6  DuckStamps example code.
// In this function, we save the instance handle
// in a global variable and
// create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    TCHAR szTitle[MAX_LOADSTRING];  // The title bar text
    TCHAR szWindowClass[MAX_LOADSTRING]; // The window class name

    // Store instance handle in our global variable
    hInst = hInstance;

    // Initialize global strings
    LoadString(hInstance, IDC_DUCKSTAMPS,
        szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle,
        WS_VISIBLE,
        0, 0,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL, NULL,
        hInstance, NULL);

    if (!hWnd)
    {
        return FALSE;
    }
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    if (hwndCB)
        CommandBar_Show(hwndCB, TRUE);
    return TRUE;
}

Listing 3.6 DuckStamps example code. (continues)
{ int wmId, wmEvent;
 switch (message)
 {
 case WM_COMMAND:  
    wmId = LOWORD(wParam);
    wmEvent = HIWORD(wParam);
    // Parse the menu selections:
    switch (wmId)
    {
    case IDM_VIEW_TRANSPARENT:
       iView = TRANSPARENT_STAMP;
       break;
    case IDM_VIEW_OPAQUE:
       iView = OPAQUE_STAMP;
       break;
    case IDM_HELP_ABOUT:
       MessageBox(hInst,
                  (LPCTSTR)IDD_ABOUTBOX,
                  hWnd, (DLGPROC)About);
       break;
    case IDM_FILE_EXIT:
       DestroyWindow(hWnd);
       break;
    default:
       return DefWindowProc(hWnd, message,
            wParam, lParam);
    }
    InvalidateRect(hWnd, NULL, TRUE);
    UpdateWindow( hWnd );
    break;
  case WM_CREATE:
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    CommandBar_AddAdornments(hwndCB, 0, 0);
    break;
  case WM_PAINT:
    MyPaint(hWnd, message, wParam, lParam);
    break;
  case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
  default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
// Message handler for the About box.

Listing 3.6 DuckStamps example code.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    RECT rt, rt1;
    int DlgWidth, DlgHeight; // dialog width and height
    // in pixel units
    int NewPosX, NewPosY;
    switch (message)
    {
    case WM_INITDIALOG:
        // trying to center the About dialog
        if (GetWindowRect(hDlg, &rt1)) {
            GetClientRect(GetParent(hDlg), &rt);
            DlgWidth    = rt1.right - rt1.left;
            DlgHeight    = rt1.bottom - rt1.top;
            NewPosX = (rt.right - rt.left - DlgWidth)/2;
            NewPosY = (rt.bottom - rt.top - DlgHeight)/2;

            // if the About box is larger than the physical screen
            if (NewPosX < 0) NewPosX = 0;
            if (NewPosY < 0) NewPosY = 0;
            SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                         0, 0, SWP_NOZORDER | SWP_NOSIZE);
        }
        return TRUE;
    case WM_COMMAND:
        if ((LOWORD(wParam) == IDOK) ||
            (LOWORD(wParam) == IDCANCEL))
        {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
        break;
    }
    return FALSE;
}

void MyPaint(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc, hdcMemory;
    PAINTSTRUCT ps;
    HBDMAP hBmp, hOldBmp;
    BITMAP  bmp;
    int i, x, y, iCaptionHeight;
    hdc = BeginPaint(hWnd, &ps);
    iCaptionHeight = GetSystemMetrics(SM_CYCAPTION) + 5;
    hdcMemory = CreateCompatibleDC(hdc);
    if (iView == OPAQUE_STAMP)
    {
        for (i = 0; i < 4; i++)
        Listing 3.6 DuckStamps example code. (continues)
{ 
    hBmp = LoadBitmap(hInst, 
                    MAKEINTRESOURCE(IDB_COCKATOO + i));
    hOldBmp = SelectObject( hdcMemory, hBmp );
    GetObject( hBmp, sizeof(BITMAP), &bmp );
    x = (( (i + 1) % 2) == 0) ? bmp.bmWidth + 5 : 0;
    y = (i > 1) ? iCaptionHeight + bmp.bmHeight + 5 : iCaptionHeight;
    BitBlt(hdc, x, y, 
           bmp.bmWidth, bmp.bmHeight, 
           hdcMemory, 
           0, 0, SRCCOPY );
    if( bFirstTime )
    {
        crTransparentColor[i] = GetPixel( hdc, x + 1, y + 1 );
    }
    DeleteObject( SelectObject( hdcMemory, hOldBmp ) );
}

bFirstTime = FALSE;
if ( iView == TRANSPARENT_STAMP )
{
    for( i = 0; i < 4; i++ )
    {
        hBmp = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_COCKATOO + i));
        hOldBmp = SelectObject( hdcMemory, hBmp );
        GetObject( hBmp, sizeof(BITMAP), &bmp );
        x = (( (i + 1) % 2) == 0) ? bmp.bmWidth + 5 : 0;
        y = (i > 1) ? iCaptionHeight + bmp.bmHeight + 5 : iCaptionHeight;

        TransparentImage(hdc, x, y, bmp.bmWidth,
                        bmp.bmHeight,
                        hdcMemory, 0, 0,
                        bmp.bmWidth, bmp.bmHeight,
                        crTransparentColor[i]);
        DeleteObject( SelectObject( hdcMemory, hOldBmp ) );
    }
}
DeleteDC( hdcMemory );
EndPaint(hWnd, &ps);
}

Listing 3.6 DuckStamps example code. (continued)

Once more, the MyPaint() function is the scene of all the action. We Begin-
Paint() and EndPaint() because we have enough information to completely
repaint the client area whenever we enter this function. The first time we load
a bitmap, we sample the background color of the image, and save it in the array, crTransparentColor :

    //global data
    COLORREF     crTransparentColor[4];
    // which color to make transparent
    //for each bitmap find the COLORREF value we'll
    // use to make transparent pixels
    if( bFirstTime )
    {
        //save the COLORREF value in a global array
        crTransparentColor[i] = GetPixel( hdc, x + 1, y + 1 );
    }

    When the user chooses “Transparent” from the View menu, we call TransparentImage() to display the bitmaps:

    TransparentImage(hdc, x, y, bmp.bmWidth, bmp.bmHeight,
                     hdcMemory, 0, 0, bmp.bmWidth,
                     bmp.bmHeight,
                     crTransparentColor[i]);

    TransparentImage() has two more parameters than BitBlt(). Here is its function declaration:

    BOOL TransparentImage(HDC hdcDest,
                           LONG DestOriginX, LONG DestOriginY,
                           LONG DestWidth, LONG DestHeight,
                           HANDLE hSrc,
                           LONG SrcOriginX, LONG SrcOriginY,
                           LONG SrcWidth, LONG SrcHeight,
                           COLORREF TransparentColor );

    The extra parameters, SrcWidth and SrcHeight allow you to isolate and use rectangular regions of the source image.

**When to Use Device-Independent Bitmaps**

There are some circumstances when it makes sense to use device-independent bitmaps. Here is a list of factors that would make DIBs desirable:

- Your Win32 application uses them
- Your application depends on photographic-quality images
- Your application sets or actively manages palettes
- Your application copies bitmaps from the screen, modifies them, and then copies them back to the screen.
If your application uses DIBs, you can load them by name using this API:

```c
HBITMAP SHLoadDIBitmap(LPCTSTR szFileName);
```

Notice that the filename parameter is typed LPCTSTR. This is a long pointer to a TCHAR, and in order to successfully load the file, the filename string must be in Unicode. This is how to use SHLoadDIBitmap() with a literal string as a filename:

```c
SHLoadDIBitmap(TEXT(szDIBFilename));
```

---

**Color**

Windows CE supports color depths of 1, 2, 4, 8, 16, 24, and 32 bits per pixel. Most of the time, devices organize some or all of the available colors using a structure called a palette. Often, even if a device is theoretically capable of displaying millions of colors, it can only display a subset of them simultaneously. The colors available for display are the ones in the currently selected palette. When you get or create a display context, (for example, using GetDC() or CreateCompatibleDC()), Windows CE creates a default palette for that device context. The exact composition of that palette may differ across devices, because there is no standard Windows CE color palette.

DIBs contain their own color information in an array of structures that follows the DIB header information. The color of individual pixels in the image is determined by indexing this color table. Windows CE matches the colors specified by the DIB as closely as it is able, using the colors currently available in the selected palette. Windows CE does not support dithered colors.

Device-dependent bitmaps simply contain RGB color information for each pixel. You don’t get the benefit of the color arbitration step that happens when a DIB is rendered, and if you try to display an image with greater color depth than the device supports, at best the attempt will fail. At worst, your application may crash and lock up the CE device.

---

**Pens, Brushes, and Fonts**

If possible, you should try to stick with the available stock pens and brushes. There are always present and live in ROM, so they come to your application at no memory cost. You also don’t have to worry about freeing them when you are finished using them. Table 3.1 lists your options.
Table 3.1  Windows CE Stock Objects

<table>
<thead>
<tr>
<th>GETSTOCKOBJECT() CONSTANT</th>
<th>OBJECT PROPERTIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACK_BRUSH</td>
<td>Black brush</td>
</tr>
<tr>
<td>DKGRAY_BRUSH</td>
<td>Dark gray brush</td>
</tr>
<tr>
<td>GRAY_BRUSH</td>
<td>Gray brush</td>
</tr>
<tr>
<td>HOLLOW_BRUSH</td>
<td>Hollow brush doesn’t erase what it paints (equivalent to NULL_BRUSH)</td>
</tr>
<tr>
<td>LTGRAY_BRUSH</td>
<td>Light gray brush</td>
</tr>
<tr>
<td>NULL_BRUSH</td>
<td>Same as HOLLOW_BRUSH</td>
</tr>
<tr>
<td>WHITE_BRUSH</td>
<td>White brush</td>
</tr>
<tr>
<td>BLACK_PEN</td>
<td>Black pen</td>
</tr>
<tr>
<td>WHITE_PEN</td>
<td>White pen</td>
</tr>
<tr>
<td>SYSTEM_FONT</td>
<td>System font. By default, the system uses the system font to draw menus, dialog box controls, and text.</td>
</tr>
<tr>
<td>DEFAULT_PALETTE</td>
<td>The default set of colors</td>
</tr>
</tbody>
</table>

The number of each type of object in the stock objects collection is instructive. There are several brushes, which in practice are small, grayscale bitmaps. There are only two pens, a black one and a white one, and what doesn’t show up in the table is that each of the pens is 1 pixel wide. Next there’s a single system font, and we finish with the default palette.

This set of objects is designed to be highly portable, and it reproduces well even on 1-bit devices. It is also designed to use an absolute minimum of memory for these necessary resources. What I take from this economy is that its possible to do an extraordinary amount of productive work without an elaborate graphical infrastructure. Think about what you need to accomplish with graphics in your application, and if at all possible consider paring down your working set of graphic objects, especially the following:

- Always use the system font.
- Don’t create pens that are more than 1 pixel wide. Drawing with wider pens is computationally intensive, slows performance, and escalates power consumption.
- Avoid the use of Ellipse() and RoundRect where possible; these are also computationally intensive.
Usability studies have shown that human vision is optimized for recognizing contrast and locating edges. The tiny, backlit screens featured in most CE devices wash out in even moderate light. This argues against using a large palette of colors and against serif, rotated, or decorative fonts.

**PORTING TIP** For greatest application usability and readability, employ graphics and fonts that display dark, sharp-edged features against a white background.

**Looking Ahead**

Windows CE device interfaces are inherently graphical, especially in the palm-top and Pocket PC form factors. More than 90 percent of the CE computers deployed today belong to this tribe of ultra-small computers, and the most important thing about this group of devices is that they lack traditional keyboards. In a sense, the tinier CE devices have brought us full circle. Thousands of years ago, we got a lot of exercise, ate vegetables in season, and wrote in the mud with sticks, this last owing in part to the absence of keyboards. Modern as we are, today’s Homo sapiens hit the salad bar after step aerobics, all the while scribbling reminders on the iPaq using a stylus. C’est plus ce change, ce plus le même chose.

Most ported applications will need to reevaluate their input handling. Users of the smaller devices simply won’t accept an interface that forces them to tap out all text input on the pop-up chicklet keyboard. Direct support for stylus input is a key feature. From a programming point of view, the significant thing about this trend is that accepting stylus input has more in common with graphical programming than with handling a keyboard or mouse input stream. In the next chapter, we’ll explore some approaches to setting up to receive and manage stylus input.
CE’s stylus-oriented interface is really a bonus for Windows CE programmers, because it allows us to dispense with a great deal of the housekeeping that goes with mouse- and keyboard-based user input on the desktop. Being a touch-screen kind of device, the stylus eliminates the need for floating cursors, inherently keeping track of insertion points and the like.

When used to tap the CE device’s screen, the stylus is conceptually treated as a 1-button mouse. For this reason, you may assume existing programming logic that detects left mouse button presses, including double clicks, will port to CE with few or no changes. Code that handles right mouse button presses will require some workarounds, but porting this code is more a matter of strategy than technique.

Many users prefer the stylus to the tiny CE keyboards for entering text. A user’s handwritten input is called inking. With this in mind, you’ll want to give some consideration to allowing users to provide textual input to your application using the inking capability of the CE device. I use the word consideration advisedly, for there are some serious issues to weigh in this matter. On the positive side, users really like being able to treat their CE devices as smart memo pads. Conversely, you’ll find that most of them really dislike typing with a stylus. This argues in favor of giving users the opportunity to interact with your
application in the ways they find most convenient. On the other hand, the implementation of inking is maddeningly asymmetrical across the various CE versions and platforms. The code you write to support inking will not be uniformly portable across CE devices. If you choose to aggressively support inking, expect to maintain multiple codebases until a single, seamless inking API evolves.

With a view to the overarching theme of this book (rapidly porting Win32 code to CE), I’ll offer my personal metric for deciding when to include inking capability in the first CE version of your Win32 app:

- Speed and ease of input are very important
- Absolute accuracy is relatively unimportant
- It’s not practical to supply a predetermined list of choices
- The application doesn’t use the input to drive runtime decision making

| PORTING TIP | Use inking to store small to medium amounts of text that don’t drive runtime decision making. |

We’ll move on to explore example applications that show how to use the stylus as a substitute for the mouse and how to use it as a pen.

**The Stylus Behaves as a One-Button Mouse**

The StylusTrack example application shows how to receive and manipulate stylus input. It captures the coordinates of the starting and ending points of the line the user specifies by dragging the stylus across the screen, then uses the point either as the diagonal of a rectangle or of the bounding rectangle of an ellipse. Figure 4.1 shows how the StylusTrack example looks on an HPC.

![Figure 4.1 The StylusTrack application.](image)
Listing 4.1 shows the main source file for the StylusTrack example. In the interest of brevity, I have omitted the support files, which can be found in their entirety on the accompanying CD.

```cpp
// StylusTrack.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "StylusTrack.h"
#include <commctrl.h>
#define MAX_LOADSTRING 100
#define RECTANGLE 1
#define ELLIPSE 2

// Global Variables:
HINSTANCE hInst; // The current instance
HWND hwndCB; // The command bar handle
int iFigure = 0; //which type of figure to draw
POINT line[2];
BOOL bLBDown = FALSE;

// Forward declarations of functions included in this code module:
ATOM MyRegisterClass (HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance,
LPTSTR    lpCmdLine,
int       nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }
    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_STYLUSTRACK);
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return msg.wParam;
}
```

Listing 4.1  StylusTrack.Cpp. (continues)
// FUNCTION: MyRegisterClass()
// PURPOSE: Registers the window class.
// COMMENTS:
// This function and its usage is only necessary if you want this code to be compatible with Win32 systems prior to the 'RegisterClassEx' function that was added to Windows 95. It is important to call this function so that the application will get 'well formed' small icons associated with it.
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_STYLUSTRACK));
    wc.hCursor = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = 0;
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);
}

// FUNCTION: InitInstance(HANDLE, int)
// PURPOSE: Saves instance handle and creates main window
// COMMENTS:
// In this function, we save the instance handle in a global variable and create and display the main program window.
//
// BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    TCHAR szTitle[MAX_LOADSTRING];
    // The title bar text

Listing 4.1  StylusTrack.Cpp.
Listing 4.1  StylusTrack.Cpp. (continues)
wmId = LOWORD(wParam);
wmEvent = HIWORD(wParam);
// Parse the menu selections:
switch (wmId)
{
    case IDM_FILE_RECTANGLE:
        iFigure = RECTANGLE;
        InvalidateRect( hWnd, NULL, TRUE );
        UpdateWindow( hWnd );
        break;
    case IDM_FILE_ELLIPSE:
        iFigure = ELLIPSE;
        InvalidateRect( hWnd, NULL, TRUE );
        UpdateWindow( hWnd );
        break;
    case IDM_HELP_ABOUT:
        DialogBox(hInst,
            (LPCTSTR)IDD_ABOUTBOX,
            hWnd, (DLGPROC)About);
        break;
    case IDM_FILE_EXIT:
        DestroyWindow(hWnd);
        break;
    default:
        return DefWindowProc(hWnd, message,
            wParam, lParam);
    }
    break;
}

Listing 4.1  StylusTrack.Cpp.
if ( bLBDown == TRUE )
{
    hdc = GetDC(hWnd);
    Polyline(hdc, line, 2);
    if( iFigure == RECTANGLE )
    {
        Rectangle( hdc, line[0].x, line[0].y,
                   line[1].x, line[1].y );
    }
    if( iFigure == ELLIPSE )
    {
        Ellipse( hdc, line[0].x, line[0].y,
                 line[1].x, line[1].y );
    }

    //reset value of initial point to end point
    line[1].x = 0;
    line[1].y = 0;
    line[0] = line[1];
    ReleaseDC(hWnd, hdc);
}  //end if LButtonDown
bLBDown = FALSE;
break;
case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
} return 0;

Listing 4.1  StylusTrack.Cpp. (continues)
DlghHeight  = rt1.bottom - rt1.top;
NewPosX     = (rt.right - rt.left - DlgWidth)/2;
NewPosY     = (rt.bottom - rt.top - DlgHeight)/2;

// if the About box is larger than the physical screen
if (NewPosX < 0) NewPosX = 0;
if (NewPosY < 0) NewPosY = 0;
SetWindowPos(hDlg, 0, NewPosX, NewPosY,
0, 0, SWP_NOZORDER | SWP_NOSIZE);
}
return TRUE;
case WM_COMMAND:
if ((LOWORD(wParam) == IDOK) ||
(LOWORD(wParam) == IDCANCEL))
{
EndDialog(hDlg, LOWORD(wParam));
return TRUE;
}
break;
}
return FALSE;
}

Listing 4.1  StylusTrack.Cpp. (continued)

All of the drawing done in the StylusTrack example is done in response to
mouse messages. Notice that the message switch has no case for application-
specific handling of the WM_PAINT message. Let’s look at the initial stylus
handling, done in the WM_LBUTTONDOWN case of the message switch:

case WM_LBUTTONDOWN :
  //If Left button is down, save mouse coordinates in Line[0]
  //point
  bLBDown = TRUE;
  line[0].x = LOWORD(lParam);
  line[0].y = HIWORD(lParam);
  break;

Notice that we don’t have to use SetCapture() to get the mouse movement
messages generated by the stylus. By default, CE sends the top window the
mouse messages. In response to the WM_LBUTTONDOWN, we set the bLB-
Down to TRUE, and copy the coordinates at which the mouse button went
down from the lParam.
When the WM_LBUTTONDOWN message arrives, the drawing work begins. First, we copy the point coordinates from the lParam of the message. We test them to see if the stylus has actually moved or if this was just a tap. If the stylus moved, we draw a line that is either the diagonal of the rectangle or the long axis of the ellipse.

case WM_LBUTTONDOWN :
    //Draw last line. Since Left button down is
    //false, no more lines are drawn
    //until next WM_LBUTTONDOWN message
    line[1].x = LOWORD(lParam);
    line[1].y = HIWORD(lParam);

    if((line[0].x != line[1].x) || (line[0].y != line[1].y))
    { Polyline(hdc, line, 2); }
    if ( bLBDown == TRUE )
    {
        hdc = GetDC(hWnd);
        Polyline(hdc, line, 2);
    }

Next, we use the coordinates to draw either the rectangle or the ellipse:

    if( iFigure == RECTANGLE )
    {
        Rectangle( hdc, line[0].x, line[0].y,
                    line[1].x, line[1].y );
    }
    if( iFigure == ELLIPSE )
    {
        Ellipse( hdc, line[0].x, line[0].y,
                 line[1].x, line[1].y );
    }

Finally, we reinitialize the coordinates in the point array, release the DC, and set bLBDown to FALSE.

    //reset value of initial point to end point
    line[1].x = 0;
    line[1].y = 0;
    line[0] = line[1];

    ReleaseDC(hWnd, hdc);
}  //end if LButtonDown
bLBDown = FALSE;
If you wanted to continuously track the stylus, you could add a WM_MOUSEMOVE case that accumulated points in a larger array, or simply did the drawing as the stylus moved.

**Using the CE Ink Control**

As mentioned in the opening paragraphs of this chapter, the inking implementation isn’t very consistent across CE platforms. Unlike the other examples in this book, you can’t count on this one working anywhere except on the platform for which it was written. (That’s why this example project is set up to target the emulator.) That’s the bad news. Now for the good news. If you simply wish to collect handwritten input, store it, and redisplay it, the RichInk control makes inking data fairly easy to acquire and handle. Though details differ among the CE platforms, the basic strategy is consistent:

- Include inkx.lib in the linker specification of your project settings
- Load the common controls
- Specifically initialize the ink control
- Communicate with the control using the messages IM_GETDATALEN, IM_GETDATA, and IM_SETDATA

![Figure 4.2](image)  The PalmtopInk example running under the CE device emulator.
Listing 4.2  The main source file for the PalmtopInk example. (continues)

// PalmtopInk.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "PalmtopInk.h"
#include <commctrl.h>
#include <commdlg.h>
#include <Inkx.h>
#include <RichInk.h>
HINSTANCE hInst = NULL; // Local copy of hInstance
HWND hwndMain = NULL; // Handle to Inking window
WNDPROC fnOldInkControl;
int iDataLen = 0;
BYTE lpbInkData = NULL;
//InitInstance
BOOL InitInstance (HINSTANCE hInstance, int CmdShow )
{
    hInst = hInstance;
    InitCommonControls();
    //initialize the inking facility before creating the control

Figure 4.3  Using the rich ink control to accept handwriting.
InitInkX();
// We create a main window of class WC_INKX
hwndMain = CreateWindow(WC_INKX,
    NULL,
    WS_VISIBLE,
    0, 0,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL, NULL, hInstance, NULL);

if (!hwndMain )
{
    return FALSE;
}
// We subclass the window procedure to implement
// app specific behavior.
fnOldInkControl = (WNDPROC) SetWindowLong (hwndMain,
    GWL_WNDPROC, (DWORD) InkProc);
// Search until we find the handle of the InkControl window menu.
HMENU hMenu = GetInkCtrlMenu(hwndMain);
// Append user defined menu items
HMENU hMenuPop = GetSubMenu (hMenu, 2);
AppendMenu (hMenuPop, MF_SEPARATOR, NULL, NULL);
AppendMenu (hMenuPop, MF_STRING, IDM_INIT_CTRL,
    TEXT("Init Ctrl"));
AppendMenu (hMenuPop, MF_STRING, IDM_GET_DATA, TEXT("Get Data"));
AppendMenu (hMenuPop, MF_STRING, IDM_SET_DATA, TEXT("Set Data"));
ShowWindow(hwndMain, CmdShow);
UpdateWindow(hwndMain);
InvalidateRect(NULL, NULL, TRUE);
return TRUE;

// WinMain
int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPWSTR     lpCmdLine,
    int        CmdShow)
{
    MSG msg;
    HWND hInkWnd = NULL;

    // if InkControl.exe is running set focus on the window
    hInkWnd = FindWindow(WC_INKX, NULL);
    if (hInkWnd)
    {
        SetForegroundWindow (hInkWnd);
        return 0;
    }
    if ( !InitInstance( hInstance, CmdShow ) )
    {
Listing 4.2  The main source file for the PalmtopInk example.
return (FALSE);
}
while ( GetMessage( &msg, NULL, 0,0 ) == TRUE )
{
    TranslateMessage (&msg);
    DispatchMessage(&msg);
}
return (msg.wParam);

// InkProc

LRESULT CALLBACK InkProc(HWND hwnd,
    UINT msg,
    WPARAM wParam,
    LPARAM lParam)
{
    switch(msg)
    {
    case WM_COMMAND:
        switch (GET_WM_COMMAND_ID(wParam, lParam))
        {
        case IDM_INIT_CTRL:
            SendMessage(hwnd, IM_REINIT, 0, 0);
            iDataLen = 0;
            if( lpbInkData != NULL)
            {
                LocalFree( lpbInkData );
                lpbInkData = NULL;
            }
            break;
        case IDM_GET_DATA:
            iDataLen = SendMessage(hwnd, IM_GETDATALEN , 0, 0);
            if( iDataLen )
            {
                lpbInkData = (LPBYTE)LocalAlloc( LPTR, iDataLen );
                if(!lpbInkData )
                { goto FAIL; }
                SendMessage(hwnd, IM_GETDATA , iDataLen,
                    (LPARAM)lpbInkData);
                SendMessage(hwnd, IM_CLEARALL, 0, 0);
            }
            break;
        case IDM_SET_DATA:
            if(iDataLen != 0 && lpbInkData != NULL )
            {
                SendMessage(hwnd, IM_SETDATA ,
                    iDataLen, (LPARAM)lpbInkData);
                iDataLen = 0;
                LocalFree( lpbInkData );
                lpbInkData = NULL;

Listing 4.2  The main source file for the PalmtopInk example. (continues)
Listing 4.2  The main source file for the PalmtopInk example.
break;
) while (hwndCurrent = GetWindow(hwndCurrent, GW_HWNDEXIELD));

if(!hwndCurrent)
{
    MessageBox(hwndInk, TEXT("Error"),
                TEXT("No CBFRAME handle"), MB_OK);
    return NULL;
}
hwndCurrent = GetWindow(hwndCurrent, GW_CHILD);
do
{
    if(MatchClass(hwndCur, TEXT("ReBarWindow32")))
        break;
) while (hwndCurrent = GetWindow(hwndCurrent, GW_HWNDEXIELD));
if(!hwndCurrent)
{
    MessageBox(hwndInk, TEXT("Error"),
                TEXT("No ReBarWindow32 handle"),
                MB_OK);
    return NULL;
}

hwndCurrent = GetWindow(hwndCurrent, GW_CHILD);
hwndCurrentChild = NULL;
do
{
    if(MatchClass(hwndCur, TEXT("ToolbarWindow32")))
    {
        hwndCurrentChild = GetWindow(hwndCurrent, GW_CHILD);
        if( MatchClass( hWndCurrentChild , TEXT("Menu")) )
            break;
        else
            hwndCurrentChild = NULL;
    }
} while (hwndCurrent = GetWindow(hwndCur, GW_HWNDEXIELD));
if(! hwndCurrentChild )
{
    MessageBox(hwndInk, TEXT("Error"), TEXT("No Menu handle"),
                MB_OK);
    return NULL;
}
return hwndCur;
}
// GetInkCtrlMenu
HMENU GetInkCtrlMenu(HWND hwndInk)
{
    HWND hwndCB = GetInkCtrlCommandBar(hwndInk);
    if (!hwndCB)
First Steps with the RichInk Control

When you add rich ink control support to your project, make sure that you remember to add the inkx.lib to the link specifications in your project. Also, you’ll need the inking header files:

```c
#include <Inkx.h>
#include <RichInk.h>
```

You must initialize the inking facility after the common controls are loaded with a call to InitInkX() and before you attempt to create an ink control. We do this initialization in InitInstance():

```c
hInst = hInstance;
InitCommonControls();
```

//initialize the inking facility before creating the control
InitInkX();

Now we’ll create an ink control by passing a special window class constant to CreateWindow(). Notice that this is the only window we create in InitInstance(), which means that it is the main frame window for our application. The window procedure for the class WC_INKX (the window class that implements the rich ink control) exists outside the scope of our application code. We must substitute a window procedure of our own, or we won’t be able to create
application-specific behaviors for the rich ink control. We do this by subclassing the rich ink control’s window procedure.

```c
// We create a main window of class WC_INKX
hwndMain = CreateWindow(WC_INKX,
    NULL,
    WS_VISIBLE,
    0,0,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL, NULL, hInstance, NULL );

if ( !hwndMain )
{
    return FALSE;
}
```

**WINDOW SUBCLASSING**

If you examine the parameters to CreateWindow(), you’ll notice that the first parameter identifies the window’s class. This parameter gives the system a way to connect a specific, individual window to a stored data structure that defines a particular window’s critical attributes. In many of the examples in this book, we supplied this data by initializing a WNDCLASS structure, and calling RegisterClass(). Here’s the typedef for the WNDCLASS structure:

```c
typedef struct _WNDCLASS {
    UINT    style;
    WNDPROC lpfnWndProc;
    int     cbClsExtra;
    int     cbWndExtra;
    HANDLE  hInstance;
    HICON   hIcon;
    HCURSOR hCursor;
    HBRUSH  hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS;
```

Notice the second member, lpfnWndProc. This member specifies the address of the main window procedure for this window class. The Windows operating system passes messages and notifications for a specific, individual window to the procedure at this address. Long ago, a clever person discovered that if you wanted to change the behavior of a window in small ways, you could replace this address (the original window procedure for the class) with a different one. This strategy is called “subclassing a window procedure.”

(continues)
We have to subclass the ink control’s window procedure to implement application-specific behavior, so we set the new window procedure, InkProc(), with a call to SetWindowLong():

```cpp
fnOldInkControl = (WNDPROC) SetWindowLong(hwndMain,
GWL_WNDPROC,
(DWORD) InkProc);
```

SetWindowLong() sets a new window procedure address and returns the old one, which we save in a global variable. Its parameters, in the order shown, are the handle to the window being subclassed, the constant GWL_WNDPROC, which indicates we are replacing the address of the window procedure, and the address of the new window procedure.

Windows OS stores system-related and optionally programmer-specific data in the window itself. This data is accessible via APIs, such as SetWindowLong and SetWindowWord. In C++, the compiler interprets function names as addresses, so this last parameter is simply the name of the new window procedures, without a parameter list.

The new procedure then receives first crack at message processing, allowing it to handle those of particular interest. It then passes the rest of the messages to the original window procedure, using the address returned by SetWindowLong(). We pass the unprocessed messages on to the original window procedure by calling CallWindowProc():

```cpp
CallWindowProc(fnAddrOldWndProc, hwnd, msg, wParam, lParam);
```

The parameters, in the order shown, are the address of the original window procedure, followed by the unmodified parameters passed in the system’s call to the window procedure.

Now we need to update the ink control’s menu with some application-specific menu items. First, we call GetInkCtrlMenu():

```cpp
//Search until we find the handle of the InkControl window menu.
HMENU   hMenu = GetInkCtrlMenu(hwndMain);
```
GetInkCtrlMenu() needs a handle to the command bar that contains the menu, so we call GetInkCtrlCommandBar():

```c
HWND hwndCB = GetInkCtrlCommandBar(hwndInk);
```

Basically, now we have to sort through all of the ink control’s child windows and find one with the class “CBFRAME”. We use the helper function MatchClass() to retrieve the window class name strings for purposes of our comparison.

```c
// Find the Menu handle in the Ink control
hwndCurrrent = GetWindow(hwndInk,GW_CHILD);
do {
    if(MatchClass(hwndCurrrent, TEXT("CBFRAME")))
        break;
} while (hwndCurrrent = GetWindow(hwndCurrrent, GW_HWNDNEXT));
```

If we are successful in finding the menu bar and its handle, things get rational again. We append the application-specific menu items, one for each ink control message we’ll be handling in our subclass procedure.

```c
// Append user defined menu items
HMENU hMenuPop = GetSubMenu (hMenu, 2);
AppendMenu (hMenuPop, MF_SEPARATOR, NULL, NULL);
AppendMenu (hMenuPop, MF_STRING, IDM_INIT_CTRL, TEXT("Init Ctrl"));
AppendMenu (hMenuPop, MF_STRING, IDM_GET_DATA, TEXT("Get Data"));
AppendMenu (hMenuPop, MF_STRING, IDM_SET_DATA, TEXT("Set Data"));
```

Now it’s mostly a matter of sending inking messages to the control in response to user menu choices:

```c
case WM_COMMAND:
    switch (GET_WM_COMMAND_ID(wParam, lParam))
    {
        case IDM_INIT_CTRL:
            SendMessage(hwnd, IM_REINIT, 0, 0);
            iDataLen = 0;
            if( lpbInkData != NULL)
            {
                LocalFree( lpbInkData );
                lpbInkData = NULL;
            }
            break;
```
Notice the use of the macro `GET_WM_COMMAND_ID(wParam, lParam)` in the switch statement. This is one of a group of macros defined in Windows.h, which provide a self-documenting way to extract switch values from a window message parameter list. Here are their definitions:

```c
#define GET_WM_COMMAND_ID(wp, lp)               LOWORD(wp)
#define GET_WM_COMMAND_HWND(wp, lp)             (HWND)(lp)
#define GET_WM_COMMAND_CMD(wp, lp)              HIWORD(wp)
```

In response to the “Init Ctrl” menu choice, we send the ink control window the IM_REINIT message. This message clears and repaints the control. If the user previously saved data, we free the memory it occupied and set the data length to zero, reinitializing the globals `iDataLen` and `lpbInkData`.

Retrieving inking data is a three-step process. We have to get the length of the ink data, allocate sufficient memory to hold it, and finally, message the control to get the data.

```c
case IDM_GET_DATA:
    iDataLen = SendMessage(hwnd, IM_GETDATALEN, 0, 0);
    if( iDataLen )
    {
        lpbInkData = (LPBYTE)LocalAlloc( LPTR, iDataLen );
        if(!lpbInkData )
        { goto FAIL; }
        SendMessage(hwnd, IM_GETDATA, iDataLen,
                    (LPARAM)lpbInkData);
        SendMessage(hwnd, IM_CLEARALL, 0, 0);
    }
    break;
```

In order to get the length of the data, we send `IM_GETDATALEN` message to the ink control window. The last two `SendMessage()` parameters are unused for this message, and should be set to zero.

If our allocation request is successful, we call `SendMessage()` again. This time the `SendMessage()` parameters are the ink control window’s handle, the message constant `IM_GETDATA`, the length in bytes of the inking data, and the address of the allocated buffer, cast to the type `LPARAM`. We follow this operation by sending an `IM_CLEARALL` message to the ink control. This is exactly equivalent to sending the `IM_REINIT` message we used earlier and is provided only for the sake of example. (We clear the control so that when we reset the data into the control there is a visible effect.)

Finally, we restore the saved ink data to the ink control window.

```c
case IDM_SET_DATA:
    if(iDataLen != 0 && lpbInkData != NULL )
    {
```
We do this by sending the IM_SETDATA message. The SendMessage() parameters, in the order shown, are the handle to the ink control window; the message constant IM_SETDATA; the length of the data in bytes, as reported by the control when we saved the data; and the address of the buffer containing the saved inking data, cast to the type LPARAM.

Take-Home Lessons

Inking is clearly going to be very important to applications targeting the smallest CE devices, possibly less so for the handheld and HPC Pro classes of devices. This imposes a considered choice on developers looking for a fast port but also one faithful to the spirit of the target platform.

As of this writing, aggressive support for inking probably isn’t a very good plan if you are interested in getting a Win32 code base on a broad variety of CE devices quickly. However, if the palmtop form factor is an important target for your application, you should probably provide users inking support if they have to input much text. If you plan to support a specific device or subset of devices exclusively, you may be able to take advantage of the advanced features in CE 3.0 and beyond which include handwriting recognition APIs.

Looking Ahead

So far we’ve been exploring the issues around moving the visual components of your application from Win32 to Windows CE. The final piece of the visual presentation puzzle has to do with your application’s use of and interaction with the Windows CE taskbar. In the next chapter, “The Windows CE Shell,” we’ll look at the key tools for this job, the Shell APIs.

The differences between shell programming for the various Windows CE devices are a significant porting issue. We’ll look at how to use conditional compilation directives to develop a single code base for CE applications. We’ll also introduce the differences in the nature of the Windows CE HPC and Windows CE PPC exposures of their file systems. We’ll show how to use shell programming APIs to find and open files in the PPC environment.
Unlike desktop Windows, Windows CE does not have a standard user interface. Rather, the hardware vendor determines which user interface components are present, and the various CE devices differ dramatically in this respect. Collectively, the user interface elements are called the “shell.” While the H/PC Pro and the HPC are highly congruent with the interface of desktop Windows, the palmtop devices share little in the way of “look-and-feel” with their larger progenitors. In terms of deciding what parts of an existing Win32 application to port, the differing nature of the CE shells is the first critical evaluation you will face. Shell programming is one of those contexts in which it’s worth weighing whether a thing that can be done should be done.

Some of the example code in this chapter is portable across most CE platforms but makes the most sense on a specific class of device. Working from the premise that the ultimate aim of application programming is to provide users with the best tools, we do want to take advantage of the strengths of various CE platforms. If there is one bit of advice I’d offer in this context, it’s this: If you have to make a choice between maintaining a single, portable code base and allowing an application to evolve in ways to amplify the unique features of a given platform, choose evolution.
The Job of the CE Shell

As far as a user can tell, the CE Shell performs two basic functions: It provides a way to launch programs, and it exposes the CE file system. From the user’s point of view, these two tasks are handled very differently by the smaller CE devices (the palmtops) than they are by the larger ones (HPC and HPC Pro) or by desktop Windows. The small size of the palmtop devices imposes a new, abbreviated set of user interface controls. In addition, the palmtop devices don’t include a “Windows Explorer” type construct.

Put another way, users can’t browse the palmtop CE file system by navigating through folders, opening files, or launching programs along the way. User file access relies exclusively on a very lightweight version of the File Open dialog box, which is displayed by applications that give users the ability to operate on specific files. The palmtop CE platforms make the Start menu the sole tool for launching programs. For the smallest devices, this restricted access to the file system reinforces the palmtop CE bias against file and text oriented applications.

PORTING TIP  As you consider porting to the CE Shell style of user interface, ask a few probing questions, with an eye to getting version 1.0 out quickly:

- Could you target the H/PC Pro or H/PC exclusively?
- If you target the palmtop devices, could you present the application’s functionality in a single dialog box?
- Can you minimize or eliminate the need for the user to locate specific files?

In general, you can assume that Win32 shell behaviors will port readily to the H/PC Pro family of devices, and with a few more modifications to the H/PC class of devices. In the palmtop class of targets, you will need to be much more discerning about which shell-related features you choose to port to CE.

Where the Files Are

CE “flattens” the standard Windows file system, and this flattening occurs in different degrees on different platforms. The practical impact of this is that the “special folders” (for example, the Recycling Bin) that your application references may be in different locations than they were on the desktop. The path to the special folders also differs among CE platforms. Our first example program, FindDirs, shows how to use the shell functions to find the paths for the special folders. This job is accomplished differently on the palmtop devices than on the others. FindDirs.cpp is the main source file for an application that
retrieves special folder locations on the palmtop CE devices. Following this example and its accompanying explanation, I’ll show how to modify the FindDirs code so that it will work properly on HPC and HPC Pro class devices.

As with earlier examples, the support files are omitted in the interest of brevity (see Figure 5.1). Full source for the FindDirs project (see Listing 5.1) can be found on the accompanying CD.

Figure 5.1 The FindDirs example running on the Pocket PC emulator.

// FindDirs.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "FindDirs.h"
#include <windows.h> // For all that Windows stuff
#include <windowsx.h> // For all that other Windows stuff
#include <winerror.h> // For all that Woops stuff
#include <commctrl.h>
#include <commdlg.h> // Command bar includes
#include <shlobj.h>
#include <objidl.h>
#define MAX_LOADSTRING 100
#define dim(x) (sizeof(x) / sizeof(x[0]))
typedef struct tagSDirs
{
  int csidl;
  LPTSTR pszCSIDL;
} SPECIAL_DIRS;

Listing 5.1 FindDirs.cpp, the main source file for the FindDirs example program. (continues)
SPECIAL_DIRS iaSpecialDirs[] =
{
    CSIDL_DESKTOP , TEXT("CSIDL_DESKTOP"),
    CSIDL_PROGRAMS , TEXT("CSIDL_PROGRAMS"),
    CSIDL_PERSONAL , TEXT("CSIDL_PERSONAL"),
    CSIDL_STARTUP , TEXT("CSIDL_STARTUP"),
    CSIDL_RECENT , TEXT("CSIDL_RECENT"),
    CSIDL_BITBUCKET , TEXT("CSIDL_BITBUCKET"),
    CSIDL_DRIVES , TEXT("CSIDL_DRIVES"),
    CSIDL_FONTS , TEXT("CSIDL_FONTS"),
    CSIDL_FAVORITES , TEXT("CSIDL_FAVORITES")
};

Listing 5.1 FindDirs.cpp, the main source file for the FindDirs example program.
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_FINDDIRS));
    wc.hCursor = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = 0;
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    TCHAR szTitle[MAX_LOADSTRING]; // The title bar text
    TCHAR szWindowClass[MAX_LOADSTRING]; // The window class name
    hinst = hInstance; // Store instance handle in our global
    // variable
    // Initialize global strings

Listing 5.1  FindDirs.cpp, the main source file for the FindDirs example program. (continues)
LoadString(hInstance, IDC_FINDDIRS, szWindowClass, MAX_LOADSTRING);
MyRegisterClass(hInstance, szWindowClass);
LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);

hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
    CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, CW_USEDEFAULT,
    NULL, NULL,
    hInstance, NULL);

if (!hWnd)
    return FALSE;

ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);

DialogBox(hInst, TEXT("SpecialDirs"), hWnd, SpecialDirsDlgProc);
return TRUE;

//
//  FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
//  PURPOSE:  Processes messages for the main window.
//  WM_COMMAND    - process the application menu
//  WM_PAINT    - Paint the main window
//  WM_DESTROY    - post a quit message and return
//

LRESULT CALLBACK WndProc(HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    int wmId, wmEvent;
    switch (message)
    {
    case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_HELP_ABOUT:
            DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX,
                hWnd, (DLGPROC)About);
            break;  
        
Listing 5.1  FindDirs.cpp, the main source file for the FindDirs example program.
break;
case IDM_FILE_EXIT:
    DestroyWindow(hWnd);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
break;
case WM_CREATE:
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    CommandBar_AddAdornments(hwndCB, 0, 0);
    break;
case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
}

// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    RECT rt, rt1;
    int DlgWidth, DlgHeight; // dialog width and height in pixel units
    int NewPosX, NewPosY;
    switch (message)
    {
    case WM_INITDIALOG:
        // trying to center the About dialog
        if (GetWindowRect(hDlg, &rt1)) {
            GetClientRect(GetParent(hDlg), &rt);
            DlgWidth   = rt1.right - rt1.left;
            DlgHeight  = rt1.bottom - rt1.top ;
            NewPosX    = (rt.right - rt.left - DlgWidth)/2;
            NewPosY    = (rt.bottom - rt.top - DlgHeight)/2;

            // if the About box is larger than the physical screen
            if (NewPosX < 0) NewPosX = 0;
            if (NewPosY < 0) NewPosY = 0;
            SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                         0, 0, SWP_NOZORDER | SWP_NOSIZE);
        }

Listing 5.1  FindDirs.cpp, the main source file for the FindDirs example program. (continues)
return TRUE;
case WM_COMMAND:
    if ((LOWORD(wParam) == IDOK) ||
(LOWORD(wParam) == IDCANCEL))
    {
        EndDialog(hDlg, LOWORD(wParam));
        return TRUE;
    }
    break;
}
return FALSE;
}

// Find the paths to "Special Folders"
BOOL CALLBACK SpecialDirsDlgProc(HWND hWnd,
    UINT wMsg,
    WPARAM wParam,
    LPARAM lParam)
{
    BOOL bOk;
    HWND hWndList;
    INT i = 0, rc;
    LPITEMIDLIST pidl = NULL;
    TCHAR szBuff[MAX_PATH];

    switch (wMsg)
    {
    case WM_INITDIALOG:
        hWndList = GetDlgItem(hWnd,IDC_SPECIAL_DIRS );
        _stprintf( szBuff, TEXT("%s"),TEXT("Special Folders"));
        ListBox_InsertString(hWndList, -1, &szBuff);
        //   if( rc != NOERROR )
        //   { return TRUE; }
        for ( i = 0; i < dim ( iaSpecialDirs ); i ++ )
        {
            rc = SHGetSpecialFolderLocation(hWnd,
                iaSpecialDirs[i].csidl, &pidl );
            if( rc == NOERROR )
            {
                _stprintf( szBuff, TEXT(" %s "),iaSpecialDirs[i].pszCSIDL);
            }
        }
    
Listing 5.1  FindDirs.cpp, the main source file for the FindDirs example program.
Examining the FindDirs Example

The rubber meets the road, so to speak, in the initialization sequence of the SpecialDirsDlgProc(). First, we get the handle to the list box and insert a title string:

```c
TCHAR szBuff[MAX_PATH];

switch (wMsg)
{
    case WM_INITDIALOG:
        hWndList = GetDlgItem(hWnd, IDC_SPECIAL_DIRS);
        _stprintf(szBuff, TEXT("%s"), TEXT("Special Folders"));
        ListBox_InsertString(hWndList, -1, &szBuff);

        break;
    }
```

Next, we loop to fill the list box with the paths to specific special directories. Notice that we use an array of structures, iaSpecialDirs. Here is the declaration of the structures that make up this arra::

```c
Listing 5.1  FindDirs.cpp, the main source file for the FindDirs example program. (continued)
```
The typedef for the SPECIAL_DIRS structure includes two members. The first, csidl, contains a CE-defined constant that identifies a special folder. The second, pszCSIDL, is a literal string that contains the CE constant’s defined name. The possible values for the csidl values and their meanings are given in Table 5.1. The physical directories identified by the constants may or may not be present on a given device, so it’s very important to test the return from SHGetSpecialFolderLocation().

**Table 5.1** SHGetSpecialFolderLocation() Constants for Special Folders

<table>
<thead>
<tr>
<th>MANIFEST CONSTANT IDENTIFIER</th>
<th>HOW THIS FOLDER IS USED</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSIDL_BITBUCKET</td>
<td>File system directory for the user’s Recycle Bin. The Recycle Bin path is not in the registry, and it is marked with the hidden and system attributes to keep users from moving or deleting it.</td>
</tr>
<tr>
<td>CSIDL_CONTROLS</td>
<td>This is a virtual folder. It contains Control Panel application icons.</td>
</tr>
<tr>
<td>CSIDL_DESKTOP</td>
<td>This is a virtual folder at the root of the namespace. It contains Windows Desktop shortcuts.</td>
</tr>
<tr>
<td>CSIDL_DESKTOPDIRECTORY</td>
<td>This is the actual physical directory used to store desktop file objects. It is not the same as the Windows Desktop folder.</td>
</tr>
<tr>
<td>CSIDL_DRIVES</td>
<td>This is a virtual folder that encapsulates the contents of the desktop “My Computer” icon. It shows everything on the local computer: storage devices, printers, and Control Panel. It may also contain mapped network drives.</td>
</tr>
<tr>
<td>CSIDL_FAVORITES</td>
<td>Physical directory that contains the user’s “Favorites” items.</td>
</tr>
<tr>
<td>CSIDL_FONTS</td>
<td>This is a virtual folder. It contains fonts.</td>
</tr>
<tr>
<td>CSIDL_NETHOOD</td>
<td>Physical directory that contains the network neighborhood objects.</td>
</tr>
<tr>
<td>CSIDL_NETWORK</td>
<td>This is a virtual folder. It represents the top level of the network hierarchy.</td>
</tr>
<tr>
<td>MANIFEST CONSTANT</td>
<td>IDENTIFIER</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------</td>
</tr>
<tr>
<td>CSIDL_PERSONAL</td>
<td></td>
</tr>
<tr>
<td>CSIDL_PRINTERS</td>
<td></td>
</tr>
<tr>
<td>CSIDL_PROGRAMS</td>
<td></td>
</tr>
<tr>
<td>CSIDL_RECENT</td>
<td></td>
</tr>
<tr>
<td>CSIDL_SENDTO</td>
<td></td>
</tr>
<tr>
<td>CSIDL_STARTUP</td>
<td></td>
</tr>
<tr>
<td>CSIDL_STARTMENU</td>
<td></td>
</tr>
<tr>
<td>CSIDL_TEMPLATES</td>
<td></td>
</tr>
</tbody>
</table>

We test for a specific subset of the constants in this version of FindDirs. Here’s the declaration and initialization of our array of SPECIAL_DIR structures, iaSpecialDirs[]:

```c
SPECIAL_DIRS iaSpecialDirs[] =
{
    CSIDL_DESKTOP               ,TEXT("CSIDL_DESKTOP") ,
    CSIDL_PROGRAMS           ,TEXT("CSIDL_PROGRAMS") ,
    CSIDL_PERSONAL            ,TEXT("CSIDL_PERSONAL") ,
    CSIDL_STARTUP               ,TEXT("CSIDL_STARTUP") ,
    CSIDL_RECENT                 ,TEXT("CSIDL_RECENT") ,
    CSIDL_BITBUCKET          ,TEXT("CSIDL_BITBUCKET") ,
    CSIDL_DRIVES                    ,TEXT("CSIDL_DRIVES") ,
    CSIDL_FONTS                      ,TEXT("CSIDL_FONTS") ,
    CSIDL_FAVORITES              ,TEXT("CSIDL_FAVORITES")
};
```

This subset was chosen because it is likely to be implemented on a palmtop class device. Now we loop through the array, calling SHGetSpecialFolderLocation() for each array element:

```c
for ( i = 0; i < dim ( iaSpecialDirs ); i ++ )
{
```
rc = SHGetSpecialFolderLocation(hWnd, 
    iaSpecialDirs[i].csidl, &pidl);

The parameters to SHGetSpecialFolderLocation(), in the order shown, are 
the handle to the window that will display any message boxes or warnings 
required during the processing of this call, the CE-defined constant that iden-
tifies the folder for which we want a path, and the address of a pointer to an 
ITEMIDLIST structure. Here’s how this variable, pidl, is declared at the top of 
the SpecialDirsDlgProc():

    LPITEMIDLIST pidl = NULL;

This fragment of code is a bit deceptive and bears a little explanation. On the 
desktop, and on larger CE platforms, pidl is a pointer to a buffer that is allocated 
by the shell. In order to free this buffer, you need to get and use the shell’s IMal-
loc COM interface. This isn’t necessary on the palmtop CE devices, because the 
value returned in pidl is a defined constant that you can pass to SHGetPath-
FromIDList() to retrieve a pathname. If the call to SHGetSpecialFolderLocation() 
is successful, we retrieve the pathname:

    if( rc == NOERROR )
    {
        _stprintf( szBuff, TEXT("%s
"), iaSpecialDirs[i].pszCSIDL);
        ListBox_InsertString(hWndList, -1, &szBuff);
        SHGetPathFromIDList( pidl, &szBuff[0] );
        ListBox_InsertString(hWndList, -1, &szBuff);
    }

Just one more thing. Note the declaration of szBuff:

    TCHAR szBuff[MAX_PATH];

Don’t be tempted to save stack space by reducing the size of this buffer. In 
some cases, the SHGetPathFromIDList() function fills the buffer with null 
padding, using the entire MAX_PATH space, even though the actual path-
name is only a few characters long.

Finding Special Folder Paths on the HPC 
and HPC Pro

Getting the paths for the special folders involves a couple of extra steps on the 
HPC and HPC Pro. These devices fully implement the “pidl” as a pointer to a 
buffer that is allocated by the shell. This buffer must be released when you are 
done using it, and in order to do so you must obtain a pointer to the shell’s
IMalloc COM interface. Here’s how the WM_INITDIALOG case looks using the IMalloc interface pointer:

```c
LPMALLOC lpMallocInterface;
case WM_INITDIALOG:
    hWndList = GetDlgItem(hWnd,IDC_SPECIAL_DIRS);
    _stprintf( szBuff, TEXT("%s"),
              TEXT("Special Folders"));
    ListBox_InsertString(hWndList, -1, &szBuff);
    //Init the IMalloc interface pointer
    lpMallocInterface = NULL;
    //Get the Shell's IMalloc Interface
    rc = SHGetMalloc( & lpMallocInterface );
    if( rc != NOERROR )
        { return TRUE; }  
    for ( i = 0; i < dim( iaSpecialDirs ); i ++ )
    {
        rc = SHGetSpecialFolderLocation(hWnd,
                                         iaSpecialDirs[i].csidl,
                                         &pidl);
        if( rc == NOERROR )
        {
            _stprintf( szBuff,
                       TEXT("   %s "),iaSpecialDirs[i].pszCSIDL);
            ListBox_InsertString(hWndList, -1,
                                  &szBuff);
            SHGetPathFromIDList( pidl, &szBuff[0] );
            ListBox_InsertString(hWndList, -1,
                                  &szBuff);
        }// end if()
    } //end for()
    //Use the interface pointer to
    //free the buffer allocation
    lpMallocInterface->Free( pidl );
    //end if()
    //end for()
    //Now remove our reference to the shell's
    //IMalloc Interface
    lpMallocInterface->Release();
    break;
```

Let’s have a recap of differences between the two implementations. First, we initialize the variable that holds the pointer to the shell’s IMalloc interface:

```c
//Init the IMalloc interface pointer
lpMallocInterface = NULL;
```

Next, we call SHGetMalloc(), passing the address of this pointer, as the only parameter. We check the return to make sure we got the interface pointer, and if not we exit the function:
//Get the Shell's IMalloc Interface
rc = SHGetMalloc( & lpMallocInterface );
if( rc != NOERROR )
{ return TRUE; }

Now when we call SHGetSpecialFolderLocation(), the address of a buffer
allocated by the shell is returned in the variable pidl:

    rc = SHGetSpecialFolderLocation(hWnd,
        iaSpecialDirs[i].csidl, &pidl );

We use this returned address to look up the actual pathname when we call
SHGetPathFromIDList():

    SHGetPathFromIDList( pidl, &szBuff[0] );

We free the shell's buffer at the bottom of the loop body:

    //Use the interface pointer to free
    //the buffer allocation
    lpMallocInterface->Free( pidl );

After we've looped the entire array of special folder constants, we release
the IMalloc interface:

    //Now remove our reference to the shell's IMalloc Interface
    lpMallocInterface->Release();

The Shell’s User Interface Elements

One of the key tasks of adapting Win32 code to the CE shell is learning how to
make applications and documents visible to the user. In the SysMenuGadgets
example (see Figure 5.2) you'll see how to add shortcuts to the HPC and HPC
Pro desktop, how to add programs and documents to the Start menu, and how
to place an application icon in the taskbar (see Listing 5.2).

Figure 5.2  The SysMenuGadgets example running on an HPC 2000 platform.
// SysMenuGadgets.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "SysMenuGadgets.h"
#include <commctrl.h>
#include <shellapi.h>
#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst; // The current instance
HWND hwndCB; // The command bar handle

// Forward declarations of functions included in this code module:
ATOM MyRegisterClass (HINSTANCE, LPTSTR);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK SysMenuGadgetsDlgProc(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPTSTR lpCmdLine,
                   int nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }
    hAccelTable = LoadAccelerators(hInstance,
                                (LPCTSTR)IDC_SYSMENUGADGETS);
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return msg.wParam;
}

Listing 5.2 SysMenuGadget.cpp, the main source file for the SysMenuGadget example. (continues)
// It is important to call this function so that the application
// will get 'well formed' small icons associated with it.

ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(hInstance,
                        MAKEINTRESOURCE(IDI_SYSMENUGADGETS));
    wc.hCursor = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = 0;
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);
}

// FUNCTION: InitInstance(HANDLE, int)

// PURPOSE: Saves instance handle and creates main window

// COMMENTS:

// In this function, we save the instance handle
// in a global variable and
// create and display the main program window.

// BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    TCHAR szTitle[MAX_LOADSTRING]; // The title bar text
    TCHAR szWindowClass[MAX_LOADSTRING]; // The window class name
    hInst = hInstance;
    // Store instance handle in our global variable
    // Initialize global strings
    LoadString(hInstance, IDC_SYSMENUGADGETS,
               szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
                         CW_USEDEFAULT,
                         CW_USEDEFAULT,
                         CW_USEDEFAULT,
                         NULL, NULL,
                         hInstance, NULL);

Listing 5.2  SysMenuGadget.cpp, the main source file for the
SysMenuGadget example.
if (!hWnd)
{
    return FALSE;
}
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);

DialogBox(hInst, TEXT("SYSMENUGADGETS"),
         hWnd, SysMenuGadgetsDlgProc );
return TRUE;
}

//
//  FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
//  PURPOSE: Processes messages for the main window.
//  WM_COMMAND    - process the application menu
//  WM_PAINT    - Paint the main window
//  WM_DESTROY    - post a quit message and return
//
// LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    switch (message)
    {
        case WM_COMMAND:
            wmId = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
            {
                case IDM_HELP_ABOUT:
                    dialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX,
                        hWnd, (DLGPROC)About);
                    break;
                case IDM_FILE_EXIT:
                    DestroyWindow(hWnd);
                    break;
                default:
                    return DefWindowProc(hWnd, message,
                        wParam, lParam);
            }
            break;
        case WM_CREATE:
            hwndCB = CommandBar_Create(hInst, hWnd, 1);
            break;
    }
}
Listing 5.2  SysMenuGadget.cpp, the main source file for the SysMenuGadget example.
return FALSE;
}

// Find which SysMenuGadgets to add

BOOL CALLBACK SysMenuGadgetsDlgProc(HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam)
{
    HWND hWndCheckbox;
    BOOL bChecked;
    NOTIFYICONDATA nidNotifyIcon;

    switch (wMsg)
    {
    case WM_COMMAND:
        switch (LOWORD (wParam))
        {
        case IDOK:
            hWndCheckbox = GetDlgItem(hWnd,IDC_ADD_START);
            bChecked = SendMessage( hWndCheckbox, BM_GETCHECK,0,0 );
            if( bChecked )
            {
                if ( ! SHCreateShortcut(
                    TEXT("\Windows\Programs\HelloWorld.lnk"),
                    TEXT("\Windows\HelloWorld.exe")))
                {
                    MessageBox( hWnd,
                    TEXT("Couldn't Create Shortcut"),
                    TEXT("ERROR!") , MB_OK);
                    return FALSE;
                }
            }
            else  
            { 
                DeleteFile(TEXT("\Windows\Programs\HelloWorld.lnk"));
            }
            hWndCheckbox = GetDlgItem(hWnd,IDC_ADD_SHORTCUT);
            bChecked = SendMessage( hWndCheckbox, BM_GETCHECK,0,0 );
            if( bChecked )
            {
                if ( ! SHCreateShortcut(
                    TEXT("\Windows\Desktop\HelloWorld.lnk"),
                    TEXT("\Windows\Programs\HelloWorld.exe")))
                {
                    MessageBox( hWnd,
                    TEXT("Couldn't Create Shortcut"),
                    TEXT("ERROR!") , MB_OK);
                }
            }
    
Listing 5.2 SysMenuGadget.cpp, the main source file for the SysMenuGadget example. (continues)
return FALSE;
}
}
else
{
    DeleteFile(TEXT("\Windows\Desktop\HelloWorld.lnk") );
}
hWndCheckbox = GetDlgItem(hWnd,IDC_ADD_TBICON );
bChecked = SendMessage( hWndCheckbox, BM_GETCHECK,0, 0 );

memset( &nidNotifyIcon, 0x0, sizeof NOTIFYICONDATA );
nidNotifyIcon.cbSize = sizeof NOTIFYICONDATA ;
nidNotifyIcon.hWnd = GetParent( hWnd );
nidNotifyIcon.uFlags = NIF_ICON;
nidNotifyIcon.hIcon =
    (HICON) LoadImage(hInst,
        MAKEINTRESOURCE(IDI_SYSMENUGADGETS),
        IMAGE_ICON, 16, 16, 0);
nidNotifyIcon.uID = 10;
if( bChecked )
{
    if ( ! Shell_NotifyIcon( NIM_ADD, &nidNotifyIcon ))
    {
        MessageBox( hWnd,
            TEXT("Couldn't Add Toolbar Icon"),
            TEXT("ERROR!"), MB_OK );
        return FALSE;
    }
}
else
{
    Shell_NotifyIcon( NIM_DELETE, &nidNotifyIcon );
}
hWndCheckbox = GetDlgItem(hWnd,IDC_ADD_RECENT );
bChecked = FALSE;
bChecked = SendMessage( hWndCheckbox, BM_GETCHECK,0,0 );
if( bChecked )
{
    SHAddToRecentDocs( SHARD_PATH,
        TEXT("\MyDocuments\Hello.txt"));
    bRecentDocAdded = TRUE;
}
else
{
    SHAddToRecentDocs( SHARD_PATH, 0 );
}
break;

Listing 5.2  SysMenuGadget.cpp, the main source file for the SysMenuGadget example.
return TRUE;
default:
    break;
}break;
default:
    break;
}//end switch (wMsg)
return FALSE;


Listing 5.2 SysMenuGadget.cpp, the main source file for the SysMenuGadget example. (continued)

Adding Shortcuts to the Start Menu

The SysMenuGadgets example collects information from the SysMenu GadgetsDlgProc() and modifies the shell elements accordingly when the user clicks the Update Shell button. The Update Shell button has the dialog control ID “IDOK”, so it also dismisses the dialog immediately after it updates the shell. The rest of the example application behave in familiar ways, so we’ll restrict our attention to SysMenuGadgetsDlgProc()’s handling of the IDOK command.

First we get the check state of the “Add Start Menu Item” control, and if it is checked, we create a Start Menu program shortcut by calling SHCreateShortcut():

```cpp
hWndCheckbox = GetDlgItem(hWnd, IDC_ADD_START);
bChecked = SendMessage(hWndCheckbox, BM_GETCHECK, 0, 0);
if( bChecked )
{
    if( ! SHCreateShortcut(
        TEXT("\\Windows\\Programs\\HelloWorld.lnk"),
        TEXT("\\Windows\\HelloWorld.exe")))
    {
        MessageBox(hWnd,
            TEXT("Couldn't Create Shortcut"),
            TEXT("ERROR!"), MB_OK);
        return FALSE;
    }
```

The parameters to SHCreateShortcut(), in the order shown, are a null terminated string specifying the fully qualified name of the shortcut and a null terminated string specifying the fully qualified name of the shortcut’s target. By creating the shortcut in the directory “\\Windows\\Programs”, we automatically end up with a start menu shortcut for the program on the HPC and
HPC Pro devices. If you want to have more flexibility in targeting, you can include the logic we used in the FindDirs example to retrieve the fully qualified path for the special folder “CSIDL_STARTMENU.”

Removing a short cut is a straightforward matter, and we do this if the “Add StartMenu Item” control is unchecked:

```c
else
{
    DeleteFile(TEXT("\\Windows\\Desktop\\HelloWorld.lnk") );
}
```

Just delete the link file created by SHCreateShortcut(), using the file’s fully qualified, null terminated pathname.

### Adding and Deleting Shortcuts Without Using the Start Menu

Here’s how you create a shortcut to an application if you don’t want it to show up on the start menu. This shortcut appears as an icon on the desktop, but it could appear in another location just as easily, also represented as an icon:

```c
hWndCheckbox = GetDlgItem(hWnd,IDC_ADD_SHORTCUT );
bChecked = SendMessage( hWndCheckbox, BM_GETCHECK,0,0 );
if( bChecked )
{
    if ( ! SHCreateShortcut(
        TEXT("\\Windows\\Desktop\\HelloWorld.lnk"),
        TEXT("\\Windows\\Programs\\HelloWorld.exe")))
```

Again, you delete this type of shortcut by calling DeleteFile() and passing the fully qualified, null terminated pathname of the link file.

### Adding Application Icons to the Taskbar

By adding an application icon to the HPC or HPC Pro taskbar, you provide the user a means of quickly starting an application any time she wants, since the taskbar is always visible on these platforms. Adding the icon is somewhat more involved than the shell operations you’ve seen so far, as you must initialize a NOTIFYICONDATA structure to do so. Table 5.2 gives the members of the structure and their meanings:
Table 5.2  NOTIFYICONDATA Structure Members and Their Meanings

<table>
<thead>
<tr>
<th>MEMBER NAME</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD cbSize;</td>
<td>Size of the structure in bytes.</td>
</tr>
<tr>
<td>HWND hWnd;</td>
<td>Handle to a window that receives notification messages.</td>
</tr>
<tr>
<td>UINT uID;</td>
<td>Application’s ID for the taskbar icon.</td>
</tr>
</tbody>
</table>
| UINT uFlags;  | Any combination of:  
NIF_ICON   The hIcon member is valid.  
NIF_MESSAGE The uCallbackMessage member is valid.  
NIF_TIP The szTip member is valid. |
| UINT uCallbackMessage; | Application defined message ID. System sends this message to the window specified by the hWnd member, in response to mouse click events over the icon's bounding rectangle. |
| HICON hIcon;  | Handle to the icon to add, delete, or modify.                          |
| WCHAR szTip[64]; | Tool tip string.                                                      |

//declaration of structure
NOTIFYICONDATA  nidNotifyIcon;
memset( &nidNotifyIcon, 0x0, sizeof(NOTIFYICONDATA));

It’s a good practice to initialize any structure we pass as a function parameter, so we do this with memset() before filling in the structure members. Next, we initialize the structure members, setting the NIF_ICON flag to indicate that we are manipulating an icon. Notice that we use LoadImage() rather than LoadIcon() to retrieve the bitmap for the icon. LoadImage() loads a 16 x 16 bit image, which is suitable for display on the toolbar. LoadIcon() retrieves a large icon bitmap, which will cause the next operation to fail:

```
    nidNotifyIcon.cbSize = sizeof(NOTIFYICONDATA);
    nidNotifyIcon.hWnd   = GetParent( hWnd );
    nidNotifyIcon.uFlags = NIF_ICON;
    nidNotifyIcon.hIcon = (HICON) LoadImage(hInst, 
                                MAKEINTRESOURCE(IDI_SYSMENUGADGETS),
                                IMAGE_ICON, 16, 16, 0);
    nidNotifyIcon.uID   = 10;
```
Table 5.3  Shell_NotifyIcon() Flags and Their Meanings

<table>
<thead>
<tr>
<th>FLAG NAME</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIM_ADD</td>
<td>Add the icon</td>
</tr>
<tr>
<td>NIM_DELETE</td>
<td>Delete the icon</td>
</tr>
<tr>
<td>NIM_DELETE</td>
<td>Modify the icon</td>
</tr>
</tbody>
</table>

**PORTING TIP** If you have difficulty getting your application to add icons to the toolbar, double-check to make sure bitmaps are exactly 16 x 16.

To add the icon to the toolbar, call Shell_NotifyIcon():

```c
Shell_NotifyIcon( NIM_ADD, &nidNotifyIcon )
```

The parameters, in the order shown, are a flag specifying the icon operation and the address of the initialized NOTIFYICONDATA structure. You may pass only one flag value in each call to the function. Table 5.3 lists the possible values and for the Shell_NotifyIcon() flag parameter.

To delete an icon from the taskbar, use the same NOTIFYICONDATA structure initialization, but pass the NIM_DELETE flag:

```c
Shell_NotifyIcon( NIM_DELETE, &nidNotifyIcon );
```

### Adding Documents to the Most Recently Used List

Add documents to the MRU by calling SHAddToRecentDocs():

```c
SHAddToRecentDocs( SHARD_PATH,
                   TEXT("\MyDocuments\Hello.txt"));
```

The parameters, in the order shown, are the flag SHARD_PATH, which indicates that we are adding the document using its fully qualified, null terminated pathname, and the pathname as a literal string. CE has an upper limit of 10 MRU documents, and if you try to add more the function will fail. You can’t delete individual documents from the MRU list. To empty the entire list, pass a zero for the pathname parameter:
The SysMenuGadgets techniques work well on the larger CE platforms, but they aren’t entirely in the spirit of the palmtop devices. If you choose to target the smallest devices, you’ll want to pare your application’s shell down to a much more concise set of behaviors.

Shell Behavior for the Palmtop CE Platforms

Even though the physical difference in screen size between the HPC and the palmtop isn’t that large, we do cross a dramatic threshold in the transition between these two devices. You can make a good, usable HPC application without huge changes in your Win32 approach to user interface issues. To make a good, usable palmtop application, you have to embrace the spirit of CE’s PPC shell, and this means leaving a lot of familiar and comfortable desktop application design ideas behind.

Although it’s possible for a palmtop CE application to do several things simultaneously, designing a PPC application that depends on the user’s ability to discern the state of background tasks constitutes a drastic error in judgment. From a user’s point of view, a palmtop CE device does one thing at a time: the thing they can see. This isn’t a technological limitation; it’s a cognitive limitation.

In a nutshell, PPC application shell strategy has two overarching tenets:

- Provide visual cues for whatever an application is doing
- Take full advantage of screen real estate

You’ll probably want to rethink many aspects of your Win32 application’s visual presentation if it’s headed for the PPC. A single dialog, which encapsulates access to all application behavior, is a good basic model with which to start. Difficult choices and possibly a bit of painful pruning lie ahead. That’s the bad news. Now for the good news: It’s true there isn’t much screen real estate available on a PPC, but when your application is active, it can take it all.

In the FullScreenDialog (see Listing 5.3) example, we’ll look into how to take over the entire screen, including the areas normally occupied by the taskbar, the SIP button, and the Windows Start logo (see Figure 5.3).
Figure 5.3 The FullScreenDlg example running on the PPC emulator.

Listing 5.3 FullScreenDlg.cpp.
MSG msg;
HACCEL hAccelTable;
// Perform application initialization:
if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}
hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_FULLSCREENDLG);
// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
return msg.wParam;

// FUNCTION: MyRegisterClass()
// PURPOSE: Registers the window class.
// COMMENTS:
// It is important to call this function so that the application
// will get 'well formed' small icons associated with it.
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS    wc;
    wc.style          = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc    = (WNDPROC) WndProc;
    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hInstance      = hInstance;
    wc.hIcon          = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_FULLSCREENDLG));
    wc.hCursor        = 0;
    wc.hbrBackground  = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName   = 0;
    wc.lpszClassName  = szWindowClass;
    return RegisterClass(&wc);
}

// FUNCTION: InitInstance(HANDLE, int)
//
// PURPOSE: Saves instance handle and creates main window
// COMMENTS:
// In this function, we save the instance handle
// in a global variable and
// create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd = NULL;
    TCHAR szTitle[MAX_LOADSTRING];  // The title bar text
    TCHAR szWindowClass[MAX_LOADSTRING];  // The window class name
    hInst = hInstance;        // Store instance handle
    //in our global variable
    // Initialize global strings
    LoadString(hInstance, IDC_FULLSCREENDLG,
                szWindowClass, MAX_LOADSTRING);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    //If it is already running, then focus on the window
    hWnd = FindWindow(szWindowClass, szTitle);
    if (hWnd)
    {
        SetForegroundWindow ((HWND) (((DWORD)hWnd) | 0x01));
        return 0;
    }
    MyRegisterClass(hInstance, szWindowClass);
    RECT rect;
    GetClientRect(hWnd, &rect);

    hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL);

    if (!hWnd)
    {
        return FALSE;
    }
    //When the main window is created using
    //CW_USEDEFAULT the height of the menubar (if one
    //is created is not taken into account).
    //So we resize the window after creating it
    //if a menubar is present
    
    RECT rc;

Listing 5.3  FullScreenDlg.cpp.
GetWindowRect(hWnd, &rc);
rc.bottom -= MENU_HEIGHT;
if (hwndCB)
    MoveWindow(hWnd, rc.left, rc.top, rc.right,
             rc.bottom, FALSE);
}
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
return TRUE;
}

Listing 5.3  FullScreenDlg.cpp. (continues)
SendMessage (hWnd, WM_CLOSE, 0, 0);
break;
default:
    return DefWindowProc(hWnd, message,
                        wParam, lParam);
}
break;
case WM_CREATE:
    hwndCB = CreateRpCommandBar(hWnd);
    break;
case WM_PAINT:
    RECT rt;
    hdc = BeginPaint(hWnd, &ps);
    GetClientRect(hWnd, &rt);
    DrawText(hdc, TEXT("Las Comidas"),
             _tcslen(TEXT("Las Comidas")), &rt,
             DT_SINGLELINE | DT_VCENTER | DT_CENTER);
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
case WM_SETTINGCHANGE:
    SHHandleWMSettingChange(hWnd, wParam, lParam, &s_sai);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
HWND CreateRpCommandBar(HWND hwnd)
{
    SHMENUBARINFO mbi;
    memset(&mbi, 0, sizeof(SHMENUBARINFO));
    mbi.cbSize = sizeof(SHMENUBARINFO);
    mbi.hwndParent = hWnd;
    mbi.nToolBarId = IDM_MENU;
    mbi.hInstRes = hInst;
    mbi.nBmpId = 0;
    mbi.cBmpImages = 0;
    if (!SHCreateMenuBar(&mbi))
        return NULL;
    return mbi.hwndMB;
}

// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg,
                        UINT message,
                        WPARAM wParam, LPARAM lParam);

Listing 5.3  FullScreenDlg.cpp.
WPARAM wParam,
LPARAM lParam)
{
    SHINITDLGINFO shidi;
    switch (message)
    {
    case WM_INITDIALOG:
        // Create a Done button and size it.
        shidi.dwMask = SHIDIM_FLAGS;
        shidi.dwFlags = SHIDIF_DONEBUTTON | SHIDIF_SIPDOWN | SHIDIF_SIZEDLGFULLSCREEN;
        shidi.hDlg = hDlg;
        SHInitDialog(&shidi);
        return TRUE;
    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK) {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
        break;
    }
    return FALSE;
}

// FullScreenDlgProc
//
BOOL CALLBACK FullScreenDlgProc(HWND hWnd, UINT wMsg,
    WPARAM wParam,
    LPARAM lParam)
{
    SHINITDLGINFO shidi;
    MENUITEMINFO mii;
    HWND hWndParent, hWndBmp;
    HMENU hSHMenu;
    RECT rc;
    HDC hdc, hdcMemory;
    BITMAP bmp;
    HBITMAP hBmp;
    int cx, cy;
    BOOL bOk;
    // Valid states
    #define SHFS_SHOWTASKBAR       0x0001
    #define SHFS_HIDETASKBAR       0x0002
    #define SHFS_SHOWSIPBUTTON     0x0004
    #define SHFS_HIDESIPBUTTON     0x0008
    #define SHFS_SHOWSTARTICON     0x0010
    #define SHFS_HIDESTARTICON     0x0020

    switch (wMsg)
{ case WM_INITDIALOG:
    hWndParent = GetParent(hWnd);
    //
    // Valid mask values
    //#define SHIDIM_FLAGS                0x0001
    shidi.hDlg = hWnd;
    shidi.dwMask = SHIDIM_FLAGS;
    shidi.dwFlags = SHIDIF_DONEBUTTON | SHIDIF_SIZEDLGFULLSCREEN;
    SHInitDialog(&shidi);
    SHFullScreen(hWnd, SHFS_HIDETASKBAR);
    SHFullScreen(hWndParent, SHFS_HIDESIPBUTTON);
    ShowWindow(hwndCB, SW_HIDE);
    hdc = GetDC(hWnd);
    cx = GetDeviceCaps(hdc, HORZRES);
    cy = GetDeviceCaps(hdc, VERTRES);
    ReleaseDC(hWnd, hdc);
    rc.left = 0;
    rc.top = 0;
    rc.right = cx;
    rc.bottom = cy;
    MoveWindow(hWnd, rc.left, rc.top, rc.right,
               rc.bottom, TRUE);
    hSHMenu = (HMENU)SendMessage((hwndCB),
                    SHCMBM_GETSUBMENU, (WPARAM)0,
                    (LPARAM)IDM_MENU);
    memset((char *)&mii, 0, sizeof(mii));
    mii.cbSize = sizeof(mii);
    mii.fMask  = MIIM_TYPE;
    mii.dwTypeData = TEXT("Enchiladas");
    mii.cch = MAX_LOADSTRING;
    mii.fType = MFT_STRING;
    GetMenuItemInfo(hSHMenu, IDM_MENU, FALSE, &mii);
    SHSetNavBarText(hWndParent, TEXT("Ole!"));
    SHDoneButton(hWndParent, SHDB_SHOW);
    break;
  case WM_COMMAND:
    switch (LOWORD (wParam))
    {
      case IDOK:
      case IDCANCEL:
        EndDialog (hWnd, 1);
    Listing 5.3  FullScreenDlg.cpp.
Listing 5.3  FullScreenDlg.cpp. *(continued)*

Looking at the FullScreenDlg Example

With the exception of FullScreenDlgProc(), almost all of this program was generated by the development environment wizard. However, we see some important differences between this project and earlier examples, because in this case we are looking at code generated specifically for the smaller CE devices. We’ll look into the generated code a bit before moving on to the specifics of handling the dialog box.

First, notice that we see some new header files in this application:

```cpp
#include <aygshell.h>
#include <sipapi.h>
```

These files are specific to the PPC platforms, and contain declarations you’ll need in order to call shell functions. If you get linker errors for shell functions, make sure you have included these.

In the global variables section, we see this new declaration for a SHACTIVATEINFO structure. This is provided as an argument to a function call we see in the main message switch, SHHandleWMSettingChange():

```cpp
static SHACTIVATEINFO s_sai;
```

One very good bit of news about the strange new world of the PPC shell is that, by default, you enjoy an almost complete lack of responsibility for maintaining it. One important difference in PPC applications is that they must share screen real estate with the Supplementary Input Panel (SIP), better known as the “chicklet keyboard.” Users show and hide this small, screen-based keyboard on demand, which could create real headaches for application programmers. We get a break on this, however, because the system arbitrates SIP screen real estate use in response to the WM_SETTINGCHANGE message:
case WM_SETTINGCHANGE:
    SHHandleWMSettingChange(hWnd, wParam, lParam, &s_sai);
    break;

The global SHACTIVATEINFO structure maintains data that allows the system to toggle the visibility of the SIP, automatically restoring whatever it covered. Now let’s take a look at how the FullScreenDlg example appropriates system screen real estate to itself.

Taking Over the Entire Screen

In the FullScreenDlg example, we start with an ordinary “Hello World” application. The only significant modification we make is the addition of three items to its menu resource template: “Tacos,” “Burritos,” and “Tamales:”

case IDM_TACO:  
case IDM_BURRITOS:  
case IDM_TAMALES:  
    DialogBox(hInst, TEXT("FullScreenDlg"),  
             hWnd, FullScreenDlgProc );  
    break;

If the user chooses any of these, we launch the FullScreenDlgProc(). In the dialog’s initialization sequence, we cause it to take over screen real estate that would normally be allotted to shell components. Notice this declaration at the top of the dialog box procedure:

//declaration of the shell's init dialog structure
SHINITDLGINFO shidi;

The SHINITDLGINFO structure contains information that determines how a dialog looks when it is invoked. Table 5.4 lists the structure members and their meanings.

Table 5.4 SHINITDLGINFO Structure Members and Their Meanings

<table>
<thead>
<tr>
<th>STRUCTURE MEMBERS</th>
<th>MEANINGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD dwMask</td>
<td>Always set to SHIDIM_FLAGS. Indicates that the flags member is initialized.</td>
</tr>
<tr>
<td>HWND hDlg</td>
<td>Handle to the dialog.</td>
</tr>
<tr>
<td>DWORD dwFlags</td>
<td>SHIDIF_DONEBUTTON Adds an OK button to the navigation bar. SHIDIF_SIZEDLG Causes the dialog to be sized.</td>
</tr>
</tbody>
</table>
We initialize the SHINITDLGINFO structure, and pass it to the SHInit-
Dialog() function.

```c
shidi.hDlg = hWnd;
shidi.dwMask = SHIDIM_FLAGS;
shidi.dwFlags = SHIDIF_DONEBUTTON |
SHIDIF_SIZEDLGFULLSCREEN;

SHInitDialog(&shidi);
```

The function SHInitDialog() is specifically intended to be used to create full
screen dialog boxes. One of its advantages is that it places an “OK” button in
the navigation bar, freeing application screen real estate that would have been
used for this purpose.

SHInitDialog is fairly flexible — you can take pretty much any combination
of screen areas for your own application, depending on the initialization
flags you specify. Using the SHIDIF_FULLSCREENNOMENUBAR, the dialog
starts up using the entire screen. However, you don’t have to start with the full
screen, because you can dynamically size and position the dialog using the
function SHFullScreen(). The parameters to SHFullScreen() (see Table 5.5) are
the handle to the window being sized and a flag which defines which real
estate it is gaining or relinquishing. Here are the flags and their meanings.

### Table 5.5  SHFullScreen Flags and Their Meanings

<table>
<thead>
<tr>
<th>FLAG NAME</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHFS_SHOWTASKBAR</td>
<td>Shows the taskbar</td>
</tr>
<tr>
<td>SHFS_HIDETASKBAR</td>
<td>Dialog hides task bar</td>
</tr>
<tr>
<td>SHFS_SHOWSIPBUTTON</td>
<td>Shows the SIP button</td>
</tr>
<tr>
<td>SHFS_HIDESIPBUTTON</td>
<td>Dialog hides SIP button</td>
</tr>
<tr>
<td>SHFS_SHOWSTARTICON</td>
<td>Shows the Windows Start icon</td>
</tr>
<tr>
<td>SHFS_HIDESTARTICON</td>
<td>Dialog hides Start icon</td>
</tr>
</tbody>
</table>
Here’s how we hide the taskbar and the SIP button:

```c
SHFullScreen(hWnd, SHFS_HIDETASKBAR);
SHFullScreen(hWndParent, SHFS_HIDESIPBUTTON);
```

Notice that we make one call for each element we change. Next, we move the dialog to its new size and position. First, we hide the window:

```c
ShowWindow(hwndCB, SW_HIDE);
```

Next we get a DC so that we can get the exact size of the screen in device units:

```c
hdc = GetDC(hWnd);
cx = GetDeviceCaps(hdc, HORZRES);
cy = GetDeviceCaps(hdc, VERTRES);
ReleaseDC(hWnd, hdc);
rc.left = 0;
rc.top = 0;
rc.right = cx;
rc.bottom = cy;
MoveWindow(hWnd, rc.left, rc.top, rc.right, rc.bottom, TRUE);
```

Finally, we move the dialog to its new size and position, with MoveWindow(). Setting the last parameter to TRUE instructs MoveWindow() to allow the window to make the window visible.

Next, we replace the navigation bar’s text string:

```c
SHSetNavBarText(hWndParent, TEXT("Ole!"));
```

Finally, we restore the “OK” button to the navigation bar so that the user will have a way to dismiss the dialog:

```c
SHDoneButton(hWndParent, SHDB_SHOW);
```

**Looking Ahead**

With our examination of the CE shells, we complete Part One: “Adapting Application Appearance to Windows CE.” In essence, we’ve seen what we can do with CE. Now that we understand the form, we are ready to tackle the substance. To me, this is the truly exciting subject. In coming chapters, we’ll learn about what’s under the hood: memory, the registry, files, databases, and more.
Because virtually everyone has to create at least some visible aspects of an application, up to this point you’ve probably at least skimmed through most examples. Henceforth, however, it’s likely that your specific interests will guide your reading. Given that fact, I’d like to encourage you to take a few minutes to read about memory management in the opening chapter of the next section. This chapter is a fairly high-level overview of CE’s approach to memory, and as such will provide you with background that is critical to making good porting decisions.
Translating Win32 Application Behaviors to Windows CE

Zen and the Art of Porting

Once you have created a visual “shell” for your newly minted Win CE application, it’s time to think about porting application behaviors. The biggest difference between porting application appearance and porting application behavior is that behaviors inherently compete for use of shared system resources: memory, I/O capacity, and power. For this reason, porting behaviors isn’t simply a matter of eliminating unsupported Win32 APIs; it’s more a matter of finding the most succinct and frugal ways to accomplish a particular task.

The first step in this process, and one that you must undertake entirely on your own, away from your workstation, is to identify the essence of the application you are porting. At the most basic practical level, there are two questions to ask:

- What task do you want to enable the user to accomplish?
- What product will the user’s effort produce?
Defining the functional essence of your application and its intended products constitutes an implicit determination about what Win32 behaviors and features are superfluous under Windows CE. *This is probably the most critical task in porting to Win CE, and it is very, very hard to accomplish.* What makes it hard is that it implies wholesale abandonment of large chunks of your application’s Win32 code base, and it is human nature to resist letting go of that which you worked hard to create. If you try to define the scope of your Win CE app with too large a group or with people who are technically naïve, you’ll find features creeping into your Win CE product design that just shouldn’t be there.

If you can answer the two questions above very specifically, then you have a blueprint for porting critical behaviors. *I would encourage you to write the answers to these questions, and if you work with a team, extend this exercise to everyone involved in managing the porting process.* Notice that I suggest specifically excluding users and junior programming team members from this deliberation. (Some things in life don’t lend themselves to group decision-making—emergency appendectomy, pro football, and porting code from Win 32 to the resource constrained environment of Windows CE, for example.)

In this spirit of “Heave ho!”, I’ll describe my own porting experiences and opinions in the coming chapters, and of course, you may disagree. But more often than not, when I have had great difficulty implementing a feature on CE, a few hours away from the problem usually brought me to the realization that the feature was inappropriate or my design was too complex.

**Mission: Possible**

Just as in the first section, the object of the discussions in the next several chapters is to help you make steady, incremental progress toward a stable Windows CE implementation of your Win32 code. To accomplish this as quickly as possible, we’ll focus on code reuse rather than on using the new and unique features of Windows CE to implement your application’s behaviors.
If the chapters in this book had been ordered on the basis of their relative importance, this chapter would have been first. Understanding how CE uses and manages memory in general is the key to your ultimate success on the CE platforms. This may seem like unnecessary detail, especially if your application doesn’t do any memory allocation directly. However, even if you don’t explicitly allocate memory, it is allocated in your behalf—by the loader—for your program’s stack, heap, and static data storage. Thus, you still have to make informed choices to create a memory-efficient application.

What does it mean to be memory efficient on a Win CE device? It means that your code, your data, and everything else in CE land should be able to gracefully coexist in 2MB. No kidding. If that didn’t get your attention, take a moment and look at the file sizes of a few of your desktop Win32 applications.

In the early days of Windows, we used to speak of the need for applications to be “well behaved.” What we meant by this was that no application should acquire and hold so many resources that other applications couldn’t execute when they got a time slice. In essence, understanding the rules and limitations of CE’s memory environment is what makes your application well behaved in the CE world.
We’ll examine four concepts in this chapter:

- How CE memory is laid out
- How to choose memory allocation strategies for your application
- How to optimize the placement of application data
- What happens when the system is in a very low memory state

An understanding of these issues will allow you to make a smooth transition to the memory-constrained environment of Windows CE.

**PORTING TIP** To be an effective CE programmer, you must understand CE memory architecture. The rules are not difficult to understand, but they are very different than what you are used to on the desktop.

**Something New Under the Sun**

Before we begin to dissect the architecture of CE, let’s step back and take in the panoramic view. Up to this point, we have framed the discussion in terms of porting Win32 code to Windows CE. This approach is informed by the bias that Windows CE is fundamentally nothing more than an “itty-bitty” Windows program. While this has been a useful fiction up to this point, it is, in fact, not at all the case.

Windows CE is an entirely new computing model, a marriage between embedded systems technology and the larger Windows world. Its defining characteristic is mobility. The CE user doesn’t “go to” a computer or a network; rather they carry these with them. In the CE world view, the network is wherever you are. Up to now, we’ve often heard the term seamless used to describe interoperable software applications. CE brings seamlessness to the human/computer interface.

As software developers, the importance of the “Windows-ness” of CE is twofold: First, CE abstracts handheld embedded computing with the stable, proven Win32 API, giving us a truly cross-platform tool set for this genre of devices; and second, we have a large existing code base that is at least nominally portable to this API. Essentially, clothing personal embedded computing in the Win32 programming model provides developers with a huge productivity advantage. However, if we view this model too literally or too credulously, we lose sight of the real potentials of CE: inherent mobility and discretionary connectivity.

We need the Win32 abstraction for productivity reasons, but we also need to know how things really work under the hood.
Anatomy 101

In part, the durability, and hence mobility, of CE devices derive from the fact that they have no moving parts (for example, disk drives). All storage is either in read only (ROM) or nonvolatile random access memory (RAM). ROM is used to store the operating system and any bundled applications a vendor distributes as part of the device.

RAM is divided into two regions: the object store and program memory. The object store is functionally equivalent to the file system of a desktop computer. Files, application data, and third-party executable code are stored here. All files in the object store are maintained in a special compressed format. This compression process isn’t visible to you, the programmer, but it’s one reason that CE programs appear to execute more slowly than desktop programs—everything, including executable code, must be decompressed when you access it and recompressed when you store it. Users can adjust the proportions of memory devoted to the object store and to program memory.

PORTING TIP Writeable memory is divided into a persistent storage area, the object store, and an area in which code executes, program memory. Users can adjust the proportion of memory devoted to each of these areas.

All third-party code and some ROM-based code is stored in compressed format. Program memory is where code executes if it was stored in compressed form. This system is highly space efficient, because Windows CE uses demand paging to load pages of executables: The loader uncompresses requested pages, and brings them into program memory a page at a time (page size is determined by device manufacturers, but in practice is either 1K or 4K), as they are needed. Older pages are swapped out as the execution point moves past them and they are no longer needed (a least recently used algorithm determines which pages are swapped).

Some ROM-resident code is not stored in compressed format. Obviously, it would be extremely wasteful to compress and uncompress parts of the kernel and OS that are almost constantly in use. This type of code executes in place, that is, in ROM, using only small amounts of program memory for stack and heap space. This means that it doesn’t incur the overhead of the compression cycle or the load process, providing a significant performance advantage and power savings, but at the cost of increased storage space.

Other less frequently used ROM resident executables and DLLs are stored in compressed format. This code is treated exactly the same as RAM resident compressed code. It is uncompressed and loaded on demand, executing RAM-based program memory.
How CE Applications Allocate Memory

The Windows CE memory management API is sophisticated, but it is lean. It consists of three groups of API functions: the VirtualAlloc family, the HeapAlloc family, and the LocalAlloc family. Don’t use the C runtime functions to allocate memory on CE; doing so will cause unpredictable behavior.

PORTING TIP Eliminate all calls to calloc(), malloc(), and alloc(). Replace these with the Windows CE memory allocation APIs that best suit your needs.

Basically, the size, duration, and consistency of your application’s memory allocation will determine which type of allocation strategy you’ll want to use. Our two goals in allocating memory under CE are to use memory as sparingly as possible, and when we are done with it, to make sure that it is made available to other processes as quickly as possible. Here are three important differences between the families of allocation functions:

- When they withdraw memory blocks from the allocation pool, giving your application exclusive access to them
- How much memory they actually remove from the allocation pool
- When they return the memory to the allocation pool after you free it

Let’s look at the allocation APIs based on types of allocation scenarios.

Making Large Allocations for Fairly Limited Durations

Say you have an application that displays large bitmapped graphics. If you are using standard bitmap file formats, you must load the entire file in order to transfer it to the screen. Depending on the color depth of the images, this could amount to a substantial allocation of memory. However, you don’t need to hold the memory for very long—only long enough to do the raster drawing. The VirtualAlloc()/VirtualFree() family of functions may be what you need for this kind of job. Below is the declaration for VirtualAlloc():

```
LPVOID VirtualAlloc(LPVOID lpAddress,
                    DWORD dwSize,
                    DWORD flAllocationType,
                    DWORD flProtect);
```

The parameters are, in the order shown, the desired base address of the requested allocation, the size in bytes of the allocation request, a flag that
specifies whether to reserve or actually commit the allocation, and the access permissions for the allocation. If the first parameter is NULL, the block can be allocated anywhere space is available. The second parameter (dwSize) is rounded up to the next full page size. Pay careful attention to the value of dwSize. If it is one byte over the physical page size, you’ll end up allocating two pages. You can get the physical page size for a device using this function:

```c
VOID GetSystemInfo( LPSYSTEM_INFO lpSystemInfo);
```

For our purposes, there are two important values for the `flAllocationType` flags: MEM_COMMIT and MEM_RESERVE. MEM_RESERVE indicates your intention, at some future point in time, to actually use space. You don’t actually have physical access to the space after reserving it. To get access, you must call VirtualAlloc() on the page (or pages) with the MEM_COMMIT flag. This two-step strategy has a pair of important advantages: Remember, we’re allocating whole pages at a time, which under CE is a very large amount of memory. You call VirtualAlloc() to reserve a block of pages. When you actually need the pages, VirtualAlloc() can commit single pages in the reserved block. This allows you to ensure space will be available before you begin a memory intensive operation. However, it doesn’t actually withdraw physical memory form the allocation pool until you need it.

To free space allocated by VirtualAlloc(), you call VirtualFree(). The key thing to know about this call is that it returns freed pages to the allocation pool immediately:

```c
BOOL VirtualFree(LPVOID lpAddress, 
                 DWORD dwSize, 
                 DWORD dwFreeType);
```

The parameters, in the order shown, are the base address of the block being freed, the size to free, and a flag that specifies what change to make in the block’s allocation status. The flag parameter `dwFreeType` specifies whether to decommit a page (`dwFreeType = MEM_DECOMMIT`) or to completely free the page (`dwFreeType = MEM_RELEASE`). When a page is decommited, it can be reallocated by the process that reserved it. When it is freed, it is returned to the system memory and can be allocated by anybody.

**PORTING TIP** Virtual Alloc is best used for large allocations of fairly short duration.

- VirtualAlloc() allocates memory in whole-page increments.
- Any unused memory inside a page is wasted; it can’t be used to satisfy other allocation requests.
Memory can be reserved without withdrawing it from the physical allocation pool.

Memory can’t be accessed until it is committed, which removes it from the physical allocation pool.

When you call VirtualFree() to release memory, it is immediately available to other processes.

---

**Using Private Heaps for Frequent Small Allocations**

Allocating too much memory and holding it for too long can obviously move you into a low memory state. A less obvious but equally insidious way to run out of memory is to make a lot of small allocations that cause fragmentation of the heap. Basically, here’s how fragmentation develops.

Your application has default heap space that is allocated by VirtualAlloc(). Initially, only one page of the heap is committed, but of course, more is reserved. When you make an allocation request, the system looks for a single, contiguous free region in the committed page(s) of your application’s heap. If it finds one, it returns a pointer (or handle) to your block. If it can’t find one, it commits enough pages to satisfy the request.

Obviously, a physical page in the heap may contain memory blocks from dozens of different allocation requests. The system simply fits them into a page wherever there is space for them. None of these pages can be decommitted and returned to the allocation pool until all of the individual allocations in the page are freed.

When an allocation request grows the heap, it may not use all of the space in the newly committed pages. If subsequent small allocations are satisfied using “leftover” space in the newly committed pages, then these pages can’t be freed when the “big” allocation is freed. Also, if the “little” allocations are in the middle of the page, they dramatically reduce the size of the largest contiguous free block that can be allocated.

In a nutshell, heap fragmentation arises from these two things:

- A page can only be freed if it is completely empty of allocations.
- Allocation requests must be satisfied with blocks of contiguous free memory.

It’s quite possible that you could have far more than the number of bytes available in the local heap to satisfy an allocation request, and still have to commit a new page in order to actually get a contiguous block of memory.

In this case, the problem is not so much that we needed to make a few big, temporary allocations of memory, it is that the big allocations are interspersed with smaller allocations that had a different lifespan.
Document processing is a good example of a situation like this. First you load data and structures, then after that you make frequent, small allocations for string buffers and the like. If you make many small allocations that share a predictable lifespan, you can conserve memory by creating a separate application-specific heap. Using this strategy, you can prevent your application’s local heap from unnecessary growth and subsequent fragmentation.

**PORTING TIP** If you make many small allocations that share a predictable lifespan, you can conserve memory and prevent local heap fragmentation by creating a private, application-specific heap and using it for small frequent allocations.

Five functions are used to create and manage a private heap:

```c
HANDLE HeapCreate(DWORD flOptions, DWORD dwInitialSize, DWORD dwMaximumSize);
LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes);
LPVOID HeapReAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes);
BOOL HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem);
BOOL HeapDestroy(HANDLE hHeap);
```

Under CE, HeapCreate() basically acts as a statement that you plan to use a private, application-specific heap. It doesn’t reserve or commit any heap space, and the parameters, though consistent with the appearance of the Win32 version of the function, are actually ignored. However, as a matter of defensive programming, you should supply valid values for them.

The private heap grows itself transparently, by committing pages when HeapAlloc() allocation requests are made. A private heap needs to keep track of some status information to manage itself (as does any heap), and this space is located in the private heap’s own memory. For this reason, the actual amount of space available for allocation is less than the total size of the private heap.

HeapAlloc(), HeapReAlloc(), and HeapFree() work in much the same way as the more familiar LocalAlloc() family: they allocate, reallocate, and free individual blocks, respectively. HeapDestroy() frees the pages committed to the private heap and returns them to the allocation pool, whether or not individual allocations within the heap have been freed with HeapFree().

### Using the Local Heap

A lot of Win32 applications allocate memory exclusively from the local heap, using LocalAlloc(), LocalReAlloc, and LocalFree(). The local heap is created when your process is loaded. Because the local heap is created as part of the
program startup sequence, you can count on at least some memory being available there initially. It is by far the easiest memory allocation scheme to use, so choose it if it makes sense, given your application’s memory allocation patterns.

**PORTING TIP** The LocalAlloc() family of memory allocation APIs is easy to use. Choose this approach if it is compatible with your application’s memory allocation needs.

If you read through your sources and find only the LocalAlloc() family of functions are used for memory allocation, you are probably in pretty good shape. There are a few things to bear in mind when porting code that allocates from the local heap:

- On CE, LocalAlloc() supports only the flags LMEM_FIXED, LPTR, and LMEM_ZEROINIT.
- Use the local heap to allocate moderate or small blocks that have various and indeterminate life spans.
- LocalFree() returns space within your heap but doesn’t necessarily cause pages to be returned to the allocation pool.

The goal in using the local heap is to keep its size fairly consistent. Use other allocation methods for objects that would cause it to grow dramatically or to become fragmented.

### Reengineering Temp Files

On the desktop, temporary files truly do provide a “one size fits all” kind of runtime data management strategy. Many document-processing applications use temporary files to buffer data and to support undo/redo behaviors. It may seem tempting to stick with this approach because it offers so many advantages and conveniences, and definitely provides a simpler programming model than concocting a variety of memory allocation schemes. The concept breaks down on CE, though. Using temporary files essentially doubles the size of your data at runtime, and this is a luxury we can ill afford.

If you use temp files, try the following instead:

- Save document state information in compressed format in runtime data structures.
- Limit the scope of undo/redo behaviors.
- Frequently write changes back to the document, but as deltas at the end of the file.
PORTING TIP  Temporary files are memory hungry. Try to devise other, more-compact state management strategies.

Stack-Based Data

Everybody’s favorite place to put data is the stack, and why not? It is effortless. Under CE, if these three things are true, then a datum belongs on the stack, period.

- The datum exists only for the scope of the function in which it is declared.
- You know its exact size, and it’s fairly small.
- It is not a constant value.

By default, you program starts up with 1K of stack space committed with the potential to grow the stack to a maximum size of 58K. Obviously, this hard limit means that you don’t want huge objects on the stack. Trying to grow the stack beyond the maximum size will cause an immediate program (and possibly system) crash. On the other hand, the stack is probably the most efficiently used memory on the system. It’s fragmentation proof.

Optimizing Your Application’s Use of Data Segments

As you’ve seen, there are a lot of things that you can do to improve your Win32 application’s runtime memory use under CE. However, you can also reap some fairly impressive gains in memory efficiency, simply by being aware of where the linker puts data in your program’s executable.

An executable file basically has two kinds of things in it: executable code and data. On all of the Windows hosts, code is “pure,” meaning that you can’t modify it at run time. For this reason, the part of a Windows executable file that contains the code is designated read only, so it is of little concern to us here. The rest of the executable file is devoted to data, of which there are two kinds: read-only data, like literal strings and constants, and read-write data, which is usually either data with global scope or data defined using the static keyword.

Since access permissions are set on a page-by-page basis once the program is in memory, these data storage areas in your program have to be created in page-sized increments. At a minimum, your executable file contains one page for static (read-only) data, and one page for read/write data. This means that there may well be some unused space in the data storage areas of your
executable file, and you could profit by using it instead of making allocation requests.

On the other side of the coin, you may discover that one or the other of your data sections is just slightly larger than one page, which causes the allocation of a lot of unnecessary space. Moving things around a bit may shrink your data sections and get you in under that critical threshold.

So how do you find out how your application file lays out the data sections? Set the linker output option for generating a mapfile; this contains data section names, lengths, and base addresses. Here’s how to interpret the parts of the link map (see Listing 6.1) that show how your program’s data is organized.

First, let’s look at the two places read-only data is stored, the .rdata section and the resource data sections, .rsrc$01 and .rsrc$02.

Here are a few lines from the ExeFileMem example. Basically, it is a typical “Hello World”-generated application, but with some data declarations added for the purposes of producing the mapfile above:

<table>
<thead>
<tr>
<th>Start</th>
<th>Length</th>
<th>Name</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001:00000000 00000c48H</td>
<td>.text</td>
<td>CODE</td>
<td></td>
</tr>
<tr>
<td>0002:00000000 0000054H</td>
<td>.rdata</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0002:00000054 0000028H</td>
<td>.idata$2</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0002:0000007c 0000014H</td>
<td>.idata$3</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0002:00000090 0000088H</td>
<td>.idata$4</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0002:00000118 0000023cH</td>
<td>.idata$6</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0002:00000354 0000000H</td>
<td>.edata</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0003:00000000 0000088H</td>
<td>.idata$5</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0003:00000088 0000004H</td>
<td>.CRT$XCA</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0003:0000008c 0000004H</td>
<td>.CRT$XCZ</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0003:00000090 0000004H</td>
<td>.CRT$XIA</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0003:00000094 0000004H</td>
<td>.CRT$XIZ</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0003:00000098 0000004H</td>
<td>.CRT$XPA</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0003:0000009c 0000004H</td>
<td>.CRT$XPZ</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0003:000000a0 0000004H</td>
<td>.CRT$XTA</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0003:000000a4 0000004H</td>
<td>.CRT$XTZ</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0003:000000a8 0000010H</td>
<td>.data</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0003:000000b8 00000810H</td>
<td>.bss</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0004:00000000 00000104H</td>
<td>.pdata</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0005:00000000 000001f0H</td>
<td>.rsrc$01</td>
<td>DATA</td>
<td></td>
</tr>
<tr>
<td>0005:000001f0 000005d4H</td>
<td>.rsrc$02</td>
<td>DATA</td>
<td></td>
</tr>
</tbody>
</table>

Listing 6.1  An abbreviated link map.
// Global Variables:
HINSTANCE hInst;     // The current instance
HWND hwndCB;     // The command bar handle

//make a large global allocation
int iLargeIntArray[512];

//the const keyword means this
datum is read-only
const int iConstDataItem = 1;

Notice the last declaration, iConstDataItem. This variable is declared with
the keyword, which makes it nonmodifiable.

PORTING TIP Nonmodifiable data is placed in the program section .rdata.

We’ve seen that memory allocation operations often have a granularity of
one page. In order to designate an item or group of items as read only, they
have to reside together in their own page(s), because that is the smallest incre-
ment on which we can set access permissions. Any datum that we declare as
with the const keyword has to end up in a read-only data section, because
that’s the only way to ensure it is protected from modification.

Now examine the mapfile line that gives the length of the .rdata section:

<table>
<thead>
<tr>
<th>Start</th>
<th>Length</th>
<th>Name</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002:00000000</td>
<td>0000054H</td>
<td>.rdata</td>
<td>DATA</td>
</tr>
</tbody>
</table>

The length of this section is 54h (84 decimal), however, when loaded this
section will consume an entire page of program memory. Yikes! We need to
find some other nonmodifiable data we can move to this page, or this space
will be wasted.

The first possibility that springs to mind is string data, and most applica-
tions have a number of them stored as resources. Resources have their own
section in the executable file layout. Here are the mapfile lines that apply to
them:

<table>
<thead>
<tr>
<th>Start</th>
<th>Length</th>
<th>Name</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0005:00000000</td>
<td>0000001f0H</td>
<td>.rsrc$01</td>
<td>DATA</td>
</tr>
<tr>
<td>0005:0000001f0</td>
<td>000005d4H</td>
<td>.rsrc$02</td>
<td>DATA</td>
</tr>
</tbody>
</table>

There’s a good reason not to move string data out of the resource files and
into the static data area. CE provides us with a special version of the Load-
String() function used to load string resources that allows you to read the
string in place. Calling LoadString() like this returns a pointer to the string:

LPCTSTR pStringResource;
    //if the last parameter is NULL,
//get back a pointer to a constant string
pStringResource = (LPCTSTR)LoadString( hInstance,
    IDS_STRING, NULL );

You can use the string, but you can’t write back to it. Always use this function to load string resources you don’t intend to modify.

There are lots of other possibilities for moving data to the .rdata section, but leave string resources in the resource section so you can use the CE version of LoadString().

Read/Write Data Sections

Any item that has greater than function level scope or is declared with the static keyword resides in the read/write data sections of the executable file. In addition, the loader requires access to something less 100 bytes of this data area.

There are two kinds of read/write data in your program: The initialized items live in the .data section, and the unitialized items live in the .bss section. These sections share a region of memory, so the sum of their length is what matters, not the individual sizes. Here are their lines in the mapfile:

<table>
<thead>
<tr>
<th>Start</th>
<th>Length</th>
<th>Name</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0003:00000a8</td>
<td>00000010H</td>
<td>.data</td>
<td>DATA</td>
</tr>
<tr>
<td>0003:00000b8</td>
<td>00000810H</td>
<td>.bss</td>
<td>DATA</td>
</tr>
</tbody>
</table>

The sum of their lengths is 820h, or 2080 decimal. Again, you can see that we are just a bit beyond the next lowest page size increment, 2048. At run time, it will require 3 pages (assuming a 1K page size) to store this data. If we could move an item or two into stack-based storage, or possibly make something constant, we could save a page.

PORTING TIP A good, rough-cut checklist for squeezing wasted memory out of your application’s executable file:

- Explicitly declare all data that is functionally constant using the const keyword.
- Recompile and see how use of const changes the size of the sections.
- Adjust the size of the read/write data section. Shrink it by moving invariant data to const and other data to the stack; fill it by using leftover space for buffering that would otherwise have been allocated from the heap.
- Leave 100 free bytes in the read/write section for use by the loader.
- Use CE-style LoadString() to access string resources in place.
Put comments in all your source files that remind you to look at a load map if you change code.

**Low-Memory Conditions**

Even when we’ve done all we can to make our program memory efficient, memory can become dangerously low. When making allocation requests, you must assume that they can fail and make provisions for handling the failure gracefully. You should always test the return value of a memory allocation function.

**PORTING TIP** Search your code for allocation function calls, and make sure all returns are tested. Provide a handler for graceful clean up if allocation fails.

Running out of memory is a more serious (and more likely) situation on CE than it is on any other Windows platform. There is no disk-based virtual memory to swap pages to, so the operating system constantly monitors the allocation status of physical memory. CE uses several mechanisms to conserve memory as it becomes more scarce, but for our purposes two of these are important: allocation request filtering and the WM_HIBERNATE construct.

**Allocation Request Filtering**

Allocation request filtering prevents a single application from taking all available memory with a single large allocation request. Basically, it does this by enforcing a flexible upper limit on the amount of memory an application can request. As memory becomes scarcer, the system lowers the limits on maximum allocation size.

You may see the effects of allocation filtering if you attempt to make an allocation with VirtualAlloc() and it fails even though the result of a call to VirtualQuery() seems to show that there is enough free memory to satisfy your request. In fact, in any kind of low memory situation, VirtualAlloc family functions will fail first, so always test returns and provide a failure handler.

**Warning Applications about Low Memory with the WM_HIBERNATE Message**

Scarcity of memory is a situation that is always looming on a CE device. In order to make the memory-constrained environment manageable, three things need to happen:
Applications must strive to be memory efficient.

The system must notify applications when a memory crisis is developing.

The applications must have a chance to clean up and gracefully relinquish whatever memory they can.

The WM_HIBERNATE message is CE’s way of giving notice to applications that the system is entering a low memory state. In order for a notification about a memory shortage to be useful, you must get it in time to do something about it. For example, the application may need to tell the user to close files, to relinquish whatever memory it can, and to save its state in preparation for an orderly close. For this reason, applications begin to get the WM_HIBERNATE message when available memory falls below the “hibernation threshold,” which is about 128K (this threshold is set slightly higher if the page size is greater than 1K). At 64K of free memory, we cross the “low memory” threshold, and the system more aggressively requests active applications to hibernate, attempts to free pages in the system heap, and as a last resort, tries to shrink the stack. Finally, at about 16K, the system displays the out of memory dialog and begins forcibly closing down applications.

When an application first receives the hibernate notification, chances are good that you can clean up, relinquish memory, and prepare for possible close. In all likelihood, you’ll still be able to allocate small amounts of memory from your local heap—remember, WM_HIBERNATE is sent as a result of a page allocation crisis, but that doesn’t mean that all committed pages are full. If you design your code so that you are never dependent (or at least not for long) on holding large allocations of memory, you can respond to WM_HIBERNATE by completing operations that are in progress, freeing memory, and possibly saving application state.

PORTING TIP  If you have been reasonably careful about your use of allocated memory, responding to WM_HIBERNATE may not be too difficult.

When you get a WM_HIBERNATE message, your first response should be to free any allocation made with VirtualAlloc(). These go back to the allocation pool immediately, and so may resolve the immediate crisis. Next, if possible, finish any pending operations that make use of a private heap, and relinquish the heap. Freeing memory devoted to a private heap also returns pages to the allocation pool immediately, so freeing a private heap may be worth incurring some data loss.

Graphic objects, especially bitmaps, consume large amounts of memory, so discard those, along with created brushes, pens, and device contexts next.

If you want to warn the user that a low memory situation exists when you first get the WM_HIBERNATE message, you can invoke the system low memory dialog using this function:

```c
int SHShowOutOfMemory(HWND hwndOwner, UINT grfFlags);
```
The parameters, in the order shown, are the handle to the window that will serve as the parent of the dialog and a reserved value that must be set to zero. Also, although the declaration shows that there is an integer return, there is no meaningful value returned. The advantage of using this function is that it comes to you with absolutely no overhead. Putting this dialog up may cause the user to close another application, which may in turn solve your memory shortage problems. At the very least, it provides feedback and won’t consume additional memory.

Memory Reconnaissance Tools

We’ve just slogged through a dozen pages of scenarios, rules, warnings, and fairly druid memory management. Now it’s time for something a bit more tangible. First of all, you’ll find it a bit reassuring to remember two things. Most of the time just being well informed about best practices in memory allocation will keep you out of trouble. Second, there are some tools you can use to proactively assess the allocation status of memory. The Memory Dialog example (see Figure 6.1) shows you how to check the physical page size of a device, how to get a reliable ballpark estimate of memory usage status, and how to get a “read-in-place” pointer to string resources.

![Figure 6.1](image)

Figure 6.1  The Memory Dialog example.

Following is the main source file for MemoryDialog (see Listing 6.2). The support files have been omitted here for the sake of brevity, but you can find them on the accompanying CD.
// MemDlg.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include <windowsx.h>                 // common dialog includes
#include <commctrl.h>
#include "MemDlg.h"
#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE            hInst;            // The current instance
HWND                hwndCB;            // The command bar handle

// Forward declarations of functions included in this code module:
ATOM                MyRegisterClass    (HINSTANCE hInstance, LPTSTR
szWindowClass);
BOOL                InitInstance    (HINSTANCE, int);
LRESULT CALLBACK    WndProc            (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK MemStatusDlgProc(HWND hWnd,
UINT wMsg, WPARAM wParam, LPARAM lParam);

int WINAPI WinMain(    HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPTSTR    lpCmdLine,
int       nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }
    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_MEMDLG);
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return msg.wParam;
}

// FUNCTION: MyRegisterClass()
// PURPOSE: Registers the window class.

// This function and its usage is only necessary if you want this code

Listing 6.2 Source code for Memory Dialog.
// to be compatible with Win32 systems prior to the 'RegisterClassEx'
// function that was added to Windows 95. It is important to call
// this function
// so that the application will get 'well formed' small icons
// associated
// with it.

ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS wc;
    wc.style            = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc        = (WNDPROC) WndProc;
    wc.cbClsExtra        = 0;
    wc.cbWndExtra        = 0;
    wc.hInstance        = hInstance;
    wc.hIcon            = LoadIcon(hInstance,
MAKEINTRESOURCE(IDI_MEMDLG)));
    wc.hCursor            = 0;
    wc.hbrBackground    = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName        = 0;
    wc.lpszClassName    = szWindowClass;
    return RegisterClass(&wc);
}

// FUNCTION: InitInstance(HANDLE, int)
// PURPOSE: Saves instance handle and creates main window
// COMMENTS:
// In this function, we save the instance handle in a global variable
// and create and display the main program window.

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    TCHAR szTitle[MAX_LOADSTRING]; // The title bar text
    TCHAR szWindowClass[MAX_LOADSTRING]; // The window class name
    hInst = hInstance; // Store instance handle in our global
    // variable
    // Initialize global strings
    LoadString(hInstance, IDC_MEMDLG, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
0, 0, CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance,
NULL);
    if (!hWnd)
    {
Listing 6.2    Source code for Memory Dialog. (continues)
return FALSE;
}
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
    //report global memory status
    DialogBox(hInst, TEXT("MemStats"), hWnd, MemStatusDlgProc);
    return TRUE;
}

// FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
// PURPOSE:  Processes messages for the main window.
// WM_COMMAND    - process the application menu
// WM_PAINT    - Paint the main window
// WM_DESTROY    - post a quit message and return
//
// LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    LPCTSTR pszHello;
    switch (message)
    {
    case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_FILE_EXIT:
            DestroyWindow(hWnd);
            break;
        default:
            default:
                return DefWindowProc(hWnd, message, wParam, lParam);
        }
        break;
    case WM_CREATE:
        hwndCB = CommandBar_Create(hInst, hWnd, 1);
        CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
        CommandBar_AddAdornments(hwndCB, 0, 0);
        break;
    case WM_PAINT:
        RECT rt;
hdc = BeginPaint(hWnd, &ps);
GetClientRect(hWnd, &rt);
pszHello = (LPCTSTR)LoadString(hInst, IDS_HELLO, NULL, 12);
DrawText(hdc, pszHello, 12, &rt,
       DT_SINGLELINE | DT_VCENTER | DT_CENTER);
EndPaint(hWnd, &ps);
break;
case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;

// MemStatus - test / warn low memory
BOOL CALLBACK MemStatusDlgProc(HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam)
{
    BOOL bOk;
    DWORD dwLargest = 0;
    MEMORYSTATUS msMemStats;
    SYSTEM_INFO siSysInfo;
    HWND hWndList;
    INT i = 0;
    TCHAR szBuff[64];

    switch (wMsg)
    {
      case WM_INITDIALOG:
        hWndList = GetDlgItem(hWnd,IDC_MEM_STATS);
        memset(&siSysInfo, 0x0, sizeof( siSysInfo) );
        GetSystemInfo( (LPSYSTEM_INFO)&siSysInfo );
        _stprintf( szBuff, TEXT("  %s : %i"),
                    TEXT("Page Size"),
                    siSysInfo.dwPageSize);
        ListBox_InsertString(hWndList, -1, szBuff);
        memset(&msMemStats, 0x0, sizeof( msMemStats) );
        msMemStats.dwLength = sizeof( msMemStats);
        GlobalMemoryStatus( (LPMEMORYSTATUS)&msMemStats );
        _stprintf( szBuff, TEXT("  %s "),
                    TEXT("% Memory in use"));
        ListBox_InsertString(hWndList, -1, &szBuff);
Listing 6.2  Source code for Memory Dialog. (continues)
_stprintf( szBuff, TEXT("  %lu "),
           msMemStats.dwMemoryLoad);
ListBox_InsertString(hWndList, -1, &szBuff);
_stprintf( szBuff, TEXT(" %s"),
           TEXT("Total Phys. Memory") );
ListBox_InsertString(hWndList, -1, &szBuff);
_stprintf( szBuff, TEXT("  %lu"),msMemStats.dwTotalPhys );
ListBox_InsertString(hWndList, -1, &szBuff);
_stprintf( szBuff, TEXT(" %s"),
           TEXT("Available Phys. Memory") );
ListBox_InsertString(hWndList, -1, &szBuff);
_stprintf( szBuff,
           TEXT("  %lu "),msMemStats.dwAvailPhys);
ListBox_InsertString(hWndList, -1, &szBuff);
break;
case WM_COMMAND:
    switch (LOWORD (wParam))
    {
    case IDOK:
        EndDialog (hWnd, 1);
        return TRUE;

        default:
            break;
    }
break;
default:
    break;
}  //end switch (wMsg)
return FALSE;

Skip down to the message switch code in WinMain() and let’s look at the
WM_PAINT case. Notice the use of the CE style LoadString() implementation.

LPCTSTR pszHello;
//read string "in place" in the resource data segment
pszHello = (LPCTSTR)LoadString(hInst, IDS_HELLO,
       NULL, 12);
DrawText(hdc, pszHello, 12, &rt,
       DT_SINGLELINE | DT_VCENTER | DT_CENTER);

Listing 6.2  Source code for Memory Dialog. (continued)

When the third parameter to LoadString() is set to NULL, the function
returns a pointer to a constant string. It is important to cast the return value to
this type, because the function is declared as returning int. Also, make sure the
variable receiving the pointer is correctly typed, using the LPCTSTR typedef.
This technique allows you to access string resources without allocating buffer
space for them. Notice the string length is hard coded. You can retrieve the
length of the string, in the word preceding the string. This length does not include the terminal null:

```c
// the length of the string, excluding the terminal null
// is in the , word before the string
iStrLen = *( pszHello - sizeof( WORD ) );
```

The real business of the MemoryDialog example is in—you guessed it—the dialog procedure, MemStatusDlgProc(). Here we employ two very useful functions for gathering information about the state of memory, GetSystemInfo() and GlobalMemoryStatus(). Here’s how to use GetSystemInfo() to find the page size on a CE device:

```c
SYSTEM_INFO siSysInfo;
memset(&siSysInfo, 0x0, sizeof( siSysInfo ) );
GetSystemInfo( (LPSYSTEM_INFO)&siSysInfo );
```

The single parameter to GetSystemInfo() is a SYSTEM_INFO structure. Notice that we explicitly initialize this before passing it to the structure. Since siSysInfo is allocated on the stack, this step ensures we get clean data back from the function. The only information we use in this example is the system page size. The page size is returned in siSysInfo.dwPageSize. If you plan to make allocations with VirtualAlloc() or HeapCreate(), use siSysInfo.dwPageSize to optimize the size of the allocation request (see Table 6.1).

However, should you want it, the SYSTEM_INFO structure can provide you with very precise detail about the device’s processor.

Table 6.1 SYSTEM_INFO Members and Their Meanings

<table>
<thead>
<tr>
<th>MEMBER TYPE AND NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>union {</td>
<td></td>
</tr>
<tr>
<td>DWORD dwOemId;</td>
<td>Obsolete field</td>
</tr>
<tr>
<td>struct {</td>
<td></td>
</tr>
<tr>
<td>WORD wProcessorArchitecture;</td>
<td>PROCESSOR_ARCHITECTURE_INTEL</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_ARCHITECTURE_MIPS</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_ARCHITECTURE_ALPHA</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_ARCHITECTURE_PPC</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_ARCHITECTURE_SHx</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_ARCHITECTURE_ARM</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_ARCHITECTURE_UNKNOWN</td>
</tr>
</tbody>
</table>

(continues)
Table 6.1 .SYSTEM_INFO Members and Their Meanings (Continued)

<table>
<thead>
<tr>
<th>MEMBER TYPE AND NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD wReserved;</td>
<td></td>
</tr>
<tr>
<td>DWORD dwPageSize;</td>
<td>Smallest allocation unit, in bytes</td>
</tr>
<tr>
<td>LPVOID lpMinimumApplicationAddress;</td>
<td>Minimum virtual address for this application</td>
</tr>
<tr>
<td>LPVOID lpMaximumApplicationAddress;</td>
<td>Minimum virtual address for this application</td>
</tr>
<tr>
<td>DWORD dwActiveProcessorMask;</td>
<td>Ignored under CE</td>
</tr>
<tr>
<td>DWORD dwNumberOfProcessors;</td>
<td>Ignored under CE</td>
</tr>
<tr>
<td>DWORD dwProcessorType;</td>
<td>PROCESSOR_INTEL_386</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_INTEL_486</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_INTEL_PENTIUM</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_INTEL_860</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_MIPS_R2000</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_MIPS_R3000</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_MIPS_R4000</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_HITACHI_SH3</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_ALPHA_21064</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_PPC_601</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_PPC_603</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_PPC_604</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_PPC_620</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_PPC_821</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_SHx_SH3</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_SHx_SH4</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_STRONGBARM</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_ARM720</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_ARM8200</td>
</tr>
<tr>
<td></td>
<td>PROCESSOR_ARM920</td>
</tr>
</tbody>
</table>
Table 6.1  (Continued)

<table>
<thead>
<tr>
<th>MEMBER TYPE AND NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD dwAllocationGranularity;</td>
<td>Bondaries to which virtual memory regions are rounded. On CE this value is 64K.</td>
</tr>
<tr>
<td>WORD wProcessorLevel;</td>
<td>Defines a specific processor within the family identified by \textit{dwProcessorType}</td>
</tr>
<tr>
<td>WORD wProcessorRevision;</td>
<td>Model number and stepping for the processor</td>
</tr>
</tbody>
</table>

Estimating Available Memory

You can get a fairly good, useful rough estimate of current memory use with \texttt{GlobalMemoryStatus()}. This is especially useful if your application is about to undertake a memory intensive operation. In the following code fragment, we use it to get a percentage estimate of memory use:

```c
MEMORYSTATUS msMemStats;
memset(&msMemStats, 0x0, sizeof(msMemStats));
msMemStats.dwLength = sizeof(msMemStats);
GlobalMemoryStatus((LPMEMORYSTATUS)&msMemStats);

_stprintf(szBuff, TEXT("  %s "), TEXT("% Memory in use"));
ListBox_InsertString(hWndList, -1, &szBuff);

_stprintf(szBuff, TEXT("  %lu "), msMemStats.dwMemoryLoad);
ListBox_InsertString(hWndList, -1, &szBuff);
```

The \texttt{GlobalMemoryStatus()} function is really mostly useful only at application launch, as the percentage of memory use is kind of a “snapshot” of the memory situation when the application is loaded.

Notice the \texttt{_stprintf()} function that is used to format the listbox strings. \texttt{_stprintf()} is a version of the more familiar \texttt{sprintf()}, but it handles the wide character string we must use for control text under CE.

Looking Ahead

Memory management is the foundation upon which every other sort of CE application behavior is built. In this chapter, we’ve covered what you need to
know to make informed decisions about memory allocation and optimizing the executable file. Now we are ready to turn our attention to the real work of application software. In the upcoming chapters, we will undertake jobs like creating and using persistent storage, the registry, and keeping an eye on power consumption. And we’ll continue to look for ways to keep the memory footprint of applications as small as possible.
From an application developers’ point of view, the registry is the place where application-specific state information is stored between executions. Predictably, the Windows CE registry is a leaner version of its desktop cousin. This means that you may want to reevaluate the kind and amount of data your application stores in the registry.

The most convenient way to become familiar with the structure of the CE registry is with the Windows CE Remote Registry Editor. To use the Remote Registry Editor to examine the registry of a specific device, you must add your device to the editor’s list. Select the Registry menu’s Add Device command, and choose a device from the list displayed in the Select A Device dialog.

The Remote Registry Viewer (see Figure 7.1) behaves in a fashion similar to its desktop counterpart, RegEdit. Let’s use it to take a closer look at the structure and content of the CE registry.
Notice that when the registry listing for the handheld PC is expanded, there are only three root keys: HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, and HKEY_LOCAL_MACHINE (see Figure 7.2). All the examples in this chapter will deal with adding, querying, and modifying keys and values under HKEY_LOCAL_MACHINE. This is the conventional location in which to store applications’ state information.
Windows CE Registry Access Functions

There are two basic groups of functions for accessing registry data on a CE device: the ones that access data in place and the ones that access the registry of the device from the desktop. We’ll be looking into the desktop registry access functions in Chapter 9, “A CE Database Primer.” For now, be aware that the remote access functions’ names are the same as the “in place” functions, except that they are prefixed with the letters “Ce.” For example, the CE side function for accessing a registry key looks like this:

\[
\text{RegQueryValueEx}(h\text{Key}, \text{lpValueName}, \text{lpReserved}, \text{lpType}, \text{lpData}, \text{lpcbData});
\]

To perform the same operation on a connected device from an application executing on the desktop, you would use this function instead of the one shown above:

\[
\text{CeRegQueryValueEx}(h\text{Key}, \text{lpValueName}, \text{lpReserved}, \text{lpType}, \text{lpData}, \text{lpcbData});
\]

And now, on to the business of designing and using the registry, CE style.

Accessing the Registry on a CE Device

Some people avoid using the registry at all, being intimidated by the implications of a misstep. Certainly, corrupting the registry could create serious problems. However, when all is said and done, the CE registry is simply a database. Like any database, we need to treat the registry in a way that ensures its consistency, keeps it as compact as possible, and accesses it in the most efficient fashion. I make this point because on the desktop, the registry has become more like a five-family garage sale with every passing year. On CE, it’s important to be organized and thrifty when deciding what data an application registers. Use the registry sparingly and efficiently. If your Win32 application stores information in the registry that isn’t specifically necessary to re-create the previous application startup state, consider eliminating these items or devising another way to get the same information with runtime queries.

Most especially, you’ll want to consider redesigning deeply nested or widely branched registry data structures. Under CE, significant size and performance penalties occur when accessing deeply nested keys. For this reason, try to make registry trees as flat as possible. Also, since registry access in general is a fairly expensive operation, another way of optimizing registry data storage is to aggregate individual named key values into a single element.
Using this strategy, your application makes one call to retrieve registry information and parses data items from the aggregate. This approach has the dual advantages of reducing both the number of key accesses and the storage space used by the values.

**PORTING TIP** Redesigning the structure of your registry tree may give your application a big performance boost.

- There is a storage and performance advantage to making registry trees as flat as possible.
- Combine multiple-named values into a single named value element that is parsed by the application, so you can retrieve startup information in a single key access.

It’s time to explore registry manipulations. In the RegDemo example, you’ll see how to create, open, write, read, and delete keys and their associated values. We’ll also see how you can use a single registry entry to store a variety of items of data. The RegDemo example is designed to be used in combination with the Remote Registry Editor. As you use the example menu choices, try inspecting the registry of the device to observe the changes in keys and values.

As in previous examples, Listing 7.1 includes only the main source file RegDemo.cpp here. You can find the complete body of sources on the accompanying CD.

```c
// RegDemo.cpp : Defines the entry point for the application.

//
#include "stdafx.h"
#include "RegDemo.h"
#include <commctrl.h>
#include <windows.h> // For all that Windows stuff
#include <windowsx.h> // More & better Windows stuff
#include <commdlg.h> // Command bar includes
#include <tchar.h> // Unicode Conversion includes
#include <winreg.h> // registry stuff

#define MAX_LOADSTRING 100
//a handy macro for finding array dimensions
#define dim(x) (sizeof(x) / sizeof(x[0]))

// Global Variables:
HINSTANCE hInst; // The current instance
HWND hwndCB; // The command bar handle

// Forward declarations of functions included in this code module:
ATOM MyRegisterClass (HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);

Listing 7.1 RegDemo.cpp.
```
BOOL CALLBACK HKeyLocalSubKeysDlg(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK HKeyN2ValuesDlg(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    // Perform application initialization:
    if (!InitInstance(hInstance, nCmdShow))
    {
        return FALSE;
    }
    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_REGDEMO);
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return msg.wParam;
}

// FUNCTION: MyRegisterClass()
// PURPOSE: Registers the window class.
// COMMENTS:
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_REGDEMO));
    wc.hCursor = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = 0;
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);

Listing 7.1 RegDemo.cpp. (continues)
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    TCHAR szTitle[MAX_LOADSTRING]; // The title bar text
    TCHAR szWindowClass[MAX_LOADSTRING]; // The window class name
    // Store instance handle in our global variable
    hInst = hInstance;
    // Initialize global strings
    LoadString(hInstance, IDC_REGDEMO, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle,
                        WS_VISIBLE,
                        0, 0,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        NULL, NULL,
                        hInstance, NULL);
    if (!hWnd)
    {
        return FALSE;
    }
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    if (hwndCB)
        CommandBar_Show(hwndCB, TRUE);
    return TRUE;
}

// FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
// PURPOSE: Processes messages for the main window.
// WM_COMMAND - process the application menu
// WM_DESTROY - post a quit message and return
//
// LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    HKEY hKeyN2;
    DWORD dwDisp;
    DWORD dwRegVal = 0x1234;
    INT i = 0, rc;
    BYTE pBuff;
    DWORD dwValType, dwDSize;

Listing 7.1  RegDemo.cpp.
PBYTE pbData;
TCHAR wszMessage[24];
switch (message) {
    case WM_COMMAND:
        wmId = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId) {
            case IDM_ADD_KEY:
                //if !n2
                //reg me
                //else open my key
                rc = RegCreateKeyEx (HKEY_LOCAL_MACHINE,
                                      TEXT("Software\n2"), 0,
                                      TEXT(""), 0, 0, NULL,
                                      &hKeyN2, &dwDisp);
                if (rc == ERROR_SUCCESS)
                {
                    MessageBox (NULL, TEXT("Software\n2"),
                                TEXT("Key Registered"), MB_OK);
                }
                else
                {
                    MessageBox (NULL, TEXT("RegCreateKeyEx Failed"),
                                TEXT("No Key Registered"), MB_OK);
                    return 0;
                }
                //clean up after ourselves
                RegCloseKey( hKeyN2 );
                break;
            case IDM_READ_KEY:
                // Dialog shows the list of
                // HKEY_LOCAL_MACHINE subkeys.
                DialogBox (hInst, TEXT ("HKEY_LOCAL_DLG"),
                            hWnd, HKeyLocalSubKeysDlg);
                break;
            case IDM_ADD_SINGLE_VALUE:
                //open the key to gain access to the value
                rc = RegOpenKeyEx (HKEY_LOCAL_MACHINE,
                                    TEXT("Software\n2"), 0,
                                    KEY_SET_VALUE, &hKeyN2);
                if (rc != ERROR_SUCCESS)
                {
                    MessageBox (NULL, TEXT("RegOpenKeyEx Failed"),
                                TEXT("Key n2 not opened"),
                                MB_OK);
                    return FALSE;
                }
// set a name and value under the open key
rc = RegSetValueEx (hKeyN2, TEXT("OneDWORD"), 0,
REG_DWORD,
(LPBYTE)&dwRegVal,
sizeof(dwRegVal));

if (rc != ERROR_SUCCESS)
{
    MessageBox (NULL,
        TEXT("RegSetValueEx Failed"),
        TEXT("Couldn't delete value"),
        MB_OK);
}
else
{
    MessageBox (NULL, TEXT("Value OneDWORD Set"),
        TEXT("Single DWORD Registered"),
        MB_OK);
}
// always clean up after ourselves
RegCloseKey( hKeyN2 );
bearing;
case IDM_ADD_BYTEARRAY_VALUE:
    // allocate some space for an aggregate of data
    pBuff = (BYTE*)LocalAlloc(LPTR, 24);
    if(!pBuff )
    {
        MessageBox (NULL,
            TEXT("LocalAlloc Failed"),
            TEXT("Fail and Bail!"), MB_OK);
        return FALSE;
    }
    // fill it with 0xff for the heck of it
    memset( pBuff, 0xff, 24);
    // open the key to gain access to values
    rc = RegOpenKeyEx (HKEY_LOCAL_MACHINE,
                TEXT ("Software\n2"), 0,
                KEY_SET_VALUE, &hKeyN2);
    // set the binary value
    rc = RegSetValueEx (hKeyN2, TEXT ("Aggregate"), 0,
                    REG_BINARY, (PBYTE)pBuff,
                    LocalSize( (HLOCAL)pBuff ));
    if (rc != ERROR_SUCCESS)
    {
        MessageBox (NULL,
            TEXT("RegSetValueEx Failed"),
            TEXT("No value set"), MB_OK);
        return FALSE;
    }

Listing 7.1 RegDemo.cpp.
Listing 7.1  RegDemo.cpp. (continues)
// open key to gain access to values
rc = RegOpenKeyEx (HKEY_LOCAL_MACHINE,
     TEXT ("Software\n2"), 0,
     KEY_ALL_ACCESS, &hKeyN2);
if(rc != ERROR_SUCCESS )
    { return FALSE; } // delete named value
rc =  RegDeleteValue( hKeyN2, TEXT("OneDWORD") );
if(rc != ERROR_SUCCESS )
    {
        MessageBox (NULL, TEXT("OneDWORD"),
        TEXT("Couldn't Delete Value"),
        MB_OK);
        RegCloseKey( hKeyN2 );
        return FALSE;
    } // clean up now
RegCloseKey( hKeyN2 );
break;
case IDM_DELETE_KEY:
    // wipe reg keys and their values
rc = RegDeleteKey(HKEY_LOCAL_MACHINE,
     TEXT ("Software\n2"));
    break;
case IDM_FILE_EXIT:
    DestroyWindow(hWnd);
    break;
default:
    return DefWindowProc(hWnd, message,
     wParam, lParam);
} break;
case WM_CREATE:
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    CommandBar_AddAdornments(hwndCB, 0, 0);
    break;
case WM_DESTROY:
    // wipe reg keys
    rc = RegDeleteKey(HKEY_LOCAL_MACHINE,
     TEXT ("Software\n2"));
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
} return 0;
}
BOOL CALLBACK HKeyLocalSubKeysDlg(HWND hDlg, UINT message, 
    WPARAM wParam, LPARAM lParam)
{
    HWND hWndList;
    INT i = 0, rc;
    DWORD dwNSize;
    DWORD dwCSize;
    TCHAR szName[MAX_PATH];
    TCHAR szClass[256];
    FILETIME ft;
    HKEY hKeyN2;
    static TCHAR* lpszDBName;

    hWndList = GetDlgItem(hDlg, IDC_SUBKEY_LIST);
    switch (message)
    {
    case WM_INITDIALOG:
        //open key to gain access to subkeys
        rc = RegOpenKeyEx(HKEY_LOCAL_MACHINE, TEXT("\"), 0,
            KEY_ENUMERATE_SUB_KEYS, &hKeyN2);
        if (rc == ERROR_SUCCESS)
        {
            dwNSize = dim(szName);
            dwCSize = dim(szClass);
            //set up enumeration of the key's subkeys
            rc = RegEnumKeyEx(HKEY_LOCAL_MACHINE, i, szName,
                &dwNSize, NULL,
                szClass, &dwCSize, &ft);
            while (rc == ERROR_SUCCESS)
            {
                ListBox_InsertString(hWndList, i, szName);
                dwNSize = dim(szName);
                rc = RegEnumKeyEx(HKEY_LOCAL_MACHINE, ++i, szName,
                    &dwNSize, NULL, NULL, 0, &ft);
            }
        } //end if( ERROR_SUCCESS )
        break;
    case WM_COMMAND:
        switch (LOWORD(wParam))
        {
        case IDCANCEL:
            EndDialog(hDlg, 0);
            return TRUE;
        default:
            break;
        }
        break;
    }
}

Listing 7.1  RegDemo.cpp. (continues)

Using the Windows CE Registry  267
default:
    break;
}//end switch (wMsg)

return FALSE;
}

BOOL CALLBACK HKeyN2ValuesDlg(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    HWND hWndList;
    INT i = 0, rc, iTest;
    DWORD dwNSize;
    DWORD dwDSize;
    DWORD dwType;
    DWORD dwDatum;
    TCHAR szName[32];
    HKEY hKeyN2;
    BYTE bData[26];
    BYTE bBuffer[48];

    switch (message)
    {
    case WM_INITDIALOG:
        hWndList = GetDlgItem(hDlg,IDC_VALUE_LIST);
        //open key to gain access to values
        rc = RegOpenKeyEx(HKEY_LOCAL_MACHINE, TEXT("Software\n2"), 0, KEY_ENUMERATE_SUB_KEYS, &hKeyN2);
        if (rc != ERROR_SUCCESS)
        {return FALSE;}
        //set up to enumerate values
        dwNSize = dim(szName);
        dwDSize = sizeof(bData);
        rc = RegEnumValue(hKeyN2, i, szName, &dwNSize, NULL, &dwType, (PBYTE)&bData, &dwDSize);
        while (rc == ERROR_SUCCESS)
        {
            ListBox_InsertString(hWndList, -1, szName);
            //distinguish between key types
            memset(bBuffer, 0x0, sizeof(bBuffer));
            if( dwType == REG_BINARY )
            {
                wsprintf((LPTSTR)&bBuffer, TEXT("  %s Data:%x "), TEXT("REG_BINARY"), *bData);
            }
            if( dwType == REG_DWORD )
            {
                dwDatum = (DWORD)(bData);
                iTest = (int)dwDatum;
            }
            rc = RegEnumValue(hKeyN2, iTest, szName, &dwNSize, NULL, &dwType, (PBYTE)&bData, &dwDSize);
        }
    Listing 7.1 RegDemo.cpp.
268 Chapter 7
Adding Registry Keys

You can add registry keys with either the RegOpenKeyEx() or RegCreateKeyEx() function. The advantage of using RegCreateKey() is that it creates a key as well as opening it if the key doesn’t already exist. In the RegDemo example, we use RegCreateKeyEx() in response to the IDM_ADD_KEY message.

case IDM_ADD_KEY:
    //if !n2
    //reg me
    //else open my key
    rc = RegCreateKeyEx (HKEY_LOCAL_MACHINE,
                          TEXT("Software\n2"), 0,
                          TEXT(""), 0, 0, NULL,
                          &hKeyN2, &dwDisp);

Listing 7.1  RegDemo.cpp. (continued)
The parameters to RegCreateKey, in the order shown, are the handle to the parent key; the name, as a Unicode string, of the key to add; a NULL placeholder for a reserved parameter; the class name of the key as a Unicode string; three more placeholders set to 0,0 and NULL, respectively; the address of a variable to receive the returned handle if the key is created successfully; and the address of a Variable to receive the disposition of the call.

Notice that the value of the first parameter is HKEY_LOCAL_MACHINE, which corresponds to one of the root key names in the registry. Used in this context, HKEY_LOCAL_MACHINE identifies a reserved handle to this key, and can be treated just as though it was returned by RegCreateKey() or RegOpenKey(). There are four reserved key handles you may use to access the top level of the CE registry (See Table 7.1).

<table>
<thead>
<tr>
<th>HANDLE NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>HKEY_CLASSES_ROOT</td>
</tr>
<tr>
<td>HKEY_CURRENT_USER</td>
</tr>
<tr>
<td>HKEY_LOCAL_MACHINE</td>
</tr>
<tr>
<td>HKEY_USERS</td>
</tr>
</tbody>
</table>

Now take a look at the second parameter:

```text(TEXT("Software\n2"),)
```

Notice that we have descended two levels in the registry tree in order to create and open the key “n2”. You don’t have to traverse the registry’s hierarchy one step at a time if you know the full name of the key you want to open.

Finally, make note of the parameter dwDisp. After a successful return, dwDisp will contain one of two values: REG_CREATED_NEW_KEY or REG_OPENED_EXISTING_KEY. Use this value to detect whether or not the key existed before the call.

Skipping over the message box code, we finish the handling of this case by closing the key handle. This is a very important step because leaving the key open may cause subsequent access to fail:

```c
//clean up after ourselves
RegCloseKey( hKeyN2 );
break;
```
Accessing and Enumerating Existing Keys

There are two ways of retrieving the information stored in keys: You can read the keys individually or you can enumerate them, looping until a particular key’s supply of subkeys is exhausted. We do both in initializing the HkeyLocal SubKeysDlg(). First, take a look at the call to RegOpenKeyEx():

```c
case WM_INITDIALOG:
    //open key to gain access to subkeys
    rc = RegOpenKeyEx (HKEY_LOCAL_MACHINE, TEXT ("\"), 0,
                        KEY_ENUMERATE_SUB_KEYS, &hKeyN2);
```

The parameters, in the order shown, are the handle to the parent key, the name of the key we want to open, the requested access mode to the key we are opening, and the address of the variable to receive the returned handle. As you saw earlier, it isn’t strictly necessary to open the HKEY_LOCAL_MACHINE root key. Notice two things here, however: In the second parameter, we set an empty subkey name, and in the next parameter, we set the access mode KEY_SET_VALUE. The value of the access mode determines what you can do with and to the opened key in subsequent manipulations using the returned handle. Table 7.2 is a list of key access modes and their meanings.

<table>
<thead>
<tr>
<th>ACCESS MODE CONSTANT</th>
<th>EFFECT OF SETTING THIS ACCESS MODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_ALL_ACCESS</td>
<td>Equivalent to KEY_QUERY_VALUE</td>
</tr>
<tr>
<td>KEY_QUERY_VALUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>KEY_CREATE_LINK</td>
<td>Allows creation of a symbolic link</td>
</tr>
<tr>
<td>KEY_CREATE_SUB_KEY</td>
<td>Allows creation of subkeys</td>
</tr>
<tr>
<td>KEY_ENUMERATE_SUB_KEYS</td>
<td>Allows enumeration of subkeys</td>
</tr>
<tr>
<td>KEY_EXECUTE</td>
<td>Allows read access</td>
</tr>
<tr>
<td>KEY_NOTIFY</td>
<td>Allows change notification</td>
</tr>
<tr>
<td>KEY_QUERY_VALUE</td>
<td>Allows query of subkey data</td>
</tr>
</tbody>
</table>

Table 7.2 Key Access Modes (continues)
As a matter of defensive programming, it’s best to set an access mode that allows no more than the capability required to accomplish what you intend.

If we manage to open the root key successfully, we set up to enumerate its subkeys. First, recall that at the entry to the HkeyLocalSubKeysDlg() function, we initialize the variable i:

```
INT i = 0, rc;
```

Next, we check the return from the key open call to make sure we don’t use an invalid handle in the subsequent calls:

```
if (rc == ERROR_SUCCESS)
{
```

We initialize the parameters that give the size of the arrays for the key name and the key’s class using the dim macro we defined at the top of this source file. Included below is the definition of dim:

```
//a handy macro for finding array dimensions
#define dim(x) (sizeof(x) / sizeof(x[0]))
dwNSize = dim(szName);
dwCSize = dim(szClass);
```

Now we begin the enumeration at the key with an index of 0:

```
//set up enumeration of the key's subkeys
rc = RegEnumKeyEx (HKEY_LOCAL_MACHINE, i, szName,
                  &dwNSize, NULL,
                  szClass, &dwCSize, NULL);
```

The parameters to RegEnumKeyEx(), in the order shown, are the handle to the key we are enumerating, the key’s index, an array of TCHARS to hold the enumerated key’s name, the size of this array in characters, including the terminal null, a NULL placeholder, an array of TCHARS to hold the enumerated
key’s class name, the size of the class buffer, and a final NULL placeholder. Before we go on, note the usefulness of the dim macro here. You can’t use sizeof() to set the buffer sizes in this call, because that returns the size in bytes, which is not the same as the size in characters when strings are in Unicode.

**PORTING TIP** If registry access code shows symptoms of memory overwrites, double-check function calls that have buffer size parameters in character counts. Make sure sizes are expressed as Unicode character counts that include a terminal null.

As we continue the enumeration, we retrieve key names but ignore their class name information. We do this by setting the class name buffer and its size to NULL:

```c
while (rc == ERROR_SUCCESS)
{
    ListBox_InsertString(hWndList, i, szName);
    dwNSize = dim(szName);
    rc = RegEnumKeyEx (HKEY_LOCAL_MACHINE, ++i, szName,
                       &dwNSize, NULL, NULL, 0, NULL);
}
```

When we are done enumerating the subkeys, we close the handle we opened above:

```c
//clean up now
RegCloseKey( hKeyN2 );
} //end if( ERROR_SUCCESS )
```

### Adding Values to Keys

In order to add values to a key, you must open it with at least KEY_SET_VALUE access permissions:

```c
case IDM_ADD_SINGLE_VALUE:
    //open the key to gain access to the value
    rc = RegOpenKeyEx (HKEY_LOCAL_MACHINE,
                        TEXT ("Software\\n2"), 0,
                        KEY_SET_VALUE, &hKeyN2);
```

If all goes well, you are presented with great flexibility in regards to what gets stored in the value. Table 7.3 lists the most useful key types supported by Windows CE.
Table 7.3 Windows CE Registry Key Data Types

<table>
<thead>
<tr>
<th>REGISTRY KEY DATA TYPES</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_BINARY</td>
<td>Byte string data</td>
</tr>
<tr>
<td>REG_DWORD</td>
<td>A 32-bit number</td>
</tr>
<tr>
<td>REG_DWORD_LITTLE_ENDIAN</td>
<td>A 32-bit number in little-endian format; same as REG_DWORD</td>
</tr>
<tr>
<td>REG_DWORD_BIG_ENDIAN</td>
<td>A 32-bit number in big-endian format</td>
</tr>
<tr>
<td>REG_MULTI_SZ</td>
<td>An array of null-terminated strings, terminated by two null characters</td>
</tr>
<tr>
<td>REG_SZ</td>
<td>A null-terminated string</td>
</tr>
</tbody>
</table>

You should use the smallest registry type that will accommodate the data you plan to store. In this case, we add a single, named value of type REG_DWORD. At the top of the function we initialized the DWORD dwRegVal in its declaration:

```c
DWORD dwRegVal = 0x1234;
//set a name and value under the open key
rc = RegSetValueEx (hKeyN2, TEXT("OneDWORD"), 0,
REG_DWORD , (LPBYTE)&dwRegVal,
sizeof(dwRegVal));
```

The parameters to RegSetValueEx(), in the order shown, are the handle to the key receiving the value, the name of the value, the type, the address of the first byte of the buffer where the data is stored, and the size of the data in bytes. Notice that we use sizeof() to set the buffer size in this case. This is a good practice, because if the data in this key were one of string types, sizeof() would automatically account for the space taken up by a NULL terminator.

Finally, skipping over the message box code, we close the key:

```c
//always clean up after ourselves
RegCloseKey( hKeyN2 );
break;
```

What if you have several items of registry data under a key? You could add a named value for each item, but that approach would use a lot of space to store the value names and require a registry access to get or set each value. If your data are all strings, you can use the REG_MULTI_SZ type, and combine all of the strings into a single named value. If the data are of mixed type, consider aggregating them and storing them in a single binary valued key. Your application can retrieve the array in a single registry access and parse them
much more quickly than if they were retrieved individually. Here is how we set a binary array as a registry value in RegDemo:

```c
//set the binary value
rc = RegSetValueEx (hKeyN2, TEXT("Aggregate"),
  0, REG_BINARY, (PBYTE)pBuff,
  LocalSize((HLOCAL)pBuff));
```

The Buffer identified by pBuff was allocated by LocalAlloc(), so the most reliable way of getting its size is with LocalSize(). The data is written to the registry as a string of bytes, which exactly preserves the order of the bytes as they were written. This means that if your application stores a mixture of types (int, float, char, and so on) in the byte array, you’ll have to take care to read the items out of the string using pointers that are properly cast to the types of data being transferred.

### Enumerating Registry Values

By its nature, the content of the registry is somewhat dynamic. Used in the conservative fashion, the values of your application’s registry keys will depend on the state the application was in when it was last used. Keys have a possible set of values, which may or may not be present. To find out which values are present, you use RegEnumValue() to enumerate the value set of a key. In the RegDemo example, we do this in the HKeyN2ValuesDlg(). Let’s dissect the value enumeration process:

First, you must open the key that owns the value with at least KEY_ENUMERATE_SUB_KEYS access permission.

```c
rc = RegOpenKeyEx (HKEY_LOCAL_MACHINE,
  TEXT("Software\n2"), 0,
  KEY_ENUMERATE_SUB_KEYS, &hKeyN2);
```

If we successfully open the key, we set up for enumeration. Notice that value of the variable `i` was initialized at the top of the function:

```c
INT i = 0, rc;
```

Next, we initialize `dwNSize` with the size, in Unicode characters, of the value name buffer. We initialize `dwDSize` with the size, in bytes, of the value data buffer:

```c
//set up to enumerate values
dwNSize = dim(szName);
dwDSize = sizeof(bData);
Now we make the initial call to RegEnumValue:

```c
rc = RegEnumValue(hKeyN2, i, szName, &dwNSize, NULL,
   &dwType, (PBYTE)&bData, &dwDSize );
```

The parameters, in the order shown, are the handle to the key that owns this value, the index of the value to retrieve, a buffer to hold the returned name of the value, the size of the buffer in Unicode characters, including the terminal null, a NULL place holder, the address of a DWORD in which the registry data type of this value is returned, the address of the data buffer, and the size of the data buffer in bytes. We capture the return of this call in the variable rc, and test it to determine when we have reached the end of the value list for this key. We loop over all of the values in order to initialize the list box:

```c
while (rc == ERROR_SUCCESS)
{
```

First, we insert the value name. The name was returned in a Unicode string, so can be directly inserted into the list box:

```c
ListBox_InsertString(hWndList, -1, szName);
```

The key data is returned a byte string, and it is essentially “typeless” when we get it. In order to use the data, we have to discriminate its type, and then use a type cast to correctly translate the bytes into the proper format for their given type:

```c
//distinguish between key types
memset( bBuffer, 0x0, sizeof(bBuffer));
if( dwType == REG_BINARY )
{
    wsprintf( (LPTSTR)&bBuffer,
            TEXT("%s Data:%x "),
            TEXT("REG_BINARY"), *bData );
}
```

In the call to wsprintf() shown above, we print the first byte of the data buffer, two hex digits, into the string that will be inserted into the list box. In this case we simply provide the address of the first byte of the data buffer. If we wanted to print more of the hex digits, we could use a call like this:

```c
//just a code fragment, not part of RegDemo
wsprintf( (LPTSTR)&bBuffer,TEXT("%s Data:%x%x%x"),
            TEXT("REG_BINARY"), *bData, *bData[1],
            *bData[2] );
```
When we want to manipulate other registry data types, we need to take a little more care:

```c
if( dwType == REG_DWORD )
{
    dwDatum = (DWORD)bData;
    iTest = (int)dwDatum;
    wsprintf( (LPTSTR)&bBuffer,
        TEXT(" %s Data:%x "),
        TEXT("REG_DWORD"), *(int*)bData );
}
```

Notice that in the wsprintf() call that displays the REG_DWORD data, we first cast the address of the data buffer to an integer pointer, and then dereference that pointer. Most numeric types are stored in memory with the low order bytes stored at the lowest address (LOWORD/HIWORD). For this reason, if you simply copy the correct number of bytes out of the data buffer and into a variable, you won't get the correct value:

```c
ListBox_InsertString(hWndList, -1, bBuffer);
dwNSize = dim(szName);
dwDSize = sizeof(bData);
rc = RegEnumValue(hKeyN2, ++i, szName,
    &dwNSize, NULL,
    &dwType, (PBYTE)&bData, &dwDSize );
```

We increment the variable i on each call to RegEnumValue() and continue looping until the function return indicates there are no more values.

### Reading a Single Named Value

If you know a value is present and you know its name, reading a single value allows you to be a little more precise. We do this in the WndProc() message switch, in response to the IDM_READ_ONE_VAL message:

```c
The first step is to open it's parent key with at least KEY_QUERY_VALUE access.

        case IDM_READ_ONE_VAL:
            //open the keys to gain access to the values
            rc = RegOpenKeyEx (HKEY_LOCAL_MACHINE,
                TEXT("Software\n2"), 0,
                KEY_QUERY_VALUE, &hKeyN2);
```
Before we actually read the value’s data, we can query the key to see how much data it has. Here’s how to do this:

```c
//How much data do we have
dwDSize = 1; // pass non null to get array size
dwValType = REG_DWORD;
rc = RegQueryValueEx (hKeyN2, TEXT("OneDWORD"),
                    NULL, &dwValType,
                    NULL,  &dwDSize);
```

The parameters to RegQueryValueEx(), in the order shown, are the handle to the parent key, the value name as a Unicode string, a NULL place holder, the address of a DWORD that will receive the registry data type of this value, a pointer to the buffer to receive the value data, and the address of a DWORD that will receive the size in bytes of the value’s data. Notice that in this call, we set the pointer to the data buffer to NULL. This causes the function to return the size of the value’s data, without returning the data itself. This allows you to allocate a buffer for the exact size of the data rather than reserving space for the largest possible amount of data:

```c
//allocate space
pbData = (PBYTE)LocalAlloc(LPTR,dwDSize);
if(!pbData )
{
    MessageBox (NULL, TEXT("LocalAlloc Failed"),
                TEXT("Fail and Bail!")), MB_OK);
    return FALSE;
}
//Get the key value
rc = RegQueryValueEx (hKeyN2, TEXT("OneDWORD"),
                    NULL, &dwValType,
                    pbData,  &dwDSize);
```

The next time we call RegQueryValueEx(), we pass the pointer to the newly allocated data buffer and retrieve the value data:

```c
if(rc == ERROR_SUCCESS )
{
    wsprintf(wszMessage, TEXT("%s: %x"),
            TEXT("OneDWORD"),
            *(int*)pbData );
    MessageBox (NULL, wszMessage,
                TEXT("Value Retrieved"), MB_OK);
    return FALSE;
}
```
Notice once more that when we copy the returned value data out of the buffer, we first cast the address of the data to an integer pointer, and then dereference. Finally, we close the key:

```c
RegCloseKey( hKeyN2 );
break;
```

## Deleting Values

Deleting a registry entry is fairly straightforward. We do this in the WndProc() message switch in response to the IDM_DELETE_VALUES message. First, open the key that owns the value. In this case we open with KEY_ALL_ACCESS permission, to ensure that we can delete it:

```c
//open key to gain access to values
rc = RegOpenKeyEx (HKEY_LOCAL_MACHINE,
                   TEXT("Software\n2"), 0,
                   KEY_ALL_ACCESS, &hKeyN2);
if(rc != ERROR_SUCCESS )
    { return FALSE; }
```

In order to delete a specific value, you must know its name:

```c
//delete named value
rc =  RegDeleteValue( hKeyN2,
                   TEXT("OneDWORD") );
```

The parameters, in the order shown, are the handle to the parent key, and the name of the value as a Unicode string. Skipping over the message box code, the last step is to close the key that owned the value:

```c
RegCloseKey( hKeyN2 );
return FALSE;
```

## Deleting Keys

Deleting keys is like pruning a tree: Everything farther out the branch is lopped off when you nip the key. This is a powerful way of cleaning a registry tree but has to be used with caution and respect. We do this in the WndProc() message switch in response to the IDM_DELETE_KEY message:
case IDM_DELETE_KEY:
    // wipe reg keys and their values
    rc = RegDeleteKey(HKEY_LOCAL_MACHINE,
                        TEXT ("Software\n2");
    break;

Notice that we don’t open the key. Keys may not be deleted unless they are closed. The parameters to RegDeleteKey(), in the order shown, are the handle to the parent key, and the name of the key to delete as a Unicode string. In this case, the key “n2”, all its values, and all of its subkeys are being deleted. This is convenient, because we don’t have to open any keys before we delete\n2, but a safe approach might be something like this:

// a safe approach to deleting keys
    // open key to gain access to values
    rc = RegOpenKeyEx (HKEY_LOCAL_MACHINE,
                        TEXT ("\Software "), 0,
                        KEY_ALL_ACCESS,
                        &hKeySoftware);
    // delete access to \n2, but not to \software
    rc = RegDeleteKey(hKeySoftware, TEXT ("n2");
    // close this key
    RegCloseKey(hKeySoftware);

This approach requires a couple of extra steps, but it has this advantage: You can’t accidentally delete a parent key, because it’s open. The safest strategy when deleting keys is to carefully limit the scope of possible deletions.

**Registry Porting Checklist**

- Keep registry tree structures as flat as possible.
- Wherever possible, aggregate multiple named key values into a single block of bytes. Have the application retrieve the entire block and parse individual values.
- All key and value names are Unicode strings.
- Double-check all size parameters passed to registry functions. String buffer sizes are the Unicode character capacity of the buffer, including the terminal NULL. Data buffer sizes are in bytes.
- When retrieving values, pay careful attention to casting the address of the data buffer. Cast the address of the buffer to a pointer to the registry data type, and then dereference: (* (int *)pDataBuffer, *(TCHAR *)pDataBuffer, and so on).
Close keys as soon as possible.

When opening keys, use the least expansive access permissions necessary.

When deleting keys, limit the scope of potential deletions by opening the key immediately above the highest node you are deleting.

Looking Ahead

As the chapters in this section unfold, we examine the nuts and bolts variety of programming. In the next chapter, we take up a topic that is central to most applications: files. Because the file-handling API is so rich and varied under Win32, it’s likely that your file management code will change in moving to Windows CE. We’ll examine strategies for simplifying file access and management. Onward.
As the owner of a brand-spanking-new Windows CE PPC, I pressed the power button in a state of great anticipation. The screen brightened and my first reaction was. . . “Where is everything?” This may seem a bit glib now, but I think most users (and probably most software developers) are a little surprised by the absence of an Explorer-style view of the Windows CE file system on the PPC devices. There’s a good reason for this, of course: We don’t have the extravagant amounts of acreage on a CE device that we do on the desktop, and so need a storage model that takes this into account. This model is called the object store, and it is informed by the bias that persistent storage is RAM based.

A couple of important aspects of the CE file system arise from this fact:

- All files are stored in compressed format.
- File organization is simplified.

While your first glimpse of the PPC shell may leave you a bit at sea, the HPC looks and behaves so much like the Windows desktop that neither you nor your application users will have any difficulty figuring out how to navigate to files and applications.
PORTING TIP There are slight differences between the classes of CE devices and fairly significant differences between CE and the desktop when it comes to how files are organized.

In practice this means that your applications must be aware of where files “land” when they are dropped onto the CE device.

One of the two porting challenges you’ll face with respect to file handling is sorting out where files are on a specific device. The second file handling hurdle you’ll need to clear has to do with what is in your application’s support files. You may well spend as much time and effort preparing your application data for porting to CE as your application’s source code. Here’s why: All control text, strings, pathnames, and textual function parameters must be in Unicode under CE. Chances are, you’ll need to translate desktop application data between the single- or multi-byte character sets typically used in Win32 applications and Unicode. Another area of potential data incompatibility has to do with some PPC devices’ limited set of floating point support. You may need to implement workarounds for these, as well.

Where the Files Are

CE’s directory structure is dramatically flattened, consisting mainly of a small number of standard directories at the root of the file system. Though it is possible to create deeply nested directory structures like those on the desktop, it is not a good idea. In order to use the CE file storage layout as efficiently as possible, you should strive to locate your application files in the default locations.

As far as fill handling is concerned, you can open, close, and access CE files using the Win32 file routines, but there are a few significant differences. First, there is no concept of current directory. To access or create a file, you always use its fully qualified pathname. Also, there are fewer standard directories on the PPC, and they are somewhat differently named than those on the HPC, so this fully qualified pathname won’t necessarily be the same on all devices. To throw light on this issue, we are going to look at the storage layout from two points of view. First, we’ll examine the CE Services connection windows on the two types of systems, and then we’ll explore an example program that demonstrates the use of the Windows common file dialog on each type of system.

On my HPC (a Compaq C Series), I open a connection using Windows CE Services, and click on the name of my device in the Mobile Devices window. Here’s the first window I see, and the one most users would drag and drop files into from the desktop Windows Explorer.
The window HPC users see when they open a CE Services connection from the desktop computer.

When I drag files into this window on my desktop computer, they end up in the directory

\Windows\Desktop

This first Connection Window has the name of my device as its caption, and it doesn’t really represent the storage structure of the device at all. It’s really just a snapshot of whatever is on the desktop of the HPC. To see storage structure, we need to tap the My Handheld PC icon (Figure 8.1).

Opening the My Handheld PC Windows CE Services depicts the HPC root directory.

File Handling, File Access, and Data Portability 285
Notice the window caption in Figure 8.2 is “My Handheld PC.” This name has more or less the same significance as “Desktop” on a Windows 9x or NT system. It looks as if

`\My Handheld PC`

is a top-level directory, but don’t be misled by this. Here’s how the fully qualified pathname for a file named `test.txt` in the My Documents directory would look on this HPC device:

`\My Documents\test.txt`

Notice two things about this path: First, there is no drive letter, and second, the directory

`\My Documents`

is at the root of the file system.

In a PPC Connection window, things are laid out a bit differently. Opening a CE Services window for my Cassiopea, the directory layout is simplified. Figure 8.3 shows how it is presented.
If I drag files from the Windows 9x or NT desktop to this window, the filenames show up at the bottom of the list of folders in the PPC Connection window. However, document files actually end up in the \My Documents directory. If you experiment with the CE Services Connection windows for your own device, they may have a slightly different appearance. The point of these specific comparisons, however, stands: There are some differences between the directory structure and default locations of files on the PPC and the HPC.

This is important because the Windows common file dialog is implemented differently on the PPC than on the HPC. The PPC’s dialog is much less flexible in terms of allowing the user to browse to the location of the file they wish to open than that of the HPC. In order to correctly initialize the common file dialog, you must be aware of where the files are.

**PORTING TIP** Make sure that the files you want users to open with the common File Open dialog are in the \My Documents directory on the PPC, or the user won’t be able to find them.

The FileOpen examples (Figures 8.4 and 8.5) show the difference between standard directories on the HPC and PPC, how to initialize and use the common file dialog on both the HPC and the PPC, and the action of different access permissions when opening files.

![FileOpen dialog on an HPC](image)

**Figure 8.4** The FileOpen dialog on an HPC.
The main source file is shown in Listing 8.1, and the support files and headers have been omitted for the sake of brevity. You can find them on the accompanying CD.

```
// FileOpen.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include <winbase.h>
#include <windowsx.h>
#include <commctrl.h>
#include <Commdlg.h>
#include "FileOpen.h"
#define MAX_LOADSTRING 100
// constants for FileOpen
#define FILE_SAVE   0
#define FILE_OPEN   1
// Global Variables:
HINSTANCE    hInst;    // The current instance
HWND         hwndCB;   // The command bar handle
DWORD        dwFileCreateFlags;        // Flags for CreateFile
CHAR         szPath[MAX_PATH];   // Initial Directory for OpenFile Dlg
// Forward declarations of functions included in this code module:
ATOM MyRegisterClass (HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK ChooseFilePathDlgProc(HWND, UINT, WPARAM, LPARAM );
BOOL CALLBACK ChooseFilePermsDlgProc(HWND, UINT, WPARAM, LPARAM );
LPOPENFILENAME DoFileOpen (HWND, int);
int WINAPI WinMain(    HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPTSTR    lpCmdLine,
                        int       nCmdShow)
{
    MSG msg;
```

**Listing 8.1** Source code for the FileOpen example.
HACCEL hAccelTable;
// Perform application initialization:
if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}
hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_FILEOPEN);
// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
return msg.wParam;

ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS    wc;
    wc.style            = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc      = (WNDPROC) WndProc;
    wc.cbClsExtra       = 0;
    wc.cbWndExtra       = 0;
    wc.hInstance        = hInstance;
    wc.hIcon            = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_FILEOPEN));
    wc.hCursor          = 0;
    wc.hbrBackground    = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName     = 0;
    wc.lpszClassName    = szWindowClass;
    return RegisterClass(&wc);
}
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{    HWND    hWnd;
    TCHAR    szTitle[MAX_LOADSTRING];       // The title bar text
    TCHAR    szWindowClass[MAX_LOADSTRING];  // The window class name
    hInst = hInstance;
    // Initialize global strings
    LoadString(hInstance, IDC_FILEOPEN, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
        0, 0, CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL);

Listing 8.1  Source code for the FileOpen example. (continues)
if (!hWnd)
{
    return FALSE;
}
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
return TRUE;

//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
    WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    TCHAR szHello[MAX_LOADSTRING];
    LPOPENFILENAME lpof;
    HANDLE hTestFile;
    switch (message)
    {
    case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_OPEN_FILE:
            DialogBox( hInst, TEXT("Pathnames"), hWnd,
                        ChooseFilePathDlgProc );
            DialogBox( hInst, TEXT("FILE_PERMISSIONS"), hWnd,
                        ChooseFilePermsDlgProc );
            lpof = DoFileOpen( hWnd, FILE_OPEN );
            if(!lpof)
            {
                break;
            }
            //Open the file for reading.
            hTestFile = CreateFile( (LPTSTR)lpof->lpstrFile,
                                GENERIC_READ | GENERIC_WRITE ,
                                FILE_SHARE_READ, NULL, dwFileCreateFlags,
                                FILE_ATTRIBUTE_NORMAL, NULL);
            if( !hTestFile )
            {
                MessageBox( hWnd,
                        TEXT("Try Different Permissions"),
                        TEXT("File Open Failed"), MB_OK );
                break;
            }
        }
    }
Listing 8.1  Source code for the FileOpen example.
if( dwFileCreateFlags == OPEN_EXISTING )
    MessageBox( hWnd, lpof->lpstrFile,
        TEXT("Existing File Opened"), MB_OK );
{
}
if( dwFileCreateFlags == CREATE_NEW )
{
    MessageBox( hWnd, lpof->lpstrFile,
        TEXT("New File Created"), MB_OK );
}
CloseHandle(hTestFile);
break;
case IDM_HELP_ABOUT:
    DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd,
        (DLGPROC)About);
    break;
case IDM_FILE_EXIT:
    DestroyWindow(hWnd);
    break;
default:
    return DefWindowProc(hWnd, message,
        wParam, lParam);
}
break;
case WM_CREATE:
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    CommandBar_AddAdornments(hwndCB, 0, 0);
    break;
case WM_PAINT:
    RECT rt;
    hdc = BeginPaint(hWnd, &ps);
    GetClientRect(hWnd, &rt);
    LoadString(hInst, IDS_HELLO, szHello, MAX_LOADSTRING);
    DrawText(hdc, szHello, _tcslen(szHello), &rt,
        DT_SINGLELINE | DT_VCENTER | DT_CENTER);
    EndPaint(hWnd, &ps);
    break;
case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam,
Listing 8.1  Source code for the FileOpen example.
hWndList = GetDlgItem(hDlg,IDC_PATHNAMES);
//Init paths list based on platform type
memset( &tszPlatType, 0x0, sizeof(tszPlatType));
rc = SystemParametersInfo( SPI_GETPLATFORMTYPE,
    sizeof(tszPlatType),
    &tszPlatType, 0);

if( (lstrcmp( tszPlatType, TEXT("Jupiter") ) == 0 )
    ||
    ( lstrcmp( tszPlatType, TEXT("HPC") ) == 0 ) )
{
    iBaseStringID = IDS_HPC_PATH1;
    iMaxStringID  = IDS_HPC_PATH11;
}

if( lstrcmp( tszPlatType, TEXT("Palm PC") ) == 0 )
{
    iBaseStringID = IDS_PPC_PATH1;
    iMaxStringID  = IDS_PPC_PATH7;
}

for( i = iBaseStringID; i < iMaxStringID; i++ )
{
    lpszPathString =
        (LPCTSTR)LoadString(hInst, i, NULL, NULL);
    lpbStrlen = (LPBYTE)lpszPathString;
    lpbStrlen -= 2 * sizeof( BYTE );
    wCharCnt = (short int)(*lpbStrlen);
    lpszPath = (LPTSTR)LocalAlloc( LPTR, (wCharCnt + 1) *
        sizeof(TCHAR));
    LoadString(hInst, i, lpszPath, (int)wCharCnt + 1);
    ListBox_InsertString(hWndList, -1, lpszPath);
    LocalFree(lpszPath);
}
return TRUE;
case WM_COMMAND:
    if(LOWORD(wParam) == IDOK)
    {
        //set path from ctrl input
        //get the sel item
        hWndList = GetDlgItem(hDlg,IDC_PATHNAMES);
        iSel = SendMessage(hWndList, LB_GETCURSEL, 0, 0 );
        //copy sel item text
        memset( szPath, 0x0, sizeof(szPath) );
        SendMessage(hWndList,LB_GETTEXT, iSel,
            (LPARAM)(LPCTSTR) &szPath );
        EndDialog(hDlg, LOWORD(wParam));
        return TRUE;
    }
    if (LOWORD(wParam) == IDCANCEL)
    {
        //set path NULL

Listing 8.1  Source code for the FileOpen example. (continues)
Listing 8.1 Source code for the FileOpen example.
return FALSE;
}

// Common File Dialog - Open a schema
LPOPENFILENAME DoFileOpen(HWND hWnd, int iOpenSaveFlag)
{
    TCHAR* pszFileName;
    const LPTSTR pszOpenFilter =
        TEXT("All Documents (*.*)\0.*\0\0");
    LPOPENFILENAME lpof;
    // Allocate space for the OPENFILENAME struct
    lpof = (LPOPENFILENAME)LocalAlloc(LPTR,sizeof (OPENFILENAME));
    if (!lpof)
        goto FAIL;
    // Allocate space for the FILENAME string
    pszFileName = (TCHAR*)LocalAlloc(LPTR, MAX_PATH);
    if (!pszFileName)
        goto FAIL;
    // Initialize File Open structure.
    lpof->lStructSize = sizeof (OPENFILENAME);
    lpof->hwndOwner = hWnd;
    lpof->lpstrFile = pszFileName;
    lpof->nMaxFile  = MAX_PATH;
    lpof->lpstrFilter = pszOpenFilter;
    lpof->lpstrInitialDir = (LPCWSTR)&szPath[0];
    if (iOpenSaveFlag ==  FILE_SAVE)
    {
        //prompt for overwrite, send files to sync directory
        lpof->Flags = OFN_OVERWRITEPROMPT;
        if( !GetSaveFileName (lpof))
            lpof = NULL;
    }
    else
    {
        if( !GetOpenFileName (lpof))
            lpof = NULL;
    }
    //it's all good
    return lpof;

FAIL:
    MessageBox( hWnd, TEXT(" Couldn't Allocate Memory"),
        TEXT( "OpenFile Failed" ), MB_OK);
    return 0;
}
The business of the FileOpen example all starts to happen when the IDM_OPEN_FILE message is received WndProc(). First, we launch ChooseFilePathDlgProc(), which allows the user to choose a path. This path will be used to initialize the common file open dialog.

```
case IDM_OPEN_FILE:
    DialogBox( hInst, TEXT("Pathnames"), hWnd,
               ChooseFilePathDlgProc );
```

Next, we call ChooseFilePermsDlgProc() to give the user a choice of file open permissions. These permissions will regulate the action of CreateFile() when we attempt to open the file.

```
DialogBox( hInst, TEXT("FILE_PERMISSIONS"), hWnd,
           ChooseFilePermsDlgProc );
```

In DoFileOpen(), we display the common file dialog and allow the user to choose an existing file or specify a filename to use in opening the file.

```
lpof = DoFileOpen( hWnd, FILE_OPEN );
if(!lpof)
    { 
        break;
    }
```

Finally, we attempt to open the file with a call to CreateFile(), and report our success of failure to the user with MessageBox().

```
//Open the file for reading.
hTestFile = CreateFile( (LPTSTR)lpof->lpstrFile,
                        GENERIC_READ | GENERIC_WRITE,
                        FILE_SHARE_READ, NULL, dwFileCreateFlags,
                        FILE_ATTRIBUTE_NORMAL, NULL);
if( !hTestFile )
    {
        MessageBox( hWnd, TEXT("Try Different Permissions"),
                    TEXT("File Open Failed"), MB_OK );
        break;
    }
if( dwFileCreateFlags == OPEN_EXISTING )
    MessageBox( hWnd, lpof->lpstrFile,
                TEXT("Existing File Opened"), MB_OK );
```

```
When we are finished, we close the file like this:

    CloseHandle(hTestFile);
    break;

Let’s take a close look at the initialization of the ChooseFilePathDlgProc() dialog. In this function, the first thing we have to do is detect the platform type. This allows us to correctly load the list box with directory name strings.

```c
case WM_INITDIALOG:
    hWndList = GetDlgItem(hDlg,IDC_PATHNAMES);
    //Init paths list based on platform type
    memset(&tszPlatType, 0x0, sizeof(tszPlatType));
    rc = SystemParametersInfo(SPI_GETPLATFORMTYPE,
                                sizeof(tszPlatType),
                                &tszPlatType, 0);
    if( (lstrcmp( tszPlatType, TEXT("Jupiter") ) == 0 ) ||
        ( lstrcmp( tszPlatType, TEXT("HPC") ) == 0 ) )
    {
        iBaseStringID = IDS_HPC_PATH1;
        iMaxStringID  = IDS_HPC_PATH11;
    }
    if( lstrcmp( tszPlatType, TEXT("Palm PC") ) == 0 )
    {
        iBaseStringID = IDS_PPC_PATH1;
        iMaxStringID  = IDS_PPC_PATH7;
    }

    The pathname strings are stored consecutively in the resource file. By initializing the iBaseStringID with the lowest string ID and the iMaxStringID with the highest, we can load them with a loop. Notice that we use a CE style call to LoadString(), which returns a pointer to the text of the string in the resource file. The return from this function must be cast to (LPCTSTR) when it is used in this way, as its declaration specifies a return type of int. The word immediately preceding the returned address contains the string length.

    for( i = iBaseStringID; i < iMaxStringID; i++ )
    {
        lpszPathString =
                        (LPCTSTR)LoadString(hInst, i, NULL, NULL);
        lpbStrlen = (LPBYTE)lpszPathString;
        lpbStrlen -= 2 * sizeof( BYTE );
        wCharCnt = (short int)(*lpbStrlen);
```
We use the string length to allocate a buffer on the local heap. Notice that the string length is reported in characters, not in bytes. Also, we add one to this length, because we need space for a terminating null. Which brings us to why we are allocating a buffer in the first place: The strings stored in the resource file are not null terminated, so you can’t use them directly in calls that require a terminal null.

```
lpszPath = (LPTSTR)LocalAlloc(LPTR, (wCharCnt + 1) * sizeof(TCHAR));
LoadString(hInst, i, lpszPath, (int)wCharCnt + 1);
```

Finally, we insert the pathname string at the end of the list and free the buffer.

```
ListBox_InsertString(hWndList, -1, lpszPath);
LocalFree(lpszPath);
}
return TRUE;
```

In the grand scheme of things, the next bit of information we acquire from the user is the file open permissions. There’s nothing unusual about the dialog code for this, so we won’t dissect it here. However, our dialog only offers nondestructive permission choices. When we open the file with a call to CreateFile(), we’ll look at a more complete list.

In DoFileOpen(), we initialize the structure that configures the common File Open dialog. Notice that we specify a filter string for the dialog using the TEXT macro. The sequence following the first null (\0) inside the quoted string sets the filter pattern the dialog will use to discriminate the filenames it makes visible to the user.

```
const LPTSTR pszOpenFilter =
    TEXT (*All Documents (*.*)\0*.*\0*\0*);
```

We allocate an OPENFILENAME structure, and we also allocate a buffer for the filename. The pointer to the filename buffer is a member of the OPENFILENAME structure, and the application is responsible both for initializing the pointer and providing space for the filename.

```
// Allocate space for the OPENFILENAME struct
lpof = (LPOPENFILENAME)LocalAlloc(LPTR, sizeof (OPENFILENAME));
if (!lpof)
    { goto FAIL; }
// Allocate space for the FILENAME string
pszFileName = (TCHAR*)LocalAlloc(LPTR, MAX_PATH);
if (!pszFileName)
    { goto FAIL; }
```
This bit of code that follows is actually superfluous in this case, because the LPTR argument to LocalAlloc() causes the allocated block to be zero initialized. However, if this were a stack-based buffer, you’d want to set the first element to null in order to display a blank edit control for the filename. Conversely, if you want a default filename displayed, initialize the filename buffer with that string.

    // Make the first char null, as we don't want the // dialog to init the filename edit ctl. // pszFileName[0] = '\0';

Setting the lpstrInitialDir member of the OPENFILENAME structure to the path the user chose from the pathname dialog causes file open dialog to be initialized with the filenames in that directory. All types of files will be displayed, because that is what we specified above, in the content of pszOpenFilter.

    // Initialize File Open structure.
    lpof->lStructSize = sizeof (OPENFILENAME);
    lpof->hwndOwner = hWnd;
    lpof->lpstrFile = pszFileName;
    lpof->nMaxFile = MAX_PATH;
    lpof->lpstrFilter = pszOpenFilter;
    lpof->lpstrInitialDir = (LPCWSTR)&szPath[0];

Once again, we have a bit of superfluous code, at least for this particular case. We test a flag passed to the function in order to see if we are opening a file or saving one. I included this to show two things: First, the initialization of the OPENFILENAME structure is nominally the same in either case. (The exception here is if you have different file extensions or initial directory locations for the two operations.) And second, it is a good practice to warn the user before allowing them to overwrite an existing file.

    if (iOpenSaveFlag == FILE_SAVE) {
        //prompt for overwrite, send files to sync directory
        lpof->Flags = OFN_OVERWRITEPROMPT;
        if( !GetSaveFileName (lpof))
        {
            lpof = NULL;
        }
    } else {
        if( !GetOpenFileName (lpof))
        {
            lpof = NULL;
        }
The DoOpenFile() function then returns either a fully initialized OPENFILENAME structure, or null if there was an error.

Next, we open the file with a call to CreateFile().

```c
//Open the file for reading.
hTestFile = CreateFile( (LPTSTR)lpof->lpstrFile,
                             GENERIC_READ | GENERIC_WRITE ,
                             FILE_SHARE_READ, NULL, dwFileCreateFlags,
                             FILE_ATTRIBUTE_NORMAL, NULL);
```

The parameters to CreateFile(), in the order shown, are the filename, the requested type of access, the sharing mode, a NULL placeholder for the unsupported security attributes parameter, the file creation mode, the file attribute flags, and another NULL placeholder for the unsupported hTemplate parameter. Table 8.1 lists the supported values for file creation, along with their meanings, and Table 8.2 lists file attributes.

<table>
<thead>
<tr>
<th>CREATE FLAG CONSTANT</th>
<th>EFFECT OF FLAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_NEW</td>
<td>Creates a new file and fails if the specified file already exists</td>
</tr>
<tr>
<td>CREATE_ALWAYS</td>
<td>Creates a new file: If the file exists, the function overwrites it and resets attributes</td>
</tr>
<tr>
<td>OPEN_EXISTING</td>
<td>Opens the file or fails if the file does not exist</td>
</tr>
<tr>
<td>OPEN_ALWAYS</td>
<td>Opens the file, if it exists, or creates the file if it doesn’t exist</td>
</tr>
<tr>
<td>TRUNCATE_EXISTING</td>
<td>Opens the and truncates it to zero length; fails if the file does not exist</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ATTRIBUTE CONSTANT</th>
<th>EFFECT OF THE ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE_ATTRIBUTE_ARCHIVE</td>
<td>Mark files for backup or removal</td>
</tr>
<tr>
<td>FILE_ATTRIBUTE_HIDDEN</td>
<td>File will not be included in an ordinary directory listing</td>
</tr>
<tr>
<td>FILE_ATTRIBUTE_NORMAL</td>
<td>No other attributes set. Overridden by any other attribute flag</td>
</tr>
<tr>
<td>FILE_ATTRIBUTE_READONLY</td>
<td>File is read only—can’t be written or deleted</td>
</tr>
<tr>
<td>FILE_ATTRIBUTE_SYSTEM</td>
<td>Used exclusively by the operating system</td>
</tr>
</tbody>
</table>
One last bit about filenames. If you want to be able to store the maximum number of files on a PC Card device. Use filenames of uppercase letters, in 8.3 format. PC Cards use more than one physical entry in their file allocation table to store long filenames and names with mixed case.

**Reading and Writing Files**

Once you find a file and open it, it’s time to confront the problem of portability of data. Specifically, getting text out of files and into controls will very often entail some translation. In the GreatWriting example (see Figure 8.6), we’ll see how to read multi-byte character set (MBCS) text and convert it to Unicode for display in an edit control.

![Figure 8.6 The GreatWriting example running on an HPC.](image)

The file GreatWriting.cpp is shown in Listing 8.2, but support files are omitted for the sake of brevity. You can find these on the accompanying CD.

```cpp
// GreatWriting.cpp : Defines the entry point for the application.

//
#endif // NPPRINT_H

Listing 8.2 Source file for GreatWriting. (continues)
```
// Forward declarations of functions included in this code module:
ATOM MyRegisterClass (HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK GreatWritersDlgProc (HWND, UINT, WPARAM, LPARAM);
LPOPENFILENAME OpenSaveFile (HWND, int);
int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPTSTR    lpCmdLine,
                    int       nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }
    hAccelTable = LoadAccelerators(hInstance,
                                    (LPCTSTR)IDC_GREATWRITING);
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        return msg.wParam;
    }
    ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
    {
        WNDCLASS wc;
        wc.style            = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc      = (WNDPROC) WndProc;
        wc.cbClsExtra       = 0;
        wc.cbWndExtra       = 0;
        wc.hInstance        = hInstance;
        wc.hIcon            = LoadIcon(hInstance,
                                         MAKEINTRESOURCE(IDI_GREATWRITING));
        wc.hCursor          = 0;
        wc.hbrBackground    = (HBRUSH) GetStockObject(WHITE_BRUSH);
        wc.lpszMenuName     = 0;
        wc.lpszClassName    = szWindowClass;
        return RegisterClass(&wc);
    }
    BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
    {
        Listing 8.2
        Source file for GreatWriting.
HWND hWnd;
TCHAR szTitle[MAX_LOADSTRING]; // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING]; // The window class name
hInst = hInstance;
// Initialize global strings
LoadString(hInstance, IDC_GREATWRITING, szWindowClass, MAX_LOADSTRING);
MyRegisterClass(hInstance, szWindowClass);
LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
  0, 0, CW_USEDEFAULT, CW_USEDEFAULT, NULL,
  NULL, hInstance, NULL);

if (!hWnd)
{
    return FALSE;
}
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
return TRUE;

Listing 8.2  Source file for GreatWriting. (continues)
default:
    return DefWindowProc(hWnd, message,
        wParam, lParam);

break;
case WM_CREATE:
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    CommandBar_AddAdornments(hwndCB, 0, 0);
    break;
case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam,
    LPARAM lParam)
{
    RECT rt, rt1;
    int DlgWidth, DlgHeight;
    int NewPosX, NewPosY;
    switch (message)
    {
    case WM_INITDIALOG:
        // trying to center the About dialog
        if (GetWindowRect(hDlg, &rt1)) {
            GetClientRect(GetParent(hDlg), &rt);
            DlgWidth = rt1.right - rt1.left;
            DlgHeight = rt1.bottom - rt1.top;
            NewPosX = (rt.right - rt.left - DlgWidth)/2;
            NewPosY = (rt.bottom - rt.top - DlgHeight)/2;

            // if the About box is larger than the physical screen
            if (NewPosX < 0) NewPosX = 0;
            if (NewPosY < 0) NewPosY = 0;
            SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                0, 0, SWP_NOZORDER | SWP_NOSIZE);
        }
        return TRUE;
    case WM_COMMAND:
        if ((LOWORD(wParam) == IDOK) ||
            (LOWORD(wParam) == IDCANCEL))
        {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
    Listing 8.2  Source file for GreatWriting.
break;
}
return FALSE;
}

BOOL CALLBACK GreatWritersDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    static int iAuthor = 0;

    HWND    hWndEdit;
    DWORD   dwBytesWritten = 0, dwFileSize;
    TCHAR*  pszUnicodeBuff;
    CHAR*   pszFileBuff = 0;
    HANDLE  hAsciiFile, hUnicodeFile;
    LPOPENFILENAME lpof = NULL;
    switch (message)
    {
    case WM_INITDIALOG:
        //get text file and translate
        if( lParam == IDM_GEISEL )
        {
            hAsciiFile =
                CreateFile( TEXT( "\My Documents\geisel.asc" ),
                                GENERIC_READ ,
                                FILE_SHARE_READ, NULL, OPEN_EXISTING,
                                FILE_ATTRIBUTE_NORMAL, NULL);
        }
        else
        {
            hAsciiFile =
                CreateFile( TEXT( "\My Documents\nietzche.asc" ),
                                GENERIC_READ ,
                                FILE_SHARE_READ, NULL, OPEN_EXISTING,
                                FILE_ATTRIBUTE_NORMAL, NULL);
        }
        //get file length
        dwFileSize = GetFileSize( hAsciiFile, NULL );
        //allocate a buffer for the script file
        pszFileBuff = (char*)LocalAlloc( LPTR, dwFileSize );
        if( !pszFileBuff )
        {
            MessageBox( hDlg, TEXT( "Couldn't Allocate Memory" ),
                        TEXT( "Open File Failed" ), MB_OK);
        }
        //Capture the identity of this author
        //we’ll use it if we write a unicode file
        iAuthor = lParam;
        //read the file
        ReadFile( hAsciiFile, (LPVOID)pszFileBuff,
        
Listing 8.2  Source file for GreatWriting. (continues)
dwFileSize, &dwBytesWritten, NULL);
//allocate a translation buffer
pszUnicodeBuff = (TCHAR*)LocalAlloc( LPTR,
    sizeof(TCHAR)*( dwFileSize + 1 ));
//translate to unicode
mbstowcs( pszUnicodeBuff, (const char *)pszFileBuff,
    (size_t)strlen(pszFileBuff) );
//fill the edit control
SetDlgItemText(hDlg, IDC_WRITING_SAMPLE, pszUnicodeBuff);

//release translation buffer
LocalFree(pszUnicodeBuff);
//release file buffer
LocalFree(pszFileBuff);
//close the file
CloseHandle(hAsciiFile);
//it's all good
return TRUE;

Listing 8.2  Source file for GreatWriting.
LocalFree( pszUnicodeBuff );
//release the file name buffer stored in the of struct
LocalFree(lpof->lpstrFile);
//release the of struct
LocalFree(lpof);
//close the file
CloseHandle(hUnicodeFile);
break;
case IDOK:
case IDCANCEL:
iAuthor = 0;
EndDialog(hDlg, LOWORD(wParam));
return TRUE;
break;
default:
    break;
    
} //end switch( LOWORD(wParam) )

default:
    break;
} //end switch( message )
return FALSE;
}
LPOPENFILENAME OpenSaveFile(HWND hDlg, int iAuthor)
{
    TCHAR* pszFileName;
    const LPTSTR pszOpenFilter =
        TEXT ("Text Documents (*.txt)\0*.txt\0\0");
    LPOPENFILENAME lpof;
    // Allocate space for the OPENFILENAME struct
    lpof = (LPOPENFILENAME)LocalAlloc(LPTR,sizeof (OPENFILENAME));
    if (!lpof)
        { goto FAIL; }
    // Allocate space for the FILENAME string
    pszFileName = (TCHAR*)LocalAlloc(LPTR, MAX_PATH);
    if (!pszFileName)
        { goto FAIL; }
    // init the filename in the ctl.
    if( iAuthor == IDM_GEISEL )
        {
            wcscpy( pszFileName, TEXT("geisel"));
        }
    else
        {
            wcscpy( pszFileName, TEXT("nietzche"));
        }
    // Initialize File Open structure.
    lpof->lStructSize = sizeof (OPENFILENAME);
    lpof->hwndOwner = hDlg;
    lpof->lpstrFile = pszFileName;

Listing 8.2  Source file for GreatWriting. (continues)
In the GreatWriting example, the user chooses a writing sample from the file menu, and then the writing is displayed in the edit control of the dialog box. Here is the code that implements menu choice in WndProc:

```cpp
    case IDM_GEISEL:
        DialogBoxParam(hInst, TEXT("WRITERS"), hWnd, GreatWritersDlgProc, IDM_GEISEL);
        break;
    case IDM_NIETZCHE:
        DialogBoxParam(hInst, TEXT("WRITERS"), hWnd, GreatWritersDlgProc, IDM_NIETZCHE);
        break;
```

Notice that we use DialogBoxParam() to invoke the dialog. This function allows you to pass a value to the dialog box when it receives the WM_INITDIALOG message in the LPARAM. In this case we pass the ID of the menu choice, which we use to set up initialization of the edit control.

In GreatWritersDlgProc(), we open one of two files:

```cpp
    case WM_INITDIALOG:
        //get text file and translate
        if( lParam == IDM_GEISEL )
        {
```

Listing 8.2  Source file for GreatWriting. (continued)
The files geisel.asc and nietzche.asc are ordinary ASCII text files. I gave them the suffix .asc here to emphasize the fact that the text in them is in single-byte characters. Notice also that they have hard-coded pathnames. I did this so that we wouldn’t need a File Open dialog, and because this particular path exists on both the PPC and HPC.

Because both files are very brief, we’ll read them into memory all at once. The parameters to GetFileSize(), in the order shown, are the handle to the file and the address of the high order DWORD. If this parameter is non-null, it is used to store the upper DWORD of the file size.

Next we capture the identity of this author, using the static integer variable iAuthor. The static keyword means that this variable retains its value across invocation of this function. We could have made it a global and accomplished the same result, but since it is only used in this function, it is cleaner to declare it inside the scope of the dialog box procedure.

Next we read the ASCII file into the newly allocated buffer, but we can’t use this code to initialize the dialog until we translate the ASCII characters to Unicode.
GetFileSize() reported the file size in bytes, which incidentally corresponds to its size in ASCII characters. Probably the two most difficult things to get used to with respect to Unicode are that you can’t treat the name of a Unicode array as an address, and you can’t treat individual characters as bytes. Notice that in calculating the size of the buffer that we use to translate the string from ASCII to Uncode, we multiply the total character count by sizeof(TCHAR).

Now we are prepared to translate, and call mbstowcs() to do the job. The name of this function is an acronym for Multi-Byte String to Wide Character String. The parameters, in the order shown, are the address of a WCS string, a pointer to a constant ASCII string, and the maximum number of source string characters to convert.

Now we set the translated text into the edit control, release the allocations, and close the file.

The GreatWriting example also allows the user to save the two files as Unicode. We call the function OpenSaveFile() to initialize an OPENFILENAME structure.

case IDC_SAVE_UNICODE:
    //get a filename for writing--
    //prompt if this is an overwrite
    lpof = OpenSaveFile( hDlg, iAuthor );
There are only a couple of differences between this open file routine and the one we examined in the FileOpen example, so we’ll limit our dissection of OpenSaveFile() to these. Notice that we use a different string for pszOpenFilter in this dialog. This time the pattern that controls the display of files in the dialog’s list is *.txt, which will allow only filenames with the extension .txt.

LPOPENFILENAME OpenSaveFile(HWND hDlg, int iAuthor)
{
    TCHAR* pszFileName;
    const LPTSTR pszOpenFilter =
        TEXT("Text Documents (*.txt)\0*.txt\0\0");
    LPOPENFILENAME lpof;

    We’ll skip over the bit of code where we allocate space to store the the filename. The pointer to this buffer, which was returned by LocalAlloc(), is pszFileName. Notice that we initialize the buffer with a default save filename this time. We use wcscpy() for this initialization because the filename we pass to CreateFile() must be a Unicode string.

    // init the filename in the ctl.
    if( iAuthor == IDM_GEISEL )
    {
        wcscpy( pszFileName, TEXT( "geisel") );
    }
    else
    {
        wcscpy( pszFileName, TEXT("nietzche") );
    }

    Also, notice that this time we initialize the lpstrDefExt member of the OPENFILENAME structure. We have exercised fairly rigid control over how and where the user can save this file, because we’ve limited display of filenames to those with text extensions, set the initial directory, provided a default save filename, and specified a default save file extension.

    // Initialize File Open structure.
    lpof->lStructSize = sizeof (OPENFILENAME);
    lpof->hwndOwner = hDlg;
    lpof->lpstrFile = pszFileName;
    lpof->nMaxFile   = MAX_PATH;
    lpof->lpstrFilter = pszOpenFilter;
    lpof->lpstrDefExt = TEXT("txt");

    Finally, notice that we use a different function to invoke the File Save common dialog.

    if( !GetSaveFileName (lpof) )
    {

lpof = NULL;

Now, back to GreatWritersDlgProc(), where we were writing out a Unicode version of the ASCII file. We open the file using the access specifier GENERIC_WRITE, set the sharing mode to FILE_SHARE_WRITE, and use the file creation flag CREATE_ALWAYS. This last parameter ensures that if the file exists, it will be reinitialized—its contents will be completely overwritten and its attributes will be reset.

    //open the file
    hUnicodeFile = CreateFile( lpof->lpstrFile,
        GENERIC_WRITE,
        FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);

Now we get the length in characters of the content of the edit control, allocate a buffer sized for the appropriate number of Unicode characters, and retrieve the text from the control.

    //get length in characters of the edit control content
    hWndEdit = GetDlgItem( hDlg, IDC_WRITING_SAMPLE );
    dwFileSize =
        SendMessage( hWndEdit, WM_GETTEXTLENGTH, 0,0 );
    //allocate a buffer
    pszUnicodeBuff = (TCHAR*)LocalAlloc( LPTR,
        (dwFileSize + 1) * sizeof(TCHAR));
    //Copy the text from the control
    SendMessage( hWndEdit, WM_GETTEXT, dwFileSize + 1,
        (LPARAM)pszUnicodeBuff );

Finally, we call WriteFile() to write a byte string to the file, and the number of bytes successfully written are returned in the dwBytesWritten parameter.

    //write to the file
    WriteFile( hUnicodeFile, (LPCVOID)pszUnicodeBuff,
        ( dwFileSize + 1 ) * sizeof(TCHAR),
        &dwBytesWritten, NULL);

Now, we clean up by releasing allocations and closing file handles.

    //release the text buffer
    LocalFree( pszUnicodeBuff );
    //release the file name buffer stored in the
    // of struct
    LocalFree(lpof->lpstrFile );
    //release the of struct
    LocalFree(lpof );
    //close the file
    CloseHandle(hUnicodeFile );
    break;
The main difficulty of Unicode is not that it is arcane or mysterious in any way—it’s that using Unicode requires that you unlearn some of the earliest and closely held ideas you assimilated as a C (or C++) programmer. A Unicode datum consists of a byte range, so you can’t treat its name as its address. For example, incrementing a PBYTE type won’t correctly iterate a wide character string.

**PORTING TIP** Remember these things about Unicode strings:

- To calculate the size of an allocation for a Unicode string, multiply the total character count by sizeof(TCHAR).
- Make sure you are consistent in typing pointers to Unicode strings if you use the increment operator to walk a string.
- Don’t try to treat the name of a Unicode string as its address. Rather, initialize a correctly typed pointer.
- Use string handling functions that begin with “wcs” to manipulate Unicode strings.
- Use mbstowcs() to translate between char and TCHAR types.

Table 8.3 lists some useful Unicode text manipulation functions.

<table>
<thead>
<tr>
<th>UNICODE CHARACTER HANDLING FUNCTION</th>
<th>BEHAVIOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>mbstowcs</td>
<td>Convert from single or multibyte characters to Unicode characters</td>
</tr>
<tr>
<td>wcstombs</td>
<td>Convert from Unicode to multibyte characters</td>
</tr>
<tr>
<td>wcscat</td>
<td>Catenate two strings</td>
</tr>
<tr>
<td>wcschr</td>
<td>Find the first occurrence of a character</td>
</tr>
<tr>
<td>wcscmp</td>
<td>Lexically compare two strings</td>
</tr>
<tr>
<td>wcscpy</td>
<td>Copy a source string to a destination string</td>
</tr>
<tr>
<td>wcsxspn</td>
<td>Return the length of a string in characters</td>
</tr>
<tr>
<td>wcsxlen</td>
<td>Catenate n characters of a source string to a destination string</td>
</tr>
<tr>
<td>wcsxncmp</td>
<td>Compare n characters of a source string to a destination string</td>
</tr>
</tbody>
</table>

(continues)
### Table 8.3  Some Useful Unicode Text Manipulation Functions (Continued)

<table>
<thead>
<tr>
<th>UNICODE CHARACTER HANDLING FUNCTION</th>
<th>BEHAVIOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>wcsncpy</td>
<td>Copy n characters of a source string to a destination string</td>
</tr>
<tr>
<td>wcspbrk</td>
<td>Scan a string for the first occurrence of any of a group of characters</td>
</tr>
<tr>
<td>wcsrchr</td>
<td>Find the first occurrence of a character, scanning backwards through the string</td>
</tr>
<tr>
<td>wcsstr</td>
<td>Find a substring</td>
</tr>
<tr>
<td>wcstok</td>
<td>Tokenize a string, replacing the token with a null</td>
</tr>
<tr>
<td>_wcsdup</td>
<td>Duplicate a string</td>
</tr>
<tr>
<td>_wcscmp</td>
<td>Lexically compare without regard to case</td>
</tr>
<tr>
<td>_wcsnicmp</td>
<td>Lexically compare n characters without regard to case</td>
</tr>
<tr>
<td>_wcslwr</td>
<td>Convert to lowercase</td>
</tr>
<tr>
<td>_wcsupr</td>
<td>Convert to uppercase</td>
</tr>
</tbody>
</table>

### Using Memory-Mapped Files

If you are handling larger files than the ones in the great writers example, memory-mapped files offer you some advantages. In a nutshell, when a file is mapped into memory, you can treat the content of the file as a writeable array of bytes. Any changes to the region in the mapped memory are automatically mirrored to the file. This means you can handle the file content directly, without calling ReadFile() and WriteFile(), and without specifically allocating memory to buffer file content (Listing 8.3).

Memory-mapped regions of the file are moveable. This means you can create a mapped region that is smaller than the file. By changing the mapping parameters, you can operate on one region of the file content, then remap and operate on another region. File maps are created in the system’s address space, so they don’t expand the local heap. We’ll explore how to set up and read from a memory-mapped file in the MemMapFiles example (see Figure 8.7). Writing to memory-mapped files is a trivial extension of this example, but before Windows CE 2.1, it is unsupported. For this reason, we confine the example to behavior that will work on all CE platforms.
MsAmericasWishList.txt: This file gets mapped into a region of memory:

Null Delimited Strings, Each Padded to 20 Bytes in Length

- World Peace
- End Of Hunger
- Waterproof Mascara

---

// Main source file for MemMapFiles
// MemMapFiles.cpp : Defines the entry point for the application.

#include "stdafx.h"
#include "MemMapFiles.h"
#include <commctrl.h>
#define MAX_LOADSTRING 100

#include "stdafx.h"
#include "MemMapFiles.h"
#include <commctrl.h>
#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst; // The current instance
HWND hwndCB; // The command bar handle
ATOM MyRegisterClass (HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK MsDlgProc (HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(    HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPTSTR    lpCmdLine,
                      int       nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }

Listing 8.3  Files for the MemMapFiles example. (continues)
hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_MEMMAPFILES);
// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
return msg.wParam;

ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS    wc;
    wc.style         = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = (WNDPROC) WndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_MEMMAPFILES));
    wc.hCursor       = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName  = 0;
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND    hWnd;
    TCHAR    szTitle[MAX_LOADSTRING]; // The title bar text
    TCHAR    szWindowClass[MAX_LOADSTRING]; // The window class name
    hInst = hInstance;
    // Initialize global strings
    LoadString(hInstance, IDC_MEMMAPFILES, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
        0, 0, CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL);
    if (!hWnd)
    {
        return FALSE;
    }
    ShowWindow(hWnd, nCmdShow);

Listing 8.3 Files for the MemMapFiles example.
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
DialogBox( hInst, TEXT("WISHLIST"), hWnd, MsDlgProc);
return TRUE;
}

Listing 8.3 Files for the MemMapFiles example. (continues)
case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam,
LPARAM lParam)
{
    RECT rt, rt1;
    int DlgWidth, DlgHeight;
    int NewPosX, NewPosY;
    switch (message)
    {
    case WM_INITDIALOG:
        // trying to center the About dialog
        if (GetWindowRect(hDlg, &rt1)) {
            GetClientRect(GetParent(hDlg), &rt);
            DlgWidth    = rt1.right - rt1.left;
            DlgHeight   = rt1.bottom - rt1.top ;
            NewPosX        = (rt.right - rt.left - DlgWidth)/2;
            NewPosY        = (rt.bottom - rt.top - DlgHeight)/2;

            // if the About box is larger than the physical screen
            if (NewPosX < 0) NewPosX = 0;
            if (NewPosY < 0) NewPosY = 0;
            SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                          0, 0, SWP_NOZORDER | SWP_NOSIZE);
        }
        return TRUE;
    case WM_COMMAND:
        if ((LOWORD(wParam) == IDOK) ||
            (LOWORD(wParam) == IDCANCEL))
        {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
        break;
    }
    return FALSE;
}
BOOL CALLBACK MsDlgProc(HWND hDlg,
                      UINT message,
                      WPARAM wParam,
                      LPARAM lParam)
{
    static HANDLE hMappedFile;

Listing 8.3 Files for the MemMapFiles example.
static HANDLE hFileMap;
static CHAR* pFileMemory;
static int iOffset;
TCHAR tszWish[24];
switch (message) {
    case WM_INITDIALOG:
        // create a file for mapping
        hMappedFile = CreateFileForMapping(
            TEXT("\My Documents\MsAmericasWishList.txt"),
            GENERIC_READ ,
            FILE_SHARE_READ, NULL, OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL, NULL);
        if( hMappedFile != INVALID_HANDLE_VALUE )
            { // create a file mapping object
                hFileMap = CreateFileMapping(hMappedFile,
                NULL,
                PAGE_READONLY,0,0,0);
            if( hFileMap )
                {
                    pFileMemory = (CHAR*)MapViewOfFile( hFileMap,
                    FILE_MAP_READ, 0,0,0 );
                if(pFileMemory )
                    {
                        memset( &tszWish[0], 0x0, sizeof(tszWish ));
                        mbstowcs( (LPTSTR)&tszWish,
                        (CHAR*)pFileMemory,
                        (size_t)strlen(pFileMemory) );
                        // update edit control
                        // with first wish list item
                        SetDlgItemText(hDlg, IDC_WISH,
                        (LPCTSTR)&tszWish);
                        // set the offset to the string we just used
                        iOffset = 0;
                    }
                }
            }
        return TRUE;
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
            case IDC_NEXT:
                iOffset += 20;
                iOffset = ( iOffset >= 60 )? 0: iOffset;
                // update edit control
                // with next wish list item
                memset( &tszWish[0], 0x0, sizeof(tszWish ));
                mbstowcs( (LPTSTR)&tszWish,
                (CHAR*)pFileMemory + iOffset,
                Listing 8.3 Files for the MemMapFiles example. (continues)
(size_t)strlen(pFileMemory) );
SetDlgItemText(hDlg, IDC_WISH, (LPCTSTR)&tszWish);
break;

case IDC_PREV:
iOffset -= 20;
iOffset = ( ioffset < 0 )? 40: iOffset;
//update edit control
//with prev wish list item
memset( &tszWish[0], 0x0, sizeof(tszWish ) );
mbstowcs( (LPTSTR)&tszWish,
(CHAR*)pFileMemory + iOffset,
(size_t)strlen(pFileMemory) );
SetDlgItemText(hDlg, IDC_WISH, (LPCTSTR)&tszWish);
break;

case IDCANCEL:
  //clean up
  UnmapViewOfFile(pFileMemory);
  CloseHandle(hFileMap);
  CloseHandle(hMappedFile);
  EndDialog(hDlg, LOWORD(wParam));
  return TRUE;
  default:
    break;
    } //end switch( LOWORD(wParam))
break;
  default:
    break;
  } //end switch( message)
  return FALSE;
}

Listing 8.3  Files for the MemMapFiles example. (continued)

We do all the setup and access of the memory-mapped file in the dialog procedure, MsDlgProc( ). Notice first that we declare a number of static variables.

BOOL CALLBACK MsDlgProc(HWND hDlg,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    static HANDLE hMappedFile;
    static HANDLE hFileMap;
    static CHAR*  pFileMemory;
    static int    iOffset;
    TCHAR  tszWish[24];
The variables hMappedFile, hFileMap, and pFileMemory are the handle to mapped file, the handle to the mapping object, and the pointer to the region of memory that mirrors the file content, respectively. These retain their value across invocations of the dialog function.

Opening a memory-mapped file and setting up the mapped region is a three-step process. You create a memory-mapped file, then create a mapping object to manage and coordinate the stored and memory resident file content, and retrieve pointer to the memory region which you use to access file data.

```c
//create a file for mapping
hMappedFile = CreateFileForMapping(  
    TEXT("\My Documents\MsAmericasWishList.txt"),  
    GENERIC_READ ,  
    FILE_SHARE_READ, NULL, OPEN_EXISTING,  
    FILE_ATTRIBUTE_NORMAL, NULL);  
if( hMappedFile != INVALID_HANDLE_VALUE )
{
    //create a file mapping object
    hFileMap = CreateFileMapping(hMappedFile,  
        NULL, PAGE_READONLY,0,0,0);  
    if( hFileMap )
    {
        pFileMemory = (CHAR*)MapViewOfFile( hFileMap,  
            FILE_MAP_READ, 0,0,0 );
    }
}
```

CreateFileForMapping() looks almost exactly like CreateFile(), and in general the only difference is that for Windows CE versions before 2.1, the flags that specify write access to the file are unsupported. The parameters to CreateFileMapping() are, in the order shown, the handle to the mapped file, a NULL placeholder for the unsupported mapping attributes parameter, protection to be applied to the committed pages of the mapped view of the file, the high-order DWORD of the mapped region’s size, the low-order DWORD of the mapped region’s size, and a pointer to a name for the mapping object created by this call. Setting the size parameters causes the entire file to be mapped into memory, and setting the pointer to the object name to zero creates the mapping object without a name.

MapViewOfFile() returns a pointer to the file’s mapped range of memory. The parameters to MapViewOfFile() are, in the order shown, the handle to the mapping object, the requested access to the object, the high-order DWORD of the file offset where the mapping begins, the low-order DWORD of the file offset where the mapping begins, and the number of bytes to map. If the last three parameters are set to zero, the entire file is mapped into view. If you want to map a smaller region, you need to make sure that parameters that set the offset into the file are an even multiple of the system allocation granularity. Use GetSystemInfo() to find this value. The allocation granularity is returned in the SYSTEM_INFO structure’s dwAllocationGranularity member.
Once you have a pointer to the mapped region, file access is really very streamlined. Here’s how we load strings into the edit control using mapped file access.

case IDC_NEXT:
    iOffset += 20;
    iOffset = ( iOffset >= 60 )? 0: iOffset;
    //update edit control
    //with next wish list item
    memset( &tszWish[0], 0x0, sizeof(tszWish ));
    mbstowcs( (LPTSTR)&tszWish,
               (CHAR*)pFileMemory + iOffset,
               (size_t)strlen(pFileMemory) );
    SetDlgItemText(hDlg, IDC_WISH,
                   (LPCTSTR)&tszWish);
    break;

If you want to change the region of the file that is mapped, first you unmap the current region, and then call MapViewOfFile() for the new region of the file.

    UnmapViewOfFile(pFileMemory);

To exit, we clean up the objects by unmapping the view, and closing handles to the mapping object and mapped file.

case IDCANCEL:
    //clean up
    UnmapViewOfFile(pFileMemory);
    CloseHandle(hFileMap);
    CloseHandle(hMappedFile);
    EndDialog(hDlg, LOWORD(wParam));
    return TRUE;

Numeric Data Portability

My PPC device runs for a fairly long time on a watch battery and a pair of AAAs. Call me a hayseed, but I think that is really incredible. I wasn’t too surprised, then, when I found part of the reason for this long battery life is that my palmtop has a limited register set and it doesn’t have on-chip support for much floating-point math. Try as I might, I simply couldn’t rearrange program logic in a way that would get rid of unresolved externals that had to do with floating-point number comparisons. I finally wrote workarounds that transform floating-point numbers to integers and then operates them. The FloatingPoint example (see Figure 8.8 and Listing 8.4) demonstrates this technique.
// FloatingPoint.cpp : Defines the entry point for the application.

#include "stdafx.h"
#include "FloatingPoint.h"
#include <commctrl.h>
#include <windowsx.h>
#include <math.h>

#define MAX_LOADSTRING 100
#define BETWEEN 1
#define LESS_THAN 2
#define GREATER_THAN 3

// Global Variables:
HINSTANCE hInst;       // The current instance
HWND hwndCB;      // The command bar handle

// Forward declarations of functions included in this code module:
ATOM MyRegisterClass (HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK FloatingPtDlgProc (HWND, UINT, WPARAM, LPARAM);

void CheckFloatingPointRange ( TCHAR*);

int WINAPI WinMain( HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPTSTR    lpCmdLine,
in t       nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }

Listing 8.4  FloatingPoint.cpp. (continues)
AccelTable = LoadAccelerators(hInstance, (LPCTSTR) IDC_FLOATINGPOINT);

// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
return msg.wParam;

//
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_FLOATINGPOINT));
    wc.hCursor = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = 0;
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);
}

//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    TCHAR szTitle[MAX_LOADSTRING]; // The title bar text
    TCHAR szWindowClass[MAX_LOADSTRING]; // The window class name
    hInst = hInstance;
    // Initialize global strings
    LoadString(hInstance, IDC_FLOATINGPOINT,
        szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle,
        WS_VISIBLE,
        0, 0, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL);
    if (!hWnd)
    {
        return FALSE;
    }
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
DialogBox( hInst, TEXT( "FLOATING_POINT" ),
    hWnd, FloatingPtDlgProc);
return TRUE;
}
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
    WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    switch (message)
    {
    case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_HELP_ABOUT:
            DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX,
                hWnd, (DLGPROC)About);
            break;
        case IDM_FILE_EXIT:
            DestroyWindow(hWnd);
            break;
        default:
            return DefWindowProc(hWnd, message,
                wParam, lParam);
        }
        break;
    case WM_CREATE:
        hwndCB = CommandBar_Create(hInst, hWnd, 1);
        CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
        CommandBar_AddAdornments(hwndCB, 0, 0);
        break;
    case WM_DESTROY:
        CommandBar_Destroy(hwndCB);
        PostQuitMessage(0);
        break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam,
LPARAM lParam)
{
    RECT rt, rtl;
    int DlgWidth, DlgHeight;
    int NewPosX, NewPosY;
    switch (message)
    {
    case WM_INITDIALOG:
        // trying to center the About dialog
        if (GetWindowRect(hDlg, &rtl)) {
            GetClientRect(GetParent(hDlg), &rt);
            DlgWidth = rtl.right - rtl.left;
            DlgHeight = rtl.bottom - rtl.top;
            NewPosX = (rt.right - rt.left - DlgWidth)/2;
            NewPosY = (rt.bottom - rt.top - DlgHeight)/2;

            // if the About box is larger than the physical screen
            if (NewPosX < 0) NewPosX = 0;
            if (NewPosY < 0) NewPosY = 0;
            SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                         0, 0, SWP_NOZORDER | SWP_NOSIZE);
        }
        return TRUE;
    case WM_COMMAND:
        if ((LOWORD(wParam) == IDOK) ||
            (LOWORD(wParam) == IDCANCEL))
        {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
        break;
    }
    return FALSE;
}

BOOL CALLBACK FloatingPtDlgProc(HWND hDlg,
                                 UINT message,
                                 WPARAM wParam,
                                 LPARAM lParam)
{
    HWND hWndEdit;
    TCHAR tszNumberBuff[8];
    switch (message)
    {
    case WM_INITDIALOG:
        hWndEdit = GetDlgItem( hDlg, IDC_FLOAT_NUMBER );
        //limit input to 6 characters
        Edit_LimitText( hWndEdit, 6 );
        break;
    case WM_COMMAND:
        switch (LOWORD(wParam))
        {
        Listing 8.4 FloatingPoint.cpp.
case IDC_TEST:
    memset( tszNumberBuff, 0x0,
            sizeof(tszNumberBuff));
    GetDlgItemText( hDlg, IDC_FLOAT_NUMBER,
                    &tszNumberBuff[0], 6 );
    if( !CheckFloatingPointRange(tszNumberBuff[0]) )
    {
        MessageBox(hDlg,
                   TEXT(" Input not in range "),
                   TEXT("Floating Point Test "),
                   MB_OK);
    }
    else
    {
        MessageBox(hDlg,
                   TEXT(" Input within range "),
                   TEXT("Floating Point Test "),
                   MB_OK);
    }
    break;
case IDCANCEL:
    EndDialog(hDlg, LOWORD(wParam));
    return TRUE;
default:
    break;
}
break;
return FALSE;

/****************************************************************************
// Some PPCs don't do floating pt compares,
// so here is a work around
****************************************************************************/
BOOL CheckFloatingPointRange( TCHAR* pszFloatString  )
{
    int  iValid,  iInputDecimal, iInputSign, iCompareInput;
    int iThreshDecimal, iThreshSign;
    int iThreshStrlen, iInputStrlen;
    int iInput, iThreshold, iExp, iDivisor;
    char *pszBuffer, *psz1stInputZero, *psz1stThreshZero;
    char szInputBuff[9];
    double dInput;
    double dThreshold = 0;
    double* pThreshold;
    TCHAR *stopstring;
    //get validation type
    iValid = BETWEEN;
    //get user input

Listing 8.4  FloatingPoint.cpp. (continues)
dInput = wcstod(pszFloatString, &stopstring);
//set the upper threshold value
dThreshold = 8.95;
//get input digits, magnitude & sign
pszBuffer = _ecvt( dInput, 8, &iInputDecimal, &iInputSign );
//copy the string before the buffer contents are destroyed
strcpy( szInputBuff, pszBuffer );
//find the location of the first 0, calculate number digits
psz1stInputZero = strchr( szInputBuff, '0');
iInputStrlen = psz1stInputZero - szInputBuff;
//convert to integer
iInput = atoi( szInputBuff );
//apply correct sign
if( iInputSign )
{
    iInput *= -1;
}

//get threshold digits, magnitude & sign
pszBuffer = _ecvt( dThreshold, 8, &iThreshDecimal, &iThreshSign );
//find the location of the first 0, calculate number digits
psz1stThreshZero = strchr( pszBuffer, '0');
iThreshStrlen = psz1stThreshZero - pszBuffer;
//convert to integer
iThreshold = atoi( pszBuffer );
//apply correct sign
if( iThreshSign )
{
    iThreshold *= -1;
}

switch( iValid )
{
    case BETWEEN:
    case LESS_THAN:
        //if the input sign is greater, fail and bail
        //if the sign == 0, the number we converted is positive
        if( iInputSign == 0 && iThreshSign != 0 )
        { return FALSE; }

        //if the input magnitude is greater, fail & bail
        //The decimal parameter points to an
        //integer value giving the
        //position of the decimal point with respect
        //to the beginning of the string.
        if( iInputDecimal > iThreshDecimal )
        { return FALSE; }

        if( iInputDecimal < iThreshDecimal )
        { return TRUE; }
}
// if we get to here, we have to compare
// the digits to find the larger number

// divide away the padding to correct the magnitudes --
// use all the decimal digits in the longest string
iExp =
    ( iInputStrlen <= iThreshStrlen )?
        8 - iThreshStrlen: 8 - iInputStrlen;
iDivisor = (int)pow( 10, (double)iExp);
iCompareInput = iInput / iDivisor;
iThreshold = iThreshold / iDivisor;
if( iThreshold < iCompareInput )
    {return FALSE;}

// return for Greater than
if(iValid == LESS_THAN)
    {return TRUE;}

// if this is a range, get 2nd bound
if(iValid == BETWEEN)
{
    // set lower range bound
    dThreshold = 4.32;
    // get threshold digits, magnitude & sign
    pszBuffer =
        _ecvt( dThreshold, 8, &iThreshDecimal,
              &iThreshSign );
    // find the location of the first 0,
    // calculate number digits
    psz1stThreshZero = strchr( pszBuffer, '0');
    iThreshStrlen = psz1stThreshZero - pszBuffer;
    // convert to integer
    iThreshold = atoi( pszBuffer );
    // apply correct sign
    if( iThreshSign )
    {
        iThreshold *= -1;
    }
}

// FALL THRU!!

case GREATER_THAN:
// test uses the first threshold val if GT, second if BTWN
// if the input sign is greater, fail and bail
// if the sign 1= 0, the number we converted is negative
if( iInputSign != 0 && iThreshSign == 0 )
    {return FALSE; }

// if the input magnitude is greater, fail & bail
// The decimal parameter points to an

Listing 8.4  FloatingPoint.cpp. (continues)
The math transforms are done in the CheckFloatingPointRange() function. The tricky thing about converting floating-point numbers to integers is getting the place values translated correctly, so if you write a floating-point routine, pay careful attention to the length of the strings that you end up with as you translate between characters and integers.

```c
//get input digits, magnitude & sign
pszBuffer = _ecvt( dInput, 8, &iInputDecimal, &iInputSign );
//copy the string before the buffer contents are destroyed
strcpy( szInputBuff, pszBuffer );
```

The parameters to _ecvt(), in the order shown, are a floating-point number (of type double), the number of digits to return in the converted string, the address of an integer that gives the offset to the location at which the decimal point would be found if it were included in the string, and the address of an integer that specifies the sign. The sign is positive if this number is zero. Notice
that we copy the result of the translation to stack based buffer immediately following the call to _ecvt(). The contents of the buffer returned by the function will be destroyed by subsequent calls, so you should preserve it by copying if it will be used later.

```c
//find the location of the first 0, calculate number digits
psz1stInputZero = strchr( szInputBuff, '0');
iInputStrlen = psz1stInputZero - szInputBuff;
//convert to integer
iInput = atoi( szInputBuff );
//apply correct sign
if( iInputSign )
{
    iInput *= -1;
}
```

Next, we find the offset of the first zero in the returned string, searching left to right. This is important because the strings are zero padded on the right-hand side if there are less input digits than we specified in the call to ecvt(). We translate the returned buffer to an integer with a call to atoi(), and then apply the sign returned by _ecvt(). We go through this conversion process for both the input number and the number to which we want to compare it. Next, let’s look at the process of comparing the two numbers:

```c
case GREATER_THAN:
    //test uses the first threshold val if GT,
    // second if BTWN
    //if the input sign is greater, fail and bail
    //if the sign 1= 0, the number we converted is negative
    if( iInputSign != 0 && iThreshSign == 0 )
    { return FALSE; }
    //if the input magnitude is greater, fail & bail
    //The decimal parameter points to an
    //integer value giving the
    //position of the decimal point with respect
    //to the beginning of the string.
    if( iInputDecimal < iThreshDecimal )
    { return FALSE; }
    if( iInputDecimal > iThreshDecimal )
    { return TRUE; }
```

There are two cases that tell us out of hand which number is greater, without doing any more work, using information returned by _ecvt(): If the input number is positive and the threshold is negative, we know that the input number is greater. Also, if there is a difference in the position of the decimal point, we can tell which number is larger. If neither of these cases occur, we have to compare the magnitudes of the integers we manufactured.
We create a divisor that is a power of 10 that will divide away all the zero padding of the least padded number, then compare.

### A Few More Thoughts on File Handling and Data Portability

Undoubtedly, as you port your application file-handling code, you’ll be nipped a few times by the hounds of Unicode. It takes some experience to unlearn the fast and loose (and convenient) string handling habits you develop as a C programmer, but here is one thing that will help: Think of Unicode characters as being numbers, rather than characters. When you access them by address, try to make a habit of using a separately declared pointer of type TCHAR*, rather than by using the addresses and doing your own address arithmetic.

The C string handling runtimes are still used (and useful) under CE, though some come up as unresolved externals on devices I have tested. To get a good idea of what is supported for a given platform, check the header file tchar.h. Finally, if you initialize control text, filenames, or function parameters using text read from files, do a global search on function names like strcpy() and strcat(). Chances are, you’ll find strings that need to be translated to Unicode in these spots.

### Looking Ahead

Files offer one kind of persistent data storage under Windows CE, but more intriguing and more exciting is CE’s integrated database API. If you are moving a data-driven application from the desktop, it may actually be quite a bit easier to use the CE database API instead of a file-based implementation.
In the most basic sense, there are really only two things you can do with a computer: store data and manipulate data. Even executable code really amounts to a very specialized kind of data, stored in a canonical database. Given this distillation of computing reality, it’s not surprising that a sleek, spare architecture like CE would build database management into the system. It’s just so sensible.

If you are porting a data management application to CE from Win32, in all likelihood you’ve given the issues around database APIs some fairly serious thought. And well you should. You’ll probably also have considered finding a way to avoid using the built-in functions and instead using files or rolling your own more familiar type of database API. If there is one bit of porting advice I would emphatically offer, it’s to make use of the CE database API. While it is certainly very different in character from contemporary relational database tools, it has some really overwhelming strengths and advantages.

**It’s simple.**  Learn six APIs and you are in business—you can do all the significant database operations with an absolutely minimal learning curve.

**It’s flexible.**  In terms of storage, it accommodates any kind of data. In addition, the database APIs are highly versatile. Use a variety of flags and argument values to implement a rich set of behaviors.
It’s storage-space efficient. The libraries for database functionality are stored in ROM. Therefore, your code incurs a miniscule amount of overhead.

It’s fast. Data access is as good as it’s going to get under CE. Anything you can design independently will either be slower or considerably more memory hungry.

Getting Started

If you take the leap of faith that brings you to the conclusion that the CE database API is the way to go, you probably have some fairly large-scale code surgery in store. If you’ve been fairly circumspect about segregating data access code in libraries or sets of source files, then the work of porting begins with these. If your circumstances are less ideal, you’ll need to locate data management operations dispersed throughout your code.

PORTING TIP Here’s how to find data management calls dispersed in the body of code you are porting.

- If you are using a third-party library, remove it from the link specification and build the application. DBM calls will show up as unresolved external references.
- If you are using a homegrown DBM that relies on files, remove the applicable header file from your application and build the application. File access functions will come up as undefined names.

Integrating Database Functionality, CE Style

The beauty of CE’s database API is that although it is spare, it is very flexible. You only need to know a handful of APIs, but the key to using them effectively is in understanding how to modify their behavior using various flags and argument values.

We are going to take a slow, careful tour of database basics in this chapter, by means of two example programs. The first one, the BirthdayReminder example, will demonstrate how to create, open, and add records to a database (see Figure 9.1 and Listing 9.1). The second example, BetterBirthdays, will expand on these topics, by showing how to open a database with a sort order active, and detect and interpret error information. In addition, BetterBirthdays shows how to position and search in a database, how to update exiting records, and how to iterate records based on a sort order. On with the show!
Listing 9.1  BirthdayReminder.cpp. (continues)
HACCEL hAccelTable;
// Perform application initialization:
if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}

hAccelTable = LoadAccelerators(hInstance,
    (LPCTSTR)IDC_BIRTHDAYREMINDER);

// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}

return msg.wParam;
}

//
ATOM MyRegisterClass(HINSTANCE hInstance,
    LPTSTR szWindowClass)
{
   WNDCLASS    wc;
   wc.style             = CS_HREDRAW | CS_VREDRAW;
   wc.lpfnWndProc       = (WNDPROC) WndProc;
   wc.cbClsExtra        = 0;
   wc.cbWndExtra        = 0;
   wc.hInstance         = hInstance;
   wc.hIcon        = LoadIcon(hInstance,
                                MAKEINTRESOURCE(IDI_BIRTHDAYREMINDER));
   wc.hCursor            = 0;
   wc.hbrBackground      = (HBRUSH) GetStockObject(WHITE_BRUSH);
   wc.lpszMenuName       = 0;
   wc.lpszClassName      = szWindowClass;
   return RegisterClass(&wc);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND    hWnd;
   TCHAR    szTitle[MAX_LOADSTRING];   // The title bar text
   TCHAR    szWindowClass[MAX_LOADSTRING];   // The window class name
   INITCOMMONCONTROLSEX icex;
   // Store instance handle in our global variable
   hInst = hInstance;
   // Initialize common controls
   icex.dwSize = sizeof (INITCOMMONCONTROLSEX);
   icex.dwICC = ICC_DATE_CLASSES;
   InitCommonControlsEx (&icex);

Listing 9.1  BirthdayReminder.cpp.
// Initialize global strings
LoadString(hInstance,
    IDC_BIRTHDAYREMINDER,
    szWindowClass, MAX_LOADSTRING);
MyRegisterClass(hInstance, szWindowClass);
LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
hWnd = CreateWindow(szWindowClass, szTitle,
    WS_VISIBLE,
    0, 0,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL, NULL, hInstance,
    NULL);

if (!hWnd)
{
    return FALSE;
}
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
DialogBox(hInst, TEXT("BirthdayDlg"),
    hWnd, (DLGPROC)BirthdayDlgProc);
return TRUE;
}

Listing 9.1  BirthdayReminder.cpp. (continues)
return DefWindowProc(hWnd, message, wParam, lParam);
}
break;
case WM_CREATE:
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
    CommandBar_AddAdornments(hwndCB, 0, 0);
    break;
case WM_DESTROY:
    CommandBar_Destroy(hwndCB);
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}

LRESULT CALLBACK About(HWND hDlg,
                        UINT message,
                        WPARAM wParam,
                        LPARAM lParam)
{
    RECT rt, rt1;
    int DlgWidth, DlgHeight;
    int NewPosX, NewPosY;
    switch (message)
    {
        case WM_INITDIALOG:
            // trying to center the About dialog
            if (GetWindowRect(hDlg, &rt1)) {
                GetClientRect(GetParent(hDlg), &rt);
                DlgWidth    = rt1.right - rt1.left;
                DlgHeight   = rt1.bottom - rt1.top ;
                NewPosX     = (rt.right - rt.left - DlgWidth)/2;
                NewPosY     = (rt.bottom - rt.top - DlgHeight)/2;

                // if the About box is larger than the physical screen
                if (NewPosX < 0) NewPosX = 0;
                if (NewPosY < 0) NewPosY = 0;
                SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                             0, 0, SWP_NOZORDER | SWP_NOSIZE);
            }
            return TRUE;
        case WM_COMMAND:
            if ((LOWORD(wParam) == IDOK) ||
                (LOWORD(wParam) == IDCANCEL))
            {
                EndDialog(hDlg, LOWORD(wParam));
                return TRUE;
            }
    }
    return TRUE;
}

Listing 9.1  BirthdayReminder.cpp.
// Message handler for the BirthdayDlgbox.
LRESULT CALLBACK BirthdayDlgProc(HWND hDlg,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    CECID ceoidDB;
    int rc, iCaption, iTitle;
    switch (message)
    {
    case WM_INITDIALOG:
        //Init the dialog controls
        InitDlgCtrls( hDlg );
        //Create data store
        if( ceoidDB = CECCreateDatabase(TEXT("Birthdays"),
            0x2468, 0, NULL))
        { return TRUE; }
        else
        {
            //either the disk is full or the db exists
            rc = GetLastError();
            //this case is ok
            if( rc == ERROR_DUP_NAME )
            { break; }
            //uh-oh...
            if( rc == ERROR_DISK_FULL )
            {
                iCaption = IDS_CANT_CREATE_DB;
                iTitle = IDS_NO_DATABASE;
                { goto FAIL; }
            }
        }
        globalCEOID = 0;
        globalHDB = CECOpenDatabase (&globalCEOID, TEXT("Birthdays"),
            0, CEDB_AUTOINCREMENT, hDlg);
        if( globalHDB == INVALID_HANDLE_VALUE )
        { goto FAIL; }
        return TRUE;
    case WM_COMMAND:
        if(LOWORD(wParam) == IDC_ADD_RECORD)
        {
            if(!AddBirthdayReminder(hDlg))
            {
                //Listing 9.1
                BirthdayReminder.cpp. (continues)
iCaption = IDS_INPUT_REQUIRED;
iTitle = IDS_NO_REC_ADDED;
goto FAIL;
}
InitDlgCtrls(hDlg);
}

if ((LOWORD(wParam) == IDOK) ||
    (LOWORD(wParam) == IDCANCEL))
{
    CloseHandle(globalHDB);
    EndDialog(hDlg, LOWORD(wParam));
    return TRUE;
}
break;
}
return FALSE;
FAIL:
    //something is wrong -- fail and bail
if (globalHDB)
{ CloseHandle(globalHDB); }
DoMessage(iCaption, iTitle);
EndDialog(hDlg, LOWORD(wParam));
return TRUE;
}

/////////////////////////////////////////////////////
//Do Message
/////////////////////////////////////////////////////
//
void DoMessage(int iCaption, int iErrorText)
{
    TCHAR szPrompt1[64];
    TCHAR szPrompt2[64];
    LoadString(hInst, iCaption, szPrompt1, sizeof(szPrompt1));
    LoadString(hInst, iErrorText, szPrompt2, sizeof(szPrompt2));
    MessageBox(NULL, (LPCTSTR)szPrompt2, (LPCTSTR)szPrompt1, MB_OK);
}

/////////////////////////////////////////////////////
//Add a record to the database
/////////////////////////////////////////////////////
//
BOOL AddBirthdayReminder(HWND hDlg)
{
    HWND hwndWho, hwndWhen;
    LPTSTR lpwszWho = NULL;
    INT iWhat, i;
    int iTextLen, iButtonState;
    SYSTEMTIME SysTime;

Listing 9.1  BirthdayReminder.cpp.
PCEPROPVAL pRecord = 0;
CEBLOB blobSystime;
PBYTE pStringStore = 0;
DWORD dwIndex;

//Find out who from the edit control
hwndWho = GetDlgItem(hDlg, IDC_WHO);
iTextLen = SendMessage(hwndWho, WM_GETTEXTLENGTH, 0, 0);
//require user input
if( !iTextLen )
  {return FALSE;}

//allocate for wide character count plus terminal null
lpwszWho = (TCHAR*) LocalAlloc(LPTR,
  (iTextLen + 1) * sizeof(WCHAR) );

//get the input string
SendMessage(hwndWho, WM_GETTEXT, (iTextLen + 1),
  (LPARAM)(LPSTR)lpwszWho);

//find out when from the date time control
hwndWhen = GetDlgItem(hDlg, IDC_WHEN);
memset(&SysTime, 0x0, sizeof(SysTime));
DateTime_GetSystemtime(hwndWhen, &SysTime);

//find out what
for( i = IDC_CANDY; i <= IDC_ANT_FARM; i++ )
{
  iButtonState =
    SendDlgItemMessage(hDlg, IDC_CANDY, BM_GETSTATE, 0, 0);
  if( iButtonState == BST_CHECKED )
    {
      iWhat = i;
    }
}

//allocate an array of CEPROPVAL structures,
//plus storage space for strings
pRecord = (PCEPROPVAL)LocalAlloc(LPTR,
  (sizeof(CEPROPVAL) * NUMBER_BIRTHDAY_PROPS)
  + ((iTextLen + 1) * sizeof(WCHAR)
    + sizeof(SysTime)));

if(!pRecord )
  {goto FAIL;}

//init pointer to record's string storage
pStringStore = (PBYTE)pRecord +
  (sizeof(CEPROPVAL) * NUMBER_BIRTHDAY_PROPS);

//populate record
  //init the unused prop val members
for( i = 0; i < NUMBER_BIRTHDAY_PROPS; i++ )
  {
    pRecord[i].wLenData = 0;
    pRecord[i].wFlags = 0;
  }
  //first we write the "who" string into a property

Listing 9.1  BirthdayReminder.cpp. (continues)
//write the prop id : ce type + index
pRecord[0].propid = MAKELONG(CEVT_LPWSTR, 0) ;
//copy the string to the data buffer at the
//end of the propval array
wcscpy((TCHAR*)pStringStore, lpwszWho);
//set the property union member to this ptr
pRecord[0].val.lpwstr = (LPWSTR)pStringStore;
//next we write the "when" value into a property
//write the prop id : ce type + index
pRecord[1].propid = MAKELONG(CEVT_BLOB, 1) ;

//now we need to init a CEBLOB structure
//move the data buffer pointer past the "who" string
blobSystime.lpb = pStringStore + (iTextLen + 1) * sizeof(WCHAR);
//copy the system time structure to the
//buffer as a byte string
memcpy(blobSystime.lpb, (PBYTE)&SysTime, sizeof(SysTime) );
//set the size of the object in the CEBLOB structure
blobSystime.dwCount = sizeof(SysTime);
// put the CEBLOB structure in the value
//union for this property.
pRecord[1].val.blob = blobSystime;
// copy the "what" data to a CEPROPVAL structure
//write the prop id : ce type + index
pRecord[2].propid = MAKELONG(CEVT_I2, 2) ;
//convert the data to int
pRecord[2].val.iVal = iWhat;
//seek to the end & write the database PROPVAL array
CeSeekDatabase(globalHDB, CEDB_SEEK_END, 1, &dwIndex);
//set oid parm to 0 to create a new rec
CeWriteRecordProps(globalHDB, 0, NUMBER_BIRTHDAY_PROPS, pRecord);
if( GetLastError() == ERROR_INVALID_PARAMETER )
{ goto FAIL;}
if( GetLastError() == ERROR_DISK_FULL )
{ goto FAIL;}
//free the cepropval struct array
LocalFree( pRecord );
//clean up and return status
if(lpwszWho)
{
    LocalFree( lpwszWho );
}
return TRUE;
FAIL:
    return FALSE;
}
void InitDlgCtrls ( HWND hDlg )
{
    HWND hwndWho, hwndWhen, hwndWhat;

Listing 9.1  BirthdayReminder.cpp.
Listing 9.1 BirthdayReminder.cpp. (continued)

Overview of the Steps

The CE database API is fairly simple to use, but you need to take sort of a canonical view. You have to do database operations in a specific order, and many operations depend on the manner in which prior operations were handled. Here is a basic chronology:

1. Create a database, and save its CE object identifier (CEOID).
2. Open the database, by name or CEOID, and save its handle.
3. Position in the database by seeking to a specific location.
4. Perform data access operations.
5. Close the database.

We’ll examine each of these actions in more depth, using the Birthday Reminder example.

Creating a Database

Most of the business of managing the Birthdays database is handled in the BirthdayDlgProc() function. We create the database in response to the WM_INITDIALOG message, with a call to the CeCreateDatabase() function:

case WM_INITDIALOG:
    //Init the dialog controls
    InitDlgCtrls( hDlg );
//Create data store
if( ceoidDB = CeCreateDatabase(TEXT("Birthdays"),
               0x2468, 0, NULL))
) { return TRUE; }

The parameters to CeCreateDatabase(), in the order shown, are the database name as a Unicode string, a DWORD value that identifies a general, application-specific database type, the number of active sort orders for this database, and a pointer to the array of SORTORDERSPEC structures for the database.

The way in which you initially create the database has great influence on the operations you’ll be able to perform subsequently. Let’s look at each of the parameters in a bit more depth.

The database name is a Unicode string. It may be no more than 32 characters in length, including the required terminal null. Longer names are used but truncated. Because you’ll most likely use the name for subsequent open operations, you want to make certain you stay within this limit.

The database type is actually an arbitrary number. You choose this number and assign it to a related group of databases. For example, if you have several databases of part numbers, you could assign them a common “part number” database type. This database type identifier is used to enumerate specific, related databases. For now, it’s enough to be aware that if you specify a database type in the create call, it should be unique. Currently, the native CE database type constants are less than 100 decimals. If you pick a large number for your database type, you are most likely safe.

The number of active sort orders specifies how many properties (fields) of your database are sortable keys. The maximum number of active sort orders is 4. To search a field by value, it must have an active sort order. If you don’t need to search records by property values, set this field to 0.

The address of the array of SORTORDERSPEC structures is set to NULL if the number of active sort orders is 0. In this example, we aren’t sorting records, so we set it to NULL. In the BetterBirthday example, we’ll use this field to set up a sort order, so we’ll explore the related issues more closely then.

A couple of points remain before we move on to opening the database. Notice that CeCreateDatabase() returns a value of type CEOID. A CEOID is a unique object identifier that specifies an object in the Windows CE Object Store. Unlike a handle, the CEOID for the database is persistent. It can be used to recover information about the database even if the database isn’t open.

Unlike file functions and the like, CE database functions often don’t return values that directly encode status information. To find out if database calls are successful, you’ll often need to use GetLastError(), as we did following the call to CeCreateDatabase():

    //either the disk is full or the db exists
    rc = GetLastError();
    //this case is ok
if( rc == ERROR_DUP_NAME )
{
    break;
}

//uh-oh...
if( rc == ERROR_DISK_FULL )
{

    iCaption = IDS_CANT_CREATE_DB;
    iTitle = IDS_NO_DATABASE;
    goto FAIL;
}

Notice that in this case we test for two conditions that might cause failure to create the database. If the database already exists, GetLastError() will report ERROR_DUP_NAME. This isn’t a problem for us, as it won’t prevent us from opening the database by name. If we get a return of ERROR_DISK_FULL we notify the user, clean up, and bail out. If we just want to unconditionally test for success or failure, we could do so like this:

//sample code fragment
if( rc == ERROR_SUCCESS )
{
    //rock on...
}

Strangely enough, a return code of ERROR_SUCCESS signifies that everything is fine. If we get this far, then the next step is to open the database we just created, with a call to CeOpenDatabase():

globalCEOID = 0;
globalHDB = CeOpenDatabase (&globalCEOID,
    TEXT("Birthdays"),
    0, CEDB_AUTOINCREMENT, hDlg);
if( globalHDB == INVALID_HANDLE_VALUE )
    goto FAIL;

The parameters to CeOpenDatabase(), in the order shown, are the address of a variable of CEOID type, which will contain the database’s CEOID on successful return; the name of the database as a Unicode string; a null placeholder for the property ID of a primary key sort order field; the flag CEDB_AUTOINCREMENT, which means that the current record location is incremented on each read operation; and the handle to the window that will receive notifications if another application modifies the database after we open it.

The fashion in which you open the database has a great deal of influence on what data access operations you are able to perform later. Also, the meaning of the parameters to this function depend on one another in subtle ways. Let’s explore them further.

First, notice that in this case, we set the globalCEOID parameter equal to zero before calling the function. This indicates that we want the CEOID returned in
this value. If we had set globalCEOID equal to a valid CEOID for the database, we could open it without using (or even knowing) its literal name.

In the example code, we use the second parameter, a Unicode literal, to open the database by its literal name. If we had chosen to open it using a CEOID value, we’d set the database name parameter to NULL.

The next parameter, the property ID of a primary sort key, is set to NULL here. The most significant effect of this is that with no active sort key, we won’t be able to locate a record by subjecting one of its properties (fields) to a specific search criteria. (For example, we can’t look for a record with a field greater than, less than, or equal to a reference value.) However, even without a sort order active, you can iteratively read records, write records, and position (seek) in the database by absolute location (for example, you can set the current record pointer at the third record from the beginning of the database) or position using a record’s CEOID.

The open mode flag parameter is set to CEDB_AUTOINCREMENT in this case. This flag causes the “current record” pointer to move ahead one record after each read operation. If you set this parameter equal to zero, the record pointer only moves when you explicitly set it to a new location by seeking.

The final parameter, in this case the handle to BirthdayDlg, identifies the window that is to be notified if the database is modified by another application. Our example includes no code to handle such a notification, and for application-specific databases, it probably isn’t necessary to worry much about this. However, if you are using a native CE database (for example, the Contacts database), you’ll need to be ready to handle such notifications.

A few final observations on opening CE databases are in order. First, notice that there is no explicit provision for open “permissions” once the database is opened. You may read, write, and delete at will. Second, how you open the database makes a lot of difference in terms of what kind of searching and positioning operations you are subsequently able to perform. For these reasons, you’ll probably find it both more convenient and more robust to open the database in the way that is most conducive to your access needs for a given circumstance, then close and reopen for different types of operations. In the example above, we opened the database in the way that makes it easiest to sequentially read records.

**PORTING TIP** Open the database in the way that most closely supports immediate access needs, and then close and reopen as access requirements change.

**Adding Records to an Open Database**

Well, after all, what would a database be without some data? It’s time to think about adding records. Before we dig into the specifics of this task, there are a
few clarifications we need to make about the relationship between CE database properties, CE database records, and the overall structure of the CE database. (In the discussion that follows, I use the terms *database property* and *field* interchangeably.)

First, recall I said in the opening paragraphs of this chapter that CE records accommodate any kind of data. This is indeed the case, although perhaps a bit disingenuous. Here’s why.

Basically, a CE database property (field) consists of a CEPROPVAL structure, which provides descriptor information, and, optionally, some application-allocated space, which contains data too large to be directly stored in the CEPROPVAL structure. To better understand this, let’s examine the typedef of the CEPROPVAL structure:

```c
typedef struct _CEPROPVAL
{
    CEPROPID propid;
    WORD wLenData;
    WORD wFlags;
    CEVALUNION val;
} CEPROPVAL;
typedef CEPROPVAL *PCEPROPVAL;
```

When we refer to a specific database property, invariably we do so using its CEPROPID, the first member of the CEPROPVAL structure. The CEPROPID is itself another typedef, which looks like this in the Windows CE header files:

```c
typedef DWORD CEPROPID;
```

The CEPROPID provides two important pieces of information about a field: the ordinal of the field in the record and the data type of the field. First, let’s take stock of the possible types for a CE database property. Table 9.1 lists the options.

<table>
<thead>
<tr>
<th>PROPERTY TYPE CONSTANT</th>
<th>DATA STORED IN THE PROPERTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEVT_BLOB</td>
<td>A CEBLOB structure</td>
</tr>
<tr>
<td>CEVT_FILETIME</td>
<td>A FILETIME structure</td>
</tr>
<tr>
<td>CEVT_I2</td>
<td>A 16-bit signed integer</td>
</tr>
<tr>
<td>CEVT_I4</td>
<td>A 32-bit signed integer</td>
</tr>
<tr>
<td>CEVT_LPWSTR</td>
<td>A null-terminated string</td>
</tr>
<tr>
<td>CEVT_UI2</td>
<td>A 16-bit unsigned integer</td>
</tr>
<tr>
<td>CEVT_UI4</td>
<td>A 32-bit unsigned integer</td>
</tr>
</tbody>
</table>
You’ve probably noticed that this is a pretty short list. There are no floating-point types, and no native date or time types. Basically, the list is a collection of the things that can either be stored in 32 bits or referenced through a long pointer. Whatever goes into a CE database record must be made to fit into one of these types. The relationship between the type of the data and the actual storage of the data will be easier to picture if we skip ahead in the CEPROPVAL structure’s members and examine the CEVALUNION member next. Here’s the typedef of CEVALUNION:

```c
typedef union _CEVALUNION
{
    short ival;
    USHORT uiVal;
    long lVal;
    ULONG ulVal;
    FILETIME filetime;
    LPWSTR lpwstr;
    CEBLOB blob;
} CEVALUNION;
```

A union, if you aren’t familiar with the idiom, is a single block of space that is large enough to accommodate the storage of its largest member. It can hold only one value, and thus represent only one of its members at any given time. Unions are a useful construct in situations where a variable might assume the value of any of several different types of data. In the case of a CEPROPVAL, its union member allows us to store seven distinct types of data in a generic database property structure.

Here’s a key relationship to visualize: A CEPROPID gives the data type of a specific property. A CEVALUNION stores either the actual data value or a pointer to the data value. Both are members of the CEPROPVAL structure, which defines the property’s attributes and (directly or indirectly) its value.

Now let’s revisit the typedef of the CEPROPID:

```c
typedef DWORD CEPROPID;
```

Recall that we said that a CEPROPID stores two pieces of information: the data type and the property’s ordinal position in the record. Here’s how we create a CEPROPID for a short integer that’s the third property in the record:

```c
MAKELONG(CEVT_I2, 2)
```

The MAKELONG macro places the CE data type constant in the LOWORD, and the zero-based index of the property in the HIWORD of the CEPROPID. By storing the property’s index, we are able to implement sparse records. In other words, if you have a record type that has 10 properties, and in a particular case you only have data for 2 of them, then you add the record using an array of only 2 CEPROPVAL structures, one for each field for which you have data.
The wDataLen and wFlags fields of the CEPROPVAL structure are unused and, by convention, are always set to 0.

Summing up, here is a litany of the relationships among the structures that constitute a CE database:

- A database is a group of records owned by a specific database object, and the database object is identified by a unique CEOID.
- A given database record consists of an array of CEPROPVAL structures, plus application-allocated memory for data that is too large to be stored in the members of the CEVALUNION.
- A record need only store CEVALPROP structures for fields that contain data.
- The CEPROPID gives the CE data type of the property combined with the zero-based index of the property with respect to the complete set of properties in a given record.
- The CEVALUNION holds either the value of the property, or a pointer to the space that holds the value.

Creating and Populating the CEPROPVAL Array

Whew! Let’s take a look at how the AddBirthdayReminder() function adds records to the CE database. The first part of the AddBirthdayReminder() function is devoted to gathering data from the user interface controls of the dialog box. The one important part of this is where we get the name of the luck recipient of our largess from the edit control:

```c
//Find out who from the edit control
hwndWho = GetDlgItem(hDlg, IDC_WHO);
iTextLen = SendMessage( hwndWho, WM_GETTEXTLENGTH, 0, 0);
//require user input
if( !iTextLen )
    {return FALSE;}
//allocate for wide character count plus terminal null
lpwszWho = (TCHAR* )LocalAlloc(LPTR,
    (iTextLen + 1) * sizeof(WCHAR) );
//get the input string
SendMessage( hwndWho, WM_GETTEXT, (iTextLen + 1),
    (LPARAM)(LPSTR)lpwszWho);
```

The code for interrogating the dialog controls is fairly self explanatory, but notice two things: First, we have found the exact length of the name string, and allocated space to hold it, plus the terminating NULL. Second, the birth date that we retrieve from the DateTime control is returned to us in a SYSTEMTIME structure, which is not among the supported CE data types.
Now we’ll skip ahead to the real work of structuring and populating the CE database record.

There are three items of data in a BirthdayReminder record: a name, a birth date, and an integer that indicates the type of gift we’ll be sending. In this example, we require the input of all three for each record. Therefore, in order to create a record, we need to allocate enough space to store 3 CEPROPVAL structures, plus enough additional space to store the name string and a SYSTEMTIME structure. Here’s how we do this:

```c
//allocate an array of CEPROPVAL structures, 
//plus storage space for strings
pRecord = (PCEPROPVAL)LocalAlloc( LPTR,
    sizeof(CEPROPVAL) * NUMBER_BIRTHDAY_PROPS)
    + ((iTextLen + 1) * sizeof(WCHAR)
    + sizeof(SysTime)));
if (!pRecord )
    {goto FAIL;}
```

We now have a single block of space large enough to hold the CEPROPVAL descriptor structures and all of the string and SYSTEMTIME data. The CEPROPVAL structures occupy the beginning of this data block; the name string and the SYSTEMTIME are stored in the remainder of the block. In order to write them into the proper location, we initialize a pointer to the byte immediately following the CEPROPVAL array:

```c
//init pointer to record's string & SYSTEMTIME storage
pStringStore = (PBYTE)pRecord +
    (sizeof(CEPROPVAL) * NUMBER_BIRTHDAY_PROPS);
```

Now we begin populating the array of CEPROPVAL structures. The wLenData and wFlags fields are unused, and, by convention, are always set to zero:

```c
//populate record 
//init the unused prop val members
for( i = 0; i < NUMBER_BIRTHDAY_PROPS; i++ )
{
    pRecord[i].wLenData = 0;
    pRecord[i].wFlags = 0;
}
```

We step through the array of property value structures by indexing the CEPRPOVAL pointer, pRecord:

```c
//first we write the "who" string into a property
//write the prop id : ce type + index
pRecord[0].propid = MAKELONG(CEVT_LPWSTR, 0) ;
//copy the string to the data buffer at the
//end of the propval array
```
To populate the name property, first we set its CEPROPID. The data type is CEVT_LPWSTR, pointer to Unicode string, and it’s index in the array of record properties is 0. Next, we copy the data from the buffer we allocated when we got the edit control input to the data storage area allocated for this record. Finally, we set the CEVALUNION member to the offset in the record’s data storage space where we copied the string. It is absolutely critical that you copy string data into the record allocation, and set the CEVALUNION member lpwstr to the correct offset in the record’s data storage area. If you set lpwstr to an address outside the record allocation (the buffer from which we copied the string, for example), when the application exits, that memory will be freed and the database will be corrupted.

Next, we’ll populate the birth date property, which is a SYSTEMTIME value. Since this type isn’t directly supported by CE, we must save it as a byte string or BLOB (Binary Large OBject):

```c
//next we write the "when" value into a property
//write the prop id : ce type + index
pRecord[1].propid = MAKELONG(CEVT_BLOB, 1) ;
```

First, we synthesize its CEPROPID, by combining its type and the index of this property in the record’s property array. A CEBLOB is yet another structure, typedef’ed like this:

```c
typedef struct _CEBLOB {
    DWORD dwCount;
    LPBYTE lpb;
} CEBLOB;
```

The size in bytes of the object is specified by dwCount, and the address of the first byte of the string is given by lpb. As with the name string, we must copy the actual data for this property into the data area of the record allocation:

```c
//move the data buffer pointer past the "who" string
blobSystime.lpb = pStringStore + (iTextLen + 1) * sizeof(WCHAR);
```

In order to do this, we move the pointer past the region into which we copied the name string. Notice that we don’t use LocalSize(lpwszWho) to advance the pointer, but instead, calculate the new location in the record’s data storage area using the string length returned by WM_GETTEXTLENGTH. LocalSize() returns the size of the allocation that was actually made in our behalf, which may have been larger than our request. Moving the buffer pointer using LocalSize() could
cause us to write beyond the end of the record allocation, corrupting the database, and possibly crashing the application as well:

```c
//copy the system time structure to the
//buffer as a byte string
memcpy(blobSystime.lpb, (PBYTE)&SysTime,
    sizeof(SysTime) );
//set the size of the object in the CEBLOB structure
blobSystime.dwCount = sizeof(SysTime);
// put the CEBLOB structure in the value
//union for this property.
pRecord[1].val.blob = blobSystime;
```

We use memcpy() to move the SYSTEMTIME structure into the record allocation as a string of bytes. (This is also a good approach for storing floating-point numbers in native format.) Finally, we initialize the dwCount member of the CEBLOB structure with the size of the SYSTEMTIME structure, and set the CEVALUNION member val.blob to the initialized blob structure.

Saving the best for last, we initialize the final CEPROPVAL structure for this record:

```c
// copy the "what" data to a CEPROPVAL structure
//write the prop id : ce type + index
pRecord[2].propid = MAKELONG(CEVT_I2, 2) ;
//convert the data to int
pRecord[2].val.iVal = iWhat;
```

Because the data for this field is a short integer, we can store the property value directly in the val.iVal member of the CEVALUNION. Now we have a completely initialized data structure for the record.

### Positioning for Data Access

With the exception of iterative reading through a database opened with the CEDB_AUTOINCREMENT flag, there isn’t really a “current record” construct that automatically operates on CE record access. As is the case with most of the rest of life, if you want to be somewhere, you have to exert a bit of effort to get there. Specifically, with respect to CE databases, you must seek to a record before you can operate on it using CeSeekDatabase():

```c
//seek to the end & write the database PROPVAL array
CeSeekDatabase(globalHDB, CEDB_SEEK_END, 1, &dwIndex );
```

The parameters to CeSeekDatabase(), in the order shown, are the handle to the database; a flag indicating the type of seek to perform; a value whose meaning depends on the seek type (in this case it is the number of records to seek beyond the end of the database); and the address of a variable that holds the
index of the new position on a successful return. Seeking is fairly simple if you want to add a record to the beginning or end of the database, and more complex if you want to find a specific record based on the value of a sorted field. For now, we’ll let this example stand on its own. We investigate the use of CeSeekDatabase() in greater detail in the next example program, BetterBirthdays.

All that remains is to write the record to the database, clean up after ourselves, and go home:

```c
//set oid parm to 0 to create a new rec
CeWriteRecordProps(globalHDB, 0,
                   NUMBER_BIRTHDAY_PROPS, pRecord);
if( GetLastError() == ERROR_INVALID_PARAMETER )
    { goto FAIL; }
if( GetLastError() == ERROR_DISK_FULL )
    { goto FAIL; }
```

First, we write the record, using a call to CeWriteRecordProps(). The parameters are, in the order shown, the handle to the database, the CEOID of the record to write, the number of CEPROPVAL structures in the array we are passing, and a pointer to the block we allocated to hold the array plus additional data storage. Notice that in this case we pass a value of 0 for the record’s CEOID. This causes a new record to be created and added to the database. CeWriteRecordProps() returns the CEOID for the new record if the call is successful. To test for success or failure, we call GetLastError(). While you are learning the CE database moves, routinely testing for a status of ERROR_INVALID_PARAMETER is a useful debugging tool, so we show that here.

Finally, we free all the allocations we made in order to build the record data structures:

```c
//free the cepropval struct array
LocalFree( pRecord );
//free the name buffer
if(lpwszWho)
    {
    LocalFree( lpwszWho );
    }
```

When the dialog box receives an IDOK or IDCANCEL message, we close the database like this:

```c
if ((LOWORD(wParam) == IDOK) ||
    (LOWORD(wParam) == IDCANCEL))
{
    CloseHandle(globalHDB);
    EndDialog(hDlg, LOWORD(wParam));
    return TRUE;
}
```

Notice that we close the database using its handle, not its CEOID.
Reviewing the BirthdayReminder Example

At this point, you’ve seen how to:

- Create a simple database using CeCreateDatabase()
- Open a database without using an active sort order using CeOpenDatabase()
- Allocate and initialize an array of CEPROPVAL structures to create a record
- Seek to the end of the database using CeSeekDatabase()
- Add a record using CeWriteRecordProps()
- Close a database using CloseHandle()
- Test for error status using GetLastError()

But wait, there’s more... 

Flexible Access, Powerful Subtleties

When it comes to data management, real power has more to do with precision of access than just about anything else. We need to be able to find specific records or groups of records, modify them, and iterate them. In the BetterBirthdays example, we tackle these issues (see Figure 9.2 and Listing 9.2). We don’t introduce a lot of new APIs for these jobs. Rather, we mostly use the ones you’ve already seen in slightly different ways.

![Figure 9.2](image)

Figure 9.2: The BetterBirthdays example running on an H/PC.

// BirthdayReminder.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "BetterBirthdays.h"
#include <windows.h>
#include <windowsx.h>

Listing 9.2 BetterBirthdays.cpp. (continues)
DispatchMessage(&msg);
}
}
return msg.wParam;

//
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(hInstance,
                       MAKEINTRESOURCE(IDI_BETTERBIRTHDAYS));
    wc.hCursor = 0;
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.lpszMenuName = 0;
    wc.lpszClassName = szWindowClass;
    return RegisterClass(&wc);
}

//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    TCHAR szTitle[MAX_LOADSTRING];
    TCHAR szWindowClass[MAX_LOADSTRING];
    INITCOMMONCONTROLSEX icex;
    hInst = hInstance
    // Initialize common controls
    icex.dwSize = sizeof(INITCOMMONCONTROLSEX);
    icex.dwICC = ICC_DATE_CLASSES;
    InitCommonControlsEx(&icex);
    // Initialize global strings
    LoadString(hInstance, IDC_BETTERBIRTHDAYS,
               szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
                         0, 0,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

    if (!hWnd)
    {
        return FALSE;
    }
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

Listing 9.2  BetterBirthdays.cpp.
if (hwndCB)
    CommandBar_Show(hwndCB, TRUE);
DialogBox(hInst, TEXT("BirthdayDlg"),
    hWnd, (DLGPROC)BirthdayDlgProc);
return TRUE;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    switch (message)
    {
    case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_HELP_ABOUT:
            DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd,
                (DLGPROC)About);
            break;
        case IDM_FILE_EXIT:
            DestroyWindow(hWnd);
            break;
        default:
            return DefWindowProc(hWnd, message,
                wParam, lParam);
        }
        break;
    case WM_CREATE:
        hwndCB = CommandBar_Create(hInst, hWnd, 1);
        CommandBar_InsertMenubar(hwndCB, hInst, IDM_MENU, 0);
        CommandBar_AddAdornments(hwndCB, 0, 0);
        break;
    case WM_DESTROY:
        CommandBar_Destroy(hwndCB);
        // get the oid, then close
        globalHDB = CeOpenDatabase (&globalCEOID,
            TEXT("Birthdays"),
            0, 0, NULL);
        CloseHandle( globalHDB );
        CeDeleteDatabase(globalCEOID);
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }

Listing 9.2 BetterBirthdays.cpp. (continues)
return 0;
}
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    RECT rt, rt1;
    int DlgWidth, DlgHeight int NewPosX, NewPosY;
    switch (message)
    {
    case WM_INITDIALOG:
        // trying to center the About dialog
        if (GetWindowRect(hDlg, &rt1)) {
            GetClientRect(GetParent(hDlg), &rt);
            DlgWidth = rt1.right - rt1.left;
            DlgHeight = rt1.bottom - rt1.top;
            NewPosX = (rt.right - rt.left - DlgWidth)/2;
            NewPosY = (rt.bottom - rt.top - DlgHeight)/2;

            // if the About box is larger than the physical screen
            if (NewPosX < 0) NewPosX = 0;
            if (NewPosY < 0) NewPosY = 0;
            SetWindowPos(hDlg, 0, NewPosX, NewPosY,
                        0, 0, SWP_NOZORDER | SWP_NOSIZE);
        }
        return TRUE;
    case WM_COMMAND:
        if ((LOWORD(wParam) == IDOK) ||
            (LOWORD(wParam) == IDCANCEL))
        {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
        break;
    return FALSE;
}

// Message handler for the BirthdayDlg box.
LRESULT CALLBACK BirthdayDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    CEOID ceoidDB = 0;
    int rc, iCaption, iTitle;
    SORTORDERSPEC sosAscendingNameSort;
    CEPROPID ceSortProp;
    ceSortProp = MAKELONG(CEVT_LPWSTR, 0);
    switch (message)
    {
Listing 9.2  BetterBirthdays.cpp.
case WM_INITDIALOG:
    // Init the dialog controls
    InitDlgCtrls( hDlg );
    // sort records ascending on name,
    // without respect to case
    sosAscendingNameSort.propid = MAKELONG(CEVT_LPWSTR, 0);
    sosAscendingNameSort.dwFlags = CEDB_SORT_CASEINSENSITIVE;
    // Create data store and save the CEOID
    globalCEOID = CeCreateDatabase(TEXT("Birthdays"),
                              0x2468, 1, &sosAscendingNameSort);
    // if the db exists, CeCreateDatabase will fail
    rc = GetLastError();
    if( rc == ERROR_DUP_NAME )
    {
        // just get the oid, then close
        globalCEOID = 0;
        globalHDB = CeOpenDatabase (&globalCEOID,
                              TEXT("Birthdays"),
                              0, 0, NULL);
        CloseHandle( globalHDB );
        globalHDB = 0;
    }
    // uh-oh...
    if( (rc == ERROR_DISK_FULL )
        ||
        ( rc == ERROR_INVALID_PARAMETER ))
    {
        iCaption = IDS_CANT_CREATE_DB;
        iTitle = IDS_NO_DATABASE;
        { goto FAIL; }
    }
    return TRUE;

Listing 9.2  BetterBirthdays.cpp. (continues)
IDS_NO_REC_ADDED);
}
break;
case IDC_CHANGE:
    if(!AddBirthdayReminder(hDlg, UPDATE_REC))
        {
            iCaption = IDS_INPUT_REQUIRED;
            iTitle = IDS_NO_REC_ADDED;
        }
    InitDlgCtrls(hDlg);
    break;
case IDC_DELETE:
    //open the database
    globalHDB = 
        CeOpenDatabase (&globalCEOID, NULL, 0, 0, NULL);
    //delete the record
    CeDeleteRecord(globalHDB, globalRecOID );
    //close the database
    CloseHandle( globalHDB );
    globalHDB = 0;
    //re-init dialog ctrls
    InitDlgCtrls(hDlg);
    break;

case IDC_SHOW_ALL:
    DialogBox(hInst, TEXT("BirthdayList"),
        hDlg, (DLGPROC)BirthdayListDlgProc);
    break;

case IDOK:
    case IDCANCEL:
    CloseHandle(globalHDB);
    EndDialog(hDlg, LOWORD(wParam));
    return TRUE;
    break;
    default:
        break;
    }//end switch(LOWORD(wParam))
    break;

}//end switch(message)
return FALSE;
FAIL:
    //something is wrong -- fail and bail
    if( globalHDB)
        {
            CloseHandle(globalHDB);
            globalHDB = 0;
        }
DoMessage( iCaption, iTitle );
EndDialog(hDlg, LOWORD(wParam));
return TRUE;
}

/////////////////////////////////////////////////////
//Do Message
/////////////////////////////////////////////////////
//
void DoMessage( int iCaption, int iErrorText)
{
    TCHAR szPrompt1[64];
    TCHAR szPrompt2[64];
    LoadString (hInst, iCaption, szPrompt1, sizeof (szPrompt1));
    LoadString (hInst, iErrorText, szPrompt2, sizeof (szPrompt2));
    MessageBox (NULL, (LPCTSTR)szPrompt2, (LPCTSTR)szPrompt1, MB_OK);
}

/////////////////////////////////////////////////////
//Add a record to the database
/////////////////////////////////////////////////////
//
BOOL AddBirthdayReminder(HWND hDlg, int iAddUpdateFlag)
{
    HWND hwndWho, hwndWhen, hwndShowAll;
    LPTSTR lpwszWho = NULL;
    INT iWhat, i;
    int iTextLen, iButtonState;
    int rc;
    SYSTEMTIME SysTime;
    FILETIME ftBirthday;

    PCEPROPVAL pRecord = 0;
    PBYTE pStringStore = 0;
    DWORD dwIndex;
    //Find out who from the edit control
    hwndWho = GetDlgItem(hDlg,IDC_WHO);
    iTextLen = SendMessage( hwndWho, WM_GETTEXTLENGTH, 0, 0);
    //require user input
    if( !iTextLen )
    {
        return FALSE;
    }
    //allocate for wide character count plus terminal null
    lpwszWho = (TCHAR* )LocalAlloc(LPTR,
    (iTextLen + 1) * sizeof(WCHAR) );
    //get the input string
    SendMessage( hwndWho, WM_GETTEXT, (iTextLen + 1),
    (LPARAM)(LPSTR)lpwszWho);
    //find out when from the date time control
    hwndWhen = GetDlgItem(hDlg,IDC_WHEN);
    memset( &SysTime, 0x0, sizeof(SysTime));
    DateTime_GetSystemtime(hwndWhen,&SysTime);

Listing 9.2  BetterBirthdays.cpp. (continues)
// translate to filetime format
SystemTimeToFileTime(&SysTime, &ftBirthday);
// find out what
for (i = IDC_CANDY; i <= IDC_ANT_FARM; i++)
{
    iButtonState = 
    SendDlgItemMessage(hDlg, i, BM_GETSTATE, 0, 0);
    if (iButtonState == BST_CHECKED)
    {
        iWhat = i;
        break;
    }
}
// allocate an array of CEPROPVAL structures,
// plus storage space for strings
pRecord = (PCEPROPVAL)LocalAlloc(LPTR,
    (sizeof(CEPROPVAL) * NUMBER_BIRTHDAY_PROPS)
    + (iTextLen + 1) * sizeof(WCHAR));
if (!pRecord)
{
    goto FAIL;
}
// init pointer to record's string storage
pStringStore = (PBYTE)pRecord +
    (sizeof(CEPROPVAL) * NUMBER_BIRTHDAY_PROPS);
// populate record
    // init the unused prop val members
for (i = 0; i < NUMBER_BIRTHDAY_PROPS; i++)
{
    pRecord[i].wLenData = 0;
    pRecord[i].wFlags = 0;
}
    // first we write the "who" string into a property
    // write the prop id : ce type + index
pRecord[0].propid = MAKELONG(CEVT_LPWSTR, 0);
    // copy the string to the data buffer at the
    // end of the propval array
wcscpy((TCHAR*)pStringStore, lpwszWho);
    // set the property union member to this ptr
pRecord[0].val.lpwstr = (LPWSTR)pStringStore;
    // next we write the "when" value into a property
    // write the prop id : ce type + index
pRecord[1].propid = MAKELONG(CEVT_FILETIME, 1);
    // now put the FILETIME structure in the value
    // union for this property.
pRecord[1].val.filetime = ftBirthday;
    // now copy the "what" data to a CEPROPVAL structure
    // write the prop id : ce type + index
pRecord[2].propid = MAKELONG(CEVT_I2, 2);
    // convert the data to int
pRecord[2].val.iVal = iWhat;

Listing 9.2 BetterBirthdays.cpp.
//open the database
globalHDB = CeOpenDatabase (&globalCEOID, NULL, 0, 0, NULL);
rc = GetLastError();
if( rc == ERROR_INVALID_PARAMETER )
    { goto FAIL; }
if( globalHDB == INVALID_HANDLE_VALUE )
    { goto FAIL; }
//seek to the end & write the database PROPVAL array
CeSeekDatabase(globalHDB, CEDB_SEEK_END, 1, &dwIndex);
if( iAddUpdateFlag == ADD_REC )
{
    //set oid parm to 0 to create a new rec
    CeWriteRecordProps(globalHDB, 0, NUMBER_BIRTHDAY_PROPS, pRecord);
}
else
{
    //set oid parm to value set by "Find"
    CeWriteRecordProps(globalHDB, globalRecOID,
                       NUMBER_BIRTHDAY_PROPS, pRecord);
}
if( GetLastError() == ERROR_INVALID_PARAMETER )
    { goto FAIL; }
if( GetLastError() == ERROR_DISK_FULL )
    { goto FAIL; }
//free the cepropval struct array
LocalFree( pRecord );

//clean up and return status
CloseHandle(globalHDB);
globalHDB = 0;
if(lpwszWho)
{
    LocalFree( lpwszWho );
}
return TRUE;
FAIL:
return FALSE;
}
void InitDlgCtrls ( HWND hDlg )
{
    HWND hwndWho, hwndWhen, hwndWhat;
    HWND hwndDelete, hwndChange, hwndShowAll;
    //get a handle to the edit ctrl
    hwndWho = GetDlgItem(hDlg,IDC_WHO);
    //Clear the input string
    SendMessage( hwndWho, WM_SETTEXT, 0,
                 (LPARAM)(LPSTR)TEXT(""));
    //Set the choice of gift to default

Listing 9.2  BetterBirthdays.cpp. (continues)
hwndWhat = GetDlgItem(hDlg,IDC_CANDY);
SendMessage( hwndWhat, BM_SETCHECK, TRUE, 0 );
hwndWhat = GetDlgItem(hDlg,IDC_FLOWERS);
SendMessage( hwndWhat, BM_SETCHECK, FALSE, 0 );
hwndWhat = GetDlgItem(hDlg,IDC_ANT_FARM);
SendMessage( hwndWhat, BM_SETCHECK, FALSE, 0 );

// Clear Date Setting
hwndWhen = GetDlgItem( hDlg,IDC_WHEN );
DateTime_SetSystemtime(hwndWhen, GDT_NONE, NULL);

// get a handle to the Change ctrl
hwndChange = GetDlgItem(hDlg,IDC_CHANGE);
// Disable it until after a "Find"
EnableWindow( hwndChange, FALSE);

// get a handle to the Delete ctrl
hwndDelete = GetDlgItem(hDlg,IDC_DELETE);
// Disable it until after a "Find"
EnableWindow( hwndDelete, FALSE);

LRESULT CALLBACK BirthdayListDlgProc(HWND hDlg,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    HWND  hwndList;
    TCHAR szBirthday[64], szWhat[12];
    PBYTE pRecord = NULL;
    PCEPROPVAL lpRecord = NULL;
    CECOID oid = 0;
    DWORD dwIndex = 0;
    DWORD dwRecSize;
    WORD wProps = 0;
    SYSTEMTIME stBirthDate;
    int rc;
    pRecord = NULL;
    switch (message)
    {
    case WM_INITDIALOG:
        // get a handle to the list control
        hwndList = GetDlgItem(hDlg,IDC_BIRTHDAY_LIST);
        break;

        // open the database in AUTOINCREMENT mode, no sort
        // order active
        globalHDB = CeOpenDatabase (&globalCEOID, NULL, 0,
            CEDB_AUTOINCREMENT ,
            NULL);

        rc = GetLastError();
        if( rc == ERROR_INVALID_PARAMETER )
            goto FAIL;
        if( globalHDB == INVALID_HANDLE_VALUE )
            goto FAIL;

Listing 9.2  BetterBirthdays.cpp.
// seek to the beginning of the database
oid = CeSeekDatabase (globalHDB, CEDB_SEEK_BEGINNING, 0, &dwIndex);
if(oid == 0 )
{
    DoMessage( IDS_NO_REC_FOUND, IDS_ABORT );
    EndDialog(hDlg, LOWORD(wParam));
    return TRUE;
}

// Read all properties for the record. Have the system
// allocate the buffer containing the data.
oid = CeReadRecordProps (globalHDB, CEDB_ALLOWREALLOC, &wProps, NULL,
    &pRecord, &dwRecSize);

while( oid != 0 )
{
    lpRecord = (PCEPROPVAL)pRecord;
    // get the Name string
    memset( szBirthday, 0x0, sizeof(szBirthday));

    // get the date
    FileTimeToSystemTime( &lpRecord[1].val.filetime, &stBirthDate);

    // what gift did we have in mind?
    memset(szWhat, 0x0, sizeof(szWhat));
    if(lpRecord[2].val.iVal == IDC_CANDY )
    { wcscpy( szWhat, TEXT("Candy"));};
    if(lpRecord[2].val.iVal == IDC_FLOWERS )
    { wcscpy( szWhat, TEXT("Flowers"));};
    if(lpRecord[2].val.iVal == IDC_ANT_FARM )
    { wcscpy( szWhat, TEXT("Ant Farm"));};

    // format the string
    wsprintf((TCHAR*)szBirthday, TEXT("%s %i/%i/%i %s"),
        lpRecord[0].val.lpwstr, stBirthDate.wDay ,
        stBirthDate.wMonth , stBirthDate.wYear ,
        szWhat);

    // insert the string
    SendMessage(hwndList, LB_INSERTSTRING, -1,
        (long)&szBirthday );

    // Read all properties for the record. Have the system
    // allocate the buffer containing the data.
    oid = CeReadRecordProps (globalHDB, CEDB_ALLOWREALLOC, &wProps, NULL,
        &pRecord, &dwRecSize);
}

CloseHandle(globalHDB);
Listing 9.2  BetterBirthdays.cpp.
CeSeekDatabase (globalHDB, CEDB_SEEK_BEGINNING, 0, &dwIndex);

globalRecOID = CeSeekDatabase (globalHDB, CEDB_SEEK_VALUEFIRSTEQUAL, (DWORD)&propBirthday, &dwIndex);

//check to make sure we found this record
if(globalRecOID == 0)
{
    return FALSE;
}

CloseHandle( globalHDB );
globalHDB = 0;
//enable the delete & update buttons
//get a handle to the Change ctrl
hwndChange = GetDlgItem(hDlg,IDC_CHANGE);
//Disable it until after a "Find"
EnableWindow( hwndChange, TRUE);
//get a handle to the Delete ctrl
hwndDelete = GetDlgItem(hDlg,IDC_DELETE);
//Disable it until after a "Find"
EnableWindow( hwndDelete, TRUE);
//A-Ok!!
return TRUE;

FAIL:
    DoMessage( IDS_CANT_OPEN_DB, IDS_ABORT );
    return FALSE;

Listing 9.2   BetterBirthdays.cpp. (continued)

Using Sort Orders

We refine our use of the database functionality in the BetterBirthdays program. The first and most noticeable change is that this time the Birthdays database has a sorted field name field. We specify this attribute at the time the database is created, and we create the database in BirthdayDlgProc() in response to the WM_INITDIALOG message:

    ceSortProp = MAKELONG(CEVT_LPWSTR, 0);
    switch (message)
    {
        case WM_INITDIALOG:
            //Init the dialog controls
            break;
    }
In order to create a database in which the properties behave like key fields in a tabular database, we provide information at database creation time that supports traversing the database records based on the values of specific properties (or fields). This information is encoded in a SORTORDERSPEC structure. The typedef for SORTORDERSPEC is shown below:

```c
typedef struct _SORTORDERSPEC {
    CEPROPID propid;
    DWORD dwFlags;
} SORTORDERSPEC;
```

The propid member is the now familiar CEPROPID. The value of the dwFlags member defines the sorting mechanism that will operate on this property (see Table 9.2).

### Table 9.2  Sort Order Flags and Their Meanings

<table>
<thead>
<tr>
<th>SORTING MECHANISM FLAG</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEDB_SORT_DESCENDING</td>
<td>The sort is done in descending order. By default, the sort is done in ascending order.</td>
</tr>
<tr>
<td>CEDB_SORT_CASEINSSENSITIVE</td>
<td>The sort operation is case-insensitive. This value is valid only for strings.</td>
</tr>
<tr>
<td>CEDB_SORT_UNKNOWNFIRST</td>
<td>Records that do not contain this property are placed before all the other records. By default, such records are placed after all other records.</td>
</tr>
<tr>
<td>CEDB_SORT_GENERICORDER</td>
<td>The default sort order—ascending order, case-sensitive—is used. Records that do not contain the sort property are placed at the end of the list.</td>
</tr>
</tbody>
</table>

A database can have a maximum of four fields with active sort orders. The call to CeCreateDatabase() for the new, improved Birthdays database looks like this:

```c
//Create data store and save the CEOID
globalCEOID = CeCreateDatabase(TEXT("Birthdays"),
```
A successful call will create a Birthdays database in which we have one sorted field, names, of type CEVT_LPWSTR, at index 0. The names are sorted in ascending alphabetical order, without sensitivity to case:

```c
//if the db exists, CeCreateDatabase will fail
rc = GetLastError();
if( rc == ERROR_DUP_NAME )
{
    //just get the oid, then close
    globalCEOID = 0;
    globalHDB = CeOpenDatabase (&globalCEOID,
    TEXT("Birthdays"),
    0, 0, NULL);
    CloseHandle( globalHDB );
    globalHDB = 0;
    CeDeleteDatabase(globalCEOID);
    goto FAIL;
}
```

If the database already exists when we try to create it, CeCreateDatabase() will fail. This case becomes important in the BetterBirthdays example because we created a database of this name in the BirthdayReminders example program. However, that database didn’t have any sort orders specified. To modify the Birthday database’s attributes, we have to delete the old database and create a new one, using CeDeleteDatabase(). Here’s the declaration for the API we use to delete a database:

```c
BOOL (CEOID oidDbase);
```

CeDeleteDatabase() has a single parameter, the database CEOID. In part, this is because you can’t delete an open database. In response to the error status ERROR_DUP_NAME, we open the existing database by name, get its CEOID, close the database, and then we are able to delete the database identified by the CEOID.

Much of the functionality of the BetterBirthdays example hinges on the ability to find a specific record. For this reason, we’ll explore the FindBirthday() function next:

```c
BOOL FindBirthday(HWND hDlg)
{
    HWND hwndWho, hwndChange, hwndDelete;
    TCHAR* lpwszWho;
    int iTTextLen, rc;
    DWORD dwIndex =0;
    CEPROPVAL propBirthday;
    //Find out who from the edit control
```
hwndWho = GetDlgItem(hDlg,IDC_WHO);
iTextLen = SendMessage( hwndWho, WM_GETTEXTLENGTH, 0, 0);
//require user input
if( !iTextLen )
    {return FALSE;}
//allocate for wide character count plus terminal null
lpwszWho = (TCHAR*)LocalAlloc(LPTR, (iTextLen + 1) * sizeof(WCHAR) );
//get the input string
SendMessage( hwndWho, WM_GETTEXT, (iTextLen + 1),
            (LPARAM)(LPSTR)lpwszWho);

The FindBirthday() function is called when the user clicks the “Find” button. The first few lines of the function get the name to find from the edit control, detecting an empty string if there was no input:

    //open the database with an active sort order
    globalHDB = CeOpenDatabase (&globalCEOID, NULL,
                                MAKELONG(CEVT_LPWSTR, 0),
                                0, NULL);
    rc = GetLastError();
    if( rc == ERROR_INVALID_PARAMETER )
        { goto FAIL; }
    if( globalHDB == INVALID_HANDLE_VALUE )
        {goto FAIL; }

Next we open the database, specifying an active sort order. The parameters to CeOpenDatabase(), in the order shown, are the CEOID of the database to open, a NULL place holder for the pointer to the database name, the CEPROPID of the field on which to activate sorting, an open flag value of zero, and a NULL placeholder for the handle of the Window to receive database change notification messages:

    //seek to this rec
    propBirthday.propid = MAKELONG(CEVT_LPWSTR, 0) ;
    propBirthday.val.lpwstr = lpwszWho;
    propBirthday.wLenData = 0;
    propBirthday.wFlags = 0;
    globalRecOID =

Finding the record that contains the name specified by the user is a three-step process. First we initialize a CEPROPVAL structure with information that specifies what field we plan to search. In this case, we set the propid member to the name property, and we set the val.lpwstr member to the string for which we are going to search:

    CeSeekDatabase (globalHDB,
                    CEDB_SEEK_BEGINNING, 0, &dwIndex);

Next we initialize the position in the database by seeking to the first record. The parameters to CeSeekDatabase(), in the order shown, are the handle to the database,
a seek type flag, a NULL placeholder for the value to seek, and the address of the variable that will contain the current record index on successful return:

```
globalRecOID =
    CeSeekDatabase (globalHDB,
    CEDB.Seek_VALUENAMEQUAL,
    (DWORD)&propBirthday, &dwIndex);
```

Finally, we perform a search for the value specified by the user. Put another way, we seek to a record containing a property that matches the search criteria.

This time the parameters to CeSeekDatabase() are the database handle, a seek flag that initiates a search for the first record containing an exact match for the value, the address of the CEVALPROP structure we initialized with the search data, and the address of the variable that contains the index of the found record on successful return. Notice that we save the returned CEOID of the matching record. This value precisely and uniquely identifies the record, so that we may manipulate it directly, referencing it by its CEOID in later operations.

Table 9.3 lists the seek type flags and their meanings.

### Table 9.3 Seek Type Flags Used by CeSeekDatabase()

<table>
<thead>
<tr>
<th>SEEK TYPE FLAG</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEDB.Seek_CEOID</td>
<td>Seek until finding an object that has the object identifier given by the dwValue parameter. This seek operation is very efficient.</td>
</tr>
<tr>
<td>CEDB.Seek_VALUESMALLER</td>
<td>Seek until finding the largest value that is less than a specified value. If no record matches, the seek pointer is left at the end of the database and the function returns zero. The dwValue parameter gives the address of a CEPROPVAL structure initialized with the search information.</td>
</tr>
<tr>
<td>CEDB.Seek_VALUENAMEQUAL</td>
<td>Seek until finding the first exact match. If no record matches, the seek pointer is left at the end of the database, and the function returns zero. The dwValue parameter gives the address of a CEPROPVAL structure initialized with the search information.</td>
</tr>
<tr>
<td>CEDB.Seek_VALUENEXTEQUAL</td>
<td>Seek one position forward in the sorted order. If the next record is an exact match, return its OID. Otherwise, return zero and leave the seek pointer at the end of the database. Useful in combination with CEDB.Seek_VALUENAMEQUAL to find all records with an equal value. The dwValue parameter gives the address of a CEPROPVAL structure initialized with the search information.</td>
</tr>
</tbody>
</table>

(continues)
### Table 9.3  Seek Type Flags Used by CeSeekDatabase() (Continued)

<table>
<thead>
<tr>
<th>SEEK TYPE FLAG CONSTANT</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEDB.Seek.VALUEGREATER</td>
<td>Find a value greater than or equal. If no records are found, the seek pointer is left at the end of the database and the function returns zero. The dwValue parameter gives the address of a CEPROPVAL structure initialized with the search information.</td>
</tr>
<tr>
<td>CEDB.Seek.BEGINNING</td>
<td>Seek an absolute number of records from the beginning of the database. The dwValue parameter gives the number of records to seek.</td>
</tr>
<tr>
<td>CEDB.Seek.CURRENT</td>
<td>Seek backward or forward an absolute number of records, given by the dwValue parameter. Seeks forward if dwValue is a positive value, or backward if it is negative.</td>
</tr>
<tr>
<td>CEDB.Seek.END</td>
<td>Seek backward from the end of the database an absolute number of records given by the dwValue.</td>
</tr>
</tbody>
</table>

**PORTING TIP** Seeks based on record values may be performed only on sorted fields.

The rest of the FindBirthday() function consists of the routine cleanup tasks such as freeing allocations and closing the database.

### Deleting a Record from the Database

Now let’s examine the ways in which we can use the record’s CEOID, which is returned by a successful call to CeSeekDatabase(). (Note that we saved this value in the global variable globalRecOID.)

Deleting a record is a straightforward operation, so we will start there. Below is a fragment of code from the message switch of the BirthdayDlgProc() function:

```c
  case IDC_DELETE:
    //open the database
    globalHDB =
      CeOpenDatabase(&globalCEOID, NULL, 0, 0, NULL);
    //delete the record
    CeDeleteRecord(globalHDB, globalRecOID);
    //close the database
    CloseHandle( globalHDB );
    globalHDB = 0;
    //re-init dialog ctrls
```
To delete a record, we use the simplest form of the CeOpenDatabase() function, identifying the database with its previously saved CEOID, globalCEOID, and setting all the remaining parameters to zero. Using the handle to the database returned by CeOpenDatabase() and the record CEOID saved after the prior call to FindBirthday(), we delete the record with a call to CeDeleteRecord(), and then close the database.

**Modifying an Existing Record**

We can also access a record by CEOID in order to make modifications. Implementing this capability requires only a few enhancements to the AddBirthdayReminder() function. We used AddBirthdayReminder() in the first example program, BirthdayReminder. Since this new version of the function is similar in many ways to the earlier one, we’ll just highlight the changes here.

First, notice that we have added an item to the original function’s parameter list. The new parameter, an int, is used as a flag. It can assume one of two values: ADD_REC or UPDATE_REC. The value of this flag determines whether the function adds a new record or updates an existing one:

```c
BOOL AddBirthdayReminder(HWND hDlg, int iAddUpdateFlag) {
```

Another change is that in this version of the program, we no longer store the birth date as the SYSTEMTIME structure that is returned by the dialog’s Date picker control:

```c
//retrieve the birthdate from the date time control
hwndWhen = GetDlgItem(hDlg,IDC_WHEN);
memit( &SysTime, 0x0, sizeof(SysTime));
DateTime_GetSystemtime(hwndWhen,&SysTime);
//translate to filetime format
SystemTimeToFileTime( &SysTime, &ftBirthday );
```

Rather, we convert the SYSTEMTIME value to a FILETIME. The FILETIME format is much more efficient, for two reasons. First, it encodes all the information in the SYSTEMTIME in a single DWORD, saving a great deal of storage space. Second, the CE native database type CEVT_FILETIME directly supports FILETIME values, so its translated value can be stored directly in the CEVALUNION member val.filetime. Here’s the new CEPROPVAL structure initialization for this property:

```c
//next we write the "when" value into a property
//write the prop id : ce type + index
```
pRecord[1].propid = MAKELONG(CEVT_FILETIME, 1);
//now put the FILETIME structure in the value
//union for this property.
pRecord[1].val.filetime = ftBirthday;

When we write the record to the database, one of two circumstances will prevail: either we’ll be creating a new record at the end of the database, or we’ll be updating an existing record that we can precisely and immediately locate by its CEOID. For this reason, we can do a simple database open operation:

//open the database
globalHDB = CeOpenDatabase(&globalCEOID, NULL, 0, 0, NULL);
rc = GetLastError();

If we detect that this is a record addition, we seek to the end of the database and write the record:

if( iAddUpdateFlag == ADD_REC )
{
    //seek to the end & write the database PROPVAL array
    CeSeekDatabase(globalHDB, CEDB_SEEK_END, 1, &dwIndex);
    //set oid parm to 0 to create a new rec
    CeWriteRecordProps(globalHDB, 0,
                       NUMBER_BIRTHDAY_PROPS, pRecord);
}

If, instead, this is an update, we don’t have to seek at all. We simply reference the existing record through its previously retrieved and saved CEOID, and write the new values directly into it:

else
{
    //set oid parm to value set by "Find"
    CeWriteRecordProps(globalHDB, globalRecOID,
                       NUMBER_BIRTHDAY_PROPS, pRecord);
}

Iterating Records

Iterative reading of CE database records is a pretty slick situation, as you probably guessed, based on the fact that there is a special database open mode for doing it. This technique is demonstrated in the BirthdayListDlgProc(), which is invoked when the user clicks the BirthdayReminder dialog’s Show All button.

The work of creating a list of birthday reminder records is all done in response to the WM_INITDIALOG message. Let’s examine the database access related parts of the BirthdayListDlgProc().
case WM_INITDIALOG:
    // get a handle to the list control
    hwndList = GetDlgItem(hDlg, IDC_BIRTHDAY_LIST);

    // open the database in AUTOINCREMENT mode, no sort
    // order active
    globalHDB = CeOpenDatabase (&globalCEOID, NULL, 0,
                                CEDB_AUTOINCREMENT ,
                                NULL);

    First, we open the database using its CEOID and setting the CEDB_ 
    AUTOINCREMENT flag. This flag causes the current record location to be 
    incremented after every read operation.

    oid = CeSeekDatabase (globalHDB, CEDB_SEEK_BEGINNING,
                          0, &dwIndex);

    Next, we seek to the beginning of the database in order to establish a current 
    record:

    pRecord = NULL;
    wProps = NULL;
    .
    .
    // Read all properties for the record. Have the system 
    // allocate the buffer containing the data.
    oid = CeReadRecordProps (globalHDB, CEDB_ALLOWREALLOC,
                              &wProps, NULL,
                              (LPBYTE )pRecord, &dwRecSize);

    Now we call CeReadRecordProps(). The parameters, in the order shown, are:

    - The handle to the database; a flag that allows the function to allocate a 
      correctly sized record buffer in our behalf
    - The address of the variable wProps has been set equal to null, which on 
      return will contain a count of the record’s properties
    - A pointer to an array of CEPROPIDs for which to retrieve values, set to 
      NULL here to indicate all properties should be returned
    - The address of a pointer to the buffer (a pointer to a pointer) containing 
      the retrieved record, set to NULL to indicate the system should allocate 
      the record buffer
    - The address of a variable that will contain the size in bytes of the 
      returned buffer on successful return

    This all sounds fairly complicated, and it could turn out to be that way in 
    certain circumstances. However, most of the time, the best and easiest way to 
    make this call is just as you see it above. Let the system allocate the record
buffer, and retrieve all the properties at one time because this is much more efficient than any other scheme:

```c
while( oid != 0 )
{
    lpRecord = (PCEPROPVAL)pRecord;
    // get the Name string
    memset( szBirthday, 0x0, sizeof(szBirthday));
}
```

We cast the pointer to the buffer returned by CeReadRecordProps() to a PCEPROPVAL so that you can use it to access the individual CEPROPVAL structures. Now we are ready to create the list box strings:

```c
// get the date
FileTimeToSystemTime( &lpRecord[1].val.filetime,
    &stBirthDate);
// what gift did we have in mind?
memset(szWhat, 0x0, sizeof(szWhat));
if(lpRecord[2].val.iVal == IDC_CANDY )
    { wcscpy( szWhat, TEXT("Candy"));};
if(lpRecord[2].val.iVal == IDC_FLOWERS )
    { wcscpy( szWhat, TEXT("Flowers"));};
if(lpRecord[2].val.iVal == IDC_ANT_FARM )
    { wcscpy( szWhat, TEXT("Ant Farm"));};
// format the string
wsprintf((TCHAR*)&szBirthday,
    TEXT("%s %i/%i/%i %s"),
    lpRecord[0].val.lpwstr,
    stBirthDate.wDay ,
    stBirthDate.wMonth ,
    stBirthDate.wYear,
    szWhat);
// insert the string
SendMessage(hwndList, LB_INSERTSTRING, -1,
    (long)&szBirthday );
```

To make list box strings out of the returned property values, we convert the birth date stored as a FILETIME back to a SYSTEMTIME, select a gift item based on the stored integer value, and use the CEVALUNION member val.lpwstr to pass the name string to wsprintf(). Presto, formatted birthday reminder record data in a Unicode string.

**A Few Final Words on CE Databases**

We’ve touched on a great many things you can do with the CE database API, a toolset that is both flexible and powerful. However, there are some inherent limitations of which to be aware. The first is that a database may contain only
one kind of record. The second, which is actually a corollary to the first, is that you can’t explicitly create one-to-many or many-to-one record relationships. In a strict sense, it’s also true that the Windows CE database API is really an object store, not a true database. It shares no relationship with standards relational database tools such as SQL and the like. Personally, I don’t see any of these facts as particularly troubling or intrusive, for the reason the CEBLOB type allows you to store sophisticated, application-specific data structures inside a record. This is a very extensible design, and so far, it’s been more than adequate to my needs.

Looking Ahead

One of the very greatest strengths of the integrated nature of CE database API is that it allows you to find other databases that exist on a device, query their structures, and dynamically incorporate their data. In upcoming chapters, we’ll build on the skills you’ve learned here. You’ll see how to connect to a Windows CE database from the desktop using the Remote Database API. You’ll learn how to remotely enumerate databases, find out what kind of information they contain, and manipulate those data in your applications.

We close this section with a brief look at the things you should do and avoid doing, with an eye to conserving battery power. Application behaviors can have a dramatic impact on the battery life, so it’s worth understanding your role in the all important task of power conservation.
We began this section with stern admonishments about keeping things lightweight: Use memory sparingly and efficiently, release allocations as quickly as possible, and find ways to condense the lavish desktop Windows feature set to which we’ve become accustomed. With a bit of experience, it becomes clear that practices such as these are the only way to keep from running out of space on a CE device. There is another, equally dire circumstance we must avoid: running out of juice.

Consider this: A pair of AA batteries powers the CE devices I own, with a watch battery for backup. Both run MIPS RISC series processors, support PC card storage and various peripherals, and integrate modem access to networks. To my way of thinking, this is next door to druid magic, and, in fact, the whole setup does rely to a certain extent on a bit of smoke and mirrors, a.k.a. the Windows CE power management strategy. It takes an extraordinary amount of parsimony to keep a sophisticated computing device running for weeks at a time on wattage that would scarcely keep a flashlight burning for a couple of hours. Ultimately, all of our conservation efforts under CE share the aim of making the most of a very limited power supply.

In this brief chapter, we explore the application programmer’s mostly collaborative role in extending the battery life of the device. This task sounds
overwhelming. However, it’s mostly a matter of staying out of the way, and allowing the inherently low power consumption design of Windows CE to do its work.

**CE Power Management Design Assumptions**

Power conservation is so important that, for the most part, the Windows CE operating system takes it out of our hands. At the heart of the design of Windows CE is a power management mechanism that puts the device into the lowest power consumption mode that is appropriate to the current state of its use. For example, when a device is “off,” it consumes only enough power to maintain volatile memory. When the device is actively executing applications, it enters a low power consumption mode whenever the processor is idle. Surprisingly enough, even a busy device idles the processor around 90 percent of the time.

The original architects of CE counted on certain power consumption patterns when they designed the operating system. Here is a summary of the original design assumptions:

- The device is in use less than two hours per day.
- The duration of individual device usage events ranges from 5 minutes to 1 hour.
- The display is fully powered when the device is in use.
- When the device is in use, the processor is busy 10 percent of the time, idle 90 percent of the time.
- The device has main batteries plus a backup battery.
- There is no nonvolatile, writable memory.
- Battery life targets are calculated without accounting for add-in devices such as PC cards or Flash storage.

In a nutshell, CE devices prolong battery life by doing nothing as often as possible! Typical Windows CE applications collaborate in this scheme passively, employing a simple and elegant strategy. Here’s how it works.

**CE’s Blocking Message Loop**

Let’s start by defining a few terms. A *process* is a single instance of an executing application. Each process has its own address space, in which reside its code and data. A *thread* is a unit of execution, and each process has at least one thread. Threads belong to a specific process and share the memory resources
of the process that creates them. In a multitasking system, individual threads run consecutively, each for a small increment of time, or *timeslice*. Threads are allotted a timeslice based on the operating system’s scheduling algorithm. The scheduling algorithm is designed to give preference to time-critical threads (device drivers, for example) and to defer lower priority jobs. By default, application threads are fairly low priority and receive a brief but adequate timeslice. When a thread isn’t scheduled, it is blocked from executing. A blocked thread uses no processor resources. From the viewpoint of a user, the overall effect of consecutively scheduling threads is that the system runs smoothly, apparently running end user applications and handling operating system housekeeping “simultaneously.” If all goes as planned, it’s not unlike animation—at 16 frames per second, Mickey Mouse sails his tugboat serenely into the sunset.

There are two key points here, and they are somewhat subtle, so I’ll emphasize them:

In a multitasking environment, threads that hog the processor by inflating their priority or the duration of their timeslice will degrade system performance, because other threads starve. The appearance of simultaneity depends on each thread getting frequent, short timeslices.

In a Windows CE system, threads that hog the processor keep it from idling and will rapidly drain power supplies.

In a typical, single-threaded Win32 application, we don’t have to worry much about either of these things happening, because the GetMessage() function that we call in our message processing code automatically blocks the application’s one and only thread whenever it is waiting for message delivery. In practice, this is far in excess of the assumed 90 percent of the time, so by doing the usual thing, we conserve battery power:

```c
// Main message loop blocks the thread if there are no messages:
while (GetMessage(&msg, NULL, 0, 0))
{
    // if (!TranslateAccelerator (msg hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

By contrast, the function PeekMessage() doesn’t block the application’s single thread, and a loop like this one will dramatically reduce battery life:

```c
//YOW!! THIS DRAINS THE BATTERIES!!!
if (!PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
{
    //doing something or other...
}
```
How to Make Sure You Don’t Sidestep the System’s Power Conservation Defenses

As you’ve seen in the first two sections of this book, a great deal of the effort involved in porting Win32 from the desktop to a CE device has to do with redefining application strategy. Put another way, there are a lot of Win32 strategies that we choose to omit or modify because they are incompatible with the scale of Windows CE. In addition to the PeekMessage() example shown above, there are a few other desktop practices that can have dramatic consequences for power consumption. Here are three types of behavior you should approach with care.

**Threading**

Threading is a highly complex application behavior. Chances are good that if you went to the trouble of implementing a design that included threading on the desktop, it was the only workable solution in that context. The key danger on Windows CE is the chance of creating threads that have a tendency to be or to become nonblocking. On Windows CE, only an application’s main thread can receive messages, and the modifying parameters used to create threads are much less numerous than on the desktop. Also, threads are not allowed to outlive the process that creates them. You may find the CE threading model doesn’t support a straight across port of your desktop design. If this turns out to be the case, consider omitting features that require threading.

If you choose to use threading, here are two caveats:

- Use mutexes or events to force threads to block when they are not engaged in their intended task.
- Test threaded code on the full range of physical devices you intend it to target.

**Timers and Other Polling Behaviors**

On the desktop, we can afford to implement polling behaviors, where every time a given interval elapses, we check status information on something. By its very nature, polling is inefficient, because in the vast majority of iterations, you’ll make a test that tells you it’s time to do nothing and check again later. Very often, timers are set for this purpose on the desktop. Check the code you are porting to see if it processes the WM_TIMER message or calls the SetTimer() function. Either of these may be the source of application behavior that tends to defeat attempts to idle the processor, causing diminished battery life.
Spinning

Spinning is a kind of looping that unintelligently persists until some arbitrary threshold is met. Here’s a loop that spins.

```c
while(1)
{
    //do something or other
    if( bBoredSenseless )
    { break; }
}
```

This kind of behavior is terrible for battery life, because the processor is active for 100 percent of the thread’s scheduled timeslice. Unfortunately, it is harder to find a spinning loop in a body of code than to find timers or multithreading. If your ported code seems to drain the batteries on a CE device noticeably, look for while()s that have a constant as the parameter and for ( ; ; ), where either the comparison or the action clause have an unusual or constant form.

Looking Ahead

In the next section, we will turn our attention to the power of Windows CE devices when used in collaboration with desktop Windows. The true distinction of Windows CE arises from the seamless pairing that occurs when desktop Windows and mobile devices connect and take advantage of distributed processing to leverage the mobility of the handheld.
To my way of thinking, the second best thing about Windows CE is that it leverages the vast body of code and knowledge that reside in the Win32 development community. The best thing about CE, and the thing in which lies its true genius and our great opportunity, is that it integrates and extends Windows, the most popular centralized computing model of our time. What we have in this technical partnership is something entirely new and profoundly greater than the sum of its parts. CE devices are poised to become eyes and ears, hands and feet, so to speak, of the existing infrastructure of computing.

This section is devoted to helping you enlarge and enrich your applications through the advantageous marriage of CE’s dispersed computing model and the power, productivity, and ubiquity of desktop Windows. We’ll accomplish this using CE’s Remote Programming Interface (RAPI), which allows us to control and interact with remote CE devices.

RAPI provides us with three important capabilities: First, we can use Windows desktop RAPI functionality to interact with any CE device, regardless of processor type. Second, we can manage and configure connected devices exclusive of user involvement. Third, and perhaps most exciting, we can dynamically perform processing and storage functions for a connected device remotely, overcoming many of the inherent limitations of CE devices.

Clearly, what makes a CE-based device an evolutionary leap is its collegiality with the Windows world. As we undertake this look at desktop Windows programming for CE, we shift gears a bit in terms of our approach to programming tools. While there are good reasons for using Windows 32 to write
native CE code, these wilt a bit when we move to the desktop. In terms of productivity and convenience, MFC is a more appealing tool for writing sophisticated desktop applications, and we’ll exploit its rich functionality in order to explore the uses of RAPI.

Aside from the productivity benefits, there are two additional reasons for this switch: The first is that demographic studies of the Win32 community tell us that most Win32 programmers have been writing Windows code a long time. They know Win32 because they started there (or moved there from earlier 16-bit versions of Windows), but most also have some experience with MFC. By contrast, most developers that program Windows using MFC exclusively are newer to the Windows world, and for them, the Win32 programming model seems both obscure and inefficient. This second group of developers is much larger than the first and may well become an important force in the evolution of CE, because they can use the familiar MFC paired with RAPI to write CE applications that run from the desktop.

In sum, if you’re an MFC programmer trying to catch the Windows CE wave, this section will read more comfortably for you. If you’re a Win32 veteran, you’ll find some MFC recipes in this section that make the job of manipulating CE devices from the desktop fast and convenient.

On to the main course—monitoring, managing, and enhancing the capability of CE devices using desktop Windows and RAPI.
Now we begin our exploration of the Windows CE Remote Application Programming Interface, a.k.a. RAPI. Here’s some good news for MFC Windows developers. Even if you can’t bear the idea of learning the Win32 interface in order to target CE, you can still write tight, fast, sophisticated code for CE devices—and run it from desktop Windows. RAPI is a mechanism for managing, monitoring, troubleshooting, and potentially even taking over the operation of connected CE devices.

In this chapter, you’ll see how to get status and configuration information from the CE side, and also how to access the file system and registration database. Many of the RAPI functions we use for this purpose are an exact parallel of their CE side counterparts, a few are slightly different, and some don’t have a CE side analog at all. Before we delve into this application, I’m going to reiterate a few points for readers who are chiefly interested in using MFC and RAPI, as opposed to writing native CE code.

**Programming Tips**

- The character set used on the desktop is technically known as the Multi-Byte Character Set, but for English and most European languages, the actual characters are one byte long.
- RAPI string arguments are Unicode, or wide characters (wchar_t). Passing single byte or multibyte character arguments will cause the RAPI functions to fail.

- To make a Unicode literal string, use the L macro: L"Unicode Literal String"

- To translate between ASCII and Unicode, use the functions mbstowcs() and wcstombs():

  ```
  //Unicode to Multi Byte Character translation
  size_t wcstombs( char *mbstr, 
                    const wchar_t *wcstr, size_t count );
  
  //MBCS to Unicode Character Translation
  size_t mbstowcs( wchar_t *wcstr, 
                   const char *mbstr, size_t count );
  ```

- To perform string manipulations on Unicode strings, use the family of string handling functions that begin with wcs; for example, wcscpy() is the Unicode analog of strcpy().

- Don’t attempt to treat the names of Unicode string buffers as addresses. To get the address of a Unicode string buffer, either create and initialize a separate pointer variable or take the address of the first element of the Unicode buffer using a typecast:

  ```
  //Using a separate pointer variable
  WCHAR wszUnicodeStringBuffer[10];
  WCHAR*pwszUnicodeStringBuffer;
  pwszUnicodeStringBuffer =
    &wszUnicodeStringBuffer[0];
  
  //Using a typecast
  (LPWSTR)& wszUnicodeStringBuffer[0]
  ```

- Don’t treat the Unicode character count as the size in bytes of the string. Multiply character counts by sizeof(WCHAR) to correctly size buffers.

- Don’t try to distinguish Unicode characters by comparing them to integers.

Just one more thing before we begin examining the RapiDemo example. If you are connecting to a CE device using a serial link (rather than Ethernet or a network connection), these programs may execute very slowly. Depending on what you are doing, the access time may be up to a couple of minutes. Walking the registry tree or getting the files from a large directory are examples of this behavior.

Because of the large volume of code necessary to run an MFC application, I have chosen to omit some of the project files from the source listings shown in this text. The omitted files fall into two categories: the files that contain the application’s resources, and those which consist of boilerplate generated by the development environment. On balance, these wouldn’t add to our explorations, and
their sheer bulk might tend to obscure the more important material. The RapiDemo project can be found in its entirety on the accompanying Wiley Website. Table 11.1 and Listing 11.1 outlines its components.

Table 11.1  RapiDemo Project Source Files and Their Uses

<table>
<thead>
<tr>
<th>SOURCE FILE NAME</th>
<th>HEADER FILE NAME</th>
<th>IMPLEMENTS</th>
<th>FILE SHOWN IN CHAPTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>MainFrame.cpp</td>
<td>MainFrame.h</td>
<td>Default frame window</td>
<td>No</td>
</tr>
<tr>
<td>RapiDemo.cpp</td>
<td>RapiDemo.h</td>
<td>Application object</td>
<td>Yes</td>
</tr>
<tr>
<td>RapiDemo.rc</td>
<td>RapiDemo.h</td>
<td>Resources</td>
<td>No</td>
</tr>
<tr>
<td>RapiDemoDoc.cpp</td>
<td>RapiDemoDoc.h</td>
<td>Default document</td>
<td>No</td>
</tr>
<tr>
<td>stdafx.cpp</td>
<td>Stdafx.h</td>
<td>MFC support</td>
<td>No</td>
</tr>
<tr>
<td>RapiDemoView.cpp</td>
<td>RapiDemoView.h</td>
<td>Default View</td>
<td>No</td>
</tr>
<tr>
<td>AllPages.cpp</td>
<td>AllPages.h</td>
<td>Property Sheet Object</td>
<td>Yes</td>
</tr>
<tr>
<td>SystemStatusPage.cpp</td>
<td>SystemStatusPage.h</td>
<td>Property Page</td>
<td>Yes</td>
</tr>
<tr>
<td>WalkReg.cpp</td>
<td>WalkReg.h</td>
<td>Property Page</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Listing 11.1  Partial source file listing for RapiDemo example. (continues)
Listing 11.1  Partial source file listing for RapiDemo example.
Listing 11.1  Partial source file listing for RapiDemo example. (continues)
void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    ///{AFX_DATA_MAP(CAboutDlg)
    ///}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    ///{AFX_MSG_MAP(CAboutDlg)
    // No message handlers
    ///}AFX_MSG_MAP
END_MESSAGE_MAP()
// App command to run the dialog
void CRapiDemoApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

Listing 11.1 Partial source file listing for RapiDemo example.

392 Chapter 11
Listing 11.1  Partial source file listing for RapiDemo example. (continues)
CAllPages::~CAllPages()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////
// CAllPages message handlers
void CAllPages::AddPropPages()
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_psh.dwFlags |= PSP_USEHICON;
    m_psh.hIcon = m_hIcon;
    m_psh.dwFlags |= PSP_NOAPPLYNOW;    // Lose the Apply Now button
    m_psh.dwFlags &= ~PSH_HASHELP;         // Lose the Help button
    AddPage(&m_RemoteFileAccessPage);
    AddPage(&m_WalkRegPage);
    AddPage(&m_SystemStatusPage);
}

RemoteFileAccessPage.h
#if !defined
(AFX_REMOTEFILEACCESSPAGE_H__48BEECA2_EFC7_11D5_90BE_24CF09C10000__INCLUDED_)
#define
AFX_REMOTEFILEACCESSPAGE_H__48BEECA2_EFC7_11D5_90BE_24CF09C10000__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// RemoteFileAccessPage.h : header file
///
//://{afxdata}CRemoteFileAccessPage
class CRemoteFileAccessPage : public CPropertyPage
{
    DECLARE_DYNCREATE(CRemoteFileAccessPage)
    // Construction
    public:
    DWORD m_FAPFlag;
    CRemoteFileAccessPage();
    ~CRemoteFileAccessPage();
    // Dialog Data
    ///{AFX_DATA(CRemoteFileAccessPage)
    enum { IDD = IDD_FILE_ACCESS_DIALOG };
Listing 11.1  Partial source file listing for RapiDemo example. (continues)
DDX_Control(pDX, IDC_LAST_WRITE_TIME_CHECK, m_GetLastWriteTime);
DDX_Control(pDX, IDC_FOLDERS_ONLY_CHECK, m_GetFoldersOnly);
DDX_Control(pDX, IDC_CREATION_TIME_CHECK, m_GetCreationTime);
DDX_Control(pDX, IDC_UPDATE_FILE_BUTTON, m_UpdateFilesList);
DDX_Control(pDX, IDC_PATH_LIST, m_SearchPath);
DDX_Control(pDX, IDC_FILES_LIST, m_FilesList);
//}}AFX_DATA_MAP

BEGIN_MESSAGE_MAP(CRemoteFileAccessPage, CPropertyPage)
//{{AFX_MSG_MAP(CRemoteFileAccessPage)
ON_BN_CLICKED(IDC_UPDATE_FILE_BUTTON, OnUpdateFileButton)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

Listing 11.1 Partial source file listing for RapiDemo example.
char szLineBuff[MAX_PATH];

for( int i = 0; i < dwFoundCount; i++ )
{
    wcstombs( szLineBuff, pFindDataArray[i].cFileName, sizeof(szLineBuff));
    m_FilesList.InsertString( -1, szLineBuff );
    if( m_GetCreationTime.GetCheck() )
    {
        CTime FileCreationTime(pFindDataArray[i].ftCreationTime, -1);
        CString csBuff = FileCreationTime.Format("%A, %B %d, %Y");
        sprintf( szLineBuff, "  %s: %s", "Creation Time", csBuff.GetBuffer(csBuff.GetLength()) );
        m_FilesList.InsertString( -1, szLineBuff );
    }
    if( m_GetLastWriteTime.GetCheck() )
    {
        CTime LastWriteTime(pFindDataArray[i].ftCreationTime, -1);
        CString csBuff = LastWriteTime.Format("%A, %B %d, %Y");
        sprintf( szLineBuff, "  %s: %s", "Last Write", csBuff.GetBuffer(csBuff.GetLength()) );
        m_FilesList.InsertString( -1, szLineBuff );
    }
    if( m_GetCEOID.GetCheck() )
    {
        sprintf( szLineBuff, "  %s: 0x%x", "CEOID", pFindDataArray[i].dwOID );
        m_FilesList.InsertString( -1, szLineBuff );
    }
} // end for()

CeRapiFreeBuffer((LPVOID) pFindDataArray);
CeRapiUninit();

BOOL CRemoteFileAccessPage::OnInitDialog()
{
    CPropertyPage::OnInitDialog();
    // Insert Search List Items
    m_SearchPath.InsertString( -1, "\*." );
    m_SearchPath.InsertString( -1, "\Windows\*.*" );
    // Set search list selection
    m_SearchPath.SetCurSel( 0 );
    // Clear File Attribute Flag members
    m_GetLastWriteTime.SetCheck(0);
    m_GetCreationTime.SetCheck(0);
    m_GetFoldersOnly.SetCheck(0);
    m_GetCEOID.SetCheck(0);
    return TRUE;  // return TRUE unless you set the focus to a control

Listing 11.1  Partial source file listing for RapiDemo example. (continues)
// EXCEPTION: OCX Property Pages should return FALSE

SystemStatusPage.h
#if
!defined(AFX_SYSTEMSTATUSPAGE_H__3491DFF3_EE4F_11D5_90BE_24CF09C10000_INCLUDED_)
#define AFX_SYSTEMSTATUSPAGE_H__3491DFF3_EE4F_11D5_90BE_24CF09C10000_INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#endif
// SystemStatusPage.h : header file
().'/

// CSystemStatusPage dialog
class CSystemStatusPage : public CPropertyPage
{
    DECLARE_DYNCREATE(CSystemStatusPage)
// Construction
public:
    CSystemStatusPage();
    ~CSystemStatusPage();
// Dialog Data
    enum { IDD = IDD_SYS_STATS };  
    CListBox m_SysStatsList;
//}}AFX_DATA
// Overrides
// ClassWizard generate virtual function overrides
//{{AFX_VIRTUAL(CSystemStatusPage)
protected:
    virtual void DoDataExchange(CDataExchange* pDX);
//}}AFX_VIRTUAL
// Implementation
protected:
    // Generated message map functions
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately
// before the previous line.
#endif // !defined(AFX_SYSTEMSTATUSPAGE_H__3491DFF3_EE4F_11D5_90BE_24CF09C10000_INCLUDED_)
SystemStatusPage.cpp
// SystemStatusPage.cpp : implementation file
//
#include "stdafx.h"
#include "RapiDemo.h"
#include <rapi.h"
#include "SystemStatusPage.h"
#ifndef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////////////////////////////////////////
/////
// CSystemStatusPage property page
IMPLEMENT_DYNCREATE(CSystemStatusPage, CPropertyPage)
CSystemStatusPage::CSystemStatusPage() :
CPropertyPage(CSystemStatusPage::IDD)
{
	//{{AFX_DATA_INIT(CSystemStatusPage)
	// NOTE: the ClassWizard will add member initialization here
	//}}AFX_DATA_INIT
}
CSystemStatusPage::~CSystemStatusPage()
{
}
void CSystemStatusPage::DoDataExchange(CDataExchange* pDX)
{
CPropertyPage::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CSystemStatusPage)
	DDX_Control(pDX, IDC_SYS_STATS_LIST, m_SysStatsList);
	//}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CSystemStatusPage, CPropertyPage)
	//{{AFX_MSG_MAP(CSystemStatusPage)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()
BOOL CSystemStatusPage::OnInitDialog()
{
CPropertyPage::OnInitDialog();
SYSTEM_INFO SystemInfo;
STORE_INFORMATION StoreInfo;
SYSTEM_POWER_STATUS_EX PowerStats;
MEMORYSTATUS MemStats;
CEOSVERSIONINFO VersionInfo;
char szInfoBuff[64];

HRESULT hr = CeRapiInit();
if ( hr != ERROR_SUCCESS )
	{ return FALSE; }

CeGetVersionEx((LPCEOSVERSIONINFO) &VersionInfo);

Listing 11.1  Partial source file listing for RapiDemo example. (continues)
```c
sprintf( szInfoBuff, "%s", "CE Version Stats:" );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, " %s : %li", "Major Version",
    VersionInfo.dwMajorVersion );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, " %s : %li",
    "Minor Version",VersionInfo.dwMinorVersion );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, " %s : %li", "Build Number",
    VersionInfo.dwBuildNumber);  
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, " %s : %li", "Platform Id",
    VersionInfo.dwPlatformId);
m_SysStatsList.InsertString( -1, szInfoBuff );
m_SysStatsList.InsertString( -1, " " );
CeGlobalMemoryStatus((LPMEMORYSTATUS) &MemStats);

sprintf( szInfoBuff, "%s", "CE Memory Stats:" );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, " %s : %li", "PerCent Usage",
    MemStats.dwMemoryLoad);
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, " %s : %li", "Total Physical Memory",
    MemStats.dwTotalPhys);
m_SysStatsList.InsertString( -1, szInfoBuff );

CeGetSystemPowerStatusEx((PSYSTEM_POWER_STATUS_EX) &PowerStats,
    TRUE);

sprintf( szInfoBuff, "%s", "CE Power Stats:" );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, " %s : %x", "AC Line Status",
    PowerStats.ACLineStatus );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, " %s : %x", "Battery Flag",
    PowerStats.BatteryFlag);
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, " %s : %x", "Battery Life Percent",
    PowerStats.BatteryLifePercent);
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, " %s : %li", "Battery Full LifeTime",
    PowerStats.BatteryFullLifeTime);
```

**Listing 11.1** Partial source file listing for RapiDemo example.
m_SysStatsList.InsertString( -1, szInfoBuff );
m_SysStatsList.InsertString( -1, "  ");
CeGetStoreInformation((LPSTORE_INFORMATION) &StoreInfo);
sprintf( szInfoBuff, "%s", "CE Store Information:" );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "    %s : %li", "Store Size", StoreInfo.dwStoreSize );
m_SysStatsList.InsertString( -1, szInfoBuff );
CeGetSystemInfo((LPSYSTEM_INFO) &SystemInfo);
sprintf( szInfoBuff, "%s", "CE System Info:" );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "    %s : %li", "Free Size", StoreInfo.dwFreeSize );
m_SysStatsList.InsertString( -1, szInfoBuff );
switch(SystemInfo.dwProcessorType )
{
    case PROCESSOR_INTEL_386:
        sprintf( szProcessorType, "%s", " PROCESSOR_INTEL_386 ");
        break;
    case PROCESSOR_INTEL_486:
        sprintf( szProcessorType, "%s", " PROCESSOR_INTEL_486 ");
        break;
    case PROCESSOR_INTEL_PENTIUM:
        sprintf( szProcessorType, "%s", " PROCESSOR_INTEL_PENTIUM ");
        break;
    case PROCESSOR_MIPS_R4000:
        sprintf( szProcessorType, "%s", " PROCESSOR_MIPS_R4000 ");
        break;
    case PROCESSOR_ALPHA_21064:
        sprintf( szProcessorType, "%s", " PROCESSOR_ALPHA_21064 ");
        break;
    case PROCESSOR_PPC_601:
        sprintf( szProcessorType, "%s", " PROCESSOR_PPC_601 ");
        break;
    case PROCESSOR_PPC_603:
        sprintf( szProcessorType, "%s", " PROCESSOR_PPC_603 ");
        break;
    case PROCESSOR_PPC_604:
        sprintf( szProcessorType, "%s", " PROCESSOR_PPC_604 ");
        break;
    case PROCESSOR_PPC_620:
        sprintf( szProcessorType, "%s", " PROCESSOR_PPC_620 ");
        break;
    case PROCESSOR_HITACHI_SH3:
        sprintf( szProcessorType, "%s", " PROCESSOR_HITACHI_SH3 ");
}

Listing 11.1  Partial source file listing for RapiDemo example. (continues)
break;
case PROCESSOR_HITACHI_SH3E:
    sprintf( szProcessorType, "%s", "PROCESSOR_HITACHI_SH3E ");
    break;
case PROCESSOR_HITACHI_SH4:
    sprintf( szProcessorType, "%s", "PROCESSOR_HITACHI_SH4 ");
    break;
case PROCESSOR_MOTOROLA_821:
    sprintf( szProcessorType, "%s", "PROCESSOR_MOTOROLA_821 ");
    break;
case PROCESSOR_SHx_SH3:
    sprintf( szProcessorType, "%s", "PROCESSOR_SHx_SH3 ");
    break;
case PROCESSOR_SHx_SH4:
    sprintf( szProcessorType, "%s", "PROCESSOR_SHx_SH4 ");
    break;
case PROCESSOR_STRONGARM:
    sprintf( szProcessorType, "%s", "PROCESSOR_STRONGARM ");
    break;
case PROCESSOR_ARM720:
    sprintf( szProcessorType, "%s", "PROCESSOR_ARM720 ");
    break;
case PROCESSOR_ARM820:
    sprintf( szProcessorType, "%s", "PROCESSOR_ARM820 ");
    break;
case PROCESSOR_ARM920:
    sprintf( szProcessorType, "%s", "PROCESSOR_ARM920 ");
    break;
case PROCESSOR_ARM_7TDMI:
    sprintf( szProcessorType, "%s", "PROCESSOR_ARM_7TDMI ");
    break;
default:
    sprintf( szProcessorType, "%s", "Unknown ");
    break;
}//end switch

    sprintf( szInfoBuff, "%s : %s", "Processor Type", szProcessorType );

    m_SysStatsList.InsertString( -1, szInfoBuff );
    sprintf( szInfoBuff, "%s : %li", "Allocation Granularity", SystemInfo.dwAllocationGranularity);
    m_SysStatsList.InsertString( -1, szInfoBuff );
    m_SysStatsList.InsertString( -1, " ");
    CeRapiUninit();
    return TRUE;  // return TRUE unless you set the focus to a control
    // EXCEPTION: OCX Property Pages should return FALSE
}

Listing 11.1  Partial source file listing for RapiDemo example.
Listing 11.1 Partial source file listing for RapiDemo example. (continues)
#include "WalkReg.h"
#include <rapi.h>
#ifdef _DEBUG
#define new DEBUG_NEW
#define THIS_FILE
#endif
static char THIS_FILE[] = __FILE__;
#define dim(x) (sizeof(x) / sizeof(x[0]))

/////////////////////////////////////////////////////////////////////
// CWalkReg property page
IMPLEMENT_DYNCREATE(CWalkReg, CPropertyPage)
CWalkReg::CWalkReg() : CPropertyPage(CWalkReg::IDD)
{
//{{AFX_DATA_INIT(CWalkReg)
//}}AFX_DATA_INIT
}
CWalkReg::~CWalkReg()
{
    m_rghItem = NULL;
    // closing the root key closes all the subkeys.
    CeRegCloseKey (HKEY_CURRENT_USER);
}
void CWalkReg::DoDataExchange(CDataExchange* pDX)
{
    CPropertyPage::DoDataExchange(pDX);
//}}AFX_DATA_MAP
BEGIN_MESSAGE_MAP(CWalkReg, CPropertyPage)
//{{AFX_MSG_MAP(CWalkReg)
ON_NOTIFY(TVN_ITEMEXPANDING, IDC_REG_TREE, OnItemexpandingRegTree)
ON_NOTIFY(TVN_SELCHANGED, IDC_REG_TREE, OnSelchangedRegTree)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CWalkReg, CPropertyPage)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

Listing 11.1
Partial source file listing for RapiDemo example.
Listing 11.1  Partial source file listing for RapiDemo example. (continues)
Listing 11.1  Partial source file listing for RapiDemo example.

```c
sprintf(&szMBCSData[0], "%x", *(DWORD *)pbValueData);
break;

case REG_BINARY:
    szMBCSData[0] = '\0';
    iLoopLimit = ( (int)dwValueDataSize < sizeof(szMBCSData) - 1) ?
        dwValueDataSize : sizeof(szMBCSData) - 1;
    for (i = 0; i < iLoopLimit; i++)
    {
        sprintf(&szMBCSData[i], "%02X", pbValueData[i]);
    }
    break;

default:
    sprintf(&szMBCSData[0], "Unknown type: %x", dwRegDataType);
}

m_RegList.InsertString(-1, szMBCSData);
    m_RegList.InsertString(-1, " ");
return;

HTREEITEM CWalkReg::InsertTreeItem(HTREEITEM hParent, TCHAR *pszName,
    LPARAM lParam, DWORD nChildren, HKEY hThisKey)
{
    TV_INSERTSTRUCT tvis;
    // Initialize the insertstruct.
    memset(&tvis, 0, sizeof(tvis));
    tvis.hParent = hParent;
    tvis.hInsertAfter = TVI_LAST;
    tvis.item.mask = TVIF_HANDLE | TVIF_TEXT | TVIF_PARAM |
        TVIF_CHILDREN | TVIF_IMAGE | TVIF_SELECTEDIMAGE;
    tvis.item.pszText = pszName;
    tvis.item.cchTextMax = lstrlen(pszName);
    tvis.item.iImage = 0;
    tvis.item.iSelectedImage = 1;
    tvis.item.lParam = (LPARAM)hThisKey;
    if (nChildren)
        tvis.item.cChildren = 1;
    else
        tvis.item.cChildren = 0;

    HTREEITEM htiSuccess = m_RegTree.InsertItem(&tvis);
    return htiSuccess;
}
BOOL CWalkReg::WalkRegTree(HKEY hKey, HTREEITEM htiParent)
{
    DWORD dwNSize;
    DWORD dwCSize;
    WCHAR wszName[MAX_PATH];
    CHAR szMBCSName[MAX_PATH];
    WCHAR wszClass[256];
```

Chapter 11
FILETIME ft;
TVITEM tviTreeViewItem;
DWORD dwNumChildren = 0;
DWORD dwNumGrandChildren = 0;
LONG rc;
HTREEITEM htiChild;
HKEY hSubKey;

// query for count of children
rc = CeRegQueryInfoKey (hKey, NULL, NULL, 0,
    &dwNumChildren, NULL, NULL, NULL,
    NULL, NULL, NULL, NULL);

// Iterate child keys
for( int iChildKeyEnumIndex = 0;
    iChildKeyEnumIndex < dwNumChildren; iChildKeyEnumIndex++ )
{
    // get child key name
dwNSize = dim(wszName);
dwCSize = dim(wszClass);
rc = CeRegEnumKeyEx (hKey, iChildKeyEnumIndex,
    wszName, &dwNSize, NULL,
    wszClass, &dwCSize, &ft);
if(rc != ERROR_SUCCESS)
    { return FALSE; }
    // open child key by name, capture HKEY
rc = CeRegOpenKeyEx(hKey, wszName, 0,
    KEY_ALL_ACCESS, &hSubKey );
if( rc != ERROR_SUCCESS )
    { return FALSE; }    
// query for count of grand children
rc = CeRegQueryInfoKey (hKey, NULL, NULL, 0,
    &dwNumGrandChildren, NULL, NULL, NULL,
    NULL, NULL, NULL, NULL);

// convert the name string from WCS to MBCS
wcstombs ( szMBCSName, wszName, sizeof(szMBCSName) );

// Add key to tree view.
htiChild = InsertTreeItem (htiParent, szMBCSName, 0,
    dwNumGrandChildren, hSubKey);
if(dwNumChildren == 0)
{
    tviTreeViewItem.hItem = htiChild;
    tviTreeViewItem.mask = TVIF_CHILDREN | TVIF_HANDLE;
    tviTreeViewItem.cChildren = 0;
    m_RegTree.SetItem( &tviTreeViewItem );
}

// if !children, return TRUE
if( dwNumChildren != 0 )
{
Listing 11.1  Partial source file listing for RapiDemo example. (continues)
// recurse walk tree
WalkRegTree( hSubKey, htiChild );
    m_RegTree.Expand(htiChild, TVE_COLLAPSE );
} // end if(dwNumChildren)
} // end for(iterate children )
return TRUE;

void CWalkReg::OnItemexpandingRegTree(NMHDR* pNMHDR, LRESULT* pResult)
{
    NM_TREEVIEW* pNMTreeView = (NM_TREEVIEW*)pNMHDR;
    // get key handle from tree item
    TVITEM tvitem = (TVITEM)pNMTreeView->itemNew;
    if(tvitem.iImage == 0 )
    {
        m_RegTree.SetItemImage( tvitem.hItem, 1, 0 );
    }
    else
    {
        m_RegTree.SetItemImage( tvitem.hItem, 0, 0 );
    }
    *pResult = 0;
}

void CWalkReg::OnSelchangedRegTree(NMHDR* pNMHDR, LRESULT* pResult)
{
    NM_TREEVIEW* pNMTreeView = (NM_TREEVIEW*)pNMHDR;
    // clear list control
    m_RegList.ResetContent();
    // get the key from the item
    TVITEM tviTreeItem = pNMTreeView->itemNew;
    HKEY  hRegKey = (HKEY)tviTreeItem.lParam;

    // does this key have values?
    INT iValueIndex = 0;
    WCHAR wszValueName[MAX_PATH];
    BYTE  bValueData[1024];
    DWORD dwValueNameSize = dim(wszValueName);
    DWORD dwValueDataSize = dim(bValueData);
    DWORD dwRegDataType;
    memset( bValueData, 0x0, sizeof(bValueData));
    memset( wszValueName, 0x0, sizeof(wszValueName));
    LONG rc = CeRegEnumValue (hRegKey, iValueIndex,
        (LPWSTR)&wszValueName, &dwValueNameSize,
        NULL, &dwRegDataType,
        (PBYTE)&bValueData,
        &dwValueDataSize);
    return;

Listing 11.1 Partial source file listing for RapiDemo example.
while (rc == ERROR_SUCCESS)
if( rc != ERROR_SUCCESS )
{
    m_RegList.InsertString( -1, "No values exist for this key");
    *pResult = 0;
    
    
    InterpKeyVal( wszValueName, (PBYTE)&bValueData,
    dwValueDataSize, dwRegDataType);
    iValueIndex++;
    memset( bValueData, 0x0, sizeof(bValueData));
    dwValueNameSize = dim(wszValueName);
    dwValueDataSize = dim(bValueData);
    rc = CeRegEnumValue (hRegKey, iValueIndex,
    (LPWSTR)&wszValueName, &dwValueNameSize,
    NULL, &dwRegDataType, (PBYTE)&bValueData,
    &dwValueDataSize);
    
}

*pResult = 0;

Listing 11.1  Partial source file listing for RapiDemo example. (continued)

An Overview of the RapiDemo Example

Because so many files and objects comprise an MFC application, I like to start with an overall strategic view of a project. In the RapiDemo example, here’s a list of the objects that are being constructed, the chronology of their construction, and the relationship between the objects:

- The application object, RapiDemoApp, is constructed first. It, in turn, creates the document, view, and mainframe window objects and establishes a relationship among them. The Visual C++ development tools generate this part of the application automatically.

- In addition, the RapiDemoApp object performs some application-specific initialization. This happens in the function CRapiDemoApp::InitInstance(). We make modifications to this function to create and launch a property sheet control.
Our application implements a property sheet control in the class CAllPages, which is derived from the standard MFC class CPropertySheet.

An application-specific class, derived from the MFC base class CPropertyPage, implements each page in the property sheet control.

CAllPages includes a member variable for each individual page of the property sheet. The type of the member variable is the class that implements the page it represents. There are three pages in the control and three member variables that represent them in CAllPages.

The three RapiDemo property sheet page classes implement pagespecific initialization and input handling behavior.

When the property sheet control is launched, the visible page receives user input. The class that implements the page handles the input.

When the user changes pages by clicking on the property sheet tabs, the property sheet control transparently handles generic page-changing behavior. The application has to do application-specific initialization for the newly visible page.

In this example, each page initializes the RAPI subsystem, makes RAPI calls, updates dialog controls on the page, and then uninitializes RAPI.

Figure 11.1 shows how the RapiDemo example looks when the application is launched. The default page, System Status, has been initialized with values queried from the CE device.
Creating the Dialog Resources for the Property Sheet Pages

The RapiDemo application relies on the MFC implementation of the Property Sheet Control. This type of control is really nothing more than a stack of dialog boxes, surmounted by a control that looks like the tabs in an index file, recipe box, or your junior high school binder. The function of the property sheet control is to arbitrate page changing behavior. When a user clicks on a tab, the property sheet control brings the corresponding dialog to the top of the z-order.

This means that each page of the property sheet must have a dialog resource from which the sheet can build a view. You don’t have to worry about making the dialog templates exactly the same size, because the property sheet control is assembled based on the size of the largest constituent page. However, the styles of the dialog resources are important. Set these styles for property page dialog resources, using the File.Properties dialog.

On the Styles tab, set Style to Child, Border to Thin, check the Title Bar checkbox, and clear the other checkboxes.

On the More Styles tab, check Disabled.

Now let’s dissect the code for the application-specific behaviors of the classes that make up RapiDemo.

The Application Object, RapiDemoApp

The code for the class that implements the application object, CRapiDemoApp, is generated for us, for the most part. We make a few small modifications to the file RapiDemoApp.cpp. First, we add the header file for the class that implements the property sheet. (If you forget to do this, you’ll get compiler error messages naming the CAllPages class members as undefined symbols.)

```
#include "AllPages.h"
```

We also modify the InitInstance() member, where the property sheet object is constructed, initialized, and launched as a modal dialog:

```
////////////////////////////////////////////////////////////////////////
//////
// The one and only CRapiDemoApp object
CRapiDemoApp theApp;
////////////////////////////////////////////////////////////////////////
//////
// CRapiDemoApp initialization
BOOL CRapiDemoApp::InitInstance()
{
```
We declare and initialize an object of the class CAllPages. This class implements the property sheet control, and is derived from CPropertySheet. CPropertySheet has three constructors. The one we use to construct the CAllPages object takes the following arguments: a character string specifying the caption for the property sheet control, the handle to the control’s parent window, and the index of the property page that is initially visible. A property page’s index is determined by the order in which it was added to the control. Passing a NULL parent window handle sets the application’s main window as parent of the property sheet control.

Next, we call the AllPages member function AddPropPages(), which adds the individual pages to the property sheet control. Calling the DoModal() member launches the property sheet control, making it visible when the application opens.

### Constructing and Initializing the Property Sheet Object, AllPages

The CAllPages class contains the code that implements the property sheet control. First, we make modifications to the class header file, AllPages.h. Next, we add some include files:

```cpp
#include "WalkReg.h"    // "Walk Registry Tree" page's
                        // class header
#include "SystemStatusPage.h"   // "System Status" page's
                        // class header
#include "RemoteFileAccessPage.h"  // "Remote File Access" page's
                        // class header
#include <rapi.h>
```

The first three #include files are for the classes that implement the behavior of the individual pages in the property sheet control. The last one is the RAPI header and is needed in any source file that calls RAPI functions. (If you forget to add this include file, RAPI function calls and structures will come up as undeclared symbols when you build.)

```cpp
// Implementation
public:
```
We add three member variables, one for each property page of the property sheet control. These members are typed according to the class that implements the behavior for its specific page. We also add a function prototype for the member AddPropPages().

Next, we make some additions to the file AllPages.cpp, adding the member function AddPropPages():

```cpp
void CAllPages::AddPropPages()
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_psh.dwFlags |= PSP_USEHICON;
    m_psh.hIcon = m_hIcon;
    m_psh.dwFlags |= PSH_NOAPPLYNOW;  // Lose the Apply Now button
    m_psh.dwFlags &= ~PSH_HASHELP;    // Lose the Help button
    AddPage(&m_RemoteFileAccessPage);
    AddPage(&m_WalkRegPage);
    AddPage(&m_SystemStatusPage);
}
```

The first part of this function manipulates the PROPSHEETHEADER structure, a base class data member that defines the appearance and behavior of the property sheet control. It’s a fairly large structure that allows great flexibility in the creation of the property sheet control. The base class data member m_psh is a pointer to this structure.

The next three lines add property page objects to the property sheet control. The pages’ indices are the reverse of the order in which they were added. Put another way, in the preceding example, the leftmost tab of this property sheet control will be the “System Status” page, the “Walk The Registry Tree” tab will be in the middle, and the rightmost tab will be for the “Remote File Access” page.

### Initializing the System Status Property Page

In this case, we don’t have to make any changes to the header file for the CSystemStatusPage class. All of our modifications are to the implementation file SystemStatusPage.cpp. by including the RAPI header.

```cpp
#include <rapi.h>
```
The behavior of the System Status page is entirely implemented in the initialization member function, `CSystemStatusPage::OnInitDialog()`:

```cpp
BOOL CSystemStatusPage::OnInitDialog()
{
    CPropertyPage::OnInitDialog();
    SYSTEM_INFO SystemInfo;
    STORE_INFORMATION StoreInfo;
    SYSTEM_POWER_STATUS_EX PowerStats;
    MEMORYSTATUS MemStats;
    CEOSVERSIONINFO VersionInfo;
    char szInfoBuff[64];
}
```

The data declaration in `OnInitDialog()` consist almost entirely of structures that will be familiar from earlier chapters. The following code illustrates the use of their members, but not exhaustively. The key thing to keep in mind about these structures is that all of their string members will be reported to your program in Unicode format. In order to display them in the list control, all strings must first be translated back into multibyte character set format:

```cpp
//Initialize RAPI before use
HRESULT hr = CeRapiInit();
if ( hr != ERROR_SUCCESS )
{
    return FALSE;
}
```

Before we can use the RAPI subsystem, we must make it available to our application by calling the function `CeRapiInit()`. Always test the return from this call before continuing with RAPI operations. When you are finished making RAPI calls, you must uninitialize the subsystem and free its resources by calling the companion function `CERapiUninit()`:

```cpp
//Uninitialize RAPI when you are finished making RAPI calls
CERapiUninit();
```

You may wonder why we didn’t just initialize RAPI in the constructor of the application object `RapiDemoApp` and uninitialize it in the destructor. In theory, this seems the sensible thing to do. However, in my experience, the RAPI connection tends to be more stable if initialization and uninitialization closely bracket your RAPI operations.

The first RAPI call we make is `CeGetVersionEx()`. RAPI function names and argument lists are generally the same as the CE side counterpart, except the function name is prefixed by `Ce`. Notice we use `sprintf()` to format strings to insert in the listbox `m_SysStatsList`. The members of the `CEOSVERSIONINFO` structure being displayed are all of type `DWORD`. Being numeric, they need no translation and can be passed as arguments to single byte string handling routines:
CeGetVersionEx((LPCEOSVERSIONINFO) &VersionInfo);
sprintf( szInfoBuff, "%s", "CE Version Stats:" );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "%s : %li",
    "Major Version", VersionInfo.dwMajorVersion );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "%s : %li",
    "Minor Version", VersionInfo.dwMinorVersion );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "%s : %li",
    "Build Number", VersionInfo.dwBuildNumber );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "%s : %li",
    "Platform Id", VersionInfo.dwPlatformId );
m_SysStatsList.InsertString( -1, szInfoBuff );
m_SysStatsList.InsertString( -1, " ");

Next we use CeGlobalMemoryStatus() to retrieve memory statistics. Once again, we can use the MEMORYSTATUS structure’s numeric members without translation:

CeGlobalMemoryStatus((LPMEMORYSTATUS) &MemStats);

sprintf( szInfoBuff, "%s", "CE Memory Stats:" );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "%s : %li",
    "PerCent Usage", MemStats.dwMemoryLoad );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "%s : %li",
    "Total Physical Memory", MemStats.dwTotalPhys );
m_SysStatsList.InsertString( -1, szInfoBuff );

The next call, CeGetSystemPowerStatusEx(), queries the power status of the CE device. The members of the returned structure specify whether the system is running on AC (wall outlet) or DC (battery) power, whether or not the batteries are currently charging, and the remaining life of the main and backup batteries. If the system being queried doesn’t support all these status requests, it returns -1 in unsupported members. Its arguments, in the order shown, are the address of a SYSTEM_POWER_STATUS_EX structure, and a Boolean flag that instructs the function to acquire the most current information directly from the driver as opposed to using older, cached information.

Notice the use of the sprintf() format string “%x” for ACLineStatus, BatteryFlag, BatteryLifePercent, BackupBatteryFlag, and BackupBattery LifePercent structure members. These are all single byte values. We haven’t used the CE side analog of this function in previous chapters, and so include here the typedef of the SYSTEM_POWER_STATUS_EX structure:
typedef struct _SYSTEM_POWER_STATUS_EX {
    BYTE ACLineStatus;
    BYTE BatteryFlag;
    BYTE BatteryLifePercent;
    BYTE Reserved1;
    DWORD BatteryLifeTime;
    DWORD BatteryFullLifeTime;
    BYTE Reserved2;
    BYTE BackupBatteryFlag;
    BYTE BackupBatteryLifePercent;
    BYTE Reserved3;
    DWORD BackupBatteryLifeTime;
    DWORD BackupBatteryFullLifeTime;
} SYSTEM_POWER_STATUS_EX;

Here’s where we query power status and publish the results:

CeGetSystemPowerStatusEx( (PSYSTEM_POWER_STATUS_EX) &PowerStats,
    TRUE);
sprintf( szInfoBuff, "%s", "CE Power Stats:");
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "    %s : %x", "AC Line Status",
    PowerStats.ACLineStatus );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "    %s : %x", "Battery Flag",
    PowerStats.BatteryFlag); 
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "    %s : %x", "Battery Life Percent",
    PowerStats.BatteryLifePercent);
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "    %s : %li", "Battery Full LifeTime",
    PowerStats.BatteryFullLifeTime);
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "    %s : %x", "Backup Battery Flag",
    PowerStats.BackupBatteryFlag);
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "    %s : %x", "Backup Battery Life Percent",
    PowerStats.BackupBatteryLifePercent);
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "    %s : %li", "Backup Battery Full LifeTime",
    PowerStats.BackupBatteryFullLifeTime);
m_SysStatsList.InsertString( -1, "  ");

Next we retrieve storage status information with a call to CeGetStoreInformation(). The returned structure gives the size of the CE device’s object store and the unused space in the object store.
CeGetStoreInformation((LPSTORE_INFORMATION) &StoreInfo);
sprintf( szInfoBuff, "%s", "CE Store Information:" );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "%s : %li", "Store Size", StoreInfo.dwStoreSize );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "%s : %li", "Free Size", StoreInfo.dwFreeSize );
m_SysStatsList.InsertString( -1, szInfoBuff );

Next we get detailed information on the CE device’s processor, with a call to CeGetSystemInfo(). (Several of the processor constants are omitted from the switch() code below, but they are shown in their entirety in the SystemStatusPage.cpp file listing.)

CeGetSystemInfo((LPSYSTEM_INFO) &SystemInfo);
sprintf( szInfoBuff, "%s", "CE System Info:" );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "%s : %li", "Page Size", SystemInfo.dwPageSize );
m_SysStatsList.InsertString( -1, szInfoBuff );

char szProcessorType[48];
switch(SystemInfo.dwProcessorType )
{
    case PROCESSOR_INTEL_386:
        sprintf( szProcessorType, "%s", "PROCESSOR_INTEL_386 ");
        break;
    .
    .
    .
    case PROCESSOR_HITACHI_SH4:
        default:
            sprintf( szProcessorType, "%s", "Unknown ");
            break;
} //end switch

sprintf( szInfoBuff, "%s : %s", "Processor Type", szProcessorType );
m_SysStatsList.InsertString( -1, szInfoBuff );
sprintf( szInfoBuff, "%s : %li", "Allocation Granularity", SystemInfo.dwAllocationGranularity);
m_SysStatsList.InsertString( -1, szInfoBuff );
m_SysStatsList.InsertString( -1, " ");

Finally, we uninitialize RAPI, and make a clean exit from the SystemStatusPage initialization member:

CeRapiUninit();
return TRUE;
Initializing the Remote File Access Page

The Remote File Access page is slightly more complicated than the SystemStatusPage owing to the fact that it has more controls and is somewhat more interactive (see Figure 11.2).

Let’s begin by looking at the modifications made to the header file for the class, RemoteFileAccessPage.h. Using the resource editor and ClassWizard, a member variable was created for each of the dialog’s controls. The variables are of the same type as the controls they encapsulate:

```cpp
// Dialog Data
//}}AFX_DATA
enum { IDD = IDD_FILE_ACCESS_DIALOG };
CButton    m_GetCEOID;
CButton    m_GetLastWriteTime;
CButton    m_GetFoldersOnly;
CButton    m_GetCreationTime;
CButton    m_UpdateFilesList;
CListBox    m_SearchPath;
CListBox    m_FilesList;
CEdit    m_FileInput;
//}}AFX_DATA
```
We have a bit more to do to implement the behavior of the Remote File Access page than the System Status page. First, we have to initialize the control variables, and after that we have to respond with the appropriate information when the user presses the Update File List button. The initialization step is handled in OnInitDialog().

We insert two search path strings in the list control m_SearchPath. Notice the double backslash characters in the path strings. In C and C++, a single backslash character is interpreted as the first character of an “escape sequence.” An example of an escape sequence is the tab character, represented as \t. By convention, \ is interpreted as a single backslash character.

We initialize the list control selection to the first visible string (0th ordinal), and we clear all of the checkbox controls. The checkbox controls correspond to items of file information that we can optionally retrieve:

```cpp
BOOL CRemoteFileAccessPage::OnInitDialog()
{
    CPropertyPage::OnInitDialog();
    //Insert Search List Items
    m_SearchPath.InsertString( -1, "\*.*" );
    m_SearchPath.InsertString( -1, "\Windows\*.*" );
    //Set search list selection
    m_SearchPath.SetCurSel( 0 );
    //Clear File Attribute Flag members
    m_GetLastWriteTime.SetCheck(0);
    m_GetCreationTime.SetCheck(0);
    m_GetFoldersOnly.SetCheck(0);
    m_GetCEOID.SetCheck(0);
    return TRUE; // return TRUE unless you set the
    // focus to a control
    // EXCEPTION: OCX Property Pages should return FALSE
}
```

The actual work of the Remote File Access page isn’t done until the user presses the Update File List button. In the OnUpdateFileButton() member, first we clear the list control with a call to ResetContent(), and then we initialize the RAPI subsystem with CeRapiInit():

```cpp
////////////////////////////////////////////////////////////////////
// CRemoteFileAccessPage message handlers
void CRemoteFileAccessPage::OnUpdateFileButton()
{
    WCHAR wszFileName[MAX_PATH];
    //clear current list
    m_FilesList.ResetContent();
    HRESULT hr = CeRapiInit();
    if ( hr != ERROR_SUCCESS )
    {
        return;
    }
}
```
Next, we create a Unicode search path string, based on the selection in the list control, \texttt{m\_SearchPath}. Notice that we use the \texttt{L} macro to prefix the literal string that we want to render in Unicode. In earlier chapters’ examples, we used the TEXT macro for this purpose. The TEXT macro doesn’t work in MFC applications unless they are specifically enabled with Unicode preprocessor definitions.

**PROGRAMMING TIP** Use the \texttt{L} macro to create Unicode literal strings in MFC applications: \texttt{L"This is a Unicode literal string"}

```c
LPCE\_FIND\_DATA pFindDataArray = NULL;
DWORD dwFoundCount = 0;
int iSel = m\_SearchPath\_GetCurSel();
if( iSel == 0 )
{
    wcscpy(&wszFileName[0], L"\*.*" );
}
else
{
    wcscpy(&wszFileName[0], L"\Windows\*.*" );
}
```

Our tool for retrieving information about files on the CE device is the powerhouse function, CeFindAllFiles(). CeFindAllFiles() has no analog on the CE side. It is specifically designed for RAPI applications, and retrieves all the files and folders under a particular node in a single call. This is advantageous, because it is much more efficient than enumerating files and folders individually. The arguments to the functions are: a pointer to a Unicode string specifying a DOS style search path, a DWORD containing flags that determine what file attributes to return, a count of the returned items, and the address of a pointer of type \texttt{LPCE\_FIND\_DATA}. On successful return, the last parameter contains the address of an array of \texttt{CE\_FIND\_DATA} structures. Here is the typedef for \texttt{CE\_FIND\_DATA}:

```c
typedef struct \_CE\_FIND\_DATA {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
    DWORD nFileSizeHigh;   //0, unless the overall file size
                            //is > MAXDWORD
    DWORD nFileSizeLow;    //file size in bytes
    DWORD dwOID;           //CE Object Identifier for this file
    WCHAR cFileName[MAX\_PATH];  //Unicode file name
} CE\_FIND\_DATA;
```
Based on the state of the checkbox controls, we add flags to \( \text{dwFileAttributes} \). At a minimum, we’ll retrieve the filename, and optionally the creation time, last write time, and CE Object ID for the file. If the GetFoldersOnly box is checked, we retrieve information for all the folders in the search path and skip over the files. Table 11.2 is a complete list of the CE\_FIND\_DATA () file attribute flags, along with their meanings.

**Table 11.2 CE\_FIND\_DATA File Attribute Flags and Their Meanings**

<table>
<thead>
<tr>
<th>FILE ATTRIBUTE FLAG</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAF_ATTRIB_CHILDREN</td>
<td>Only retrieve information for directories with child items</td>
</tr>
<tr>
<td>FAF_ATTRIB_NO_HIDDEN</td>
<td>Omit files or directories that have the hidden attribute set</td>
</tr>
<tr>
<td>FAF_FOLDERS_ONLY</td>
<td>Retrieve information for directories only</td>
</tr>
<tr>
<td>FAF_NO_HIDDEN_SYS_ROMMODULES</td>
<td>Omit ROM files or directories</td>
</tr>
<tr>
<td>FAF_ATTRIBUTES</td>
<td>Return file attributes in ( \text{dwFileAttributes} )</td>
</tr>
<tr>
<td>FAF_CREATION_TIME</td>
<td>Return file creation time in ( \text{ftCreationTime} )</td>
</tr>
<tr>
<td>FAF_LASTACCESS_TIME</td>
<td>Return last access time in ( \text{ftLastAccessTime} )</td>
</tr>
<tr>
<td>FAF_LASTWRITE_TIME</td>
<td>Return last write time in ( \text{ftLastWriteTime} )</td>
</tr>
<tr>
<td>FAF_SIZE_HIGH</td>
<td>Return the high-order DWORD value of the file size in ( \text{nFileSizeHigh} )</td>
</tr>
<tr>
<td>FAF_SIZE_LOW</td>
<td>Return the low-order DWORD value of the file size in ( \text{nFileSizeLow} )</td>
</tr>
<tr>
<td>FAF_OID</td>
<td>Return the CE object identifier of the file in ( \text{dwOID} )</td>
</tr>
<tr>
<td>FAF_NAME</td>
<td>Return the Unicode file name in ( \text{cFileName} )</td>
</tr>
</tbody>
</table>

Now we retrieve file information from the Windows CE side of the connection:

```c
DWORD dwFileAttributeFlags = FAF_NAME;
if( m_GetCreationTime.GetCheck() )
{ dwFileAttributeFlags |= FAF_CREATION_TIME ; }
if( m_GetLastWriteTime.GetCheck() )
```
( dwFileAttributeFlags |= FAF_LASTWRITE_TIME ; )
if( m_GetFoldersOnly.GetCheck() )
( dwFileAttributeFlags |= FAF_FOLDERS_ONLY; )
if( m_GetCEOID.GetCheck() )
( dwFileAttributeFlags |= FAF_OID; )
BOOL bOk = CeFindAllFiles( (LPCWSTR)&wszFileName[0],
dwFileAttributeFlags, &dwFoundCount, &pFindDataArray );

if(!bOk )
{ return; }

On successful return from CeFindAllFiles, we loop through the array of
returned structures, transferring the retrieved information to the list control,
m_FilesList. Notice that we use the count of found items, dwFoundCount,
returned by CeFindAllFiles to set the loop limit. Because the returned file
name is a Unicode string, we must translate it to multibyte character format
using wcstombs() before attempting to insert it into the list control.

char szLineBuff[MAX_PATH];

for( int i = 0; i < dwFoundCount; i++ )
{
   wcstombs( szLineBuff, pFindDataArray[i].cFileName,
              sizeof(szLineBuff));
   m_FilesList.InsertString( -1, szLineBuff );
   if( m_GetCreationTime.GetCheck() )
   {
      CTime FileCreationTime(pFindDataArray[i].ftCreationTime,
                              -1 );
      CString csBuff = FileCreationTime.Format("%A, %B %d, %Y");
      sprintf( szLineBuff, " %s: %s", "Creation Time",
                     csBuff.GetBuffer(csBuff.GetLength()) );
      m_FilesList.InsertString( -1, szLineBuff );
   }
   if( m_GetLastWriteTime.GetCheck() )
   {
      CTime LastWriteTime(pFindDataArray[i].ftCreationTime,
                           -1 );
      CString csBuff = LastWriteTime.Format("%A, %B %d, %Y");
      sprintf( szLineBuff, " %s: %s", "Last Write",
                     csBuff.GetBuffer(csBuff.GetLength()) );
      m_FilesList.InsertString( -1, szLineBuff );
   }
   if( m_GetCEOID.GetCheck() )
   {
      sprintf( szLineBuff, " %s: %s%x", "CEOID",
                     "0x", pFindDataArray[i].dwOID );
      m_FilesList.InsertString( -1, szLineBuff );
   }
}
//      m_FilesList.InsertString( -1, " ");
};//end for()

Notice that our call to CeFindAllFiles() returned a pointer to a dynamically allocated array of CE_FIND_DATA structures. We must free this buffer, which was allocated in our behalf before we uninitialize the RAPI subsystem, or memory will be leaked:

    //free the CE_FIND_DATA structure array buffer
    hr = CeRapiFreeBuffer((LPVOID) pFindDataArray);
    //now uninitialize RAPI.
    CeRapiUninit();

---

**Initializing and Handling the Walk Registry Tree Page**

The Walk Registry Tree page is by far the most involved of the problems we tackle in the RapiDemo example because it uses the highly sophisticated MFC control, CTreeCtrl. Tree controls aren’t easy to code, but they are very powerful tools for depicting hierarchies of variable structure and depth. File systems, databases, and the Windows registry are all examples of such hierarchies.

In the CWalkReg class, we take an in-depth look at how to use CTreeCtrl to display the Windows CE Registry.

**How the “Walk Registry Tree” Page Uses the MFC Tree Control**

The natural choice for displaying hierarchically organized data is the MFC Tree Control. To use a tree control, we have to do three things:

- Discern hierarchical relationships between tree items.
- Add items to the tree control, by initializing the TVITEM structure with information that defines the appearance of the item and its relationship to neighboring tree items.
- Handle tree control notification messages.

First we’ll see how to add a single item to the tree control, and then we’ll see how to handle tree control notifications. Finally, we’ll tie the concepts together by walking out the registry tree, adding tree items hierarchically. The Walk Registry Tree page is implemented in the class CWalkReg, which creates and initializes the tree control when the page is created, and which handles notification messages from the tree control.
Adding a Single Tree Item to the Tree Control

Assume for a moment that we have a tree control, and an item we want to add to it. We need to know a few specific things about an item in order to add it to the tree:

- What item is the new item’s parent?
- Does the new item have children?
- Will the icon next to the new item change if the user selects it from the tree control?

We pass this and other information to the tree control in the TV_INSERTSTRUCT structure, and another structure, TVITEM, which is a member of TV_INSERTSTRUCT. The content of these structures define the rank, behavior, and appearance of the tree item we insert. Here’s the typedef for TV_INSERTSTRUCT:

```c
typedef struct tagTVINSERTSTRUCT {
    HTREEITEM hParent;
    HTREEITEM hInsertAfter;
    TVITEM item;
} TV_INSERTSTRUCT;
```

The first member, hParent, is the handle to the parent of the item being inserted. This handle, hParent, was returned when the parent was inserted. Passing a NULL in hParent establishes an item as the root of the hierarchy. The next member, hInsertAfter, is actually a flag, which is assigned one of the predefined values listed in Table 11.3.

The item member (which is a structure of type TVITEM) of the TV_INSERTSTRUCT structure contains the item’s configuration information. TVITEM is used both to set item information when adding items to the tree control and to retrieve information about a particular item. Here’s the typedef of TVITEM:

```c
typedef struct tagTVITEM{
    UINT mask;   //tells which members are valid or
    // should be returned
    HTREEITEM hItem;   // handle to item for which we are
    //retrieving data, unused if we are
    // adding the item
    UINT state;      //Item State Flags
    UINT stateMask;  //Mask that identifies which bits of the
    // state mask are valid
    LPTSTR pszText;   //Item caption
    int cchTextMax;  // caption buffer size, in characters
```
int iImage; //tree control image list index of the
    //icon to use when the item is
    //unselected
int iSelectedImage; //tree control image list
    //index of the
    //icon to use when the item
    //is selected
int cChildren; //treated as Boolean flag- 1= has
    //children, 0= no children
LPARAM lParam; //application specific data stored
    //in the item
} TVITEM, FAR *LPTVITEM;

The key to using the TVITEM structure to set or retrieve information about
an item is in understanding the mask, state, and stateMask members:

- The mask member indicates which members of the TVITEM structure
  contain valid data if you are adding the item.
- The mask member indicates which members of the TVITEM structure
  should contain valid data on return if you are querying the item.
- Bits 0–7 of the state member contain flags that specify the tree view
  item’s state.
- Bits 8–11 of the state member give the one based index of the item’s
  overlay image.
- Bits 12–15 of the state member give the application-defined state
  image.
- The stateMask member defines which bits of the state member are valid.

Notice that the cChildren member is treated as a Boolean flag, even though
it is of type int. If the item has children, you should set this value equal to one,
not the count of the children, as its name and type seem to imply.

In the CWalkReg::InsertTreeItem() member, initializing the TV_INSERTSTRUCT
with a call to memset(). We set the hParent member to the value that was
passed as the first parameter to the function. (You’ll see later that we call this
function repetitively when we detect that an item has children, passing in the

<table>
<thead>
<tr>
<th>FLAG NAME</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>TVI_FIRST</td>
<td>Insert item at the head of the list</td>
</tr>
<tr>
<td>TVI_LAST</td>
<td>Insert item at the end of the list</td>
</tr>
<tr>
<td>TVI_SORT</td>
<td>Insert sorted alphabetically</td>
</tr>
</tbody>
</table>
handle to the parent of the item being added.) We set the hInsertAfter to TVI_LAST, which means this item will be added to the bottom of the list. The item.mask member signals that we are passing the handle of the parent item, a caption string, and application-defined lParam value, the children flag, and the zero-based indexes in the tree control’s image list for the item’s selected and nonselected state. Notice that we set tvis.item.lParam to the value hThisKey. hThisKey was passed to the InsertTreeItem() member as a parameter, and is the handle to an open registry key corresponding to the item we are adding to the tree view:

```
HTREEITEM CWalkReg::InsertTreeItem( HTREEITEM hParent, TCHAR *pszName, 
   LPARAM lParam, DWORD nChildren, HKEY hThisKey)
{
  TV_INSERTSTRUCT tvis;
  // Initialize the insertstruct.
  memset (&tvis, 0, sizeof (tvis));
  tvis.hParent = hParent;
  tvis.hInsertAfter = TVI_LAST;
  tvis.item.mask = TVIF_HANDLE | TVIF_TEXT | TVIF_PARAM |
                  TVIF_CHILDREN | TVIF_IMAGE | TVIF_SELECTEDIMAGE ;
  tvis.item.pszText = pszName;
  tvis.item.cchTextMax = lstrlen (pszName);
  tvis.item.iImage = 0;
  tvis.item.iSelectedImage = 1;
  tvis.item.lParam = (LPARAM)hThisKey;
  if (nChildren)
    tvis.item.cChildren = 1;
  else
    tvis.item.cChildren = 0;

  Once the structures are properly initialized, a call to m_RegTree.InsertItem() 
  adds the item and updates the tree control’s appearance. We return the handle 
  to the newly created tree item to the calling function:

  HTREEITEM htiSuccess = m_RegTree.InsertItem(&tvis );
  return htiSuccess;
}
```

### Responding to Tree Control Notification Messages

Once you’ve added all of the items, the tree control manages most of the visual 
work of collapsing and expanding the tree control. However, there are some 
parts of the tree control’s behavior in which we want to exert some control. 
First, we need to change the appearance of the icon next to the tree item when 
its selection state changes. We also must add a newly selected key’s values (or 
a message saying it has no values) to the listbox control.
To do this, we need to:

- Detect and handle tree control notification when the selected item changes.
- Detect and handle tree control notification when the tree item is expanding.
- Retrieve the handle to the opened registry key from the item’s lParam value.
- Enumerate the key values.
- Interpret and display the key values in the list control.

Here’s our plan of attack:

We created message handlers in CWalkReg to process the notification message from the tree control. Each of these functions was added to CWalkReg using the ClassWizard, which generated the following message map entry:

```cpp
protected:
    // Generated message map functions
    // AFX_MSG(CWalkReg)
    afx_msg void OnItemexpandingRegTree(NMHDR* pNMHDR,
                                        LRESULT* pResult);
    afx_msg void OnSelchangedRegTree(NMHDR* pNMHDR,
                                       LRESULT* pResult);
    //AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

We’ll get the handle to the registry keys for which we want to enumerate values from the lParam of the selected item’s TVITEM structure. (Recall that we stored the corresponding opened key’s handle in each item when we initialized the tree control.)

We’ll retrieve the keys’ values by looping through them with the RAPI function CeRegEnumValue().

Finally, we’ll display the values in the list control, using the CWalkReg member InterpretKeyValue(). This member detects the registry-defined data type of the value, writes it to a properly formatted string, and inserts it in the list control.

**Handling Item Expansion**

Let’s start with a look at our simplest tree control notification handler, OnItemexpandingRegTree(). You’ll notice that like ordinary Windows
messages handling functions, notification handlers have a standard form. Here are the first two lines of our TVN_ITEMEXPANDING notification handler:

```c
void CWalkReg::OnItemexpandingRegTree(NMHDR* pNMHDR,
    LRESULT* pResult)
{
    NM_TREEVIEW* pNMTreeView = (NM_TREEVIEW*)pNMHDR;
}
```

The parameters, in the order shown, are a pointer to a notification message header structure, and a pointer to a variable through which this function can return a result. Notifications extend the Windows messaging architecture by allowing controls to send more information than could be encapsulated in a windows message, and by providing a standard form of communication that is consistent across various control types. To better understand this, let’s look more closely at the structures used to communicate the notification.

The NMHDR typedef looks like this:

```c
typedef struct tagNMHDR {
    HWND hwndFrom;      // handle of control
    // sending
    // message
    UINT idFrom;     // identifier of control
    // sending message
    UINT code;          // control specific
    // notification code
} NMHDR;
```

Examining this structure definition, it is apparent that the NMHDR structure explicitly provides the handler function with both the window handle and the ID of the control sending the notification. What is subtler is that the code member may be used to deliver either a code or a pointer to additional data.

We chose to handle the notification TVN_ITEMEXPANDING because it is sent after the user has opened an item in the tree control. When this happens we need to toggle the icon image. If it was the selected icon, we’ll set the unselected icon, and vice versa.

To change the icon, we need a pointer to the tree view control object. We get this pointer by casting the NMHDR structure pointer to an NM_TREEVIEW* and assigning the pointer to pNMTreeView.

```c
void CTreeWalker::OnSelchanged(NMHDR* pNMHDR, LRESULT* pResult)
{
    NM_TREEVIEW* pNMTreeView = (NM_TREEVIEW*)pNMHDR;
}
```

Here’s where the notification message magic comes in. All notify messages have at least a populated NMHDR structure, but many control notifications are actually a larger, more-complex structure that includes NMHDR as the first
member of the control-specific notification message. Here’s the typedef of the
NM_TREEVIEW notification structure:

```
typedef struct tagNMTREEVIEW {
    NMHDR    hdr;    //the NMHDR structure for this
                    // notification
    UINT       action;  //code indication action taken
    TVITEM    itemOld;  //the item losing selection
    TVITEM    itemNew;  //item receiving selection
    POINT     ptDrag;   //point where user started
                    // dragging item
} NM_TREEVIEW, FAR *LP_NMTREEVIEW;
```

For the purposes at hand, what is important about the NM_TREEVIEW
structure is that it contains a fully populated TVITEM structure for the item
receiving selection. In order to manipulate the item icon, we get the item’s han-
dle from itemNew:

```c
// get key handle from tree item
TVITEM tvitem = (TVITEM)pNMTreeView->itemNew;
```

We set the new image with the CTreeCtrl member SetItemImage(). The para-
eters to SetItemImage(), in the order shown, are the handle to the tree control
item, the zero-based image list index for the new icon, and the index of the
item’s image when the item is selected:

```c
if(tvitem.iImage == 0 )
{
    m_RegTree.SetItemImage( tvitem.hItem, 1, 0 );
}
else
{
    m_RegTree.SetItemImage( tvitem.hItem, 0, 0 );
}
*pResult = 0;
```

### Handling a Change of Item Selection

We chose to handle the TVN_SELCHANGED notification because it is sent
after the user has selected a new registry key from the tree control pane. At the
least, this means that we have to clear the list control, because we only display
values for one key at a time. If the newly selected registry key has values, we
also have to update the list view with the key’s values.

The first step is to get a pointer to the tree view control object. We get a
pointer to the treeview by casting the NMHDR structure pointer to an
NM_TREEVIEW* and assigning the pointer to pNMTreeView:
void CWalkReg::OnSelchangedRegTree(NMHDR* pNMHDR, LRESULT* pResult)
{
    NM_TREEVIEW* pNMTreeView = (NM_TREEVIEW*)pNMHDR;

    // get key handle from tree item
    HKEY hRegKey = (HKEY)pNMTreeView->itemNew.lParam;

    // clear list control with the ClistBox member ResetContent():
    // clear list control
    m_RegList.ResetContent();

    // does this key have values?
    int iValueIndex = 0;
    WCHAR wszValueName[MAX_PATH];
    BYTE bValueData[1024];
    DWORD dwValueNameSize = dim(wszValueName);
    DWORD dwValueDataSize = dim(bValueData);
    DWORD dwRegDataType;
    memset(bValueData, 0x0, sizeof(bValueData));
    LONG rc = CeRegEnumValue(hRegKey, iValueIndex,
                              wszValueName, &dwValueNameSize,
                              NULL, &dwRegDataType,
                              (PBYTE)&bValueData, &dwValueDataSize);

    if( rc != ERROR_SUCCESS )
    {
        pWalkRegPage->m_RegList.InsertString(-1,
                      "No values exist for this key");
        *pResult = 0;
        return;
    }
If key values are found, we pass the value name, data, data size, and registry data type to the CWalkReg member InterpretKeyValue(). This function formats the key data according to its registry data type and inserts it in the list. (We’ll explore registry data types exhaustively when we examine InterpretKeyValue().)

Notice that before making the next call to CeRegEnumValue(), increment the value index, reinitialize dwValueNameSize and dwValueDataSize, and initialize the value data buffer. These steps are critical to iterating through the values. The index parameter moves the scan through the key’s list of values. The name size and data size parameters are used in two ways: When passed in to the function, they establish the size of the buffers. When passed back, they report the size of the data that was actually copied to the buffers.

We continue to loop, testing for ERROR_SUCCESS. When the test fails, there are no more keys:

```c
while (rc == ERROR_SUCCESS)
{
    InterpretKeyValue( wszValueName, (PBYTE)&bValueData,
                        dwValueDataSize, dwRegDataType);
    iValueIndex++;
    dwValueNameSize = dim(wszValueName);
    dwValueDataSize = dim(bValueData);
    memset(bValueData, 0x0, sizeof(bValueData));
    rc = CeRegEnumValue (hRegKey, iValueIndex,
                         wszValueName, &dwValueNameSize,
                         NULL, &dwRegDataType,
                         (PBYTE)&bValueData, &dwValueDataSize);
}

*pResult = 0;
```

### How Key Values Are Inserted in the List Control

Registry key values are limited to a specific set of data types. Accessing the key values is a two-step process. First, we find the values by enumerating them, then we need to format the raw value data in order to make it useful for our particular purpose. In the InterpretKeyValue() member, we handle the second step. The parameters to InterpretKeyValue(), in the order shown, are a WCHAR string containing the value’s name, the address of the data buffer, the size of the data, and the registry data type. Just a couple of points before we push on with the job of inserting values in the listbox. First, registry values can be unnamed, so it’s possible to have data, but not have a value name. Second, we copied a maximum of 1,024 bytes of data per value, but this is an artificial limit. The value
could actually have more data than that. If you call CeRegEnumValue() with the data buffer address and size set to NULL, the function returns the actual size of the data, and you can allocate a buffer accordingly:

```c
//insert the name of the value
memset(szMBCSData, 0x0, sizeof(szMBCSData));
wcsombs(&szMBCSData[0], pwszValueName, sizeof(szMBCSData));
m_RegList.InsertString(-1, szMBCSData);
```

First, we insert the name of the value, and then we use a switch to appropriately handle the value data. Notice that all string data passed back from the CE registry is Unicode, and so must be translated to multibyte character format before it can be inserted in the listbox control:

```c
memset(szMBCSData, 0x0, sizeof(szMBCSData));
switch (dwRegDataType)
{
    case REG_MULTI_SZ:
    case REG_EXPAND_SZ:
    case REG_SZ:
        wcsombs(&szMBCSData[0], (WCHAR*)pbValueData, sizeof(szMBCSData));
        break;
    case REG_DWORD:
        sprintf(&szMBCSData[0],"%x " , *(DWORD *)pbValueData);
        break;
    case REG_BINARY:
        szMBCSData[0] = '\0';
        iLoopLimit =
            ( (int)dwValueDataSize < sizeof(szMBCSData) - 1)?
                dwValueDataSize : sizeof(szMBCSData) - 1;
        for ( i = 0; i < iLoopLimit; i++)
        {
            //len = lstrlen ((LPTSTR)szData[0]);
            sprintf (&szMBCSData[i], "%02X " , pbValueData[i]);
            // if (len > dim(szData) - 6)
            //    break;
        }
        break;
    default:
        sprintf (&szMBCSData[0],"Unknown type: %x", dwRegDataType);
        break;
}
m_RegList.InsertString(-1, szMBCSData);
m_RegList.InsertString(-1, " ");
return;
```

Table 11.4 is a complete list of registry data types and their constants.
Table 11.4 CE Registry Data Types and Their Uses

<table>
<thead>
<tr>
<th>REGISTRY DATA TYPE NAME</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_BINARY</td>
<td>Byte stream; Good for application defined / interpreted complex types.</td>
</tr>
<tr>
<td>REG_DWORD</td>
<td>A 32-bit number.</td>
</tr>
<tr>
<td>REG_DWORD_LITTLE_ENDIAN</td>
<td>A 32-bit number in little-endian format. Windows CE is designed to run on little-endian computer architectures. Some UNIX systems are big endian.</td>
</tr>
<tr>
<td>REG_DWORD_BIG_ENDIAN</td>
<td>A 32-bit number in big-endian format.</td>
</tr>
<tr>
<td>REG_EXPAND_SZ</td>
<td>A null-terminated string that contains unexpanded references to environment variables (for example, “%PATH%”).</td>
</tr>
<tr>
<td>REG_MULTI_SZ</td>
<td>An array of null-terminated strings, terminated by two null characters.</td>
</tr>
<tr>
<td>REG_NONE</td>
<td>No defined value type.</td>
</tr>
<tr>
<td>REG_RESOURCE_LIST</td>
<td>A device-driver resource list.</td>
</tr>
<tr>
<td>REG_SZ</td>
<td>A null-terminated string. It will be a Unicode or ANSI string, depending on whether you use the Unicode or ANSI functions.</td>
</tr>
</tbody>
</table>

Walking the Registry Tree

To correctly initialize the tree view control, we have to walk all the way to the end of each of the branches of the registry tree. Since we don’t know the exact depth of the registry hierarchy, and since it isn’t likely to be symmetrical, we use recursion to find all the nodes in the tree. Recursion, in a nutshell, is when a function calls itself, until some condition is met. (Figure 11.3)

As mentioned earlier, recursion is indispensable for traversing hierarchical data, but it has risks. A runaway recursion will overflow the stack and probably cause an unrecoverable failure (a.k.a. crash). Also, deep recursion is likely to cause the stack to be reallocated, which at least will slow your code down and at worst will expose other silent code defects. Use recursion if it makes sense, but don’t overuse it. Here are a few tips on recursion:

- Try to minimize the size of parameter lists for recursive functions.
- Try to minimize the size of local data in a recursive function.
Don’t use recursion unless it is clearly required by the problem you are trying to solve.

Don’t use recursion unless you can reasonably estimate the depth of the recursion.

Now we’ll take a detailed look at the recursive function we use to walk the registry, WalkRegTree(), initializing the tree control to depict the structure of the registry. The parameters, in the order shown, are the handle to an open registry key and the handle to a tree item:

```c
BOOL CWalkReg::WalkRegTree(HKEY hKey, HTREEITEM htiParent)
{
    DWORD dwNSize;
    DWORD dwCSize;
    WCHAR wszName[MAX_PATH];
    CHAR szMBCSName[MAX_PATH];
    WCHAR wszClass[256];
    FILETIME ft;
    TVITEM tviTreeViewItem;
    DWORD dwNumChildren = 0;
    DWORD dwNumGrandChildren = 0;
    LONG rc;
    HTREEITEM htiChild;
    HKEY hSubKey;
```

![Figure 11.3](image-url) The Walk Registry tree page of the RapiDemo app.
Using the handle to the key passed in to the function, we query to find out if the key has any children. We do this with the function CeRegQueryInfoKey(), passing NULLs for all parameters except dwNumChildren. This instructs the function to return only a count of the child keys of this key:

```c
// query for count of children
rc = CeRegQueryInfoKey (hKey, NULL, NULL, 0,
                        &dwNumChildren, NULL, NULL, NULL,
                        NULL, NULL, NULL, NULL);
```

Here’s the declaration for CeRegQueryInfoKey(), which shows the other items of information you can retrieve about the key:

```c
LONG RegQueryInfoKey ( HKEY hKey,       //Handle to open reg key
                        LPWSTR lpClass,           //Unicode key class name
                        LPDWORD lpcbClass,        //Unicode buffer size for
                        // class name
                        LPDWORD lpReserved,       //unused
                        LPDWORD lpcSubKeys,        //count of child keys
                        LPDWORD lpcbMaxSubKeyLen, //size of longest child
                        // key name
                        LPDWORD lpcbMaxClassLen   //longest child key class
                        // name
                        LPDWORD lpcValues,         //count of key values
                        LPDWORD lpcbMaxValueNameLen, //longest value name
                        LPDWORD lpcbMaxValueLen,  // unused
                        PDWORD lpcbSecurityDescriptor, // unused
                        PFILETIME lpftLastWriteTime    // unused
                        );
```

Passing valid addresses in any of the parameters causes the corresponding data to be returned. To get the class name for a key, you must initialize both the lpClass and lpcbClass parameters.

Now we use the returned count of child keys, dwNumChildren, to enumerate the child keys of the key passed to the function:

```c
//Iterate child keys
for( int iChildKeyEnumIndex = 0;
    iChildKeyEnumIndex < dwNumChildren;
    iChildKeyEnumIndex++ )
{
    //get child key name
    dwNSize = dim(wszName);
    dwCSize = dim(wszClass);
    rc = CeRegEnumKeyEx (hKey, iChildKeyEnumIndex,
                          wszName, &dwNSize, NULL,
                          wszClass, &dwCSize, &ft);
    if(rc != ERROR_SUCCESS)
    { return FALSE; }
```
When we find a child key, we open it using the name returned by CeRegEnumKeyEx() and capture the child key’s handle in hSubKey:

```
//open child key by name, capture HKEY
rc = CeRegOpenKeyEx(hKey, wszName, 0,
                      KEY_ALL_ACCESS, &hSubKey );
if( rc != ERROR_SUCCESS )
{ return FALSE; }  
```

Now we find out if the child key has children by passing its handle to CeRegQueryInfoKey():

```
// query for count of grand children
rc = CeRegQueryInfoKey (hSubKey, NULL, NULL, 0,
                        &dwNumGrandChildren, NULL,
                        NULL, NULL,
                        NULL, NULL, NULL, NULL);
```

Every time we find hSubKey exists, we insert a new item in the tree control. We need to translate the key’s Unicode name to multibyte format to make a caption for the tree item, then we call our CWalkReg class member InsertTreeItem() to create and position the new tree item:

```
//convert the name string from WCS to MBCS
wcstombs( szMBCSName, wszName, sizeof(szMBCSName) );
// Add key to tree view.
htiChild = InsertTreeItem (htiParent, szMBCSName, 0,
                          dwNumGrandChildren, hSubKey);
```

If we know that the item has no children, we tell it so by passing a 0 (FALSE) flag in the children member of the TVITEM structure and setting this new item data with the CTreeCtrl member SetItem():

```
if(dwNumChildren == 0)
{
    tviTreeViewItem.hItem = htiChild;
    tviTreeViewItem.mask =
                      TVIF_CHILDREN | TVIF_HANDLE;
    tviTreeViewItem.cChildren = 0;
    m_RegTree.SetItem( &tviTreeViewItem );
}
```

If the subkey has children, we need to recurse in order to continue walking out the registry tree:
Initialization of the Walk Registry Tree Dialog

Well, it took a darn long time to get here, but we finally understand all the constituent parts of the Walk Registry Tree page, and it’s time to initialize it. First, we initialize the RAPI subsystem with a call to RapiInit(). We call the balancing RapiUninit() in the destructor for the class CWalkReg():

```cpp
BOOL CWalkReg::OnInitDialog()
{
    CImageList *pImageList;
    CBitmap bitmap;
    CRapiDemoApp *pApp;
    HKEY hKey;
    LONG rc;

    HRESULT hr = CeRapiInit();
    if ( hr != ERROR_SUCCESS )
        return FALSE;

    The next few lines are commented, but included here for the following reason. As you’ve noticed if you’ve built and run the RapiDemo program over a serial link, it takes a long time to initialize the tree control that displays the registry. For this reason, we only show one of the root registry hierarchies in RapiDemo’s tree control. However, in any practical application, you’d probably want to see the whole registry, not just the shortest tree. The commented lines below show the HKEY constants you’d use if you wanted to open and traverse all of the registry hierarchy, along with properly formatted tree control captions:

    CString szBaseKeys[4];
    HKEY hRootKeys[4];
    szBaseKeys[0] = _T("HKEY_CLASSES_ROOT");
    szBaseKeys[1] = _T("HKEY_CURRENT_USER");
    szBaseKeys[2] = _T("HKEY_LOCAL_MACHINE");
```
In order to have icons next to the tree control items, we must create and initialize an image list control. The parameters to Create() are the x and y dimensions of the bitmaps we’ll add to the image list, the mask that is used to display the images, the number of bitmaps the image list will initially hold, and the number of bitmaps by which the image list can grow:

```c
CPropertyPage::OnInitDialog(); // let the base class do
// the default work
pImageList = new CImageList();
pImageList->Create(16, 16, ILC_COLOR, 2, 0);
```

Next, we load a single bitmap resource, which contains both image bitmaps. We add the bitmaps to the image list, and delete the bitmap object. Because we allocated the image list on the heap with new, we use delete to dispose of it in the destructor for this class:

```c
bitmap.LoadBitmap(IDB_FOLDER_BITMAPS);
pImageList->Add(&bitmap, (CBitmap *)NULL);
bitmap.DeleteObject();
```

Next, we set the image list in the tree control. The parameters to SetImageList() are a pointer to the initialized image list object and a flag that means that these images are to be used as tree item icons:

```c
m_RegTree.SetImageList(pImageList, TVSIL_NORMAL);
```

Now we open one of the four root registry keys, HKEY_CURRENT_USER:

```c
rc = CeRegOpenKeyEx(HKEY_CURRENT_USER, NULL,
0, KEY_ALL_ACCESS, &hKey );
if( rc != ERROR_SUCCESS )
{ return FALSE; }
```

We get a count of the children for this key, calling CeRegQueryInfoKey():

```c
// query for count of children
DWORD dwNumChildren;
CeRegQueryInfoKey (HKEY_CURRENT_USER,
NULL, NULL, 0, &dwNumChildren,
NULL, NULL, NULL,
NULL, NULL, NULL, NULL);
```
Now we insert the root key in the tree view, calling the CWalkReg member InsertTreeItem(), and capture the handle to the tree item it returns:

```cpp
m_rghItem = InsertTreeItem( NULL, "HKEY_CURRENT_USER", 0,
                           dwNumChildren, hKey );
```

If the key has children, we call WalkRegTree(), which will recursively walk out the registry tree, adding and initializing tree control items as it goes:

```cpp
if( dwNumChildren > 0 )
{
    WalkRegTree( hKey, m_rghItem );
}

return FALSE;
```

Looking Ahead

In the next chapter, we’ll use RAPI to find databases on the remote machine. In particular, you’ll see how to find and manipulate the native databases for Windows CE-integrated applications.
In this chapter we examine the use of RAPI to find and access remote databases. This capability is one of the most strategic aspects of the CE architecture. If you doubt this, consider the following. In 1985, if you were sitting in a darkened theater and heard someone’s pager go off, you’d naturally assume that there was an obstetrician in the room who was going to have to rely on second-hand information as to how the movie ended. Today, even my plumber has a pager. At present, CE devices (HPCs and PPCs) are mostly the province of fairly affluent, early-adopter types. Like my plumber and his digital pager, it won’t be all that long before high-end handhelds give way to palmtop CE devices as the dominant form factors. Equipped with environmental sensors, image capture, and GPS capability, we’ll see CE gadgets taking on jobs that are too tedious, dangerous, or time consuming for humans, and too small or remote to be practical for PC hardware.

**The RemoteDBScan Example**

In this chapter’s example, you’ll see how to find all of the databases of a remote device and retrieve database attributes. You’ll also see how to read the
records from a specific remote database, dynamically detecting property types and appropriately interpreting property data. Once again we use MFC as the basis of the example.

Here are a few basics about the features of RemoteDBScan (see Figure 12.1) and the way in which they are implemented.

For the view, we use a specialized version of the standard MFC CListView class, CListViewEx. The sources for this class have been distributed with the last several generations of Visual C++, and although they are not formally part of the MFC libraries, for all intents and purposes they have become a standard. When the user selects an item shown in a CListViewEx view, the entire row that contains the item is selected. You’ll see how to derive the CRemoteDBScanView class from CListViewEx.

When the application launches, the view contains a list showing the attributes of all of the databases on the remote system. To see the records in a specific database, select it by clicking on its row in the list view, and then choose the Remote Database Access.Get Database Records for the main frame window’s menu. This menu item launches a dialog that displays all of the records for the selected database (see Figure 12.2).

Because you can use the RemoteDBScan application to access any database of the CE device, we don’t include facility to add, delete, or modify records. However, you will see how to capture the CE object identifiers (CEOID) for records as they are found. Once you have the CEOID, writing and deleting individual records is a straightforward process. You can learn more by looking at Chapter 9, “A CE Database Primer.” Table 12.1 describes the RemoteDBScan project source files.

<table>
<thead>
<tr>
<th>SOURCE FILENAME</th>
<th>HEADER FILENAME</th>
<th>IMPLEMENTS</th>
<th>FILE SHOWN IN CHAPTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>MainFrm.cpp</td>
<td>MainFrm.h</td>
<td>Default frame window and menu behaviors</td>
<td>Yes</td>
</tr>
<tr>
<td>RemoteDBScan.cpp</td>
<td>RemoteDBScan.h</td>
<td>Application object</td>
<td>No</td>
</tr>
<tr>
<td>RemoteDBScan.rc</td>
<td>resource.h</td>
<td>Resources</td>
<td>No</td>
</tr>
<tr>
<td>RemoteDBScanDoc.cpp</td>
<td>RemoteDBScanDoc.h</td>
<td>Default document</td>
<td>No</td>
</tr>
<tr>
<td>stdafx.cpp</td>
<td>Stdafx.h</td>
<td>MFC support</td>
<td>No</td>
</tr>
<tr>
<td>RemoteDBScan View.</td>
<td>RemoteDBScanView.h</td>
<td>Displays list of remote databases</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 12.1  RemoteDBScan Project Source Files and Their Uses (Continued)

<table>
<thead>
<tr>
<th>SOURCE FILENAME</th>
<th>HEADER FILENAME</th>
<th>IMPLEMENTS</th>
<th>FILE SHOWN IN CHAPTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecordListDialog.cpp</td>
<td>RecordListDialog.h</td>
<td>Displays records from selected database</td>
<td>Yes</td>
</tr>
<tr>
<td>ListViewEx.cpp</td>
<td>ListViewEx.h</td>
<td>Extensions to the MFC CListView</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 12.1  RemoteDBScan’s main window.

Figure 12.2  Records from the selected database.
MainFrm.h
// MainFrm.h : interface of the CMainFrame class
//
#if !defined(AFX_MAINFRM_H__C022A313_F464_11D5_90BE_24CF09C10000__INCLUDED_)
define AFX_MAINFRM_H__C022A313_F464_11D5_90BE_24CF09C10000__INCLUDED_
#include "RecordListDialog.h"    // Added by ClassView
#include "rapi.h"    // Added by ClassView
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
class CMainFrame : public CFrameWnd
{

protected: // create from serialization only
    CMainFrame();
    DECLARE_DYNCREATE(CMainFrame)

public:
    // Operations
public:
    // Overrides
    // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CMainFrame)
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
   //}}AFX_VIRTUAL
    // Implementation
public:
    CRecordListDialog m_RecordListDialog;
    virtual ~CMainFrame();
#endif
#if _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
protected:  // control bar embedded members
    CStatusBar  m_wndStatusBar;
    CToolBar    m_wndToolBar;

protected:
    // Generated message map functions
   //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnGetDatabaseRecords();
   //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

Listing 12.1   Important source files for the RemoteDBScan project.
// MainFrm.cpp : implementation of the CMainFrame class
//
#include "stdafx.h"
#include "RemoteDBScan.h"
#include "ListViewEx.h"
#include "RemoteDBScanView.h"
#include "rapi.h"
#include "MainFrm.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////////////////////////////////////////
// CMainFrame
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
//}}AFX_MSG_MAP
ON_WM_CREATE()  
ON_COMMAND(IDM_GET_DATABASE_RECORDS, OnGetDatabaseRecords) 
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
static UINT indicators[] =
{
    ID_SEPARATOR,  // status line indicator
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction
CMainFrame::CMainFrame()
{
}
CMainFrame::~CMainFrame()
{
}
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT,
    WS_CHILD | WS_VISIBLE | CBRS_TOP |
    CBRS_GRIPPER | CBRS_TOOLTIPS |
    CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
    !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
    TRACE0("Failed to create toolbar\n");
Listing 12.1  Important source files for the RemoteDBScan project. (continues)
return -1; // fail to create
}
if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators,
    sizeof(indicators)/sizeof(UINT)))
{
    TRACE0("Failed to create status bar\n");
    return -1; // fail to create
}
// TODO: Delete these three lines if you don't want the toolbar to
// be dockable
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
return 0;
}
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs
    return TRUE;
}

Listing 12.1
Important source files for the RemoteDBScan project.
return;

int ipos = (int)--pos;
int iLex = strcmp(szBuff, "0");
if ( iLex == 0 )
{
    AfxMessageBox( "This database has no records.", MB_OK, 0);
    return;
}

//got a selection & some records, so put up the dialog
m_RecordListDialog.DoModal();

RemoteDBScanView.h
// RemoteDBScanView.h : interface of the CRemoteDBScanView class
//@
#define AFX_REMOTEDBSCANVIEW_H__C022A317_F464_11D5_90BE_24CF09C10000_INCLUDED_#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "MainFrm.h"
#include "RemoteDBScanDoc.h"
#include "ListViewEx.h"
#include "rapi.h"
class CRemoteDBScanView : public CListViewEx
{
protected: // create from serialization only
    CRemoteDBScanView();
    DECLARE_DYNCREATE(CRemoteDBScanView)
    // Attributes

Listing 12.1 Important source files for the RemoteDBScan project. (continues)
```cpp
public:
    CRemoteDBScanDoc* GetDocument();
// Operations
public:
// Overrides
// ClassWizard generated virtual function overrides
//}}AFX_VIRTUAL(CRemoteDBScanView)
public:
virtual void OnDraw(CDC* pDC);  // overridden to draw this view
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
virtual void OnInitUpdate();
virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
//}}AFX_VIRTUAL
// Implementation
public:
    BOOL m_bColumnsExist;
    BOOL AddRowToList( CEOID, CEOIDINFO*);
    virtual ~CRemoteDBScanView();
#endif
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
protected:
// Generated message map functions
protected:
//}}AFX_MSG(CRemoteDBScanView)
DECLARE_MESSAGE_MAP()
};
#ifndef _DEBUG  // debug version in RemoteDBScanView.cpp
inline CRemoteDBScanDoc* CRemoteDBScanView::GetDocument()
{ return (CRemoteDBScanDoc*)m_pDocument; }
#endif
/**************************************************************************
///
//{{AFX_INSERT_LOCATION}}
#endif  // !defined(AFX_REMOTEDBSCANVIEW_H__C022A317_F464_11D5_90BE_24CF09C10000__INCLUDED_)
# include "ListViewEx.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////
// CRemoteDBScanView
IMPLEMENT_DYNCREATE(CRemoteDBScanView, CListViewEx)
BEGIN_MESSAGE_MAP(CRemoteDBScanView, CListViewEx)
//{{AFX_MSG_MAP(CRemoteDBScanView)
//}}AFX_MSG_MAP
// Standard printing commands
ON_COMMAND(ID_FILE_PRINT, CListViewEx::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CListViewEx::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CListViewEx::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////
// CRemoteDBScanView construction/destruction
CRemoteDBScanView::CRemoteDBScanView()
{
// TODO: add construction code here
m_bColumnsExist = FALSE;
}
CRemoteDBScanView::~CRemoteDBScanView()
{
}
BOOL CRemoteDBScanView::PreCreateWindow(CREATESTRUCT& cs)
{
// TODO: Modify the Window class or styles here by modifying
// the CREATESTRUCT cs
return CListViewEx::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////
// CRemoteDBScanView drawing
void CRemoteDBScanView::OnDraw(CDC* pDC)
{
CRemoteDBScanDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
}
void CRemoteDBScanView::OnInitialUpdate()
{
CListViewEx::OnInitialUpdate();
CListViewEx::OnInitialUpdate();
if( m_bColumnsExist )
{
return;

Listing 12.1 Important source files for the RemoteDBScan project. (continues)
} else
{
    m_bColumnsExist = TRUE;
}
CRemoteDBScanApp* pApp = (CRemoteDBScanApp*)AfxGetApp();

//don't add columns if they are already there
LV_COLUMN lvc1;
memset(&lvc1, 0x0, sizeof(lvc1));
lvc1.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
CListCtrl& ListCtrl = GetListCtrl();
BOOL bColumns = ListCtrl.GetColumn(0, &lvc1);
if( bColumns)
    return;
// insert columns
int i;
LV_COLUMN lvc;
lvc.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
CRect rectClient;
this->GetClientRect( &rectClient );
for(i = IDS_DATABASE_NAME; i<= IDS_DATABASE_CEOID; i++)
{
    CString str;
    str.LoadString(i);
    lvc.iSubItem = i - IDS_DATABASE_NAME;
    lvc.pszText = (LPSTR)(LPCTSTR)str;
    lvc.cx = rectClient.right / (IDS_DATABASE_CEOID -
        IDS_DATABASE_NAME);
    lvc.fmt = LVCFMT_LEFT;
    ListCtrl.InsertColumn(i,&lvc);
}
HANDLE hEnum;
CEOID enumCeoid;
CEOIDINFO oidInfo;
BOOL bOk;
HRESULT hr = CeRapiInit();
if( hr != ERROR_SUCCESS )
    {return ;}
//enum databases
hEnum = CeFindFirstDatabase(0);
while( enumCeoid = CeFindNextDatabase( hEnum))
{
    bOk = CeOidGetInfo(enumCeoid, &oidInfo);
    AddRowToList( enumCeoid, &oidInfo );
}
CeRapiUninit();

Listing 12.1  Important source files for the RemoteDBScan project.
BOOL CRemoteDBScanView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

/////////////////////////////////////////////////////////////////////
// CRemoteDBScanView diagnostics
#ifdef _DEBUG
void CRemoteDBScanView::AssertValid() const
{
    CListViewEx::AssertValid();
}
void CRemoteDBScanView::Dump(CDumpContext& dc) const
{
    CListViewEx::Dump(dc);
}
CRemoteDBScanDoc* CRemoteDBScanView::GetDocument()
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CRemoteDBScanDoc)));
    return (CRemoteDBScanDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////
// CRemoteDBScanView message handlers
/
BOOL CRemoteDBScanView::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.style |= LVS_SHOWSELALWAYS | LVS_REPORT;
    return CListViewEx::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////
// CRemoteDBScanView initialization
void CRemoteDBScanView::OnInitialUpdate()
{
}

/////////////////////////////////////////////////////////////////////
// CPrinterView drawing
void CRemoteDBScanView::OnDraw(CDC* pDC)
{
    LV_ITEM     lvitem;   // List View Item Descriptor struct
    int         iLineSpacing, iThisLineY;
    int         iStops[5];    // strlen of longest string in column
    int*        piStops;
    CString     strLineBuff;
    TEXTMETRIC  tm;
    static _TCHAR szLineBuff[256];
    static _TCHAR szBuff[128];
    static _TCHAR szWhiteSpace[64];
    static _TCHAR* pBuff;

Listing 12.1  Important source files for the RemoteDBScan project. (continues)
memset(szWhiteSpace, 0x20, sizeof(szWhiteSpace));
CListCtrl& ListCtrl = this->GetListCtrl();
lvitem.mask = LVIF_TEXT;
lvitem.pszText = szBuff;
lvitem.cchTextMax = sizeof(szBuff);
int iNumberItems = ListCtrl.GetItemCount();
//don't print empty window
CString str, str1;
if ( !(iNumberItems))
{
    str1.LoadString(IDS_ERR_PRINT_EMPTY_WINDOW);
    str.LoadString(IDS_ERR_PRINT_EXCEPTION);
    CWnd::MessageBox(str1, str, MB_ICONHAND | MB_OK);
    return;
}
int iNumberSubItems = 4;
// Calculate column stops
piStops = (int*)&iStops;
CalcColumnStops(piStops, pDC);
//calc line height
pDC->GetTextMetrics(&tm);
iLineSpacing = tm.tmHeight + tm.tmExternalLeading;
iThisLineY = 10;
CRemoteDBScan* pApp = (CRemoteDBScan*)AfxGetApp();
CString csWindowCaption;
pApp->m_pMainWnd->GetWindowText(csWindowCaption);
pDC->TextOut(iStops[0], iThisLineY, csWindowCaption);
iThisLineY += 2 * iLineSpacing;
//write out column headings, skip one line
int iStringID = IDS_FNAME_COL_HEADING;
for (int i = 0; i < iNumberSubItems; i++)
{
    str.LoadString(iStringID);
    pDC->TextOut(iStops[i], iThisLineY, str);
iStringID++;
}
iThisLineY += 2 * iLineSpacing;
//write a line
for (i = 0; i < iNumberItems; i++)
{
    //clean the buffer & init moveable ptr
    lvitem.iItem = i;
    for (int j = 0; j < 4; j++)
    {
        //write the item
        lvitem.iSubItem = j;
        ListCtrl.GetItem(&lvitem);
pDC->TextOut(iStops[j], iThisLineY, lvitem.pszText);
    }

Listing 12.1 Important source files for the RemoteDBScan project.
//if there is a sel list,  
//write it out  
CString*  pstr;  
CPtrList* pList =  
(CPtrList*)ListCtrl.GetItemData(i);  
DWORD dwListFlag = (DWORD)pList;  
POSITION pos;  
if(pList != NULL)  
{  
    for( pos = pList->GetHeadPosition(); pos != NULL; )  
    {    //get a ptr from the list  
        pstr = (CString*)pList->GetNext( pos );  
        //print these in the constraints col, 1 per line  
        iTThisLineY += iLineSpacing;  
        pDC->TextOut(iStops[3],  
                    iTThisLineY,  
                    pstr->GetBuffer(pstr->GetLength()));  
    } //end for (pos)  
} //end if(pList != NULL)  

// do the next line  
iThisLineY += iLineSpacing;  
} //end for j

Listing 12.1  Important source files for the RemoteDBScan project. (continues)
wcestombs(szTempBuff, poidInfo->infDatabase.szDbaseName,
    sizeof(szTempBuff));
lvitem.pszText = szTempBuff;
ListCtrl.InsertItem(&lvitem);
// save the ceoid for this database in item data
ListCtrl.SetItemData( lvitem.iItem, (DWORD)enumCeoid );
lvitem.iSubItem = 1;
sprintf( szTempBuff, "%li", poidInfo->infDatabase.dwDbaseType);
lvitem.pszText = szTempBuff;
ListCtrl.SetItem(&lvitem);
lvitem.iSubItem = 2;
sprintf( szTempBuff, "%x", poidInfo->infDatabase.dwFlags);
ListCtrl.SetItem(&lvitem);
lvitem.iSubItem = 3;
sprintf( szTempBuff, "%li", poidInfo->infDatabase.wNumRecords);
ListCtrl.SetItem(&lvitem);
lvitem.iSubItem = 4;
sprintf( szTempBuff, "%li", poidInfo->infDatabase.dwSize);
ListCtrl.SetItem(&lvitem);
lvitem.iSubItem = 5;
CTime ctLastWrite(poidInfo->infDatabase.ftLastModified );
CString csLastWrite = ctLastWrite.Format("%A, %B %d, %Y" );
sprintf( szTempBuff, "%s", csLastWrite.GetBuffer( csLastWrite.GetLength()));
ListCtrl.SetItem(&lvitem);
lvitem.iSubItem = 6;
sprintf( szTempBuff, "%x", enumCeoid);
ListCtrl.SetItem(&lvitem);
return(TRUE);
}

Listing 12.1 Important source files for the RemoteDBScan project.
HANDLE m_globalHDB;
CRecordListDialog(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
//}}AFX_DATA(CRecordListDialog)
enum { IDD = IDD_RECORD_DIALOG }; CListBox m_RecordListCtrl;
//}}AFX_DATA

public:

// Overrides
//}}AFX_VIRTUAL(CRecordListDialog)
protected:
virtual void DoDataExchange(CDataExchange* pDX);
//}}AFX_VIRTUAL

// Implementation
protected:

// Generated message map functions
//}}AFX_MSG(CRecordListDialog)
virtual BOOL OnInitDialog();
//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};

Listing 12.1  Important source files for the RemoteDBScan project. (continues)
void CRecordListDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CRecordListDialog)
    DDX_Control(pDX, IDC_RECORD_LIST, m_RecordListCtrl);
    //}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CRecordListDialog, CDialog)
    //{{AFX_MSG_MAP(CRecordListDialog)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

Listing 12.1
Important source files for the RemoteDBScan project.

Chapter 12
if( m_globalHDB == INVALID_HANDLE_VALUE )
{
    return FALSE;
}
//seek to the beginning of the database
CEOID oid = 0;
DWORD dwIndex = 0;
int iRecIndex = 1;

oid = CeSeekDatabase( m_globalHDB, CEDB_SEEK_BEGINNING,
iRecIndex, &dwIndex);
if(GetLastError != ERROR_SUCCESS)
{
    oid = oid;
}
//get the first record

// Read all properties for the record. Have the system
// allocate the buffer containing the data.
DWORD dwRecSize = 0;
WORD wProps = 0;
PBYTE pRecord = NULL;
oid = CeReadRecordProps( m_globalHDB, CEDB_ALLOWREALLOC,
    &wProps, NULL,
    (LPBYTE )pRecord, &dwRecSize);
if( oid == ERROR_INSUFFICIENT_BUFFER )
{
    return FALSE;
}
//while more records
while (oid != 0 )
{
    //format the record and insert data in the list control
    PublishRecord( oid, wProps, (PCEPROPVAL)pRecord, dwRecSize );
    //add a line between records
    m_RecordListCtrl.InsertString( -1, "" );
    //read the record
    oid = CeReadRecordProps( m_globalHDB, CEDB_ALLOWREALLOC,
        &wProps, NULL,
        (LPBYTE )pRecord, &dwRecSize); 
}
//close db
CloseHandle(m_globalHDB);
//uninit rapi
CeRapiUninit();

return TRUE; // return TRUE unless you set the focus to a control
             // EXCEPTION: OCX Property Pages should return FALSE
}
void CRecordListDialog::PublishRecord( CEOID oid, WORD wProps,
PCEPROPVAL pRecord,
(DWORD dwRecSize)
{
    char szInsertString[124];

    sprintf(szInsertString, "%s: 0x%x", "Record CEOID", oid );
m_RecordListCtrl.InsertString(-1, szInsertString);
sprintf(szInsertString, "%s: %i", "Properties Returned", wProps);
m_RecordListCtrl.InsertString(-1, szInsertString);
sprintf(szInsertString, "%s: %i", "Record Size", dwRecSize);
m_RecordListCtrl.InsertString(-1, szInsertString);
m_RecordListCtrl.InsertString(-1, "");
WORD iPropId;
int iPropDataType;
for (int i = 0; i < wProps; i++)
{
    iPropId = HIWORD(pRecord[i].propid);
    sprintf(szInsertString, "%s: %i", "Property Id", iPropId);
    m_RecordListCtrl.InsertString(-1, szInsertString);
    iPropDataType = LOWORD(pRecord[i].propid);
    CTime tFileTime;
    CString csFileTime;
    switch (iPropDataType)
    {
    //ce type
    case CEVT_I2:
        sprintf(szInsertString, "%s: %s", "Data Type", "CEVT_I2" );
        m_RecordListCtrl.InsertString(-1, szInsertString);
        sprintf(szInsertString, "%i", pRecord[i].val.iVal );
        m_RecordListCtrl.InsertString(-1, szInsertString);
        break;
    case CEVT_UI2:
        sprintf(szInsertString, "%s: %s", "Data Type", "CEVT_UI2" );
        m_RecordListCtrl.InsertString(-1, szInsertString);
        sprintf(szInsertString, "%ui", pRecord[i].val.uiVal );
        m_RecordListCtrl.InsertString(-1, szInsertString);
        break;
    case CEVT_I4:
        sprintf(szInsertString, "%s: %s", "Data Type", "CEVT_I4" );
        m_RecordListCtrl.InsertString(-1, szInsertString);
        sprintf(szInsertString, "%li", pRecord[i].val.lVal );
        m_RecordListCtrl.InsertString(-1, szInsertString);
        break;
    case CEVT_UI4:
        sprintf(szInsertString, "%s: %s", "Data Type", "CEVT_UI4" );
        m_RecordListCtrl.InsertString(-1, szInsertString);
        sprintf(szInsertString, "%ul", pRecord[i].val.ulVal );
        m_RecordListCtrl.InsertString(-1, szInsertString);
        break;
    case CEVT_FILETIME:
        sprintf(szInsertString, "%s: %s", "Data Type", "CEVT_FILETIME" );
        m_RecordListCtrl.InsertString(-1, szInsertString);
        break;
    
Listing 12.1 Important source files for the RemoteDBScan project.
```c
    tFileTime = pRecord[i].val.filetime;
    csFileTime = tFileTime.Format("%A, %B %d, %Y" );
    m_RecordListCtrl.InsertString(-1,
        csFileTime.GetBuffer(csFileTime.GetLength()) );
    break;
    case CEVT_LPWSTR: 
        sprintf(szInsertString, "%s: %s", "Data Type",
            "CEVT_LPWSTR" );
        m_RecordListCtrl.InsertString(-1, szInsertString);
        wcstombs(szInsertString, pRecord[i].val.lpwstr, sizeof(szInsertString));
        m_RecordListCtrl.InsertString(-1, szInsertString);
        break;
    case CEVT_BLOB: 
        sprintf(szInsertString, "%s: %s", "Data Type",
            "CEVT_BLOB" );
        m_RecordListCtrl.InsertString(-1, szInsertString);
        sprintf(szInsertString, "%s: %li", "Size in bytes",
            pRecord[i].val.blob.dwCount);
        m_RecordListCtrl.InsertString(-1, szInsertString);
        sprintf(szInsertString, "%s: 0x%x", "Buffer Address",
            pRecord[i].val.blob.lpb);
        break;
    default: 
        break;
}
//blank line between proprties for readability
m_RecordListCtrl.InsertString(-1, "");
```

**Listing 12.1** Important source files for the RemoteDBScan project. *(continued)*

---

**Creating a View by Deriving from CListviewEx**

RemoteDBScan began life as a typical MFC AppWizard project. In Step 6 of the project creation dialog, AppWizard gives you a chance to choose the base class for the application’s view. At this point, we chose CListView as the base class for the view. After the AppWizard generated files exist, we make a few changes in order to derive the view from CListViewEx.

First, add this include file to the header file for our view class, RemoteDBScanView.h:

```c
#include "ListViewEx.h"
```
Next we change the line where we derive our view class to use CListViewEx as a base class.

    class CRemoteDBScanView : public CListViewEx

We add the CListViewEx class to the project, and after the files are generated we copy the implementing source to our newly generate a project files, ListViewEx.h and ListViewEx.cpp, being careful not to overwrite the #defines at the top of our application’s generated files. (The source files that contain the implementation of CListViewEx are part of the MFC sample application ROWLIST.)

After we change the derivation of the class, we update source files in which pointers to the view class appear in generated code. Pointers to the view must now be of type CListViewEx*.

### Initializing the View with the List of Remote Databases

When the RemoteDBScan application opens, it displays a list control populated with a list of the remote databases and their attributes. We build and initialize the list control in the OnInitialUpdate() member of the CRemoteDBScanView class. Building a list view is a two-step process. First, we add columns to the list control, and after that we can add individual rows, one item at a time.

Before we begin to build the list control, we do the base class initialization, calling CListViewEx:: OnInitialUpdate(). Next, we test the CRemoteDBScanView member variable, m_bColumnsExist, to see if columns have already been added to the control. (m_bColumnsExist is set to FALSE in the constructor for the class.) If there are already columns, we bail out of the initialization. Otherwise, we set this member to TRUE and proceed.

    void CRemoteDBScanView::OnInitialUpdate()
    {
        CListViewEx::OnInitialUpdate();
        // TODO: You may populate your ListView with items by directly accessing
        // its list control through a call to GetListCtrl().
        CListViewEx::OnInitialUpdate();
        if( m_bColumnsExist )
        {
            return;
        }
        else
        {
            m_bColumnsExist = TRUE;
        }
    }
Now we add seven columns to the list control. To add columns, first we get a reference to the list control associated with the view.

```cpp
// insert columns
CListCtrl& ListCtrl = GetListCtrl();
```

We define the attributes of individual columns by initializing an `LV_COLUMN` structure. Here’s the typedef for `LV_COLUMN`:

```cpp
typedef struct _LVCOLUMN {
    UINT mask;         //mask flags define which members are valid
    int fmt;           //flag for column heading alignment
    int cx;            //column width in pixels
    LPTSTR pszText;    //column heading
    int cchTextMax;    //length of heading buffer
    int iSubItem;      //this column’s sub item index
    int iImage;        //image list index for column icon
    int iOrder;        //0 based column index
} LVCOLUMN;
```

The `LV_COLUMN` is used both to set and retrieve information about a column. The mask member defines which members of the structure are to be considered valid. Table 12.2 lists its possible values.

The mask flags may be combined with logical OR, so that you can set or retrieve as many or few `LV_COLUMN` members as you like.

The `fmt` member controls the column heading alignment and the placement of an optional image next to the heading. The heading of the leftmost column must be left aligned. You can use one of heading styles shown in Table 12.3.

### Table 12.2 LV_COLUMN Mask Flag Constants and Their Meanings

<table>
<thead>
<tr>
<th>MASK FLAG CONSTANT</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVCF_FMT</td>
<td><code>fmt</code> is valid or should be returned</td>
</tr>
<tr>
<td>LVCF_IMAGE</td>
<td><code>iImage</code> is valid or should be returned</td>
</tr>
<tr>
<td>LVCF_ORDER</td>
<td><code>iOrder</code> is valid or should be returned</td>
</tr>
<tr>
<td>LVCF_SUBITEM</td>
<td><code>iSubItem</code> is valid or should be returned</td>
</tr>
<tr>
<td>LVCF_TEXT</td>
<td><code>pPszText</code> is valid or should be returned</td>
</tr>
<tr>
<td>LVCF_WIDTH</td>
<td><code>cx</code> is valid or should be returned</td>
</tr>
</tbody>
</table>
Table 12.3  LV_COLUMN Heading Style Flags and Their Meanings

<table>
<thead>
<tr>
<th>HEADING FORMAT FLAG</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVCFMT_CENTER</td>
<td>Center heading text</td>
</tr>
<tr>
<td>LVCFMT_LEFT</td>
<td>Left align heading text</td>
</tr>
<tr>
<td>LVCFMT_RIGHT</td>
<td>Right align heading text</td>
</tr>
</tbody>
</table>

Now we declare and initialize an LV_COLUMN structure. We set the mask for fmt, cx, pszText, and iSubItem members. We get the dimensions of the client area and use it to set the column width. Looping through the string resources, we load the caption string for each column and also use the loop index to set the value of iSubItem. We set each column heading format to left alignment, and insert the column with a call to ListCtrl.InsertColumn().

```c
int i;
LV_COLUMN lvc;
lvc.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
CRect rectClient;
this->GetClientRect( &rectClient );
for(i = IDS_DATABASE_NAME; i<= IDS_DATABASE_CEOID; i++)
{
    CString str;
    str.LoadString(i);
    lvc.iSubItem = i - IDS_DATABASE_NAME;
    lvc.pszText = (LPSTR)(LPCTSTR)str;
    lvc.cx = rectClient.right / (IDS_DATABASE_CEOID - IDS_DATABASE_NAME);
    lvc.fmt = LVCFMT_LEFT;
    ListCtrl.InsertColumn(i,&lvc);
}
```

Once we have columns in the list control, we are ready to add rows containing the remote database attributes. The first step is a familiar one—we initialize RAPI with a call to CeRapiInit().

```c
HRESULT hr = CeRapiInit();
if( hr != ERROR_SUCCESS )
    {return ;}
```

If we are successful, we’ll begin our search for remote databases. Setting up to enumerate the databases is a two-step process. First we call CeFindFirstDatabase(). The parameter we pass indicates what type of database we wish to enumerate. The database type is an arbitrary constant that is set by the application that initially creates the database. It’s not a required parameter for database creation, but it can be useful if an application creates and uses a family of databases.
Setting their types equal makes it easy to find all of them again if you need to manipulate them as a group. Passing 0 as the parameter to CeFindFirstDatabase() causes all databases to be enumerated, regardless of type. On success, CeFindFirstDatabase() returns a handle to an enumeration context.

To actually find a database, we call CeFindNextDatabase(), passing the handle returned by CeFindFirstDatabase() as its only parameter. CeFindNextDatabase() returns the CE object identifier (CEOID) of the found database. An OID can uniquely identify many types of objects: files, directories, databases, or individual database records. We don’t have a special function for getting information for each of these types of objects. Instead, we have a function that returns attribute information for any of the object types, CeOidGetInfo().

CeOidGetInfo() takes two parameters: the CEOID for the object being queried and the address of a CEOIDINFO structure. Here is the typedef for CEOIDINFO:

```c
typedef struct _CEOIDINFO {
    WORD wObjType;       //the type of object being queried
    DWORD dwSize;        //the size of this structure
    WORD wPad;           //used for structure alignment
    union {
        CEFILEINFO infFile;   //is returned populated with
                             //attribute data
        CEDIRINFO infDirectory;
        CEDBASEINFO infDatabase;
        CERECORDINFO infRecord;
    };
} CEOIDINFO;
```

The information in which we are interested is returned in the CEDBASEINFO member of the union. We’ll get down to business populating the list after we take a look at this key structure:

```c
typedef struct _CEDBASEINFO {
    DWORD dwFlags;        //which structure members are valid
    WCHAR szDbaseName[CEDB_MAXDBASENAMELEN];
    DWORD dwDatabaseType; //application defined database type
    WORD wNumRecords;    //number records in this database
    WORD wNumSortOrder;  //number of active sort orders
    DWORD dwSize;        //database size in bytes
    FILETIME ftLastModified; //last modified time
    //an array of structures
    //defining sort order specs
    SORTORDERSPEC rgSortSpecs[CEDB_MAXSORTORDER];
} CEDBASEINFO;
```

The information returned in the CEDBASEINFO structure gives you all of the information you need to open any database on the CE device, applying sort orders if they exist. You can use the dwFlags member to detect which
other structure members are valid. The flags are found in the low order word of dwFlags, and can be isolated like this:

```c
//get the aggregated flags
WORD wFlagWord = LOWORD( dwFlags );
//is the CEDB_VALIDDBFLAGS flag set
WORD wFlagTest;
WFlagTest = wFlagWord & CEDB_VALIDDBFLAGS;
if( wFlagWord )
{
    //do something with the flags
}
```

Table 12.4 lists the possible values of dwFlags.
With this background, we are ready to find and examine remote databases.

```c
HANDLE hEnum;
CEOID enumCeoid;
CEOIDINFO oidInfo;
BOOL bOk;
//enum databases
hEnum = CeFindFirstDatabase(0);
while( enumCeoid = CeFindNextDatabase( hEnum))
{
    CeOidGetInfo(enumCeoid, &oidInfo);
    AddRowToList( enumCeoid, &oidInfo );
}
```

Now we are ready to add a row for this database with a call to AddRowToList(). Before we move on to this job, notice that after we’ve exhausted the enumeration loop, we close OnInitUpdate() by uninitializing RAPI.

```c
CeRapiUninit();
```

**Table 12.4** CEDBASEINFO Flags

<table>
<thead>
<tr>
<th>DWFLAGS VALUE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEDB_VALIDMODTIME</td>
<td><code>ftLastModified</code> member is valid.</td>
</tr>
<tr>
<td>CEDB_VALIDNAME</td>
<td><code>szDbaseName</code> member is valid.</td>
</tr>
<tr>
<td>CEDB_VALIDTYPE</td>
<td><code>dwDbaseType</code> member is valid.</td>
</tr>
<tr>
<td>CEDB_VALIDSORTSPEC</td>
<td><code>rgSortSpecs</code> member is valid.</td>
</tr>
<tr>
<td>CEDB_VALIDDBFLAGS</td>
<td>The low-order word of the <code>dwFlags</code> member is valid.</td>
</tr>
</tbody>
</table>
Adding Database Attribute Rows to the List Control

The ListView Control treats its contents as a matrix, where iItem is the zero-based row index and iSubItem is the zero-based column index. Although the CListViewEx class on which our view is based appears to the user to treat list control item as full, the underlying list control deals with individual matrix elements. For this reason, when we add the “row” containing a database’s attributes, we must add each item separately.

Adding a list item is not unlike adding a column to the list control, in that it is again a two-step process. First we initialize an LV_ITEM structure that defines the item’s attributes, then we add the item with a call to the ClistCtrl member InsertItem().

Here’s the typedef for the LV_ITEM structure:

```c
typedef struct _LVITEM {
    UINT  mask;      //flags that define which members
                    // are valid
    int  iItem;      // 0 based row index in list ctrl matrix
    int  iSubItem;   // 0 based column index in list
                    // control matrix
    UINT state;     //the item's state, state image
                    // and overlay image
    UINT stateMask; //tells which bits of the state are valid
    LPTSTR pszText;  //item caption
    int  cchTextMax; //caption buffer length
    int  iImage;     //index of image in image list
    LPARAM lParam;   //used for app specific data
} LVITEM, FAR *LPLVITEM;
```

Tables 12.5 and 12.6 show the possible values for the mask and state members of the LV_ITEM structure:

<table>
<thead>
<tr>
<th>MASK</th>
<th>FLAG</th>
<th>CONSTANT</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVIF_TEXT</td>
<td></td>
<td>pszText is valid</td>
<td></td>
</tr>
<tr>
<td>LVIF_IMAGE</td>
<td></td>
<td>iImage is valid</td>
<td></td>
</tr>
<tr>
<td>LVIF_INDENT</td>
<td></td>
<td>iIndent is valid</td>
<td></td>
</tr>
<tr>
<td>LVIF_PARAM</td>
<td></td>
<td>lParam is valid</td>
<td></td>
</tr>
<tr>
<td>LVIF_STATE</td>
<td></td>
<td>state is valid</td>
<td></td>
</tr>
</tbody>
</table>
Now let’s tour the AddRowToList() member function.

```cpp
BOOL CRemoteDBScanView::AddRowToList(CEOID enumCeoid,
                                      CEOIDINFO * poidInfo )
{
    LV_ITEM         lvitem;   //List View Item Descriptor struct
    //Get the List Control Object
    CListCtrl& ListCtrl = GetListCtrl();

    // add at the last row
    lvitem.iItem = ListCtrl.GetItemCount();
    lvitem.mask  = LVIF_TEXT | LVIF_STATE;
    lvitem.stateMask = LVIS_FOCUSED;
    lvitem.state     = LVIS_FOCUSED;

    This function is designed to add a full row at the bottom of the list control,
    so our first step is to get a count of list control items. Recall that items are syn-
    onymous with rows in this context, while subitems are synonymous with columns.

    The first item we add is the database name. This was returned in the
    CEOIDINFO structure member CEDBASEINFO’s szDbaseName member. The
    returned name string is in Unicode, so we have to translate it to multibyte
    character format before inserting it in the list control. After translation, we set
    the address of the MBCS string in lvitem.pszText, and call InsertItem().

    //add attribute string to bottom row of the listctl
    //loop thru the subitems
    char szTempBuff[512];
    lvitem.iSubItem = 0;
    wcstombs(szTempBuff, poidInfo->infDatabase.szDbaseName,
             sizeof(szTempBuff));
    lvitem.pszText = szTempBuff;
    ListCtrl.InsertItem(&lvitem);
```

### Table 12.6  LV_ITEM State Flags and Their Meanings

<table>
<thead>
<tr>
<th>ITEMSTATE FLAG</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVIS_ACTIVATING</td>
<td>Item is activating on LVN_ITEMACTIVATE notification</td>
</tr>
<tr>
<td>LVIS_CUT</td>
<td>Item is marked for cut and paste</td>
</tr>
<tr>
<td>LVIS_DROPHILITED</td>
<td>Item is a drag and drop target</td>
</tr>
<tr>
<td>LVIS_FOCUSED</td>
<td>Item has the input focus</td>
</tr>
<tr>
<td>LVIS_SELECTED</td>
<td>Item is selected</td>
</tr>
</tbody>
</table>

466 Chapter 12
The next step is a key strategy. It helps us optimize performance and makes opening the database to enumerate its records a bit more convenient. We set the OID value for this database as item data in the list control cell that holds the database name. Here’s why we do this.

CE object identifiers are permanent attributes of the objects they identify. Put another way, they are created when the object is created, and stick with the same object throughout its lifetime. For our purposes, they are worth preserving, because you can open a database (or a file) by CEOID as well as by name. Preserving the CEOID allows us to open the database without having to translate the name string in the list control back to Unicode. Also, opening the database by CEOID is a good bit faster than opening by name.

After the item has been added, we set its item data like this:

```c
//save the ceoid for this database in item data
ListCtrl.SetItemData( lvitem.iItem, (DWORD)enumCeoid );
```

Now we add the rest of the database attributes to the row. All we really have to do differently for the successive items is increment the lvitem.iSubItem, dereference the applicable member of the CEDBASEINFO structure, and format the datum using sprintf().

```c
lvitem.iSubItem = 1;
sprintf( szTempBuff, "%li", poidInfo->infDatabase.dwDbaseType);
lvitem.pszText = szTempBuff;
ListCtrl.SetItem(&lvitem);

lvitem.iSubItem = 2;
sprintf( szTempBuff, "%x", poidInfo->infDatabase.dwFlags);
ListCtrl.SetItem(&lvitem);

lvitem.iSubItem = 3;
sprintf( szTempBuff, "%i", poidInfo->infDatabase.wNumRecords);
ListCtrl.SetItem(&lvitem);

lvitem.iSubItem = 4;
sprintf( szTempBuff, "%li", poidInfo->infDatabase.dwSize);
ListCtrl.SetItem(&lvitem);

lvitem.iSubItem = 5;
CTime ctLastWrite(poidInfo->infDatabase.ftLastModified);
CString csLastWrite = ctLastWrite.Format("%A, %B %d, %Y");
sprintf( szTempBuff, "%s",
    csLastWrite.GetBuffer( csLastWrite.GetLength()));
ListCtrl.SetItem(&lvitem);

lvitem.iSubItem = 6;
sprintf( szTempBuff, "%x", enumCeoid);
ListCtrl.SetItem(&lvitem);
return(TRUE);
```
Getting Records from the Remote Device

When the user selects a database from the list control and chooses the Remote Database Access.Get Database Records menu item, we land in the mainframe window’s menu handler.

This member function has three jobs:

- It checks to make sure that the user has selected a database from which to retrieve records.
- It makes sure that the database actually contains records.
- If both these conditions are met, it launches the dialog that displays the database records.

First, we check for a row selection by calling the CListCtrl member GetFirstSelectedItemPosition(). This function returns a value of type POSITION, but if there is no selected row, it returns NULL. This is subtle and tricky—the value returned, pos, is a 1 based index. The list control refers to items using zero-based indices. In the first step, we are only interested in whether or not there is a selection. If we want to manipulate the returned selection, we have to decrement and properly cast pos in order to land on the correct list view item.

```cpp
void CMainFrame::OnGetDatabaseRecords()
{
    //get list ctrl
    CRemoteDBScanView* pListView =
        (CRemoteDBScanView*)this->GetActiveView();
    CListCtrl& ListCtrl = pListView->GetListCtrl();
    //get currently selected row
    POSITION pos = ListCtrl.GetFirstSelectedItemPosition();
    if( pos == NULL )
    {
        AfxMessageBox( "You must select a database from the list.",
                        MB_OK, 0 );
        return;
    }

    Now we use the returned position to query the item in which we stored the number of records contained in this database. We decrement pos, and assign this new value to a variable of type int, ipos. We set lvitem.iItem to ipos, and lvitem.iSubItem to 3, which is the zero-based index of the list control column containing the database record counts. We set the lvitem.mask to LVIF_TEXT, which signals that we are only interested in retrieving the item’s caption string. We call ListCtrl.GetItem(), passing the initialized LVITEM structure as its parameter.
```
//pos is a 1 based index, so we must decrement
int ipos = (int)--pos;
//get the item
LVITEM lvitem;
CHAR szBuff[12];
memset( &szBuff, 0x0, sizeof(szBuff));
lvitem.mask = LVIF_TEXT ;
//pos is a 1 based index, so we must decrement
lvitem.iItem =     ipos;
lvitem.iSubItem = 3;
lvitem.pszText = szBuff;
lvitem.cchTextMax = sizeof(szBuff);
ListCtrl.GetItem(&lvitem);

On successful return, we have the caption, and we compare it to 0. If there
are no records in the database, we message the user and bail out here.

   int iLex = strcmp( szBuff, "0" );
   //message and bail if the item has no records
   if ( iLex == 0 )
   {
      AfxMessageBox( "This database has no records.", MB_OK, 0);
      return;
   }

If we clear all of these hurdles, it’s time to open the remote database and
recover the records. The CMainFrame class declares a member variable for the
dialog that does this job. Here is an excerpt from the class header that depicts
the relationship between the dialog and the CMainFrame class:

   // Implementation
   public:
       CRecordListDialog m_RecordListDialog;
       At the bottom of CMainFrame::OnGetDatabaseRecords() we put up the modal
dialog:
       //got a selection & some records, so put up the dialog
       m_RecordListDialog.DoModal();

Initializing the CRecordListDialog

First off, we initialize the dialog’s window caption to reflect the name of the
database from which we are retrieving records. To do this, first we get a
pointer to the CMainFrame object that owns the dialog, then we get a pointer
to the CRemoteDBScanView object that owns the list control, and finally, we get the list control itself.

BOOL CRecordListDialog::OnInitDialog()
{
  CDialog::OnInitDialog();
  //Set dialog title to name of this database

  CMainFrame* pFrame = (CMainFrame*)this->GetParent();
  CRemoteDBScanView* pListView =
    (CRemoteDBScanView*)pFrame->GetActiveView();
  CListCtrl& ListCtrl = pListView->GetListCtrl();

  Once again we query the selected item position, receiving a POSITION value that we decrement and cast so that we can use it as a list control item index.

  //get currently selected row
  POSITION pos = ListCtrl.GetFirstSelectedPosition();
  //pos is a 1 based index, so we must decrement
  int ipos = (int)--pos;

  We retrieve the database name by initializing lvitem.iItem with the index of the selected row and lvitem.iSubItem with 0, the index of the list control column that contains the database names.

  //set caption for the DB we are scanning
  LVITEM lvitem;
  CHAR szBuff[124];
  memset( &szBuff, 0x0, sizeof(szBuff));
  lvitem.mask = LVIF_TEXT;
  //pos is a 1 based index, so we must decrement
  lvitem.iItem = ipos;
  lvitem.iSubItem = 0;
  lvitem.pszText = szBuff;
  lvitem.cchTextMax = sizeof(szBuff);

  The database name is returned in szBuff, which was sized so that we could safely catenate the rest of the dialog’s caption text.

  ListCtrl.GetItem(&lvitem);
  strcat(szBuff, " Records" );

  We set the string in the dialog’s title bar by calling the CWnd base class function, SetWindowText().

  this->SetWindowText(szBuff);
Now we’ll open the selected database using the CEOID value we stored in the database name’s item data. We save the database CEOID in a CRecordListDialog public member variable, m_globalCEOID.

```cpp
//get the saved CEOID
m_globalCEOID = (CEOID)ListCtrl.GetItemData(ipos);
```

We initialize RAPI and, if successful, proceed to open the database.

```cpp
//init rapi
HRESULT hr = CeRapiInit();
if( hr != ERROR_SUCCESS )
{return FALSE;}
```

We looked at this function in depth in the course of our treatment of the CE database API, but if you skipped over that part of the book, we’ll examine it again briefly here.

The parameters to CeOpenDatabase(), in the order shown, are the address of a variable of type CEOID, a pointer to a Unicode string containing the name of the database to open, the index of the SORTORDERSPEC to apply when the database is opened, a flag that tells whether to let the application increment the current position in the database of serial read operations or whether the system should automatically increment the current position, and the handle to a window that can display message boxes for the called function.

In this case, we are opening the database by CEOID value, so pass NULL for the address of the name string. Also, we are opening the database with the express intent of serially reading through its record set, so we set the fourth parameter to CEDB_AUTOINCREMENT. This means that we won’t have to handle the database seek operation except before our first read, when we set the current position to the first database record. The last parameter, m_hWnd, is a CWnd base class member variable that stores this object’s window handle.

```cpp
//open using CEOID
m_globalHDB =
    CeOpenDatabase (&m_globalCEOID, NULL,
                    0, CEDB_AUTOINCREMENT, m_hWnd);
if( m_globalHDB == INVALID_HANDLE_VALUE )
{ return FALSE; }
```

If we are successful, we save the returned database handle in the CRecordListDialog member variable, m_globalHDB. In order to do any sort of operation on a CE database, you must first explicitly set the current record by calling CeSeekDatabase(). The parameters to CeSeekDatabase(), in the order shown, are the handle to the open database, the seek type flag, the offset to
which to seek from the seek flag point, and the address of a variable that receives the index of the record to which the seek operation positioned. Notice that when we seek to the beginning of the database, the seek offset is set to 1 rather than to 0.

```c
//seek to the beginning of the database
CEOID oid = 0;
DWORD dwIndex = 0;
int iRecIndex = 1;
oid = CeSeekDatabase (m_globalHDB, CEDB.Seek.BEGINNING,
iRecIndex, &dwIndex);
```

We read the first record, with a call to CeReadRecordProps(). The parameters to this function, in the order shown, are the handle to the open database; a flag that tells the system to dynamically allocate a buffer to hold the record data; the address of a DWORD that receives the count of properties returned for this record; a NULL flag, which indicates we want to retrieve all properties for this record; the address of a pointer to the buffer in which CeReadRecordProps() placed the record data; and the address of a DWORD that gives the size in bytes of the record data.

```c
//get the first record
// Read all properties for the record. Have the system
// allocate the buffer containing the data.
DWORD dwRecSize = 0;
WORD wProps = 0;
PBYTE pRecord = NULL;
oid = CeReadRecordProps (m_globalHDB, CEDB.Allow REALLOC,
&wProps, NULL,
&(LPBYTE )pRecord, &dwRecSize);
if( oid == ERROR.INSUFFICIENT_BUFFER )
{return FALSE;}
```

If we successfully read the record, we format the record data and display it in the list box, using the CRecordListDialog member function PublishRecord. We insert a blank line in the list control to provide a separation between records, and continue reading until CeReadRecordProps() returns 0, indicating there are no more records.

```c
//while more records
while (oid != 0 )
{
  //format the record and insert data in the list control
  PublishRecord( oid, wProps, (PCEPROPVAL)pRecord,
dwRecSize );
  //add a line between records
  m_RecordListCtrl.InsertString( -1, "\n");
//read the record
oid = CeReadRecordProps (m_globalHDB, CEDB_ALLOWREALLOC,
    &wProps, NULL,
    &(LPBYTE)pRecord, &dwRecSize);
}

When the OnInitDialog() function’s work is done, we close the remote database with a call to CloseHandle(), and uninitialize the RAPI subsystem.

//close db
CloseHandle(m_globalHDB);
//unint rapi
CeRapiUninit();

return TRUE; EXCEPTION: OCX Property Pages should return FALSE

Interpreting the Retrieved Records and Formatting Them for Display

A CE database record is a compact and self-documenting block of variable size data, comprised of an array of CEPROPVAL structures. Here is the typedef for CEPROPVAL:

typedef struct _CEPROPVAL {
    CEPROPID propid; //LOWORD is CE value type
        //HIWORD is property ID
    WORD wLenData; //unused
    WORD wFlags; //flags: CEDB_PROPNOTFOUND returned
        // if prop isn't found
        // passing in CEDB_PROPDELETE causes
        // property to be deleted from record
    CEVALUNION val; //Union includes members for
        // all CE database types
} CEPROPVAL;

A CE database record may be sparse. Put another way, a record may not contain a value for each property. Also, a given CE database may contain only one record type, and there is no explicit support for hierarchically structured data.

Here’s how to interpret the array of CEPROPVAL structures in order to create and display a relational-style record from the property value array:

Set up an iteration of the CEPROPVAL array, using the count of record properties returned by CeReadRecordProps().

For each CEPROPVAL, retrieve the property index by taking the LOWORD of the propid and use it to set the column index for the property.
Next, determine the CE value type for the data stored in the record by taking the HIWORD of the propid. Use the CE data type to select the correct member of the CEVALUNION and proper type of formatting for the data.

In CRecordListDialog::PublishRecord() we discriminate the property ID and data type of each property, but for the sake of brevity, we insert them serially in a list box rather than laying them out as rows in another list control.

At the top of each record’s set of properties, we insert the record OID, the number of properties returned for this record, and the total size of the record in bytes. These data were passed to PublishRecord() as parameters. They were reported by CeReadRecordProps() in the caller.

```cpp
void CRecordListDialog::PublishRecord( CEOID oid, WORD wProps, PCEPROPVAL pRecord, DWORD dwRecSize)
{
    char szInsertString[124];
    sprintf(szInsertString, "Record CEOID: 0x%x", oid );
    m_RecordListCtrl.InsertString( -1, szInsertString );
    sprintf(szInsertString, "Properties Returned: %i", wProps );
    m_RecordListCtrl.InsertString( -1, szInsertString );
    sprintf(szInsertString, "Record Size: %i", dwRecSize );
    m_RecordListCtrl.InsertString( -1, szInsertString );
}
```

Here we set up an iteration of the array of CEPROPVAL structures. We use the HIWORD macro to recover the property ID from the propid member. The property ID is formatted and inserted in the list box.

```cpp
WORD iPropId;
int iPropDataType;
for( int i = 0; i < wProps; i++ )
{
    iPropId = HIWORD(pRecord[i].propid );
    sprintf(szInsertString, "Property Id: %i", iPropId );
    m_RecordListCtrl.InsertString( -1, szInsertString );
    iPropDataType = LOWORD(pRecord[i].propid );
}
```

Now we have to retrieve and format the record’s value data. The way we do this depends on the data’s type. Table 12.7 lists the type constants for the data in a CE database record, along with the name of the structure member in the CEVALUNION in which that type is stored.
Table 12.7  CE Database Types and Corresponding CEVALUNION Members

<table>
<thead>
<tr>
<th>CE DATABASE TYPE VALUE</th>
<th>_CEVALUNION MEMBER TYPE AND NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEVT_I2</td>
<td>short iVal;</td>
</tr>
<tr>
<td>CEVT_UI2</td>
<td>USHORT uiVal;</td>
</tr>
<tr>
<td>CEVT_I4</td>
<td>long lVal;</td>
</tr>
<tr>
<td>CEVT_UI4</td>
<td>ULONG ulVal;</td>
</tr>
<tr>
<td>CEVT_FILETIME</td>
<td>FILETIME filetime;</td>
</tr>
<tr>
<td>CEVT_LPWSTR</td>
<td>LPWSTR lpwstr;</td>
</tr>
<tr>
<td>CEVT_BLOB</td>
<td>CEBLOB blob;</td>
</tr>
</tbody>
</table>

We declare a variable to handle translation of FILETIME data outside the switch, then use the CE value type as a switch test. Notice that with the integer and filetime types, all of the record’s data is stored within the CEPROPVAL structure.

```c
CTime tFileTime;
CString csFileTime;
switch( iPropDataType )
{
    //ce type
    case CEVT_I2:
        sprintf(szInsertString, "%s: %s", "Data Type", "CEVT_I2" );
        m_RecordListCtrl.InsertString( -1, szInsertString );
        sprintf(szInsertString, "%i", pRecord[i].val.iVal );
        m_RecordListCtrl.InsertString( -1, szInsertString );
        break;
    case CEVT_UI2:
        sprintf(szInsertString, "%s: %s", "Data Type", "CEVT_UI2" );
        m_RecordListCtrl.InsertString( -1, szInsertString );
        sprintf(szInsertString, "%ui", pRecord[i].val.uiVal );
        m_RecordListCtrl.InsertString( -1, szInsertString );
        break;
    case CEVT_I4:
        sprintf(szInsertString, "%s: %s", "Data Type", "CEVT_I4" );
        m_RecordListCtrl.InsertString( -1, szInsertString );
        sprintf(szInsertString, "%li", pRecord[i].val.lVal );
        m_RecordListCtrl.InsertString( -1, szInsertString );
        break;
```
case CEVT_UI4:
sprintf(szInsertString, "%s: %s", "Data Type", "CEVT_UI4" );
m_RecordListCtrl_InsertString(-1, szInsertString);
sprintf(szInsertString, "%uli", pRecord[i].val.ulVal);
m_RecordListCtrl_InsertString(-1, szInsertString);
break;

To format the FILETIME for insertion in the listbox, we wrap the raw data in the tFileTime object, and use the CTime Format() member to create a printable string.

case CEVT_FILETIME:
sprintf(szInsertString, "%s: %s", "Data Type", "CEVT_FILETIME" );
m_RecordListCtrl_InsertString(-1, szInsertString);
tFileTime = pRecord[i].val.filetime;
ctFileTime = tFileTime.Format("%A, %B %d, %Y");
m_RecordListCtrl_InsertString(-1,
    csFileTime.GetLength());
break;

The CEVT_LPWSTR member represents a slightly different case than the previous types. A pointer to the Unicode string is stored in pRecord[i].val.lpwstr, but the actual string data is stored elsewhere. This isn’t of that much importance if you are reading records. However, if you are writing records, remember that when you allocate memory to hold record data you must allocate a block of memory large enough for the array of CEPROPVAL structures and for the string data to which pRecord[i].val.lpwstr points. (The actual strings are stored in the memory beyond what is occupied by the CEPROPVAL array.)

case CEVT_LPWSTR:
sprintf(szInsertString, "%s: %s", "Data Type", "CEVT_LPWSTR" );
m_RecordListCtrl_InsertString(-1, szInsertString);
wcstombs(szInsertString, pRecord[i].val.lpwstr,
    sizeof(szInsertString));
m_RecordListCtrl_InsertString(-1, szInsertString);
break;

The CEVT_BLOB data type is useful for typical types of binary data, but also as a way to implement application-defined data types and one-to-many record relationships. The CEVALUNION member blob is actually yet another structure, CEBLOB. Here’s the typedef for CEBLOB:

typedef struct _CEBLOB {
    DWORD dwCount;  //size of the BLOB data in bytes
    LPBYTE lpb;       //address of byte stream
} CEBLOB;
We insert the data type, the size in bytes of the binary data, and the first byte of the datum in the list box.

case CEVT_BLOB:
sprintf(szInsertString, "%s: %s",
"Data Type", "CEVT_BLOB" );
m_RecordListCtrl.InsertString( -1, szInsertString );
sprintf(szInsertString, "%s: %li", "Size in bytes",
pRecord[i].val.blob.dwCount );
m_RecordListCtrl.InsertString( -1, szInsertString );
sprintf(szInsertString, "%s: 0x%x", "Buffer Address",
pRecord[i].val.blob.lpb );
break;
default:
    break;
}
//blank line between properties for readability
m_RecordListCtrl.InsertString( -1, "" );

Whew! That’s remote database access. Once you are connected to the remote databases via RAPI, a cut and paste of the sample code shown in the earlier chapter on CE database management will take you most anywhere you want to go. The function names and parameter lists are identical for the RAPI and CE side versions of the following functions:

- CeOpenDatabase
- CeReadRecordProps
- CeWriteRecordProps
- CeDeleteRecord
- CeDeleteDatabase
- CeSeekDatabase

A final reminder: Any string parameters passed into or returned from these functions are in Unicode. The functions will fail if you forget to translate between character formats.

**Looking Ahead**

In the next chapter, we explore the most powerful RAPI capability of all: the ability to remotely invoke a function on the CE device. We’ll pair this sophisticated capability with the Windows CE HTML viewer to create a flexible and dynamic means of communicating with the user.
As initially conceived, palmtop devices were nothing more than special-purpose standalone computers—fun and intriguing, but of limited practical use, precisely because they were inherently and imperviously encapsulated. Windows CE’s RAPI technology dramatically expands the palmtop computing metaphor by integrating seamless connectivity with the larger Windows world. In effect, this allows CE devices to borrow the storage, processing power, and networking infrastructure of the Windows Desktop. These capabilities also transform the CE programming model. Being a good CE application designer is about more than knowing how to write native code for the CE side. We need a larger world view. The big picture is one in which the desktop and the CE device collaborate to produce a result neither could accomplish independently. This is the ultimate promise of CE.

In the preceding chapters in this part, you’ve seen how to query a CE device from the desktop to retrieve status information, access the registry, and manipulate files and databases. These are powerful tools, indeed, because they redefine the very nature of palmtop and handheld computing, giving us flexibility that doesn’t exist in the standalone world. In this last chapter, we equip ourselves with the most powerful of tools for blending the strengths of the desktop with the mobility of the handheld device: remote procedure calls from the desktop to the CE device.
The ability to dynamically invoke functions on the CE side enables a spectrum of possibility in terms of embedded applications and end user oriented applications. We’ll use our examples both to explore how to use RAPI’s RPC mechanism and the ways in which it broadens the scope of handheld and palmtop computing. In this sense, the subject matter of this chapter is as much about programming strategy as about programming technique. Thinking beyond the handiness of having a contacts database or lightweight word processor in your pocket, we must imagine more dynamic and flexible kinds of scenarios. Here are some possibilities for what you might do with remotely invoked functions:

- Awaken device sensors and relay data directly to a network rather than storing them locally.
- Dynamically provide tailored information, including sound and images.
- Unload CE device storage, saving contents on the desktop side in order to free space for dynamically invoked tasks. Replace storage contents when processing is complete.

In this chapter, our example employs a desktop MFC application to transfer a text or HTML file to the handheld, and then allows you to invoke an application on the CE side which displays the Transferred file in an HTML viewer control.

The DesktopRAPIInvoker Example

In the DesktopRAPIInvoker example, we’ll explore two useful technologies: First, we’ll see how to call a CE side function from the desktop using CeRapiInvoke(), and how to create a corresponding DLL on the CE device from which we can invoke a function. Second, we’ll see how to use the Windows CE HTML Viewer control to display HTML or plain text.

There are three critical elements in this process. First, we create a desktop application to initialize RAPI, transfer the files we want to display in the HTML Viewer control, and invoke the remote process. On the CE side, we need a DLL that contains a specific type of function we can invoke from the desktop. Also on the CE side, we need the HTML viewer application that displays the transferred file.

In the interest of brevity, we include only the critical files for each of the components (see Table 13.1 and Listing 13.1). You can find the full source code for each component on the accompanying CD.
Table 13.1  Source Files for DesktopRAPIInvoker example

<table>
<thead>
<tr>
<th>SOURCE FILENAME</th>
<th>HEADER FILENAME</th>
<th>IMPLEMENTS</th>
<th>LISTING INCLUDED IN TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DesktopRapiInvoker.cpp</td>
<td>DesktopRapiInvoker.h</td>
<td>Application object</td>
<td>No</td>
</tr>
<tr>
<td>DesktopRapiInvokerDoc.cpp</td>
<td>DesktopRapiInvokerDoc.h</td>
<td>Document Object</td>
<td>No</td>
</tr>
<tr>
<td>DesktopRapiInvokerView.cpp</td>
<td>DesktopRapiInvokerView.h</td>
<td>View Object</td>
<td>No</td>
</tr>
<tr>
<td>MainFrm.cpp</td>
<td>MainFrm.h</td>
<td>Application’s main frame window</td>
<td>Yes</td>
</tr>
<tr>
<td>StdAfx.cpp</td>
<td>StdAfx.h</td>
<td>Standard application frameworks support</td>
<td>No</td>
</tr>
<tr>
<td>DesktopRapiInvoker.rc</td>
<td>Resource.h</td>
<td>Resources</td>
<td>No</td>
</tr>
</tbody>
</table>
MainFrm.h
// MainFrm.h : interface of the CMainFrame class
//
//-----------------------------------------------------------------------------
/////
#ifdef !defined
( 
AFX_MAINFRM_H__55089750_FB15_11D5_8B88_00A0D2168AEA__INCLUDED_)
#define AFX_MAINFRM_H__55089750_FB15_11D5_8B88_00A0D2168AEA__INCLUDED_
#endif
#pragma once
#endif // _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
class CMainFrame : public CFrameWnd
{

protected: // create from serialization only
    CMainFrame();
    DECLARE_DYNCREATE(CMainFrame)
    // Attributes
public:
    // Operations
public:
    // Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMainFrame)
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    //}}AFX_VIRTUAL
    // Implementation
public:
    virtual ~CMainFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
protected:
    // control bar embedded members
    CStatusBar  m_wndStatusBar;
    CToolBar    m_wndToolBar;
    // Generated message map functions
protected:
    //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnDownloadHtml();
    afx_msg void OnLaunchViewer();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/ {{AFX_INSERT_LOCATION}}

Listing 13.1  Important source files for DesktopRAPIInvoker example.
// Microsoft Visual C++ will insert additional declarations immediately
before the previous line.
#endif // !defined
{
AFX_MAINFRM_H__55089750_FB15_11D5_8B88_00A0D2168AEA__INCLUDED_)
MainFrm.cpp
// MainFrm.cpp : implementation of the CMainFrame class

#include "stdafx.h"
#include "DesktopRapiInvoker.h"
#include "MainFrm.h"
#include <direct.h>
#include <rapi.h>
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////////////////////////////////////////
/////
// CMainFrame
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
ON_WM_CREATE()
ON_COMMAND(IDM_DOWNLOAD_HTML, OnDownloadHtml)
ON_COMMAND(IDM_LAUNCH_VIEWER, OnLaunchViewer)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

static UINT indicators[] =
{
   ID_SEPARATOR,           // status line indicator
   ID_INDICATOR_CAPS,
   ID_INDICATOR_NUM,
   ID_INDICATOR_SCRL,
};

////////////////////////////////////////////////////////////////////////
/////
// CMainFrame construction/destruction
CMainFrame::CMainFrame()
{
   // TODO: add member initialization code here
}

CMainFrame::~CMainFrame()
{
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{

Listing 13.1 Important source files for DesktopRAPIInvoker example. (continues)
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT,
    WS_CHILD | WS_VISIBLE |
    CBRS_TOP | CBRS_GRIPPER |
    CBRS_TOOLTIPS | CBRS_FLYBY |
    CBRS_SIZE_DYNAMIC) ||
    !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
    TRACE0("Failed to create toolbar\n");
    return -1; // fail to create
}
if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators,
    sizeof(indicators)/sizeof(UINT)))
{
    TRACE0("Failed to create status bar\n");
    return -1; // fail to create
}
// TODO: Delete these three lines if you don't want the toolbar to
// be dockable
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
return 0;

bool CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if (!CFrameWnd::PreCreateWindow(cs))
        return FALSE;
    return TRUE;
}

// CMainFrame diagnostics
#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}
void CMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}
#endif // _DEBUG

// CMainFrame message handlers
void CMainFrame::OnDownloadHtml()
{
CString str1;

CFileDialog dlg( TRUE, NULL, "", 
    OFN_HIDEREADONLY , 
    szFilter, this);

str1.LoadString( IDS_DOWNLOAD_DLGCAPTION );
dlg.m_ofn.lpstrTitle = str1;
_chdir("\My Documents");
CString str;
CString strFileName;
CFile fileHTML;
str.LoadString( IDS_DOWNLOAD_PATH );
dlg.m_ofn.lpstrInitialDir = str.GetBuffer(str.GetLength());
if( dlg.DoModal() == IDOK )
{
    strFileName = dlg.GetFileExt();
    strFileName = dlg.GetPathName();
    //open a stream file

    if(!fileHTML.Open(
            strFileName.GetBuffer(strFileName.GetLength()),
            CFile::modeRead | CFile::typeBinary) )
    {
        str1.LoadString( IDS_ERR_FILE_OPEN );
        CWnd::MessageBox( str1, strFileName,
            MB_ICONHAND | MB_OK );
        return;
    }
    strFileName = dlg.GetFileName();
    //open a file on the CE device
    HRESULT hr = CeRapiInit();
    if ( hr != ERROR_SUCCESS )
    {
        return;
    }
    //how big is the desktop file
    DWORD dwFileSize = fileHTML.GetLength();
    //allocate a buffer
    BYTE pbFileBuff = (BYTE) LocalAlloc(LPTR, dwFileSize);
    if ( !pbFileBuff )
    {
        str1.LoadString( IDS_ALLOCATION_FAILURE );
        CWnd::MessageBox( str1, strFileName,
            MB_ICONHAND | MB_OK );
        return;
    }
    //read it
    fileHTML.Read(pbFileBuff, dwFileSize );
    //open a CE side file
    HANDLE hFile = CeCreateFile (L"MyHTMLViewerText",
            (HANDLE) LocalAlloc(LPTR, dwFileSize));
    if ( !hFile )
    {
        str1.LoadString( IDS_ERR_UNKNOWN );
        CWnd::MessageBox( str1, strFileName,
            MB_ICONHAND | MB_OK );
        return;
    }
    //read the CE side file
    DWORD dwFileSizeCE = CeGetFileSize(hFile);
    if ( !dwFileSizeCE )
    {
        str1.LoadString( IDS_ERR_UNKNOWN );
        CWnd::MessageBox( str1, strFileName,
            MB_ICONHAND | MB_OK );
        return;
    }
    //allocate a buffer
    BYTE pbFileBuffCE = (BYTE) LocalAlloc(LPTR, dwFileSizeCE);
    if ( !pbFileBuffCE )
    {
        str1.LoadString( IDS_ALLOCATION_FAILURE );
        CWnd::MessageBox( str1, strFileName,
            MB_ICONHAND | MB_OK );
        return;
    }
    //read it
    CeFileRead(hFile, pbFileBuffCE, dwFileSizeCE );
    hFile = CeCloseFile(hFile);
}
//end if(dlg.DoModal() == IDOK )

Listing 13.1 Important source files for DesktopRAPIInvoker example. (continues)
How the DesktopRAPIInvoker Application Transfers the File

In the CMainFrame member OnDownloadHtml(), we choose a file to transfer to the desktop, initialize the RAPI subsystem, create and open a CE side file, and copy the desktop file to the CE side file.
The first step is to initialize and display the common file dialog to allow the user to choose a file for transfer. Note that this process of file selection is intentionally naïve. We don’t use a filter string to ensure that the file is actually text or HTML, and we don’t check to see if the CE device has adequate space to accept the transferred file before we send it. We create an object of CFileDialog type, passing the following parameters to the constructor. Passing TRUE as the first parameter indicates that we are opening rather than saving the file. Setting the second parameter to NULL has the effect of eliminating the appending of a default file extension to the name specified by the user. The third parameter sets a flag that prevents read-only files from being displayed in the dialog. The fourth parameter, set to NULL, means that we are not supplying a filter string to control the types of files displayed in the dialog; and the final parameter is the handle to a window that can display error messages for the common file dialog if necessary.

```cpp
void CMainFrame::OnDownloadHtml()
{
    CString str1;
    CFileDialog dlg( TRUE, NULL, "",
                    OFN_HIDEREADONLY ,
                    NULL, this);

    CString str1;
    CFileDialog dlg( TRUE, NULL, "",
                    OFN_HIDEREADONLY ,
                    NULL, this);

    str1.LoadString( IDS_DOWNLOAD_DLG_CAPTION );
    dlg.m_ofn.lpstrTitle = str1;
    _chdir("\My Documents");
    CString str;
    CString strFileName;
    CFile fileHTML;
    /* This is another way to set the initial directory for the dialog
       str.LoadString( IDS_DOWNLOAD_PATH );
       dlg.m_ofn.lpstrInitialDir = str.GetBuffer(str.GetLength());
    */

    Next, we invoke the dialog, and if it returns with IDOK, we get the filename the user has chosen. We open the file combining the flags for read only and binary access.
if( dlg.DoModal() == IDOK )
{
    strFileName = dlg.GetFileNameExt();
    strFileName = dlg.GetPathName();
    //open a stream file

    if( !fileHTML.Open(
            strFileName.GetBuffer(strFileName.GetLength()),
            CFile::modeRead | CFile::typeBinary) )
    {
        str1.LoadString( IDS_ERR_FILE_OPEN );
        CWnd::MessageBox( str1, strFileName,
                   MB_ICONHAND | MB_OK );
        return;
    }
} //end if(dlg.DoModal() == IDOK )
else
{
    return;
}

Now we are ready to copy the file to the CE device. First we initialize the RAPI subsystem. We check the size of the file, allocate a buffer for it, and read the entire file into memory on the desktop.

    // open a file on the CE device
    HRESULT hr = CeRapiInit();
    if ( hr != ERROR_SUCCESS )
    {
        return;
    }
    //how big is the desktop file
    DWORD dwFileSize = fileHTML.GetLength();
    //allocate a buffer
    PBYTE pbFileBuff = (PBYTE) LocalAlloc(LPTR, dwFileSize);
    if ( !pbFileBuff )
    {
        str1.LoadString( IDS_ALLOCATION_FAILURE );
        CWnd::MessageBox( str1, strFileName,
                   MB_ICONHAND | MB_OK );
        return;
    }
    //read it
    fileHTML.Read(pbFileBuff, dwFileSize );

Now we open a file on the CE device, in this case using the Unicode string MyHTMLViewerText as the filename. We open for both reading and writing; allow shared reading; pass a NULL placeholder for the unused security attributes parameter; set the CREATE_ALWAYS flag, which creates the file if it doesn’t exist or opens an existing file, clears all attributes, and zeroes the file length, sets default file attributes, and passes a NULL placeholder for the unused template handle parameter.
// open a CE side file
HANDLE hFile = CeCreateFile(L"MyHTMLViewerText",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);

Next, we copy the file data from the buffer on the desktop to the CE side. We
don’t do any error checking in this example, but in practice, you’d want to
check to see that and dwBytesWritten and dwFileSize were equal after the call,
and take some remedial action otherwise.

// copy from the desktop to the CE device
DWORD dwBytesWritten;
CeWriteFile(hFile, (LPCVOID)pbFileBuff,
    dwFileSize, &dwBytesWritten, NULL);

Finally, we close the remote file, close the desktop file, and uninitialize the
RAPI subsystem.

// close both files
CeCloseHandle(hFile);
fileHTML.Close();
// uninit rapi
CeRapiUninit();

}

How the DesktopRAPIInvoker Launches the HTML
Viewer on the CE Device

To examine this step, we are going to skip over what’s taking place on the CE
side for now and concentrate exclusively on the desktop code that invokes the
remote function. (We’ll look at the CE-side DLL and the CE-side HTML viewer
application next.) You can’t invoke just any CE side function from the desktop.
There are two big restrictions. First, the function you invoke must reside in a
DLL on the CE device before you try to call it, and second, the invoked function
must conform to a specific prototype.

The job of invoking the remote function is handled by the RAPI call CeRapi-
Invoke(), which loads a remote DLL, and then jumps to a specific entry point
corresponding to the supplied function name. As usual, first, we initialize the
RAPI subsystem. Next, we call the function CeRapiInvoke().
HRESULT hr = CeRapiInit();
if ( hr != ERROR_SUCCESS )
{
    return;
}
//invoke the LaunchViewer Fxn
DWORD cbOutput;
BYTE pOutput;

The key to using CeRapiInvoke is making sure the parameters are squeaky clean with respect to type and value. Here’s the declaration for CeRapiInvoke():

```c
HRESULT CeRapiInvoke( LPCWSTR pDllPath,
                        //Unicode path to remote DLL
LPCWSTR pFunctionName, //C style Unicode function
                        // name
 DWORD cbInput,         //number bytes input buffer
                        // data
 BYTE *pInput,         //address of input data buffer,
                        //allocated by the caller
 DWORD *pcbOutput,     //address of DWORD to hold
                        // byte count of returned data
 BYTE **ppOutput,      //address of ptr to returned
                        // data
 IRAPIStream **ppIRAPIStream, //address of pointer to
                        //an IRAPIStream interface
 DWORD dwReserved );         //supply a NULL placeholder
```

CeRapiInvoke() operates in one of two modes, depending on the value of the IRAPIStream ** parameter to the function. If this parameter is set equal to NULL, the function returns after all processing is completed on the CE side. This is called synchronous behavior, or block mode. The advantage of block mode is that it’s simple to implement, but it’s also slower and less flexible than the asynchronous stream mode. If the ppIRAPIStream parameter contains a valid pointer to an interface, then CeRapiInvoke() operates in stream mode. In stream mode, the desktop side and the application can communicate during the CE side processing, sending data of arbitrary size back and forth via the RAPI stream interface.

We use block mode here, as the CE application and the desktop aren’t exchanging data.

```c
hr = CeRapiInvoke( L"CeSideRapiInvoke.dll",
                   L"LaunchViewer",
                   0, NULL, &cbOutput,
                   &pOutput, NULL, 0 );
```

The trickiest part of getting this mechanism to work is making sure that the DLL is loaded on the CE side and the function you are trying to invoke is resolved. Two things are likely to throw a spanner in the works at this point. First, if the path or function names aren’t Unicode, they won’t be resolved. Second, if
you are using a combination of C and C++ languages, you have to be certain that
the DLL on the CE side exports C-style function names, rather than C++-style
decorated names. If the CeRapiInvoke() call is failing, checking for the following
errors will help you to isolate the problem.

```c
if (hr == ERROR_FILE_NOT_FOUND )
    {return;}
if (hr == ERROR_CALL_NOT_IMPLEMENTED )
    {return;}
if (hr == ERROR_EXCEPTION_IN_SERVICE )
    {return;}
//uninit rapi
CeRapiUninit();
```

When we are done processing, we uninitialize the RAPI subsystem.
To see exactly what function names the CE-side DLL exports, you can use
the Visual C++ Dependency Walker utility to check the exports from the
library. We’ll see how to do this in the next section.

## Creating the DLL for the Remotely Invoked Function

There are two keys to creating a CE function that may be remotely invoked.
First, the function prototype must conform exactly to the declaration you see
below:

```c
HRESULT (STDAPICALLTYPE RAPIEXT) MyRapiInvokableFxn (  
    DWORD      cbInput,          // [IN]  
    BYTE        *pInput,         // [IN]  
    DWORD      *pcbOutput,       // [OUT]  
    BYTE        **ppOutput,      // [OUT]  
    IRAPIStream  *pIRAPIStream    // [IN]  
);  
```

Second, you must make sure the name exported for the function is a C-style
function name, not a C++-style decorated name. C++ compilers generate arbi-
trary function names that encode things such as parameter list information,
function return types, and class membership. These names are not standard-
ized, so you can’t safely use them to invoke functions in the DLL.

If you use Visual C++ Wizards or other similar development tools to generate
the skeleton files for your DLL, a thicket of macros and definitions are invoked
in your behalf which make it hard to tell just what kind of function names you’ll
end up with. To see exactly what is exported by the DLL, you can use the Visual
C++ Dependency Walker to look at the exported symbols in the DLL.
Figure 13.1 shows the Dependency Walker’s view of the DLL that contains the function we invoke from DesktopRAPIInvoker, LaunchViewer(). The DLL was built using files generated by App Wizard. When the files were initially created, I chose the option that includes an exported dummy variable and an exported dummy function in the generated files. Notice that there are three more-or-less “English-looking” names in the Exports pane: LaunchViewer, our exported function, and fnDRICompanion and nDRICompanion, the automatically generated dummy exports. By contrast, there are two long (ugly) names above, which happen to be the class constructor and destructor for CDRICompanion. If a function you are trying to call from CeRapiInvoke() looks something like the first two names in the Exports pane, you’ve gotten C++-style function naming and the call won’t be resolved.

Now let’s look at the source that implements the DLL DRICompanion (see Table 13.2). Once again, we’ll include only the files that are pertinent to the explanation at hand. To see the code in its entirety, check the accompanying CD.

Table 13.2  Source Files for DRICompanion Example

<table>
<thead>
<tr>
<th>SOURCE FILENAME</th>
<th>HEADER FILENAME</th>
<th>USAGE</th>
<th>LISTING INCLUDED IN TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRICompanion.cpp</td>
<td>DRICompanion.h</td>
<td>Main DLL source files</td>
<td>Yes</td>
</tr>
<tr>
<td>StdAfx.cpp</td>
<td>StdAfx.h</td>
<td>Standard Application Frameworks support</td>
<td>No</td>
</tr>
</tbody>
</table>
DRICompanion.h
// The following ifdef block is the standard way of creating macros
// which make exporting
// from a DLL simpler. All files within this DLL are compiled with the
// DRICOMPANION_EXPORTS symbol defined on the command line. This symbol
// should not be defined on any project that uses this DLL. This way
// any other project
// whose source files include this file see DRICOMPANION_API functions
// as being
// imported from a DLL, where as this DLL sees symbols defined with this
// macro as
// being exported.
#ifdef DRICOMPANION_EXPORTS
#define DRICOMPANION_API __declspec(dllexport)
#else
#define DRICOMPANION_API __declspec(dllimport)
#endif
// This class is exported from the DRICompanion.dll
class DRICOMPANION_API CDRICompanion {
public:
   CDRICompanion(void);
   // TODO: add your methods here.
};
extern DRICOMPANION_API int nDRICompanion;
DRICOMPANION_API int fnDRICompanion(void);
// Not included in a server-side include file
typedef enum tagRAPISTREAMFLAG {
   STREAM_TIMEOUT_READ
} RAPISTREAMFLAG;
DECLARE_INTERFACE_ (IRAPIStream, IStream)
{
   STDMETHOD(SetRapiStat)(THIS_ RAPISTREAMFLAG Flag,
      DWORD dwValue) PURE;
   STDMETHOD(GetRapiStat)(THIS_ RAPISTREAMFLAG Flag,
      DWORD *pdwValue) PURE;
};
// Function prototypes declared as exports from the DLL.
DRICOMPANION_API INT LaunchViewer (DWORD cbInput, BYTE *pInput,
   DWORD *pcbOutput, BYTE **ppOutput,
   IRAPIStream *pIRAPIStream);

Listing 13.2  Important source files for DRICompanion DLL.
Understanding the DLL Header File

The most important job of the header file in this example is to make sure the function we want to expose to CeRapiInvoke() is exported properly. First, we want to make sure it is treated as a C-style symbol, and second, we must be scrupulous about the parameter list and their types. Notice this generated code at the top of the header file:

```c
#ifdef DRICOMPANION_EXPORTS
#define DRICOMPANION_API __declspec(dllexport)
#else
#define DRICOMPANION_API __declspec(dllimport)
#endif
```

These conditionally applied macros do two things. First, they correctly apply the `__declspec` attribute based on whether the header is being included in the compilation of a DLL (which is exporting functions for use by other modules) or a module which is importing the same functions. The fragment `__declspec(dllexport)` explicitly defines the calling convention and interface to the function it modifies. Declaring functions as `dlexport` eliminates the need for a module-definition (.DEF) file, and makes the function available to be called by another application (as in the case at hand) or by another DLL. In the following code, the macro `DRICOMPANION_API` applies the `__declspec(dllexport)` modifier to the function we export for use by CeRapiInvoke().

Recall from the typedef at the beginning of this part that one of the parameters to an invokable function is of type `IRAPIStream**`. In order to properly type this parameter, you must first add a declaration for the interface and its data structures. The following lines provide the necessary declarations:

```c
// Not included in a server-side include file
typedef enum tagRAPISTREAMFLAG {
    STREAM_TIMEOUT_READ
} RAPISTREAMFLAG;
DECLARE_INTERFACE_ (IRAPIStream, IStream)
{
    STDMETHOD(SetRapiStat)(THIS_ RAPISTREAMFLAG Flag,
                           DWORD dwValue) PURE;
    STDMETHOD(GetRapiStat)(THIS_ RAPISTREAMFLAG Flag,
                           DWORD *pdwValue) PURE;
};
```

Finally, and most critically, you must include a prototype that exactly matches the expectations of CeRapiInvoke() and properly exports the function.
Function prototypes declared as exports from the DLL.

```c
DRICOMPANION_API INT LaunchViewer (DWORD cbInput, BYTE *pInput,
    DWORD *pcbOutput,
    BYTE **ppOutput,
    IRAPIStream *pIRAPIStream);
```

The implementing source code for the DLL is surprisingly brief. Our only
real interest here is the function LaunchViewer().

```c
DRICompanion.cpp
// DRICompanion.cpp : Defines the entry point for the DLL application.
//
#include "stdafx.h"
extern "C"
{
    #include "DRICompanion.h"
}
BOOL APIENTRY DllMain( HANDLE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
)
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
// This is an example of an exported variable
DRICOMPANION_API int nDRICompanion=0;
// This is an example of an exported function.
DRICOMPANION_API int fnDRICompanion(void)
{
    return 42;
}
// This is the constructor of a class that has been exported.
// see DRICompanion.h for the class definition
CDRICompanion::CDRICompanion()
{
    return;
}
DRICOMPANION_API int LaunchViewer(DWORD cbInput, BYTE *pInput,
    DWORD *pcbOutput, BYTE **ppOutput,
    IRAPIStream *pIRAPIStream)
{
PROCESS_INFORMATION piHtmlViewer;

CreateProcess( TEXT("MyHtmlViewer.exe"),
    NULL, NULL, NULL, FALSE,
    0, NULL, NULL, NULL,
    &piHtmlViewer );

return 0;
}

Notice that LaunchViewer() includes but a single call: CreateProcess(). To my way of thinking, this bit of logic epitomizes the elegance of CE’s remote procedure call functionality. While there are very strict limitations on how you can invoke a function, the function you invoke is virtually unlimited in its capability because it can in turn create processes. In a nutshell, the whole device is your oyster once you successfully invoke the first function.

Here are the parameters to CreateProcess():

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

Notice that like CeRapiInvoke(), CreateProcess() provides a means by which to pass data to the function being invoked (via a command line). By using these functions in concert, you create a fundamentally unlimited integration of the CE and desktop Windows worlds.

**Using the CE HTML Viewer Control to Create Powerful, Customized Presentations**

The keystone of this example is the application on the CE side, MyHTMLViewer, which hosts the HTML viewer control. The HTML viewer control is a tool that I consider to be a “gestalt” item. It enables a great many application strategies that are really much more than the sum of their parts. Though this application (like the rest of the exclusively CE side code in this book) is written in Win32, it could very easily be written in MFC for Windows CE. The code for implementing the HTMLViewer on the CE side is fairly sparse, so the impact wouldn’t be unconscionable, especially if the application was downloaded from the desktop side on demand and then deleted at the end of the invoked processing. Table 13.3 lists the source files. Listing 13.3 gives the code.
Table 13.3  Source Files for MyHTMLViewer Example

<table>
<thead>
<tr>
<th>SOURCE FILENAME</th>
<th>HEADER FILENAME</th>
<th>USAGE</th>
<th>LISTING INCLUDED IN TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyHtmlViewer.cpp</td>
<td>MyHtmlViewer.h</td>
<td>Implements HTML Viewer and displays downloaded file</td>
<td>Yes</td>
</tr>
<tr>
<td>MyHtmlViewer.rc</td>
<td>Resource.h</td>
<td>Contains Application resources</td>
<td>No</td>
</tr>
<tr>
<td>StdAfx.cpp</td>
<td>StdAfx.h</td>
<td>Standard Application Frameworks Support</td>
<td>No</td>
</tr>
</tbody>
</table>

MyHtmlViewer.h
#if !defined
{
AFX_MYHTMLVIEWER_H__7217EE8A_FB01_11D5_8B87_00A0D2168AEA__INCLUDED_
}
#defineAFX_MYHTMLVIEWER_H__7217EE8A_FB01_11D5_8B87_00A0D2168AEA__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "resource.h"
#endif // !defined
{
AFX_MYHTMLVIEWER_H__7217EE8A_FB01_11D5_8B87_00A0D2168AEA__INCLUDED_
}
MyHtmlViewer.cpp
// MyHtmlViewer.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "MyHtmlViewer.h"
#include <commctrl.h>
#include <htmlctrl.h>
//Forward References
void PopulateHTMLViewer();
#define DISPLAYCLASS TEXT("DISPLAYCLASS")
BOOL g_bMakeFit = TRUE; // DTM_ENABLESHRINK Shrink-enable flag
TCHAR const c_szHTMLControlLibrary[] = TEXT("htmlview.dll");
HINSTANCE g_hInstHTMLCtrl; // HTML Control Viewer instance
HINSTANCE hInst; // Application instance
HWND hwndHtml; // Handle to HTML DISPLAYCLASS window
#define MAX_LOADSTRING 100

Listing 13.3  Important source files for MyHTMLViewer example. (continues)
// Global Variables:
HWND hwndCB; // The command bar handle
// Forward declarations of functions included in this code module:
ATOM MyRegisterClass (HINSTANCE hInstance, LPTSTR szWindowClass);
BOOL InitInstance (HINSTANCE, int);
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About (HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPTSTR lpCmdLine,
                    int nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }
    hAccelTable = LoadAccelerators(hInstance,
                                       (LPCTSTR)IDC_MYHTMLVIEWER);
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        return msg.wParam;
    }
    //
    ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR szWindowClass)
    {
        WNDCLASS wc;
        wc.style = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc = (WNDPROC) WndProc;
        wc.cbClsExtra = 0;
        wc.cbWndExtra = 0;
        wc.hInstance = hInstance;
        wc.hIcon
                        = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_MYHTMLVIEWER));
        wc.hCursor = 0;
        wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
        wc.lpszMenuName = 0;
        wc.lpszClassName = szWindowClass;
        return RegisterClass(&wc);
    }

Listing 13.3  Important source files for MyHTMLViewer example.
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    TCHAR szTitle[MAX_LOADSTRING];
    TCHAR szWindowClass[MAX_LOADSTRING];

    hInst = hInstance;
    // Initialize global strings
    LoadString(hInstance, IDC_MYHTMLVIEWER,
        szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance, szWindowClass);
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
        0, 0,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL);
    if (!hWnd)
    {
        return FALSE;
    }
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    if (hwndCB)
    {
        CommandBar_Show(hwndCB, TRUE);
    }
    return TRUE;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    int wmId, wmEvent;
    RECT rc;
    switch (message)
    {
    case WM_COMMAND:
        wmId = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_HELP_ABOUT:
            DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX,
                hWnd, (DLGPROC)About);
            break;
        case IDM_FILE_EXIT:
            break;
        }
    Listing 13.3  Important source files for MyHTMLViewer example. (continues)
DestroyWindow(hWnd);
break;
default:
    return DefWindowProc(hWnd, message,
                       wParam, lParam);
}
break;
case WM_CREATE:
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    CommandBar_InsertMenuBar(hwndCB, hInst, IDM_MENU, 0);
    CommandBar_AddAdornments(hwndCB, 0, 0);
    g_hInstHTMLCtrl = LoadLibrary(c_szHTMLControlLibrary);
    InitHTMLControl(hInst);
    GetClientRect(hWnd, &rc);
    rc.top += CommandBar_Height(hwndCB);
    hwndHtml = CreateWindow(DISPLAYCLASS, NULL,
                            WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_CLIPSIBLINGS,
                            rc.left, rc.top, rc.right - rc.left,
                            rc.bottom - rc.top,
                            hWnd, (HMENU)IDC_MYHTMLVIEWER, hInst, NULL);
    SetFocus(hwndHtml);
    PostMessage(hwndHtml, DTM_ENABLESHRINK, 0, g_bMakeFit);
    PopulateHTMLViewer();
    break;
case WM_DESTROY:
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;

// Message handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
  RECT rt, rt1;
  int DlgWidth, DlgHeight;       int NewPosX, NewPosY;
  switch (message)
  {
    case WM_INITDIALOG:
      // trying to center the About dialog
      if (GetWindowRect(hDlg, &rt1)) {
        GetClientRect(GetParent(hDlg), &rt);
        DlgWidth   = rt1.right - rt1.left;
        DlgHeight  = rt1.bottom - rt1.top ;
        NewPosX    = (rt.right - rt.left - DlgWidth)/2;
        NewPosY    = (rt.bottom - rt.top - DlgHeight)/2;
Listing 13.3  Important source files for MyHTMLViewer example.
// if the About box is larger than the physical screen
if (NewPosX < 0) NewPosX = 0;
if (NewPosY < 0) NewPosY = 0;
SetWindowPos(hDlg, 0, NewPosX, NewPosY,
            0, 0, SWP_NOZORDER | SWP_NOSIZE);
}
return TRUE;
case WM_COMMAND:
    if ((LOWORD(wParam) == IDOK) ||
        (LOWORD(wParam) == IDCANCEL))
    {
        EndDialog(hDlg, LOWORD(wParam));
        return TRUE;
    }
    break;
return FALSE;
}
void PopulateHTMLViewer()
{
    DWORD dwFileSize, dwHighWord, dwBytesWritten;
    BYTE pbFileBuff;
    HANDLE hFile;
    BOOL bOk;
    //Open the HTML file.
    hFile = CreateFile(TEXT("MyHTMLViewerText"), GENERIC_READ,
                      FILE_SHARE_READ, NULL, OPEN_EXISTING,
                      FILE_ATTRIBUTE_NORMAL, NULL);
    //get the file size
    dwFileSize = GetFileSize(hFile, &dwHighWord);
    //buffer the file
    pbFileBuff = (BYTE)LocalAlloc(LPTR, dwFileSize);
    if(!pbFileBuff)
        { return; }
    //read the file into the buffer
    ReadFile(hFile,(LPVOID)pbFileBuff, dwFileSize,
             &dwBytesWritten, NULL);
    //close the handle
    CloseHandle(hFile);
    //send html formatted text to the control
    bOk = SendMessage(hwndHtml, WM_SETTEXT, 0, (LPARAM)TEXT(""));
    SendMessage(hwndHtml, DTM_ADDTEXTW, 0, (LPARAM)pbFileBuff);
    //send end of source message
    SendMessage(hwndHtml, DTM_ENDOFSOURCE, 0, 0);
    //delete the buffer
    LocalFree(pbFileBuff);
}
We handle two jobs in MyHTMLViewer: We create the control, and then we populate it with data from the file we downloaded using the DesktopRAPI-Invoker application. First, let’s look at creation of the control, which we do in response to the WM_CREATE message.

After we handle the normal CE window creation processing, we explicitly load the library that implements the HTML viewer control. We load using a global variable initialized with the name of the DLL. Here’s the declaration, which appears at the top of the source file:

```c
TCHAR const c_szHTMLControlLibrary[] = TEXT("htmlview.dll");
```

The reason that we didn’t just link this library at compile time is that the HTML Viewer control isn’t part of the core Windows CE system definition, so it may not be present on every platform. Explicit loading allows you to test for success before continuing with initialization processing.

After we load the HTML Viewer DLL, we initialize the control with a call to InitHTMLControl().

```c
case WM_CREATE:
    hwndCB = CommandBar_Create(hInst, hWnd, 1);
    CommandBar_InsertMenuBar(hwndCB, hInst, IDM_MENU, 0);
    CommandBar_AddAdornments(hwndCB, 0, 0);
    g_hInstHTMLCtrl = LoadLibrary(c_szHTMLControlLibrary);
    InitHTMLControl(hInst);
```

With the library loaded and the control initialized, we are ready to create the HTML Viewer control. We do this with a call to CreateWindow(), passing DISPLAYCLASS as the class name parameter. Notice also, that when we create the HTML Viewer window, we set its upper edge just below the bottom of the command bar.

```c
GetClientRect( hWnd, &rc );
rc.top += CommandBar_Height(hwndCB);
hwndHtml = CreateWindow(DISPLAYCLASS, NULL,
    WS_CHILD | WS_VISIBLE |
    WS_VSCROLL |
    WS_CLIPSIBLINGS,
    rc.left, rc.top,
    rc.right - rc.left,
    rc.bottom - rc.top,
    hWnd, (HMENU)IDC_MYHTMLVIEWER,
    hInst, NULL);
```

We give the HTMLViewer control the initial focus with a call to SetFocus(), and we send the control the DTM_ENABLESHRINK. This message toggles the image display state of the control so that it sizes images to allow them to be
displayed in the control at its current size. We follow this initialization with a call to our local function, PopulateHTMLViewer().

```c
SetFocus(hwndHtml);
PostMessage(hwndHtml, DTM_ENABLESHRINK, 0, g_bMakeFit);
PopulateHTMLViewer();
break;
```

We populate the control with text using the simple expedient of opening our downloaded file, clearing the control of preexisting content, and then passing the file contents to the control.

```c
void PopulateHTMLViewer()
{
    DWORD dwFileSize, dwHighWord, dwBytesWritten;
    PBYTE pbFileBuff;
    HANDLE hFile;
    BOOL bOk;
    //Open the HTML file.
    hFile = CreateFile (TEXT("MyHTMLViewerText"), GENERIC_READ ,
                        FILE_SHARE_READ, NULL, OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL, NULL);
    //get the file size
    dwFileSize = GetFileSize(hFile, &dwHighWord );
    //buffer the file
    pbFileBuff = (PBYTE)LocalAlloc( LPTR, dwFileSize );
    if( !pbFileBuff )
    { return; }
    //read the file into the buffer
    ReadFile( hFile,(LPVOID)pbFileBuff, dwFileSize,
              &dwBytesWritten, NULL);
    //close the handle
    CloseHandle( hFile );
    //send html formatted text to the control
    bOk = SendMessage( hwndHtml, WM_SETTEXT, 0, (LPARAM)TEXT("" ) );
    SendMessage( hwndHtml, DTM_ADDTEXTW , 0, (LPARAM)pbFileBuff );
    //send end of source message
    SendMessage( hwndHtml, DTM_ENDOFSOURCE , 0, 0 );
    //delete the buffer
    LocalFree(pbFileBuff);
}
```

Notice that we clear the control before adding the new content by sending the WM_SETTEXT message. This isn’t strictly necessary in this case, but is included in the example as it is one of the few messages that is useful for communicating with the HTML Viewer that is not prefixed “DTM_”. Table 13.4 is a list of HTML Viewer Control messages, along with their meanings and parameter settings.
### Table 13.4  HTML Viewer Control Window Messages and Their Meanings

<table>
<thead>
<tr>
<th>MESSAGE AND MEANING</th>
<th>WPARAM</th>
<th>LPARAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTM_ADDTEXT: Add MBCS text</td>
<td>(WPARAM) (BOOL)</td>
<td>bPlainText; TRUE : Add plain text</td>
</tr>
<tr>
<td>FALSE: Add HTML</td>
<td>(LPARAM) (LPSTR)</td>
<td>szText</td>
</tr>
<tr>
<td>DTM_ADDTEXTW: Add Unicode Text</td>
<td>(WPARAM) (BOOL)</td>
<td>bPlainText; TRUE : Add plain text</td>
</tr>
<tr>
<td>FALSE: Add HTML</td>
<td>(LPARAM)(LPWSTR)</td>
<td>szwText</td>
</tr>
<tr>
<td>DTM_SETIMAGE: Associates a bitmap with an image</td>
<td>0</td>
<td>(LPARAM) (LPIINLINEIMAGEINFO) lpInlineImageInfo</td>
</tr>
<tr>
<td>DTM_ENDOFSOURCE: Signal end of addition of content</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DTM_ANCHOR: Jump to anchor named by MBCS string</td>
<td>0</td>
<td>(LPARAM)(LPSTR)</td>
</tr>
<tr>
<td>DTM_ANCHORW: Jump to anchor named by Unicode string</td>
<td>0</td>
<td>(LPARAM)(LPWSTR)</td>
</tr>
<tr>
<td>DTM_ENABLESHRINK: Toggle “size to image”</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DTM_IMAGEFAIL: Control can’t load image identified by dwCookie</td>
<td>0</td>
<td>(LPARAM)(DWORD)</td>
</tr>
<tr>
<td>DTM_SELECTALL: Select all the text in the current page</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Looking Ahead

You may wonder what we could look ahead to, in the last chapter of a long book like this one. Here’s what I hope: I hope you are looking ahead to the delight and satisfaction of creating new kinds of applications, alive with the spirit and promise of Windows CE’s decentralized style of computing. Go forth, and prosper!
Like almost every other porting job, moving Help files to Windows CE is a matter of casting away all but the absolutely necessary items. Pare help facilities down to the minimum amount of information that allows a user to be functional: how commands work, how to find application related files or resources, and the like. If you want to provide more-complete instructions, you can always use the approach demonstrated in Chapter 13 by dynamically downloading information to the device and displaying it in the HTML viewer when the CE device is connected to a network.

CE help files are created in HTML format, so you can create them in any word processor. In order for the help facility to launch successfully, the help files must have a specific file extension, and this differs among versions of the CE operating system. Table A.1 lists valid file extensions and OS version pairs.

<table>
<thead>
<tr>
<th>WINDOWS CE VERSION NUMBER</th>
<th>HELP FILENAME EXTENSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0 or lower</td>
<td>.htp</td>
</tr>
<tr>
<td>2.01 or better</td>
<td>.htm</td>
</tr>
</tbody>
</table>
Here’s a code fragment that shows how to invoke help from a CE application. In this case, the application is loading a help file that has the base name pft, and it will have an .htp extension for versions 2.0 and lower. If it fails to find this file, it tries to load the same filename with the extension .htm. We could have checked the OS version.

Some systems won’t find help files unless they have the same root name as the application. For example, in the code shown below, the application pft.exe is loading the help file pft.htm or pft.htp. Also, observe the 8.3 file naming convention with help files. Some systems won’t load a file whose base filename is longer than eight characters or includes spaces.

**PORTING TIP**

- Use the appropriate file extension for the version of the CE OS.
- Observe the 8.3 file naming convention.
- If the help file doesn’t load properly, try giving it the same base filename as the application.

The DoHelp() function is called in response to the WM_HELP message. This message is dispatched when the user taps the ? icon on the application’s command bar.

```
//------------------------------------------------------------------
// DoHelp - Display Help File
LRESULT DoHelp (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam)
{
    // if ce 2.0, help extension is htp
    if(!CreateProcess( TEXT("PegHelp.exe"), TEXT("pft.htp"), NULL,
        NULL, FALSE, 0, NULL, NULL, NULL, NULL ))
    {
        // else help extension is htm
        CreateProcess( TEXT("PegHelp.exe"), TEXT("pft.htm"), NULL,
            NULL, FALSE, 0, NULL, NULL, NULL, NULL );
    }
    return 0;
}
```

With the exception of the differences in the file extension, help files are portable across CE devices. To see how various help features are implemented, open a CE help file in a Web browser and look at its source. You can do this using the browser’s View:Source menu command.
It is important to test your application across as many physical devices as possible. While the emulator environments are valuable, ultimately the composition of a given implementation of the CE OS on a device is entirely in the hands of the hardware vendor. To know for sure that your application functions correctly on a given device, it’s safest to test it there.

How to Use the Remote Debugger

If you are targeting a standard CE platform, you can use the Visual C++ Remote Debugger to find problems in your code. For this purpose, you’ll probably want to set up an Ethernet connection between the CE device and the development workstations. Downloads are about 100 times faster with an Ethernet connection, and the connection itself tends to be much more stable and durable.

To make a direct Ethernet connection between a CE device and the workstation, you’ll need an Ethernet card for the CE side and an Ethernet card in the workstation. I have had a very good experience with the Xircom card I use in my CE devices, and I recommend it. It was easy to install and has been tolerant of a lot of traveling, jostling, and general commotion around the lab. Also, you’ll need a crossover cable to connect the two devices.
To use the remote debugger, open the Build menu and set the active configuration to Debug and the processor type of your connected device. When you build, the development environment will detect the Ethernet connection, download the newly built version of the executable, and launch the remote debug process.

How to Find Unresolved Externals When Moving Between Win CE Platforms

As mentioned above, it's quite possible for two CE devices of the same class and version to present slight differences in implementation. Sometimes these differences show up as unresolved external references when you try to move a body of code from one target to another. If the reference is a high-level function name (for example, DoSomethingOrOther() ), that doesn't present a big problem. Just use the text search facility to find the offending reference, and you can begin devising a workaround. However, if the unresolved reference is an instruction, you can't search your own code for it. Instead, you need to find it in a combined source/assembly listing. Here's how to do this:

1. Compile with map file output.
2. Search for the unresolved external name in map file text.
3. This gives you the line number of the non-portable instruction.
4. Change the source view pane to source/assembly by right clicking anywhere in the pane. Select Source/Assembly from the floating menu.
5. Examine all the instructions BELOW the given line number.
6. Change the source line to eliminate unresolved instructions.

Preemptive Strategies

Pretty much as my Mom predicted, the best way to get out of trouble is not to get in it. Here is a list of simple precautions that will spare you hours of debugging:

- Always initialize variables, especially structs that are used as function parameters.
- Assume Unicode sizes for text buffers.
- Don’t rely on character counts for allocations.
- Always test memory allocations and provide handlers for failure.
Test returns from functions that return status.
Always provide handlers for failure to open files, databases, and registry keys.

Using Conditional Compilation to Include Diagnostics

If life was fair, you would be finished debugging after debug builds of your code passed validation. However, since we don’t release debug code, we aren’t really done testing at this point. We have to have some method of validating release builds of code, because, well, that is what we release. This is problematic; it’s harder to find bugs that are masked by the debugger’s protections. We have to invent our own tools and strategies for this.

Here’s my technique for validating release builds. Set up separate targets for debug, release test, and release. As you write the application, make a point of keeping your functions small and tightly focused. If you have bugs that emerge after you switch to release build, you can write diagnostic code that calls your small, single-purpose functions and reports on the application’s state fairly easily. Define a conditional compilation constant “TESTING” in the build settings for your release test configuration like this:

1. Open the Project menu.
2. Choose “Settings.”
3. Open the C/C++ tab of the dialog.
4. Choose “Preprocessor” from the Category dropdown list.
5. Add a comma with no preceding or trailing space to the end of the list of preprocessor definitions, followed by the symbol TESTING.
6. Add items to the release test menu to allow you to manually invoke the diagnostic code. Bracket diagnostic code in the application’s message switch with #ifdef and #endif.

```c
switch( wMenuItem )
{
    #ifdef TESTING
        case IDM_TEST_FILE_ACCESS:
            // check it out
            break;
    #endif
    .
    .
    .
```
Conditional compilation allows you to maintain the release code and the diagnostics in a single code base.

Finally, Table B.1 lists some of the tools provided with the development environment that are useful for debugging.

**Table B.1 Useful Debugging Tools**

<table>
<thead>
<tr>
<th>DEBUGGING AND VALIDATION TOOLS</th>
<th>WHAT TO USE IT FOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote Zoomin</td>
<td>Captures the screen of the CE device on the desktop. Great for screen shots as well as visual debugging of graphics.</td>
</tr>
<tr>
<td>Remote Registry Editor</td>
<td>Runs from the desktop, and manipulates the connected CE device’s registry. Functions like desktop RegEdit, for which there is no CE side analog.</td>
</tr>
<tr>
<td>Remote Object Viewer</td>
<td>Runs from the desktop. Walks the object store on a connected CE device.</td>
</tr>
<tr>
<td>Remote Spy</td>
<td>Runs from the desktop, and functions in the same way as spy. Allows you to see window message traffic, thread activity, and process information.</td>
</tr>
</tbody>
</table>
Most applications underuse the portion of memory devoted to static data. Here is a technique that effectively reclaims some of that dead space, and shrinks message code dramatically at the same time.

I call it “Duncanizing” because legendary programmer Ray Duncan first described it. Basically, it involves three steps. First, you typedef two structures: one each to associate Windows messages and commands with the names of functions that handle them. Next, you populate the structures with application specific data. Finally, you modify your application’s main message handler so that it loops arrays of the structures to dispatch messages to the proper handler.

Normally, you put the typedefs in your application’s header. Here’s the typedef for the message handling structure:

```c
struct handleMsg {
    // Structure associates
    UINT Msg;            // messages
    // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
```
Notice that the first member, Msg, is of the same type as a Windows message constant; the second member, Fxn, is a pointer to a function with a parameter list exactly matching that of the main Windows message handling function. The typedef for the structure that associates Windows commands with handlers is similar, and looks like this:

```c
struct handleCmd {
    UINT Cmd;    // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);  // function.
};
```

The only difference between the two structures is in the parameter list of Fxn, which in this case is matched to the parameters that accompany a menu message.

In the application source file, we use these structures to set up an association between messages or commands and their handlers. Here's the message/handler array:

```c
// Message dispatch table for MainWindowProc
const struct handleMsg MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_COMMAND, DoCommandMain,
    WM_DESTROY, DoDestroyMain,
    WM_HELP, DoHelp,
};
```

Here's the populated array of structures that sets the association between menu commands and handlers:

```c
// Command message dispatch for MainWindowProc
const struct handleCmd MainCommandItems[] = {
    ID_FILE_NEWDATABASE,         DoFileCommandNewDB,
    ID_FILE_OPENEXISTINGDATABASE, DoFileCommandOpenDB,
    ID_FILE_EXPORTDATA,          DoFileCommandExportDB,
    #ifdef TESTING
    ID_RUNAUTOTESTS,             DoDiagCommandRunTests,
    IDM_CLEANUP,                 DoMainCommandClean,
    #endif
};
```

Notice that this command handler array conditionally includes some diagnostic menu items. If you'd like to read more about how to conditionally include diagnostics in release code, check Appendix B.
Here’s how the message processing code looks. The first of these functions, MainWndProc(), processes Windows messages, and the second, DoCommandMain(), processes application-specific command messages.

```c
LRESULT CALLBACK MainWndProc (HWND hWnd,
    UINT wMsg,
    WPARAM wParam,
    LPARAM lParam)
{
    INT i;
    //Loop over the structures till we find the matching message
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
        {
            //call the handler and pass it the parameters that came with
            // this message
            return (*MainMessages[i].Fx)(hWnd, wMsg, wParam, lParam);
        }
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
LRESULT DoCommandMain (HWND hWnd,
    UINT wMsg,
    WPARAM wParam,
    LPARAM lParam) {
    WORD idItem, wNotifyCode;
    HWND hwndCtl;
    INT i;
    // Parse the parameters.
    idItem = (WORD) LOWORD (wParam);
    wNotifyCode = (WORD) HIWORD (wParam);
    hwndCtl = (HWND) lParam;
    // Call routine to handle control message.
    for (i = 0; i < dim(MainCommandItems); i++) {
        if (idItem == MainCommandItems[i].Code)
        {
            //if we find a match, call the handler
            return (*MainCommandItems[i].Fx)(hWnd, idItem, hwndCtl,
                wNotifyCode);
        }
    }
    return 0;
}
```

There are advantages and disadvantages to this approach. The main advantage is that it eliminates the message switch code, which is itself rather bulky. The arrays of function pointers are static data, so they occupy a part of your
application’s data space that would likely be wasted otherwise. Some people believe switch code is also more error prone over time than the Duncanized approach.

I use the Duncanized approach in my own work for these reasons, but I chose not to use it in this book for the following reasons. For many people, switch code is easier to follow because it provides a literal road map of the application: If message X is received, then we fall through the switch to the set of steps which corresponds to message X’s case constant. This is a fairly clear cause-and-effect pairing. Without the switch, you have to be able to visualize the flow of the application. In my experience, this detracts from the clarity of example code. Also, in Duncanized code, debug tracing is more difficult and time consuming. If you don’t know exactly which message is causing problems, you have to sit on a breakpoint in the loop, which can be really exasperating.
SYMBOLES
<> (angle brackets), 135
\ (backslash), 39, 419

A
About... box, 9–11
accelerators, 4, 9–10
access permissions, 241–242, 346
    file systems and, 287–288, 296, 298
    Registry and, 274, 275, 279, 281
AddBands function, 86, 127
AddBirthdayReminder function, 349, 373
AddBitmap function, 31
AddPropPages function, 412, 413
AddRowToList function, 464, 466
adornments, for command bars, 29, 42
    alloc function, 236
AllPages.cpp, 389, 413
AllPages function, 412
AllPages.h, 389, 392–393, 412
angle brackets (<>), 135
animation, 381
ANSI (American National
    Standards Institute), 433
APIs (Application Program
    Interfaces), 9, 188
database, 333–334, 343, 354, 376–377
graphics, 133–134, 136–137, 160, 168
    identify unsupported, 133
memory management and, 234,
    236, 240
See also RAPI (Windows CE Remote
    Application Programming
    Interface)
application object, 389, 409,
    411–412, 414
AppWizard projects, 459–460, 492
array(s)
    CommandBand controls and, 86–87
    declaration of, 61
graphics behaviors and, 144, 167
    initialization of, 87–88, 130
    loop through, 62
    REBARANDINFO structures and, 83
    Registry and, 272
    stylus input and, 179–180
    Windows CE shell and, 201–204, 206
ASCII (American Standard Code for
    Information Interchange), 39,
    309–313, 388
.asm file extension, 136
aspect ratios, 9
Assembly with Source listings, 135–136
atoi function, 331
backgrounds, white, advantages of, 170
backslash (\), 39, 419
band(s)
  add, 86
  handles to, 88
  indices for, 125–126
  initialization of, 127
  loops, 87
  move, between command bars, 125
  resizable, 106–131
  specify the behavior of, 81–82
  See also CommandBand controls
batteries, 322, 379–386
BeginPaint function, 145, 166
BetterBirthdays.cpp, 354–367
BetterBirthdays example, 334, 353–372
Binary Large OBjects (BLOBs), 351
BirthdayDlgProc function, 343, 367–368, 372–373
BirthdayListDlgProc function, 374–375
BirthdayReminder.cpp, 335–343
BirthdayReminder example, 334–354
BitBlt function, 152, 159
bitmap(s)
  button image, 31–34
  color restrictions for, 31, 65
  cursor image, 8–9
  custom, 33–34
  device-dependent (DDBs), 159–160
  device-independent (DIBs), 159–160, 167–168
  icon image, 65, 215
  lists, 32
  memory management and, 246
  raster operations, 152–158
  resource IDs, 32–33
  system, 34
  transparent backgrounds for, 160–167
  troubleshoot problems with, 75
  See also graphics
BLACK_BRUSH object, 169
BLACK_PEN object, 169
BLOBs (Binary Large OBjects), 351
brushes, 168–170
buffer(s)
  allocate, 89
  combo boxes and, 66
  CommandBand controls and, 89, 105
  database management and, 351–352
  debugging and, 508
  file systems and, 298–299, 311, 314
  memory management and, 240–241, 252
  RAPI and, 388, 423, 430–432, 488
  Registry and, 273, 275–276, 278, 280
  size, 204, 388
  stack-based, 127, 299
  string character counts and, 36
  stylus input and, 190
  temporary files and, 240–241
  Windows CE shell and, 204–206
  See also memory
Build Toolbar, 14
button(s)
  add, to command bars, 30–32, 39–42
  images, 31–34, 105
  navigation, for form pages, 105
  rectangles for, 41
  size restrictions for, 31
  troubleshoot problems with, 75
C
C (high-level language)
  C-style function names, 491
  memory management and, 236
  RAPI and, 491
  runtime functions, 236
  use of the backslash in, 419
C++ (high-level language)
  C++-style function names, 491
  compilers, 188, 491
  function names, 492
  RAPI and, 491–492
  use of the backslash in, 419
  calloc function, 236
CAllPages class, 410, 412–413
Cancel button, 64
Cassiopea, 286
CBFRAME class, 189
CBS_AUTOHSCROLL flag, 38, 66
CBS_DROPDOWN flag, 37, 38
CBS_DROPDOWNLIST flag, 37, 38
CBS_LOWERCASE flag, 38
CBS_SIMPLE flag, 37
CBS_UPPERCASE flag, 38
CEBLOB structure, 347, 351–352, 377, 476–477
CeCreateDatabase function, 343–344, 354, 368–369
CEDB_AUTOINCREMENT flag, 345–346, 352, 375, 471
CEDB_SEEK_BEGINNING flag, 372
CEDB_SEEK_CEOID flag, 371
CEDB_SEEK_CURRENT flag, 372
CEDB_SEEK_END flag, 372
CEDB_SEEK_VALUEFIRSTEQUAL flag, 371
CEDB_SEEK_VALUEGREATER flag, 372
CEDB_SEEK_VALUENEXTEQUAL flag, 371
CEDB_SEEK_VALUESMALLER flag, 371
CEDB_SORT_CASEINSENSITIVE flag, 368
CEDB_SORT_DESCENDING flag, 368
CEDB_SORT_UNKNOWNFIRST flag, 368
CEDB_VALIDDBFLAGS flag, 464
CEDB_VALIDMODTIME flag, 464
CEDB_VALIDNAME flag, 464
CEDB_VALIDSORTSPEC flag, 464
CEDB_VALIDTYPE flag, 464
CeDeleteDatabase function, 477
CeDeleteRecord function, 373
CeFindAllFiles function, 420, 422
CE_FIND_DATA structure, 421–422
CeFindFirstDatabase function, 462–463
CeFindNextDatabase function, 463
CeGetStoreInformation function, 416
CeGetSystemInfo function, 417
CeGetSystemPowerStatusEx function, 415
CeGetVersionEx function, 414
CeGlobalMemoryStatus function, 415
capture, 442
defined, 463
open database with, 471
preserve, 467
return, 463
CEOIDINFO structure, 463–464, 466
CeOpenDatabase function, 345, 354, 370, 373, 471, 477
CEOSVERSIONINFO structure, 414
CePapiInit function, 414, 419
CEPROPID structure, 347–348, 351, 368, 370
CEPROPVAL structure, 347–352, 354, 370, 473–476
CERapiInit function, 462
CeRapiInvoke function, 480, 489–491, 494, 496
CeReadRecordProps function, 375–376, 472–474, 477
CeRegEnumKeyEx function, 436
CeRegEnumValue function, 427, 430–432
CeRegQueryInfoKey function, 435–436, 438–439
CeSeekDatabase function, 352–354, 370–372, 471, 477
CE Services connection windows, 284–288
CEVALPROP structure, 349, 371
CEVALUNION structure, 348–352, 373, 376, 474, 476
CEVT_BLOB constant, 347, 475, 476
CEVT_FILETIME constant, 347, 473, 475
CEVT_12 constant, 347, 475
CEVT_U12 constant, 347, 475
CeWriteRecordProps function, 353, 477
CFileDialog class, 487
character(s)
character sets
ASCII, 39, 309–313, 388
See also character sets
See also characters
See also characters
char data type, 313
check box(es)
  basic description of, 35
  behavior, add buttons with, 35–36
  RAPI and, 419, 421
  states, 35
CheckFloatingPointRange function, 330
chicklet keyboard. See SIP (Supplementary Input Panel)
ChooseFilePathDlgProc function, 296, 297
classes. See also MFC (Microsoft Foundation Classes)
  CallPages class, 410, 412–413
  CBFRAME class, 189
  CFileDialog class, 487
  CListCtrl class, 468
  CListCtrl class, 468
  CLView class, 442–443, 459
  CLViewEx class, 459–460, 465
  CMainFrame class, 469, 486
  CPropertySheet class, 410, 412
  CRecordListDialog class, 469–474
  CRemoteDBScanView class, 442, 460, 470
  CSystemStatusPage class, 413–414
  CTreeCtrl class, 423, 429, 436
  CWalkReg class, 423, 425, 427, 431, 436–439
  CWnd class, 470
Clipboard, 51
CListCtrl class, 468
CLView class, 442–443, 459
CLViewEx class, 459–460, 465
Close button, inclusion of, on menu bars, 29
CloseHandle function, 473
CMainFrame class, 469, 486
CMDBAR_HELP flag, 29
CMDBAR_OK flag, 29
code listings
  AllPages.cpp, 389, 413
  BetterBirthdays.cpp, 354–367
  BirthdayReminder.cpp, 335–343
  DesktopRapiInvoker.cpp, 481
  DesktopRapiInvokerDoc.cpp, 481
  DesktopRapiInvokerView.cpp, 481
  Dinner.cpp, 153–160
  DRICompanion.cpp, 492
  EtchASketch.cpp, 137–143
  Etch2.cpp, 146–151
  FileOpen.cpp, 288–301
  FindDirs.cpp, 194–204
  FloatingPoint.cpp, 323–330
  FullScreenDlg.cpp, 218–225
  GreatWriting.cpp, 301–314
  ListViewEx.cpp, 443, 460
  MainFrame.cpp, 389
  MainFrm.cpp, 442, 444–447, 481, 483–486
  MemDlg.cpp, 248–252
  MemMapFiles.cpp, 315–322
  MyHTMLViewer.cpp, 497
  RapiDemo.cpp, 389
  RapiDemoDoc.cpp, 389
  RapiDemoView.cpp, 389
  RecordListDialog.cpp, 443
  RegDemo.cpp, 260–269
  RemoteDBScan.cpp, 442
  RemoteDBScanDoc.cpp, 442
  RemoteDBScanView.cpp, 442
  RemoteFileAccessPage.cpp, 389
  stdafx.cpp, 389, 442, 481, 492, 497
  StylusTrack.cpp, 173–178
  SysMenuGadgets.cpp, 207–213
  SystemStatusPage.cpp, 389, 413–414, 417
  TabbedDlg.cpp, 51–60
  WalkReg.cpp, 389
See also header files
color
  buttons and, 31, 32, 65
  CommandBand controls and, 83
  conservative use of, 31
cursors and, 9
depth, 8, 159, 168
grayscale, 31, 65, 159
icons and, 8
supported by Windows CE, 159–160, 168
tables, 168
See also color palettes
color palettes
  basic description of, 168
default, 168
  grayscale, 31, 65, 159
combo box(es)
   add, 36–39
   CommandBand controls and, 86, 89–90
   style flags, 37–38
   troubleshoot problems with, 76
   width calculations for, 37
CommandBand control(s)
   add, 86, 89–90
   style flags, 37–38
   troubleshoot problems with, 76
   width calculations for, 37
   CommandBand control(s)
   add buttons to, 105
   create, 68–69, 82
   defined, 68–69
   examples, 68–80, 90–106
   gridded appearance of, 83
   gripper controls for, 85
   images and, 83
   long forms and, 106
   moveable, resizable bands for, 106–125
   REBARBANDINFO structure and,
   68–69, 81–86
   restore information for, 127
   retrieve input from controls in, 89
   strategies for, 87
   See also specific controls
   Control Panel, 202
   Copy command, 51
   copyright information, 11
   CPropertySheet class, 410, 412
   CPUs (central processing units),
   12–14, 381, 382
   CREATE_ALWAYS flag, 488, 300, 312
   CreateCursor function, 9
   CreateDataBands function, 81–82,
   104, 129
   CreateDC function, 152
   CreateDIBSection function, 136
   CreateFileForMapping function, 321
   CreateFile function, 296, 298, 300, 311
   Create function, 82, 438
   CREATE_NEW flag, 300
   CreateProcess function, 496
   CreateWindow function, 186, 187, 502
   CRecordListDialog class, 469–474
   CRemoteDBScanView class,
   442, 460, 470
   CSIDL_BITBUCKET constant
   identifier, 202
   CSIDL_CONTROLS constant
   identifier, 202
   CSIDL_DESKTOP constant
   identifier, 202
   CSIDL_DESKTOPDIRECTORY
   constant identifier, 202
   CSIDL_DRIVES constant
   identifier, 202
   CSIDL_FAVORITES constant
   identifier, 202
   CSIDL_FONTS constant identifier, 202
   CSIDL_NETHOOD constant
   identifier, 202
   CSIDL_NETWORK constant
   identifier, 202
   CSIDL_PERSONAL constant
   identifier, 203
organize, into sets of pages, 90–106
register, 27
retrieve input from, 89
size of, 37, 50
style flags, 27
width of, 66
See also specific controls
CSIDL_PRINTERS constant identifier, 203
CSIDL_PROGRAMS constant identifier, 203
CSIDL_RECENT constant identifier, 203
CSIDL_SENDTO constant identifier, 203
CSIDL_STARTMENU constant identifier, 203
CSIDL_STARTUP constant identifier, 203
CSIDL_TEMPLATES constant identifier, 203
CSystemStatusPage class, 413–414
CTreeCtrl class, 423, 429, 436
current point, concept of, 144
cursors, 8–9
CWalkReg class, 423, 425, 427, 431, 436–439
CWnd class, 470
cxIdeal mask, 84, 86
cx mask, 84, 86

D
data
compression, 283
portability, 283–332
validation, 64, 65, 69, 84, 507–510
DataBands.rc, 69–80
database(s)
add attribute rows to, 465–467
add records to, 346–349
BetterBirthdays example and, 334, 353–372
BirthdayReminder example and, 334–354
CEPROPVAL array for, 349–352
create, 343–346
delete, 369
flexible access to, 354–372
functionality, integration of, 334–349
lists of, 442, 460–465
overview of, 333–377
Registry as a, 259
remote access to, 441–477
sort order and, 344–345, 367–373, 471
views, 459–460
See also records
DataDlg dialog, 42–50
DataDlg.rc, 49–50
data types
combo boxes and, 38, 66
database management and, 347–348
RAPI and, 431–433, 474–475
Registry and, 273–274, 277, 431–433
See also data types (listed by name)
data types (listed by name)
LPARAM data type, 178–179, 190, 191, 308
REG_BINARY data type, 274, 432
REG_DWORD_BIG_ENDIAN data type, 274, 432
REG_DWORD data type, 274, 277, 432
REG_DWORD_LITTLE_ENDIAN data type, 274, 432
REG_EXPAND_SZ data type, 432
REG_MULTI_SZ data type, 274, 432
REG_NONE data type, 432
REG_RESOURCE_LIST data type, 432
REG_SZ data type, 274, 432
WCHAR data type, 38, 66, 431
See also data types; TCHAR data type
debugging
conditional compilation and, 509–510, 512
database management and, 353
Duncanized approach and, 511–516
preemptive strategies for, 408
tools, 507–510
unresolved externals and, 508
See also errors
DEFAULT_PALETTE object, 169
DeleteFile function, 214
desktop
connect to Windows CE from, 387–439
shortcuts, 202, 206, 214
Windows CE shell and, 202, 214
desktop folder, 202
DesktopRapiInvoker.cpp, 481
DesktopRapiInvokerDoc.cpp, 481
DesktopRapiInvokerDoc.h, 481
DesktopRAPIInvoker example, 480–504
file transfers and, 486–489
launch HTML Viewer with, 489–491
See also data types; TCHAR data type

Dial, Dwight, 67

dialog(s)
controls, subdivide into groups, 42–43
definition of, in resource files, 4, 42–48
edit control, 301–314
for file transfers, 487–489
handles to, 88
initialization of, 437–439, 469–473
memory management and, 247–255
modeless, 81
RAPI and, 387–389, 411, 437–439
screen size and, 42, 48, 50, 65, 67
templates, 11–12, 51, 60–62, 411
Windows CE shell and, 217–228
DIBs (device-independent bitmaps), 159–160, 167–168
dim macro, 272, 273
Dinner.cpp, 153–160
Dinner example, 152–158
directories
paths to, 134–135, 194–206
root, 285–286
structure of, 284–301
top-level, 286
See also file systems; folders
DLLs (Dynamic Link Libraries), 32, 235
create, for remotely-invoked functions, 491–493
exported symbols in, 491–493
header files for, 494–496
HTML Viewer and, 480, 489–493, 502
load remote, 489–491
RAPI and, 480, 489–491
DoCommandMain function, 513
documents, adding to the MRU, 216–217
DoFileOpen function, 296, 298
DoHelp function, 506
DoModal function, 412, 487
DoOpenFile function, 300
download speed, 507
DRICOMPANION_API macro, 494–495
DRICompanion.cpp, 492

DROPINvoker.h, 481
DROPINvoker.rc, 481
DROPINvokerView.cpp, 481
DROPINvokerView.h, 481
Device Emulator, 180
diagnostics, 509–510, 512
Dial, Dwight, 67
dialog(s)
controls, subdivide, into groups, 42–43
definition of, in resource files, 4, 42–48
edit control, 301–314
for file transfers, 487–489
handles to, 88
initialization of, 437–439, 469–473
memory management and, 247–255
modeless, 81
RAPI and, 387–389, 411, 437–439
screen size and, 42, 48, 50, 65, 67
templates, 11–12, 51, 60–62, 411
Windows CE shell and, 217–228
DIBs (device-independent bitmaps), 159–160, 167–168
dim macro, 272, 273
Dinner.cpp, 153–160
Dinner example, 152–158
directories
paths to, 134–135, 194–206
root, 285–286
structure of, 284–301
top-level, 286
See also file systems; folders
DLLs (Dynamic Link Libraries), 32, 235
create, for remotely-invoked functions, 491–493
exported symbols in, 491–493
header files for, 494–496
HTML Viewer and, 480, 489–493, 502
load remote, 489–491
RAPI and, 480, 489–491
DoCommandMain function, 513
documents, adding to the MRU, 216–217
DoFileOpen function, 296, 298
DoHelp function, 506
DoModal function, 412, 487
DoOpenFile function, 300
download speed, 507
DRICOMPANION_API macro, 494–495
DRICompanion.cpp, 492
ellipse, 172, 179
EndDialog function, 48
EndPaint function, 146, 166
ERROR_DISK_FULL, 345
ERROR_DUP_NAME, 345, 369
ERROR_INVALID_PARAMETER, 353
errors
  database management and, 344–345, 353–354, 369
  RAPI and, 353
  status, test for, 353–354, 369
See also debugging
ERROR_SUCCESS, 345, 431
escape characters, 39
EtchASketch.cpp, 137–143
EtchASketch example, 137–152, 159
Etch2.cpp, 146–151
Etch2 example, 146–152
Ethernet, 388, 507
evolution, of applications, 193
tables (listed by name)
  BetterBirthdays, 334, 353–372
  BirthdayReminder, 334–354
  DesktopRAPIInvoker, 480–504
  Dinner, 152–158
  DuckStamps, 160–167
  EtchASketch, 137–152, 159
  Etch2, 146–152
  ExeFileMem, 242–243
  FidDirs, 194–201
  FileOpen, 194, 287–301, 311
  FindDirs, 214
  Floating Point, 322–332
  FullScreenDlg, 217–228
  GreatWriting, 301–314
  Hello World, 13–14, 133–135, 226–228, 242
  MemMapFiles, 314–322
  Memory Dialog, 247–255
  MenuBar, 15–42, 68, 84
  PagedBands, 90–106
  PalmtoInk, 180–186
  RapiDemo, 388–410
  RegDemo, 260–269, 275
  RemoteDBScan, 441–477
  StylusTrack, 172–180
  SysMenuGadgets, 206–214
  TabbedDlg, 50–60
  Unresolved, 133–136
  ExeFileMem example, 242–243
  Exit command, 11
  Explorer, 194, 283, 284

F
  FAF_ATTRIB_CHILDREN flag, 421
  FAF_ATTRIB_NO_HIDDEN flag, 421
  FAF_ATTRIBUTES flag, 421
  FAF_CREATION_TIME flag, 421
  FAF_FOLDERS_ONLY flag, 421
  FAF_LASTACCESS_TIME flag, 421
  FAF_LASTWRITE_TIME flag, 421
  FAF_NAME flag, 421
  FAF_NO_HIDDEN_SYS_ROMMODULES flag, 421
  FAF_OID flag, 421
  FAF_SIZE_HIGH flag, 421
  FAF_SIZE_LOW flag, 421
field(s)
  CommandBand controls and, 69, 86–87, 127
  database management and, 347–351, 368
  include all, simultaneously, 68
  initialization information for, 106
  long forms and, 107, 108
  use of the term, 347
See also properties
file(s)
  access, 283–332
  adding, to the MRU, 216–217
  compression, 283
  find, 283–301
  help, 10–12, 50, 505–506
  management, 283–332
  memory-mapped, 314–322
  open, 283–301
  overwrite, 299
  paths, 134–135
  read, 301–314
  size, 309, 310
  write, 301–314
See also filenames; file systems
FILE_ATTRIBUTE_ARCHIVE
  attribute, 300
FILE_ATTRIBUTE_HIDDEN attribute, 300
FILE_ATTRIBUTE_NORMAL attribute, 300
FILE_ATTRIBUTE_READONLY attribute, 300
FILE_ATTRIBUTE_SYSTEM attribute, 300
File menu
Exit command, 11
New command, 13
Print Setup command, 11
filename(s)
buffers, 298–299
default save, 311
extensions, 136, 299, 311, 505–506
format for, 301
Unicode, 42, 89
See also files
FileOpen.cpp, 288–301
FileOpen example, 194, 287–301, 311
File Save dialog, 311–312
FILE_SHARE_WRITE attribute, 312
file system(s)
character sets and, 284, 301, 309–314
find files in, 283–301
"flattened," in Windows CE, 194–201
memory-mapped files and, 314–322
numeric data portability and, 322–332
open files in, 283–301
read files in, 301–314
structure of, 284–301
Unicode and, 284, 301, 309–314, 332
Windows CE shell and, 194–201
write files in, 301–314
See also files
FILETIME structure, 347, 373, 376, 475–476
FindBirthday function, 369–370, 372–373
Find button, 370
FindDirs.cpp, 194–204
FindDirs example, 194–201, 214
flags (listed by name)
CBS_AUTOHSCROLL flag, 38, 66
CBS_DROPDOWN flag, 37, 38
CBS_DROPDOWNLIST flag, 37, 38
CBS_LOWERCASE flag, 38
CBS_SIMPLE flag, 37
CBS_UPPERCASE flag, 38
CEDB_AUTOINCREMENT flag, 345–346, 352, 375, 471
CEDB_SEEK_BEGINNING flag, 372
CEDB_SEEK_CEOID flag, 371
CEDB_SEEK_CURRENT flag, 372
CEDB_SEEK_END flag, 372
CEDB_SEEK_VALUEFIRSTEQUAL flag, 371
CEDB_SEEK_VALUEGREATER flag, 372
CEDB_SEEK_VALUENEXTEQUAL flag, 371
CEDB_SORT_CASEINSSENSITIVE flag, 368
CEDB_SORT_DESCENDING flag, 368
CEDB_SORT_UNKNOWNFIRST flag, 368
CEDB_VALIDDBFLAGS flag, 464
CEDB_VALIDMODTIME flag, 464
CEDB_VALIDNAME flag, 464
CEDB_VALIDSORTSPEC flag, 464
CEDB_VALIDTYPE flag, 464
CMDBAR_HELP flag, 29
CMDBAR_OK flag, 29
CREATE_ALWAYS flag, 488, 300, 312
CREATE_NEW flag, 300
FAF_ATTRIB_CHILDREN flag, 421
FAF_ATTRIB_NO_HIDDEN flag, 421
FAF_ATTRIBUTES flag, 421
FAF_CREATION_TIME flag, 421
FAF_FOLDERS_ONLY flag, 421
FAF_LASTACCESS_TIME flag, 421
FAF_LASTWRITE_TIME flag, 421
FAF_NAME flag, 421
FAF_NO_HIDDEN_SYS_ROMMODULES flag, 421
FAF_OID flag, 421
FAF_SIZE_HIGH flag, 421
FAF_SIZE_LOW flag, 421
ICC_ANIMATE_CLASS flag, 27
ICC_BAR_CLASSES flag, 27
ICC_COOL_CLASSES flag, 27
ICC_DATE_CLASSES flag, 27
ICC_HOTKEY_CLASS flag, 27
flags (continued)
ICC_INTERNET_CLASSES flag, 27
ICC_LISTVIEW_CLASSES flag, 27
ICC_PROGRESS_CLASSES flag, 27
ICC_TAB_CLASSES flag, 27
ICC_TREEVIEW_CLASSES flag, 27
ICC_UPDATE_CLASSES flag, 27
ICC_USEREX_CLASSES flag, 27
LMEM_FIXED flag, 240
LMEM_ZEROINIT flag, 240
L PTR flag, 83, 240
LVCF_FMT flag, 461
LVCF_IMAGE flag, 461
LVCFMT_CENTER flag, 462
LVCFMT_LEFT flag, 462
LVCFMT_RIGHT flag, 462
LVCF_ORDER flag, 461
LVCF_SUBITEM flag, 461
LVCF_TEXT flag, 461
LVCF_WIDTH flag, 461
LVIF_IMAGE flag, 465
LVIF_INDENT flag, 465
LVIF_PARAM flag, 465
LVIF_STATE flag, 465
LVIF_TEXT flag, 465, 468
LVIS_ACTIVATING flag, 466
LVIS_DROPHILITED flag, 466
LVIS_FOCUSED flag, 466
LVIS_SELECTED flag, 466
MEM_COMMIT flag, 237
MEM_RESERVE flag, 237
NIM_ADD flag, 216
NIM_DELETE flag, 216
OPEN_ALWAYS flag, 300
OPEN_EXISTING flag, 300
RBBS_BREAK flag, 85
RBBS_NOGRIPPER flag, 85
RBS_AUTOSIZE flag, 83
RBS_BANDBORDERS flag, 83
RBS_FIXEDORDER flag, 83, 125
SHFS_HIDESIBUTTON flag, 227
SHFS_HIDESTARTICON flag, 227
SHFS_HIDETASKBAR flag, 227
SHFS_SHOWSIBUTTON flag, 227
SHFS_SHOWSTARTICON flag, 227
TRUNCATE_EXISTING flag, 300
TVI_FIRST flag, 425
TVI_LAST flag, 425
TVI_SORT flag, 425
floating menus, 12, 39, 42
FloatingPoint.cpp, 323–330
Floating Point example, 322–332
fmask flag, 84, 86
focus
  maintenance of, 64–65
transfer of, 64
folders
  special, find paths to, 194–206
  virtual, 202
Windows CE shell and, 194–206
  See also directories
fonts, 168–170. See also text
footprints, of applications, 10, 151
Format function, 476
forms
  long, 90–106
  screen size limitations and, 67
  single, visual metaphor of, 68
  strategies for, 86
  See also specific components
fStyle mask, 84
FullScreenDlg.cpp, 218–225
FullScreenDlg example, 217–228
functions
  AddBands function, 86, 127
  AddBirthdayReminder function, 349, 373
  AddBitmap function, 31
  AddPropPages function, 412, 413
  AddRowToList function, 464, 466
  alloc function, 236
  AllPages function, 412
  atoi function, 331
  BeginPaint function, 145, 166
  BirthdayDlgProc function, 343, 367–368, 372–373
  BirthdayListDlgProc function, 374–375
  BitBlt function, 152, 159
  calloc function, 236
  CeCreateDatabase function, 343–344, 354, 368–369
  CeDeleteDatabase function, 477
  CeDeleteRecord function, 373
  CeFindAllFiles function, 420, 422
  CeFindFirstDatabase function, 462–463
CeFindNextDatabase function, 463
CeGetStoreInformation function, 416
CeGetSystemInfo function, 417
CeGetSystemPowerStatusEx function, 415
CeGetVersionEx function, 414
CeGlobalMemoryStatus function, 415
CeOpenDatabase function, 345, 354, 370, 373, 471, 477
CePapiInit function, 414, 419
CERapiInit function, 462
CeRapiInvoke function, 480, 489–491, 494, 496
CeReadRecordProps function, 375–376, 472–474, 477
CeRegEnumKeyEx function, 436
CeRegEnumValue function, 427, 430–432
CeRegQueryInfoKey function, 435–436, 438–439
CeSeekDatabase function, 352–354, 370–372, 471, 477
CeWriteRecordProps function, 353, 354, 477
CheckFloatingPointRange function, 330
ChooseFilePathDlgProc function, 296, 297
CloseHandle function, 473
CreateCursor function, 9
CreateDataBands function, 81–82, 105, 129
CreateDC function, 152
CreateDIBSection function, 136
CreateFileForMapping function, 321
CreateFile function, 296, 298, 300, 311
Create function, 82, 438
CreateProcess function, 496
CreateWindow function, 186, 187, 502
DeleteFile function, 214
DoCommandMain function, 513
DoFileOpen function, 296, 298
DoHelp function, 506
DoModal function, 412, 487
DoOpenFile function, 300
_ecvt function, 330–331
EndDialog function, 48
EndPaint function, 146, 166
FindBirthday function, 369–370, 372–373
Format function, 476
GetCommandBar function, 88
GetDC function, 146, 152
GetDeviceCaps function, 159–160
GetDlgItemText function, 88
GetDlgItemText function, 89
GetFilesize function, 309, 310
GetFirstSelectedltemPosition function, 468
GetInspectorMenu function, 188–189
GetInspectorCommandBar function, 189
GetItem function, 468
GetLastError function, 344–345, 353, 354
GetStockObject function, 169
GetSystemInfo function, 253, 321
GetSystemMetrics function, 37, 66
GlobalMemoryStatus function, 253, 255
GetWritersDlgProc function, 308–309, 312
HeapAlloc function, 236, 239
HeapCreate function, 239, 253
HeapDestroy function, 239
HeapFree function, 239
HeapReAlloc function, 239
HideRecord function, 128
HkeyLocalSubKeysDlg function, 271, 272
InitCommonControlsEx function, 27
InitHTMLControl function, 502
InitInstance function, 28, 144, 186, 409
InsertColumn function, 462
InsertComboBox function, 37, 66
InsertItem function, 465, 466
InsertTreeItem function, 425–426, 436, 439
InterpretKeyValue function, 427, 431
InvalidateRect function, 137
LaunchViewer function, 492, 495–496
lineTo function, 137
LoadCursorFromFile function, 9
LoadCursor function, 9
LoadIcon function, 215
LoadImage function, 215
LoadMenu function, 42
functions (continued)
LoadString function, 243–244, 252, 297
LocalAlloc function, 83, 236, 239–240,
275, 299, 311
LocalFree function, 239–240
LocalReAlloc function, 239–240
LocalSize function, 275, 351–352
MainWndProc function, 513
malloc function, 236
MapViewOfFile function, 321–322
MapWindowPoints function, 41, 146
MatchClass function, 189
mbstowcs function, 66, 313, 388
memset function, 27, 61, 215, 425
MemStatusDlgProc function, 253
MessageBox function, 11–12, 296
MoveToEx function, 137
MoveWindow function, 228
MsDlgProc function, 320
MyPaint function, 159, 166
MyRegisterClass function, 28
OnDownloadHtml function, 486
OnInitDialog function, 414, 419, 473
OnInitialUpdate function, 460, 464
OnItemexpandingRegTree function, 427–428
OnSelChangedRegTree function, 430
OnUpdateFileButton function, 419
OpenSaveFile function, 310, 311
Polygon function, 152
Polyline function, 144, 146, 152
PopulateHTMLViewer function, 503
PropertySheet function, 60
PublishRecord function, 472, 474
RapiUninit function, 437
ReadFile function, 314
RegCreateKeyEx function, 269–270
RegCreateKey function, 269–270
RegDeleteKey function, 280
RegEnumKeyEx function, 272
RegEnumValue function, 275–276
RegOpenKeyEx function, 269–271
RegQueryValueEx function, 278
RegSetValueEx function, 274
ReleaseDC function, 146, 152
ResetControl function, 419
RoundRect function, 169
SaveDC function, 152
SendMessage function, 41, 126,
190, 191
SetCapture function, 178
SetFocus function, 502
SetImageList function, 438
SetItem function, 436
SetItemImage function, 429
SetTimer function, 382
SetWindowLong function, 128, 188
SetWindowText function, 470–471
SetWindowWord function, 188
SHAddToRecentDocs function, 216
SHCreateShortcut function, 213, 214
Shell_NotifyIcon function, 216
SHFullScreen function, 227–228
SHGetMalloc function, 205
SHGetPathFromIDList function,
204, 206
SHGetSpecialFolderLocation function, 202–204, 206
SHHandleWMSettingChange function, 225
SHInitDialog function, 227
SHLoadDIBitmap function, 168
ShowRecDlgProc function, 126
sizeof function, 27, 36, 274, 388
SpecialDirsDlgProc function, 201, 204
sprintf function, 38, 414, 415, 467
_stprintf function, 255
strcpy function, 332, 388
StretchBlt function, 152, 159
SysMenuGadgetsDlgProc function, 213
SystemParameterInfo function, 85
TrackPopupMenuEx function, 41–42
TransparentImage function, 160, 167
UpdateWindow function, 145
VirtualAlloc function, 236–238,
245–246, 253
VirtualFree function, 237, 238
WalkRegTree function, 434, 439
wcsat function, 313
wcschr function, 313
wcsncmp function, 313
wcsncpy function, 311
wcspsn function, 313
wcsncpy function, 313–314, 388
wcsdup function, 314
wcsicmp function, 314
wcslen function, 313
wcslen function, 313
wcsncmp function, 313
wcsnicmp function, 314
wcsupper function, 314
WndProc function, 28, 105, 128, 277, 279–280, 296, 308
WriteFile function, 312, 314
wsprintf function, 38, 276–277, 376

G
GetCommandBar function, 88
GetDC function, 146, 152
GetDeviceCaps function, 159–160
GetDlgItem function, 88
GetDlgItemText function, 89
GetFileSize function, 309, 310
GetFirstSelectedItemPosition function, 468
GetInkCtrlMenu function, 188–189
GetItem function, 468
GetLastError function, 344–345, 353, 354
GetStockObject function, 169
GetSystemInfo function, 253, 321
GetSystemMetrics function, 37, 66
globalCEOID parameter, 345–346
GlobalMemoryStatus function, 253, 255
globalRecOid variable, 372
graphics
APIs, 133–134, 136–137, 160, 168
CommandBand controls and, 83
for cursors, 8–9
Dinner example and, 152–158
EtchASketch example and, 137–152, 159
Etch2 example and, 146–152
lines, 137–152, 159, 179
memory management and, 246
overview of, 133–170
photographic-quality, 167
rectangles, 179
size specifications for, 37
usability/readability of, 170
See also bitmaps; graphics behaviors; icons
graphics behaviors
choose subset of, 133
copy code for, 133
linker and, 133–136
port, 133–136
test, 133, 134
See also graphics
GRAY_BRUSH object, 169
gray scale palette, 31, 65, 159
GreatWritersDlgProc function, 308–309, 312
GreatWriting.cpp, 301–314
GreatWriting example, 301–314
gripper controls, 85
GWL_USERDATA constant, 128
GWL_WNDPROC constant, 188

H
Handheld PC devices. See HPC (Handheld PC) devices
header files
AllPages.h, 389, 392–393, 412
DesktopRapiInvokerDoc.h, 481
DesktopRapiInvoker.h, 481
DesktopRapiInvokerView.h, 481
DRICompanion.h, 492–493
ListViewEx.h, 443, 460
MainFrame.h, 389
MainFrm.h, 442, 444, 481, 482–483
MyHtmlViewer.h, 497–502
prsht.h, 60–61
RapiDemoDoc.h, 389
RapiDemo.h, 389–393
RapiDemoView.h, 389
RecordListDialog.h, 443, 454–459
RemoteDBScanDoc.h, 442
RemoteDBScan.h, 442
RemoteDBScanView.h, 442, 447–454, 459–460
Resource.h, 43–48, 442, 481, 497
stdafx.h, 389, 442, 481, 492, 497
SystemStatusPage.h, 389, 398–402
header files (continued)
Unresolved.h, 135
WalkReg.h, 389, 402–409
Windows.h, 190
winuser.h, 63–64
heap
file systems and, 298
fragmentation, 239
local, 236, 238–240, 298–299, 311
memory management and, 233,
  235, 238–240
private, 238–240
string tables and, 10
See also memory
HeapAlloc function, 236, 239
HeapCreate function, 239, 253
HeapDestroy function, 239
HeapFree function, 239
HeapReAlloc function, 239
Hello World example, 13–14,
  133–135, 226–228, 242
Help button, 29, 64
help files, 10–12, 50, 505–506
Help menu, 11–12
hibernation, 245–247
HideRecord function, 128
HIWORD macro, 348, 474
HKEY_CLASSES_ROOT key, 258, 270
HKEY_CURRENT_USER key,
  258, 270, 438
HKEY_LOCAL_MACHINE key,
  258, 270–271
HkeyLocalSubKeysDlg function,
  271, 272
HKEY_USERS key, 270
HOLLOW_BRUSH object, 169
HPC (Handheld PC) devices
  CommandBand controls and, 68
  file systems and, 283–285, 287, 309
  long forms and, 90
  options for, in the WCE
    Configuration bar, 14
  RAPI and, 441
  stylus input and, 172–173, 191
  Windows CE shell and, 193–195,
    204–206, 213–216
See also HPC Pro devices
HPC Pro devices, 193–195, 204–206,
  213–216. See also HPC (Handheld
  PC) devices
hTemplate parameter, 300
HTML Viewer control
  basic description of, 479–504
  create presentations with, 496–504
  display help files in, 505–506
  file transfers and, 486–489
  launch, 489–491
  window messages, 503–504
HWND_DESKTOP constant, 146

I
ICC_ANIMATE_CLASS flag, 27
ICC_BAR_CLASSES flag, 27
ICC_COOL_CLASSES flag, 27
ICC_DATE_CLASSES flag, 27
ICC_HOTKEY_CLASS flag, 27
ICC_INTERNET_CLASSES flag, 27
ICC_LISTVIEW_CLASSES flag, 27
ICC_PROGRESS_CLASSES flag, 27
ICC_TAB_CLASSES flag, 27
ICC_TREEVIEW_CLASSES flag, 27
ICC_UPDATE_CLASSES flag, 27
ICC_USEREX_CLASSES flag, 27
icons
  definition of, in resource files, 4, 8
  image size restrictions for, 8, 65
  troubleshoot problems with, 75
See also graphics
IDB_DINNER constant, 159
IDB_ONE_TWO constant, 31, 32
IDB_STD_SMALL_COLOR constant, 32
IDB_THREE_FOUR constant, 32
IDB_VIEW_SMALL_COLOR
  constant, 32
IDCANCEL message, 353
IDM_ADD_KEY message, 269–270
IDM_DELETE_KEY message, 279–280
IDM_DELETE_VALUES message, 279
IDM_MENU message, 30
IDM_OPEN_FILE message, 296
IDM_READ_ONE_VAL message,
  277–278
IDOK message, 213, 353, 487
images. See graphics
IMalloc COM interface, 204–205
IM_CLEARALL message, 190
IM_GETDATALLEN message, 180, 190
IM_GETDATA message, 180, 190
IM_REINIT message, 190
IM_SETDATA message, 180, 191
INCLUDE environment variable, 135
InitCommonControlsEx function, 27
InitHTMLControl function, 502
InitInstance function, 28, 144, 186, 409
inking, 171–172
inkx.lib, 180, 186
InsertColumn function, 462
InsertComboBox function, 37, 66
InsertItem function, 465, 466
InsertTreeItem function, 425–426, 436, 439
InterpretKeyValue function, 427, 431
int return type, 297
InvalidateRect function, 145
ITEMIDLIST structure, 204

K
kernel, 235
KEY_ALL_ACCESS constant, 271, 279
keyboards, screen-based. See SIP
(Supplementary Input Panel)
KEY_CREATE_LINK constant, 271
KEY_CREATE_SUB_KEY constant, 271
KEY_ENUMERATE_SUB_KEYS constant, 271, 275
KEY_EXECUTE constant, 271
KEY_NOTIFY constant, 271
KEY_QUERY_VALUE constant, 271, 272
KEY_READ constant, 272
keys
add, 269–270
add values to, 273–275
delete, 279–281
enumerate, 271–273
HKEY_CLASSES_ROOT key, 258, 270
HKEY_CURRENT_USER key, 258, 270, 438
HKEY_LOCAL_MACHINE key, 258, 270–271
HKEY_USERS key, 270
types of, supported by Windows CE, 273–274
See also Registry
KEY_SET_VALUE constant, 271–273
KEY_WRITE constant, 272

L
LaunchViewer function, 492, 495–496
libraries. See also DLLs (Dynamic Link Libraries)
graphics behaviors and, 134
paths to, 134–135
line(s)
breaks, 85
draw, 137–152, 159, 179
See also graphics
lineTo function, 137
linkers, 133–136, 180, 225, 242
list box(es)
add database attribute rows to, 465–467
CommandBand controls and, 88–90
dropdown arrow control for, 37–38
memory management and, 255
multiple, 89–90
RAPI and, 414, 419–420, 422–423, 460–467, 470
Registry and, 276, 431–433
RemoteDBScan example and, 460–465
style flags for, 37–38
troubleshoot problems with, 66
Unicode strings for, 39
Windows CE shell and, 201–202
ListView control, 465–467
ListViewEx.cpp, 443, 460
ListViewEx.h, 443, 460
L macro, 420
LMEM_FIXED flag, 240
LMEM_ZEROINIT flag, 240
LoadCursorFromFile function, 9
LoadCursor function, 9
LoadIcon function, 215
LoadImage function, 215
LoadMenu function, 42
LoadString function, 243–244, 252, 297
LocalAlloc function, 83, 236, 239–240, 275, 299, 311
LocalFree function, 239–240

local heap
  file systems and, 298–299, 311
  memory management and, 236, 238–240
  See also heap
LocalReAlloc function, 239–240
LocalSize function, 275, 351–352
  lowercase, convert combo box input to, 38
LPARAM data type, 178–179, 190, 191, 308
lpcbClass parameter, 435
LPCE_FIND_DATA structure, 420
LPCTSTR parameter, 168, 252, 297
LPNMTOOLBAR structure, 63
LPTR flag, 83, 240
LTGRAY_BRUSH object, 169
LVCF_FMT flag, 462
LVCF_IMAGE flag, 461
LVCFM_CENTER flag, 462
LVCFMT_LEFT flag, 462
LVCFMT_RIGHT flag, 462
LVCF_ORDER flag, 461
LVCF_SUBITEM flag, 461
LVCF_TEXTURE flag, 461
LVCF_WIDTH flag, 461
LV_COLUMN structure, 461–462
LVIF_IMAGE flag, 465
LVIF_INDENT flag, 465
LVIF_PARAM flag, 465
LVIF_STATE flag, 465
LVIF_TEXT flag, 465, 468
LVIS_ACTIVATING flag, 466
LVIS_CUT flag, 466
LVIS_DROPHILITED flag, 466
LVIS FOCUSED flag, 466
LVIS_SELECTED flag, 466
LV_ITEM structure, 465, 468

MAKELONG macro, 348

Registry and, 272, 273, 277
stilus input and, 190
See also TEXT macro
MainFrame.cpp, 389
MainFrame.h, 389
MainFrm.cpp, 442, 444–447, 481, 483–486
MainFrm.h, 442, 444, 481, 482–483
MainWndProc function, 513
MAKELONG macro, 348
malloc function, 236
mapfiles, 242
MapViewOfFile function, 321–322
MapWindowPoints function, 41, 146
MatchClass function, 189
MBCS (multi-byte character set), 284, 301, 387–388, 466
mbstowcs function, 66, 313, 388
MEM_COMMIT flag, 237
MemDlg.cpp, 248–252
MemMapFiles.cpp, 315–322
MemMapFiles example, 314–322
memory
  accelerators and, 10
  allocation, 83, 126, 152, 236–240
  architecture, 234–235
  availability estimates, 255
  CommandBand controls and, 83, 127
  cursors and, 9
  data segment use and, 241–242
  debugging and, 508
  devoted to static data, 511–514
  graphics behaviors and, 152, 159
  management, 233–256
  mapped files, 314–322
  menus and, 11–12
  overwrites, 273
  power conservation and, 379, 380
  RAPI and, 476, 488
  read / write data sections and, 241, 244–245
  reconnaissance tools, 247–255
  release, 237, 238
  shortages, 245–247
  statistics, retrieve, 415
  storage space calculations, 36
  string tables and, 10

M

macros
  dim macro, 272, 273
  DRICOMPANION_API macro, 494–495
  HIWORD macro, 348, 474
  L macro, 420
stylus input and, 190
temporary files and, 240–241
See also heap; RAM (random-access memory); stack
Memory Dialog example, 247–255
MEMORYSTATUS structure, 415
MEM_RESERVE flag, 237
memset function, 27, 61, 215, 425
MemStatusDlgProc function, 253
menu bar(s)
  add buttons to, 30–33
  add combo boxes to, 36–39
  basic description of, 12–14
  CommandBand controls and, 68, 84
dynamic modification of, 30
examples, 14–42, 68, 84
number of, design guidelines for, 29
style flags, 84
stylus input and, 189
troubleshoot problems with, 76
versatility of, 12–14
width settings, 85
WinMain function and, 26–28
See also menus
MenuBar example, 15–42, 68, 84
MenuBar.rc, 15–26, 31
menus
  definition of, in resource files, 4, 10–12, 28, 30, 39
  eliminate items from, 42
  floating, 12, 39, 42
  nested, 14–26
  popup, 12, 39, 41–42
  rich ink control for, 188–190
  screen coordinates for, 41
  screen size and, 66, 85
troubleshoot problems with, 76
  update, 188–190
  Windows CE shell and, 226–228
See also menu bars
MessageBox function, 11–12, 296
messages
  DTM_ADDTEXT message, 504
  DTM_ADDTEXTW message, 504
  DTM_ANCHOR message, 504
  DTM_ANCHORW message, 504
  DTM_ENABLESHRINK message, 502, 504
  DTM_ENDOFSOURCE message, 504
  DTM_IMAGEFAIL message, 504
  DTM_SELECTALL message, 504
  DTM_SETIMAGE message, 504
  IDCANCEL message, 353
  IDM_ADD_KEY message, 269–270
  IDM_DELETE_KEY message, 279–280
  IDM_DELETE_VALUES message, 279
  IDM_MENU message, 30
  IDM_OPEN_FILE message, 296
  IDM_READ_ONE_VAL message, 277–278
  IDOK message, 213, 353, 487
  IM_CLEARALL message, 190
  IM_GETDATALEN message, 180, 190
  IM_GETDATA message, 180, 190
  IM_REINIT message, 190
  IM_SETDATA message, 180, 191
  PSN_APPLY message, 64, 65
  PSN_HELP message, 64
  PSN_KILLACTIVE message, 64
  PSN_QUERYCANCEL message, 64, 65
  PSN_RESET message, 64
  PSNRET_INVALID NOCHANGE-PAGE message, 65
  PSNRET_NOERROR message, 64–65
  PSN_SETACTIVE message, 64
  WM_COMMAND message, 29, 105
  WM_CREATE message, 28, 81, 502
  WM_GETTEXTLENGTH message, 126, 351
  WM_HELP message, 29, 506
  WM_HIBERNATE message, 245–247
  WM_INITDIALOG message, 205, 308, 343, 367–368, 374
  WM_LBUTTONDOWN message, 178–179
  WM_MOUSEMOVE message, 180
  WM_NOTIFY message, 40, 51, 63
  WM_PAINT message, 145, 146, 151–152, 178
  WM_SETTEXT message, 503
  WM_SETTINGCHANGE message, 225–226
  WM_TIMER message, 382
MFC (Microsoft Foundation Classes)
  RAPI and, 309–310, 387–389, 442
  remote presentations with, 480–504
MFC (continued)
    TEXT macro and, 420
    Tree control, 423–426
tips for, 387–310
See also classes
microprocessors, 12–14, 381, 382
Microsoft Visual C++
copy controls with, 51
Dependency Walker utility, 491–493
RAPI and, 409, 442
Remote Debugger, 507–508
resource file and, 4
start new project with, 12–14
Wizards, 491
Microsoft Windows 9x, 286, 287
Microsoft Windows NT, 286, 287
MM_TEXT mapping mode, 144–145
mobility, importance of, 234, 235
modulus operator, 104
most recently used list. See MRU
    (most recently used) list
MoveAndSize CommandBands
    control, 106–131
MoveAndSize.rc, 106–131
MoveToEx function, 137
MoveWindow function, 228
MRU (most recently used) list, 216–217
MsDlgProc function, 320
multitasking, 380–381
mutexes, 382
My Computer, 202
My Documents folder, 286, 287
MyHTMLViewer, 496–504
MyHTMLViewer.cpp, 497
MyHTMLViewer.h, 497–502
MyHTMLViewer.rc, 497
MyPaint function, 159, 166
MyRegisterClass function, 28

NOTIFYICONDATA structure, 214–216
NULL values, 61, 412, 432, 487, 490
database management and, 344, 346, 349, 370–371, 375
file systems and, 298, 300, 321
Registry and, 272–274, 276, 278, 280

O
object identifiers (CEOIDs), 344–346, 349, 353, 369–370, 372–375
capture, 442
defined, 463
open database with, 471
preserve, 467
return, 463
object store, 235, 283
OK button, 29, 64
OnDownloadHtml function, 486
OnInitDialog function, 414, 419, 473
OnInitialUpdate function, 460, 464
OnItemexpandingRegTree function, 427–428
OnSelchangedRegTree function, 430
OnUpdateFileButton function, 419
OPEN_ALWAYS flag, 300
OPEN_EXISTING flag, 300
OPENFILENAME structure, 298–300, 310–311, 487
OpenSaveFile function, 310, 311

P
PagedBands example, 90–106
PagedBands.rc, 90–104
palettes
    basic description of, 168
default, 168
grayscale, 31, 65, 159
PalmtopInk example, 180–186
parameters
dwAllocationGranularity parameter, 321
dwBytesWritten parameter, 312, 489
dwCount parameter, 351
dwDisp parameter, 270
dwDSize parameter, 275
dwFileAttributeFlags parameter, 421
dwFileSize parameter, 489

N
New command, 13
newline character, 39
Next button, 105
NIM_ADD flag, 216
NIM_DELETE flag, 216
NMHDR structure, 63–64, 428, 429–430
NM_TREEVIEW structure, 428–421
Index  533

dwFlags parameter, 464
dwFoundCount parameter, 422
dwFreeType parameter, 237
dwNSize parameter, 275
dwNumChildren parameter, 435
dwSize parameter, 237
dwValueDataSize parameter, 431
dwValueNameSize parameter, 431
lpcbClass parameter, 435
LPCTSTR parameter, 168, 252, 297
tszBuff parameter, 38
Paste command, 51
pathnames, 42, 297
  fully-qualified, 286
  hard-coded, 309
PC Cards, 301, 379, 380
pens, 168–170
permissions, 241–242, 346
  file systems and, 287–288, 296, 298
  Registry and, 274, 275, 279, 281
pFileMemory variable, 321
pidl variable, 204–205
  See also screen size
Platform Developer’s Kit
  (Windows CE), 85
platforms
  detect types of, 297
  target, in the Platforms pane, 13, 14
  unresolved externals and, 508
Pocket PC. See PPC (Pocket PC) devices
POINT structure, 41, 144
polling, 382
Polygon function, 152
Polyline function, 144, 146, 152
PopulateHTMLViewer function, 503
popup menu(s)
  alignment of, 41–42
  dropdown button and, 39
  initialization of, 41
  nested, 12
  tracking structures, 41–42
  See also menus
power conservation
  importance of, 380
  strategies, 379–386
power status, query, 415
PPC (Pocket PC) devices, 90, 218, 441
  CommandBand controls and, 68
  file systems and, 283–284, 286–288, 309, 322
  strategy for, 217
  Windows CE shell and, 217–226
preprocessor definitions, 420
processors, 12–14, 381, 382
program memory, defined, 235. See also memory
project(s)
  build settings for, 38
  choose initial files for, 13–14
  create, 12–14, 134–135
  release builds, validation of, 509–510
  start new, 12–14
  See also projects (listed by name)
projects (listed by name)
  BetterBirthdays, 334, 353–372
  BirthdayReminder, 334–354
  DesktopRAPIInvoker, 480–504
  Dinner, 152–158
  DuckStamps, 160–167
  EtchASketch, 137–152, 159
  Etch2, 146–152
  ExeFileMem, 242–243
  FidDirs, 194–201
  FileOpen, 194, 287–301, 311
  FindDirs, 214
  Floating Point, 322–332
  FullScreenDlg, 217–228
  GreatWriting, 301–314
  Hello World, 13–14, 133–135, 226–228, 242
  MemMapFiles, 314–322
  Memory Dialog, 247–255
  MenuBar, 15–42, 68, 84
  PagedBands, 90–106
  PalmtopInk, 180–186
  RapiDemo, 388–410
  RegDemo, 260–269, 275
  RemoteDBScan, 441–477
  StylusTrack, 172–180
  SysMenuGadgets, 206–214
  TabbedDlg, 50–60
  Unresolved, 133–136
  See also projects
properties
  database management and, 347–351
  data types for, 347–348
  references to, 347–348
  use of the term, 347
See also fields; property sheets
property sheet(s)
  behavior of, 51
  control of, 418–423
  data validation and, 64, 65
  define page groups for, 62
  dialog resources for, 411
  notification messages, 63–65
  object, initialization of, 412–413
  RAPI and, 389, 409–411
  setup, 60–65
  templates, 51
PropertySheet function, 60
PROPSHEETHEADER structure, 413
PROPSHEETPAGE structure, 60–61
prsht.h, 60–61
PSN_APPLY message, 64, 65
PSN_HELP message, 64
PSN_KILLACTIVE message, 64
PSN_QUERYCANCEL message, 64, 65
PSN_RESET message, 64
PSNRET_INVALID_NOCHANGE-PAGE message, 65
PSNRET_NOERROR message, 64–65
PSN_SETACTIVE message, 64
pszCSIDL constant, 202
pszFileName constant, 311
pszOpenFilter constant, 299, 311
PublishRecord function, 472, 474
records
  delete, 372–373
  get, from remote devices, 468–469
  iterate, 374–376
  modify, 373–374
  retrieved, interpret and format, 473–477
  sparse, implement, 348
See also databases
rectangles, 179. See also graphics
recursion, 434
Recycle Bin, 202
references, unresolved external,
  133–136, 508
REG_BINARY data type, 274, 432
RegCreateKeyEx function, 269–270
RegCreateKey function, 269–270
RegDeleteKey function, 280
RegDemo.cpp, 260–269
RegDemo example, 260–269, 275
REG_DWORD_BIG_ENDIAN data
type, 274, 432
REG_DWORD data type, 274, 277, 432
REG_DWORD_LITTLE_ENDIAN data type, 274, 432
RegEnumKeyEx function, 272
RegEnumValue function, 275–276
REG_EXPAND_SZ data type, 432
Registry
access, 259, 271–273
delete entries in, 279
enumerate values in, 275–277
manipulate data in, 260–269
overview of, 257–281
permissions and, 273
porting checklist, 280–291
RAPI and, 388, 431–433
read single-named values in, 277–279
retrieve information stored in, 271–273
structure/content of, 257–281
tree, 260, 280, 388, 423–431
See also Registry keys
Registry keys
add, 269–270
add values to, 273–275
delete, 279–281
enumerate, 271–273
HKEY_CLASSES_ROOT key, 258, 270
HKEY_CURRENT_USER key, 258, 270, 438
HKEY_LOCAL_MACHINE key, 258, 270–271
HKEY_USERS key, 270
types of, supported by Windows CE, 273–274
See also Registry
REG_MULTI_SZ data type, 274, 432
REG_NONE data type, 432
RegOpenKeyEx function, 269–271
RegQueryValueEx function, 278
REG_RESOURCE_LIST data type, 432
RegSetValueEx function, 274
REG_SZ data type, 274, 432
release builds, validation of, 509–510
ReleaseDC function, 146, 152
Remote Application Programming Interface. See RAPI (Windows CE Remote Application Programming Interface)
RemoteDBScan.cpp, 442
RemoteDBScanDoc.cpp, 442
RemoteDBScan example, 441–477
RemoteDBScan.h, 442
RemoteDBScan.rc, 442
RemoteDBScanView.cpp, 442
RemoteDBScanView.h, 442, 447–454, 459–460
RemoteFileAccessPage.cpp, 389
Remote Object Viewer, 510
Remote Registry Editor, 257, 260, 510
Remote Registry Viewer, 257–258
Remote Spy, 510
request filtering, 245
ResetControl function, 419
resolution, 9
Resource Editor, 51
resource file
defined, 4–8
dialog definitions, 4, 42–48
icon definitions in, 4, 8
menu definitions in, 4, 10–12, 28, 30, 39
resource_file.rc, 4–5
Resource.h, 43–48, 442, 481, 497
Resource workspace tab, 51
rich ink control
basic description of, 180–191
menu, 188–189
window procedure, subclassing, 187–189
ROM (read-only memory), 168, 235, 334. See also memory
RoundRect function, 169
S
SaveDC function, 152
scheduling algorithms, 381
screen size
combo boxes and, 66
CommandBand controls and, 85–86
dialogs and, 42, 48, 50, 65, 67
lack of standard, 37
limitations on user behavior by, 12
long forms and, 90
menus and, 66, 85
scroll action, 38
seamlessness, characteristic of, 234
seek type flags, 371–372
SendMessage function, 41, 126, 190, 191
SetCapture function, 178
SetFocus function, 502
SetImageList function, 438
SetItem function, 436
SetItemImage function, 429
SetTimer function, 382
SetWindowLong function, 128, 188
SetWindowText function, 470–471
SetWindowWord function, 188
SHACTIVATEINFO structure, 225–226
SHAddToRecentDocs function, 216
SHARD_PATH flag, 216
shell
application icons and, 206, 214–216
desktop shortcuts and, 206, 214
FullScreenDlg example and, 217–228
MRU (most recently used) list and, 216–217
overview of, 193–229
palmtop CE platforms and, 217–225
paths to special folders and, 194–206
SIP and, 217, 225–228
Start menu shortcuts and, 214
two basic functions performed by, 194
user interface elements, 206–213
Shell_NotifyIcon function, 216
SHFS_HIDESIPBUTTON flag, 227
SHFS_HIDESTARTICON flag, 227
SHFS_HIDETASKBAR flag, 227
SHFS_SHOWSIPBUTTON flag, 227
SHFS_SHOWSTARTICON flag, 227
SHFS_SHOWTASKBAR flag, 227
SHFullScreen function, 227–228
SHGetMalloc function, 205
SHGetPathFromIDList function, 204, 206
SHGetSpecialFolderLocation function, 202–204, 206
SHHandleWMSettingChange function, 225
SHInitDialog function, 227
SHINITDLGINFO structure, 226–228
SHLoadDIBitmap function, 168
ShowRecDlgProc function, 126
SIP (Supplementary Input Panel), 217, 225–228
SIP button, 217, 228
sizeof function, 27, 36, 274, 388
sort order, 344–345, 367–373, 471
SORTORDERSPEC structure, 344, 368, 471
source code
AllPages.cpp, 389, 413
BetterBirthdays.cpp, 354–367
BirthdayReminder.cpp, 335–343
DesktopRapiInvoker.cpp, 481
DesktopRapiInvokerDoc.cpp, 481
DesktopRapiInvokerView.cpp, 481
Dinner.cpp, 153–160
DRICompanion.cpp, 492
EtchASketch.cpp, 137–143
Etch2.cpp, 146–151
FileOpen.cpp, 288–301
FindDirs.cpp, 194–204
FloatingPoint.cpp, 323–330
FullScreenDlg.cpp, 218–225
GreatWriting.cpp, 301–314
ListViewEx.cpp, 443, 460
MainFrame.cpp, 389
MainFrm.cpp, 442, 444–447, 481, 483–486
MemDlg.cpp, 248–252
MemMapFiles.cpp, 315–322
MyHTMLViewer.cpp, 497
RapiDemo.cpp, 389
RapiDemoDoc.cpp, 389
RapiDemoView.cpp, 389
RecordListDialog.cpp, 443
RegDemo.cpp, 260–269
RemoteDBScan.cpp, 442
RemoteDBScanDoc.cpp, 442
RemoteDBScanView.cpp, 442
RemoteFileAccessPage.cpp, 389
stdafx.cpp, 389, 442, 481, 492, 497
StylusTrack.cpp, 173–178
SysMenuGadgets.cpp, 207–213
SystemStatusPage.cpp, 389, 413–414, 417
TabbedDlg.cpp, 51–60
WalkReg.cpp, 389
SpecialDirsDlgProc function, 201, 204
SPECIAL_DIRS structure, 202–203
sprintf function, 38, 414, 415, 467
stack, 10, 433
    based buffer, 127, 299
    memory management and, 233, 235, 241, 244
Windows CE shell and, 204
See also memory
Start menu
    add shortcuts to, 206, 213–214
    remove shortcuts from, 214
Windows CE shell and, 194, 206, 213–214
static keyword, 241, 309
stdafx.cpp, 389, 442, 481, 492, 497
stdafx.h, 389, 442, 481, 492, 497
 STD_COPY identifier, 32
 STD_CUT identifier, 32
 STD_DELETE identifier, 32
 STD_FILENEW identifier, 32
 STD_FILEOPEN identifier, 32
 STD_FILESAVE identifier, 32
 STD_FIND identifier, 32
 STD_HELP identifier, 32
 STD_PASTE identifier, 32
 STD_PRINT identifier, 33
 STD_PRINTPRE identifier, 33
 STD_PROPERTIES identifier, 32
 STD_REDOW identifier, 32
 STD_REPLACE identifier, 32
 STD_UNDO identifier, 33
stock objects, 168–170
_stprintf function, 255
strcpy function, 332, 388
StretchBlt function, 152, 159
string(s)
    caption, 86, 88
    character counts in, 36
    combo boxes and, 66
    CommandBand controls and, 85, 87–89
    file systems and, 297–298, 313
    graphics behaviors and, 135, 168
    handling functions, 313, 388
    identifiers (IDs), 88
    length, 298
    literals, 36, 39, 66
memory management and,
    243–244, 252–253
null terminated, 298
references to, 36
Registry and, 275, 276, 278, 280
resource file code for, 10
start address for, 36
tables, 4, 10
tips for, 313
type declarations, 36
Windows CE shell and, 202, 228
structCmdBarInit structure, 88–90, 130
structure(s)
    initialization of, 27–28, 61, 83–84, 86
    size settings, 27–28
See also structures (listed by name)
structures (listed by name)
CEBlob structure, 347, 351–352, 377, 476–477
CE_FIND_DATA structure, 421–422
CEOIDINFO structure, 463–464, 466
CEOSVERSIONINFO structure, 414
CEPROPID structure, 347–348, 351, 368, 370
CEPROPVAL structure, 347–352, 354, 370, 473–476
CEVALPROP structure, 349, 371
CEVALUNION structure, 348–352, 373, 376, 474, 476
COMMANDBANDSRESTOREINFO structure, 127
FILETIME structure, 347, 373, 376, 475–476
ITEMIDLIST structure, 204
LPCE_FIND_DATA structure, 420
LPNMTOOLBAR structure, 63
LV_COLUMN structure, 461–462
LV_ITEM structure, 465, 468
MEMORYSTATUS structure, 415
NMHDR structure, 63–64, 428, 429–430
NM_TREEVIEW structure, 428–421
NOTIFYICONDATA structure, 214–216
OPENFILENAME structure,
    298–300, 310–311, 487
POINT structure, 41, 144
structures (continued)
PROPSEETHEDER header structure, 413
PROPSEETPAGE structure, 60–61
REBARBANDINFO structure, 68–69,
81–86, 129
SHACTIVATEINFO structure,
225–226
SHINITDLGINFO structure, 226–228
SORTORDERSPEC structure, 344,
368, 471
SPECIAL_DIRS structure, 202–203
structCmdBarInit structure, 88–90, 130
SYSTEM_INFO structure, 253–255, 321
SYSTEM_POWER_STATUS_EX
structure, 415–416
SYSTEMTIME structure, 349–352,
373, 376
TBBUTTON structure, 30–31, 33, 39–40
TVITEM structure, 423–430, 436
WNDCLASS structure, 28, 187
See also structures
stylus, 9, 171–191
advantages of, over keyboard
input, 171
behavior of, as a one-mouse
button, 172–180
rich ink control and, 180–191
StylusTrack.cpp, 173–178
StylusTrack example, 172–180
sub-classes, 187–189
Supplementary Input Panel. See SIP
(Supplementary Input Panel)
SysMenuGadgets.cpp, 207–213
SysMenuGadgetsDlgPro function, 213
SysMenuGadgets example, 206–214
SYSTEM_FONT object, 169
SYSTEM_INFO structure, 253–255, 321
SystemParameterInfo function, 85
SYSTEM_POWER_STATUS_EX
structure, 415–416
System Status page, 389, 398–402,
413–419
SystemStatusPage.cpp, 389,
413–414, 417
SystemStatusPage.h, 389, 398–402
SYSTEMTIME structure, 349–352,
373, 376
T
tabbed dialogs, 50–68
TabbedDlg.cpp, 51–60
TabbedDlg example, 50–60
taskbar
add application icons to, 206, 214–216
hide, 228
TBBUTTON structure, 30–31, 33, 39–40
TCHAR data type, 38, 66, 310, 313, 331
CommandBand controls and, 89, 127
graphics behaviors and, 168
Registry and, 272
templates
dialog, 11–12, 51, 60–62, 411
property sheet, 51, 60–61
temporary files, 240–241
test
error status, 353–354, 431
graphics behaviors, 133, 134
threads, 382
text
fields, in list boxes, 37–38
fonts, 168–170
manipulation functions, 313–314
TEXT macro, 36, 38–39, 42, 66
FileOpen example and, 298
RAPI and, 420
threads
allotment of timeslices to, 381
basic description of, 380, 382
blocked, 381
power conservation and, 380–382
test, 382
timers, 382
timeslice, 381
toolbars
Build toolbar, 14
troubleshoot problems with, 75
TrackPopupMenuEx function, 41–42
TransparentImage function, 160, 167
TRUNCATE_EXISTING flag, 300
tszBuff parameter, 38
TVI_FIRST flag, 425
TVI_LAST flag, 425, 426
TVI_SORT flag, 425
TVITEM structure, 423–430, 436
.txt file extension, 311
typedef, 347
U
UFO dialog, 48–50, 60–61, 81
Unicode, 42, 467, 476–477
  ASCII and, translation between, 309–313, 388
combo boxes and, 36, 38–39, 66
  CommandBand controls and, 89
convert MBCS text to, 301, 309–313
database management and, 344, 351, 376
debugging and, 508
file systems and, 284, 301, 309–314, 332
HTML Viewer example and, 488, 490
L macro and, 420
  RAPI and, 388, 414, 420, 422, 430–433, 436
  Registry and, 273, 275, 276, 278, 280
rules, 36
  TCHAR type and, 38
text manipulation functions, 313–314
  unions, 348
UNIX, 433
Unresolved.asm, 136
Unresolved example, 133–136
unresolved externals, 133–136, 508
Unresolved.h, 135
updates, 145, 213, 373–374
UpdateWindow function, 145
uppercase, convert combo box input to, 38
usability studies, 170

V
validation, 64, 65, 84, 507–510
  CommandBand controls and, 69
tools, 507–510
variable(s)
  debugging and, 508
file systems and, 309, 320–321
initialization of, 508
long forms and, 107
  RAPI and, 410, 413, 418–419
static, declarations, 320–321
version information, 11–12, 85
VIEW_DETAILS identifier, 33
VIEW_LARGEICONS identifier, 33
VIEW_LIST identifier, 33
VIEW_SMALLICONS identifier, 33
VIEW_SORTDATE identifier, 33
VIEW_SORTNAME identifier, 33
VIEW_SORTSIZE identifier, 33
VIEW_SORTTYPE identifier, 33
VirtualAlloc function, 236–238, 245–246, 253
VirtualFree function, 237, 238
Visual C++ (Microsoft)
copy controls with, 51
  Dependency Walker utility, 491–493
IDE, 4
  RAPI and, 409, 442
Remote Debugger, 507–508
resource file and, 4
  start new project with, 12–14
Wizards, 491

W
wait cursor, 9
WalkReg.cpp, 389
WalkReg.h, 389, 402–409
Walk Registry Tree page, 389, 402–439
WalkRegTree function, 434, 439
WCHAR data type, 38, 66, 431
WCS (Wide Character String), 310–314, 388
wcscat function, 313
wcschr function, 313
wcscmp function, 313
wcscopy function, 311
wcscpsn function, 313
wcsncpy function, 313–314, 388
wcsdup function, 314
wcsicmp function, 314
wcslen function, 313
wcslwr function, 314
wcsncat function, 313
wcsncmp function, 313
wcsnicmp function, 314
wcspbrk function, 314
wcsrchr function, 314
wcstok function, 314
wcsstr function, 314
wcstombs function, 313, 422, 388
wcsupr function, 314
WHITE_BRUSH object, 169
WHITE_PEN object, 169
window(s)
  attributes for, 187
  classes, 186–188
  hide, 228
  titles, 104–106
Windows Explorer, 194, 283, 284
Windows.h, 190
Windows 9x (Microsoft), 286, 287
Windows NT (Microsoft), 286, 287
Windows Start logo, 217–218
WinMain, 26–28
WinPro, 81
winuser.h, 63–64
WM_COMMAND message, 29, 105
WM_CREATE message, 28, 81, 502
WM_GETTEXTLENGTH message,
  126, 351
WM_HELP message, 29, 506
WM_HIBERNATE message, 245–247
WM_INITDIALOG message, 205, 308,
  343, 367–368, 374
WM_LBUTTONDOWN message,
  178–179
WM_MOUSEMOVE message, 180
WM_NOTIFY message, 40, 51, 63
WM_PAINT message, 145, 146,
  151–152, 178
WM_SETTEXT message, 503
WM_SETTINGCHANGE message,
  225–226
WM_TIMER message, 382
WNDCLASS structure, 28, 187
WndProc function, 28, 105, 128
  GreatWriting example and, 308
  IDM_OPEN_FILE message and, 296
  Registry and, 277, 279–280
WriteFile function, 312, 314
wsprintf function, 38, 276–277, 376
x
Xircom cards, 507
FieldTrip 1.0 pairs your Windows CE device and your desktop computer to create a powerful, database independent forms based data collection system. For more information and free demo downloads visit www.fieldtrip2go.com

Use your Windows Desktop computer to create a form and specify validation criteria:

Download the form description to any Windows CE device using CE Services. The form is dynamically constructed, and custom fit to the screen of the device:

- Site Name: Text, Variable Length
- Location: Latitude, Longitude
- Time: Hours, Minutes, Seconds
- Date: Year, Month, Day
- Species: Text, Variable Length
- Count: Integer

For Help, press F1