allows the programmer to specify which bands won't be displayed when the command band is in a vertical orientation. Bands containing menus or wide controls are candidates for this flag because they won't be displayed correctly on vertical bands.

You can fill the *clrFore* and *clrBack* fields with a color that the command band will use for the foreground and background color when your application draws the band. These fields are used only if the RBBIM_COLORS flag is set in the mask field. These fields, along with the *hbmBack* field, which specifies a background bitmap for the band, are useful only if the band contains a transparent command bar. Otherwise, the command bar covers most of the area of the band, obscuring any background bitmap or special colors. I'll explain how to make a command bar transparent in the section, "Configuring individual bands."

The *lpText* field specifies the optional text that labels the individual band. This text is displayed at the left end of the bar immediately right of the gripper. The *iImage* field is used to specify a bitmap that will also be displayed on the left end of the bar. The *iImage* field is filled with an index to the list of images contained in the image list control. The text and bitmap fields take added significance when paired with the RBS_SMARTLABELS style of the command band control. When that style is specified, the text is displayed when the band is restored or maximized and the bitmap is displayed when the band is minimized. This technique is used by the H/PC Explorer on its command band control.

The *wID* field should be set to an ID value that you use to identify the band. The band ID is important if you plan on configuring the bands after they have been created or if you think you'll be querying their state. Even if you don't plan to use band IDs in your program, it's important that each band ID be unique because the control itself uses the IDs to manage the bands. This field is checked only if the RBBIM_ID flag is set in the *fMask* field.

The *hwndChild* field is used if the default command bar control in a band is replaced by another control. To replace the command bar control, the new control must first be created and the window handle of the control then placed in the *hwndChild* field. The *hwndChild* field is checked only if the RBBIM_CHILD flag is set in the *fMask* field.

The *cxMinChild* and *cyMinChild* fields define the minimum dimensions to which a band can shrink. When you're using a control other than the default command bar, these fields are useful for defining the height and minimum width (the width when minimized) of the band. These two fields are checked only if the RBBIM_CHILDSIZE flag is set.

The *cxIdeal* field is used when a band is maximized by the user. If this field isn't initialized, a maximized command band stretches across the entire width of the control. By setting *cxIdeal*, the application can limit the maximized width of a band, which is handy if the controls on the band take up only part of the total width of the control. This field is checked only if the RBBIM_IDEALSIZE flag is set in the *fMask* field.

The *lParam* field gives you a space to store an application-defined value with the band information. This field is checked only if the RBBIM_LPARAM flag is set in

the *fMask* field. The other fields in the REBARBANDINFO apply to the more flexible rebar control, not the command band control. The code below creates a command bands control, initializes an array of three REBARBANDINFO structures, and adds the bands to the control.

```
// Create a command bands ctl.
hwndCB = CommandBands_Create (hInst, hWnd, IDC_CMDBAND, RBS_SMARTLABELS |
                              RBS_VARHEIGHT, himl);

// Init common REBARBANDINFO structure fields.
for (i = 0; i < dim(rbi); i++) {
    rbi[i].cbSize = sizeof (REBARBANDINFO);
    rbi[i].fMask = RBBIM_ID | RBBIM_IMAGE | RBBIM_SIZE | RBBIM_STYLE;
    rbi[i].fStyle = RBBS_FIXEDBMP;
    rbi[i].wID = IDB_CMDBAND+i;
}
// Init REBARBANDINFO structure for each band.
// 1. Menu band.
rbi[0].fStyle |= RBBS_NOGRIPPER;
rbi[0].cx = 130;
rbi[0].iImage = 0;

// 2. Standard button band.
rbi[1].fMask |= RBBIM_TEXT;
rbi[1].cx = 200;
rbi[1].iImage = 1;
rbi[1].lpText = TEXT ("Std Btns");

// 3. Edit control band.
hwndChild = CreateWindow (TEXT ("edit"), TEXT ("edit ctl"),
                          WS_VISIBLE | WS_CHILD | WS_BORDER,
                          0, 0, 10, 5, hWnd, (HMENU)IDC_EDITCTL,
                          hInst, NULL);

rbi[2].fMask |= RBBIM_TEXT | RBBIM_STYLE | RBBIM_CHILDSIZE | RBBIM_CHILD;
rbi[2].fStyle |= RBBS_CHILDEDGE;
rbi[2].hwndChild = hwndChild;
rbi[2].cxMinChild = 0;
rbi[2].cyMinChild = 25;
rbi[2].cyChild = 55;
rbi[2].cx = 130;
rbi[2].iImage = 2;
rbi[2].lpText = TEXT ("Edit field");

// Add bands.
CommandBands_AddBands (hwndCB, hInst, 3, rbi);
```

The command bands control created above has three bands, one containing a menu, one containing a set of buttons, and one containing an edit control instead of a command bar. The control is created with the RBS_SMARTLABELS and RBS_VARHEIGHT styles. The smart labels display an icon when the bar is minimized and a text label when the band isn't minimized. The RBS_VARHEIGHT style allows each line on the control to have a different height.

The common fields of the REBARBANDINFO structures are then initialized in a loop. Then the remaining fields of the structures are customized for each band on the control. The third band, containing the edit control, is the most complex to initialize. This band needs more initialization since the edit control needs to be properly sized to match the standard height of the command bar controls in the other bands.

The *iImage* field for each band is initialized using an index into an image list that was created and passed to the *CommandBands_Create* function. The text fields for the second and third bands are filled with labels for those bands. The first band, which contains a menu, doesn't contain a text label because there's no need to label the menu. You also use the RBBS_NOGRIPPER style for the first band so that it can't be moved around the control. This fixes the menu band at its proper place in the control.

Now that we've created the bands, it's time to see how to initialize them.

Configuring individual bands

At this point in the process, the command bands control has been created and the individual bands have been added to the control. We have one more task, which is to configure the individual command bar controls in each band. (Actually, there's little more to configuring the command bar controls than what I've already described for command bars.)

The handle to a command bar contained in a band is retrieved using

```
HWND CommandBands_GetCommandBar (HWND hwndCmdBands, UINT uBand);
```

The *uBand* parameter is the zero-based band index for the band containing the command bar. If you call this function when the command bands control is being initialized, the index value correlates directly with the order in which the bands were added to the control. However, once the user has a chance to drag the bands into a new order, your application must obtain this index indirectly by sending a RB_IDTOINDEX message to the command bands control, as in

```
nIndex = SendMessage (hwndCmdBands, RB_IDTOINDEX, ID_BAND, 0);
```

This message is critical for managing the bands because many of the functions and messages for the control require the band index as the method to identify the band. The problem is that the index values are fluid. As the user moves the bands around, these index values change. You can't even count on the index values being consecutive. So, as a rule, never blindly use the index value without first querying the proper value by translating an ID value to an index value with RB_IDTOINDEX.

Once you have the window handle to the command bar, simply add the menu or buttons to the bar using the standard command bar control functions and messages. Most of the time, you'll specify only a menu in the first bar, only buttons in the second bar, and other controls in the third and subsequent bars.

The following code completes the creation process shown in the earlier code fragments. This code initializes the command bar controls in the first two bands. Since the third band has an edit control, you don't need to initialize that band. The final act necessary to complete the command band control initialization is to add the close box to the control using a call to *CommandBands_AddAdornments*.

```
// Add menu to first band.
hwndBand = CommandBands_GetCommandBar (hwndCB, 0);
CommandBar_InsertMenubar (hwndBand, hInst, ID_MENU, 0);

// Add std buttons to second band.
hwndBand = CommandBands_GetCommandBar (hwndCB, 1);
CommandBar_AddBitmap (hwndBand, HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
                      15, 0, 0);
CommandBar_AddButtons (hwndBand, dim(tbCBStdBtns), tbCBStdBtns);

// Add exit button to command band.
CommandBands_AddAdornments (hwndCB, hInst, 0, NULL);
```

Saving the band layout

The configurability of the command bands control presents a problem to the programmer. Users who rearrange the bands expect their customized layout to be restored the next time the application is started. This task is supposed to be made easy using the following function.

```
BOOL CommandBands_GetRestoreInformation (HWND hwndCmdBands,
                    UINT uBand, LPCOMMANDBANDSRESTOREINFO pcbr);
```

This function saves the positioning information from an individual band into a COMMANDBANDSRESTOREINFO structure. The function takes the handle of the command bands control and an index value for the band to be queried. The following code fragment shows how to query the information from each of the bands in a command band control.

```
// Get the handle of the command bands control.
hwndCB = GetDlgItem (hWnd, IDC_CMDBAND);

// Get information for each band.
for (i = 0; i < NUMBANDS; i++) {
```

(continued)

```
    // Get band index from ID value.
    nBand = SendMessage (hwndCB, RB_IDTOINDEX, IDB_CMDBAND+i, 0);

    // Initialize the size field and get the restore information.
    cbr[i].cbSize = sizeof (COMMANDBANDSRESTOREINFO);
    CommandBands_GetRestoreInformation (hwndCB, nBand, &cbr[i]);
}
```

The code above uses the RB_IDTOINDEX message to convert known band IDs into the unknown band indexes required by *CommandBands_GetRestoreInformation*. The data from the structure would normally be stored in the system registry. I'll talk about how to read and write registry data in Chapter 7, "Files, Databases, and the Registry."

The restore information should be read from the registry when the application is restarted, and used when creating the command bands control.

```
// Restore configuration to a command band.
COMMANDBANDSRESTOREINFO cbr[NUMBANDS];
REBARBANDINFO rbi;

// Initialize size field.
rbi.cbSize = sizeof (REBARBANDINFO);

// Set only style and size fields.
rbi.fMask = RBBIM_STYLE | RBBIM_SIZE;

// Set the size and style for all bands.
for (i = 0; i < NUMBANDS; i++) {
    rbi.cx = cbr[i].cxRestored;
    rbi.fStyle = cbr[i].fStyle;

    nBand = SendMessage (hwndCB, RB_IDTOINDEX, cbr[i].wID, 0);
    SendMessage (hwndCB, RB_SETBANDINFO, nBand, (LPARAM)&rbi);
}

// Only after the size is set for all bands can the bands
// needing maximizing be maximized.
for (i = 0; i < NUMBANDS; i++) {
    if (cbr[i].fMaximized) {
        nBand = SendMessage (hwndCB, RB_IDTOINDEX, cbr[i].wID, 0);
        SendMessage (hwndCB, RB_MAXIMIZEBAND, nBand, TRUE);
    }
}
```

This code assumes that the command bands control has already been created in its default configuration. In a real-world application, the restore information for the size and style could be used when first creating the control. In that case, all that would remain would be to maximize the bands depending on the state of the

fMaximized field in the COMMANDBANDSRESTOREINFO structure. This last step must take place only after all bands have been created and properly resized.

One limitation of this system of saving and restoring the band layout is that you have no method for determining the order of the bands in the control. The band index isn't likely to provide reliable clues because after the user has rearranged the bands a few times, the indexes are neither consecutive nor in any defined order. The only way around this problem is to constrain the arrangement of the bands so that the user can't reorder the bands. You do this by setting the RBS_FIXEDORDER style. This solves your problem, but doesn't help users if they want a different order. In the example program at the end of this section, I use the band index value to guess at the order. But this method isn't guaranteed to work.

Handling command band messages

The command bands control needs a bit more maintenance than a command bar. The difference is that the control can change height, and thus the window containing the command bands control must monitor the control and redraw and perhaps reformat its client area when the control is resized.

The command bands control sends a number of different WM_NOTIFY messages when the user rearranges the control. To monitor the height of the control, your application needs to check for a RBN_HEIGHTCHANGE notification and to react accordingly. The code below does just this:

```
// This code is inside a WM_NOTIFY message handler.
LPNMHDR pnmh;

pnmh = (LPNMHDR)lParam;
if (pnmh->code == RBN_HEIGHTCHANGE) {
    InvalidateRect (hWnd, NULL, TRUE);
}
```

If a RBN_HEIGHTCHANGE notification is detected, the routine simply invalidates the client area of the window forcing a WM_PAINT message. The code in the paint message then calls

```
UINT CommandBands_Height (HWND hwndCmdBands);
```

to query the height of the command bands control and subtracts this height from the client area rectangle.

As with the command bar, the command bands control can be hidden and shown with a helper function:

```
BOOL CommandBands_Show (HWND hwndCmdBands, BOOL fShow);
```

The visibility state of the control can be queried using

```
BOOL CommandBands_IsVisible (HWND hwndCmdBands);
```

The CmdBand Example Program

The CmdBand program demonstrates a fairly complete command bands control. The example creates three bands: a fixed menu band, a band containing a number of buttons, and a band containing an edit control. Transparent command bars and a background bitmap in each band are used to create a command bands control with a background image.

You can use the View menu to replace the command bands control with a simple command bar by choosing Command Bar from the View menu. You can then recreate and restore the command bands control to its last configuration by choosing Command Bands from the View menu. The code for the CmdBand program is shown in Figure 5-5 .

CmdBand.rc

```
//=============================================================================
// Resource file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//=============================================================================
#include "windows.h"                    //
#include "CmdBand.h"                    // Program-specific stuff

//-----------------------------------------------------------------------------
// Icons and bitmaps
//
ID_ICON         ICON   "cmdband.ico"     // Program icon
CmdBarBmps      BITMAP "cbarbmps.bmp"    // Bmp used in cmdband image list
CmdBarEditBmp   BITMAP "cbarbmp2.bmp"    // Bmp used in cmdband image list
CmdBarBack      BITMAP "backg2.bmp"      // Bmp used for cmdband background

//-----------------------------------------------------------------------------
// Menu
//
ID_MENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit",                       IDM_EXIT
    END
    POPUP "&View"
```

Figure 5-5. *The CmdBand program.*

```
    BEGIN
        MENUITEM "Command Bar",                 IDM_VIEWCMDBAR
        MENUITEM "Command Band",                IDM_VIEWCMDBAND
    END
    POPUP "&Help"

    BEGIN
        MENUITEM "&About...",                   IDM_ABOUT
    END
END
//------------------------------------------------------------------------
// About box dialog template
//
aboutbox DIALOG discardable 10, 10, 160, 40
STYLE  WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU | DS_CENTER |
       DS_MODALFRAME
CAPTION "About"
BEGIN
    ICON  ID_ICON,                  -1,  5,   5,  10,  10
    LTEXT "CmdBand - Written for the book Programming Windows \
           CE Copyright 1998 Douglas Boling"
                                    -1,  40,  5, 110,  30
END
```

CmdBand.h

```
//------------------------------------------------------------------------
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//------------------------------------------------------------------------
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))

//------------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                             // Structure associates
    UINT Code;                                  // messages
                                                // with a function.
```

(continued)

Figure 5-5. *continued*

```
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                                  // Structure associates
    UINT Code;                                      // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);         // function.
};
//------------------------------------------------------------------------
// Defines used by application
//
#define  IDC_CMDBAND          1                     // Command band ID
#define  IDC_CMDBAR           2                     // Command bar ID

#define  ID_ICON              10                    // Icon ID
#define  ID_MENU              11                    // Main menu resource ID
#define  IDC_EDITCTL          12

#define  IDB_CMDBAND          50                    // Base ID for bands
#define  IDB_CMDBANDMENU      50                    // Menu band ID
#define  IDB_CMDBANDBTN       51                    // Button band ID
#define  IDB_CMDBANDEDIT      52                    // Edit control band ID

// Menu item IDs
#define  IDM_EXIT             100

#define  IDM_VIEWCMDBAR       110
#define  IDM_VIEWCMDBAND      111

#define  IDM_ABOUT            120
#define  NUMBANDS             3
//------------------------------------------------------------------------
// Function prototypes
//
int CreateCommandBand (HWND hWnd, BOOL fFirst);
int DestroyCommandBand (HWND hWnd);

int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoNotifyMain (HWND, UINT, WPARAM, LPARAM);
```

```
LRESULT DoCommandMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);

// Command functions
LPARAM DoMainCommandViewCmdBar (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandVCmdBand (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandExit (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandAbout (HWND, WORD, HWND, WORD);

// Dialog procedures
BOOL CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM);
```

CmdBand.c

```
//========================================================================
// CmdBand - Dialog box demonstration
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//------------------------------------------------------------------------
#include <windows.h>                    // For all that Windows stuff
#include <commctrl.h>                   // Command bar includes

#include "CmdBand.h"                    // Program-specific stuff

//------------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("CmdBand");
HINSTANCE hInst;                        // Program instance handle

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_PAINT, DoPaintMain,
    WM_NOTIFY, DoNotifyMain,
    WM_COMMAND, DoCommandMain,
    WM_DESTROY, DoDestroyMain,
};
// Command message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDM_VIEWCMDBAR, DoMainCommandViewCmdBar,
    IDM_VIEWCMDBAND, DoMainCommandVCmdBand,
    IDM_EXIT, DoMainCommandExit,
    IDM_ABOUT, DoMainCommandAbout,
};
```

(continued)

Figure 5-5. *continued*

```
// Command band button initialization structure
const TBBUTTON tbCBStdBtns[] = {
//  BitmapIndex      Command  State              Style     UserData  String
    {STD_FILENEW,    210,     TBSTATE_ENABLED, TBSTYLE_BUTTON, 0,    0},
    {STD_FILEOPEN,   211,     TBSTATE_ENABLED, TBSTYLE_BUTTON, 0,    0},
    {STD_FILESAVE,   212,     TBSTATE_ENABLED, TBSTYLE_BUTTON, 0,    0},
    {0,              0,       TBSTATE_ENABLED, TBSTYLE_SEP,    0,    0},
    {STD_CUT,        213,     TBSTATE_ENABLED, TBSTYLE_BUTTON, 0,    0},
    {STD_COPY,       214,     TBSTATE_ENABLED, TBSTYLE_BUTTON, 0,    0},
    {STD_PASTE,      215,     TBSTATE_ENABLED, TBSTYLE_BUTTON, 0,    0},
    {0,              0,       TBSTATE_ENABLED, TBSTYLE_SEP,    0,    0},
    {STD_PROPERTIES, 216,     TBSTATE_ENABLED, TBSTYLE_BUTTON, 0,    0},
};

// Command bar initialization structure
const TBBUTTON tbCBViewBtns[] = {
//  BitmapIndex   Command  State                Style          UserData String
    {0,               0, 0,
                                               TBSTYLE_SEP,          0, 0},
    {VIEW_LARGEICONS, 210, TBSTATE_ENABLED | TBSTATE_CHECKED,
                                               TBSTYLE_CHECKGROUP, 0, 0},
    {VIEW_SMALLICONS, 211, TBSTATE_ENABLED,
                                               TBSTYLE_CHECKGROUP, 0, 0},
    {VIEW_LIST,       212, TBSTATE_ENABLED,
                                               TBSTYLE_CHECKGROUP, 0, 0},
    {VIEW_DETAILS,    213, TBSTATE_ENABLED,
                                               TBSTYLE_CHECKGROUP, 0, 0},
    {0,               0, 0,                    TBSTYLE_SEP,          0, 0},
    {VIEW_SORTNAME,   214, TBSTATE_ENABLED | TBSTATE_CHECKED,
                                               TBSTYLE_CHECKGROUP, 0, 0},
    {VIEW_SORTTYPE,   215, TBSTATE_ENABLED,
                                               TBSTYLE_CHECKGROUP, 0, 0},
    {VIEW_SORTSIZE,   216, TBSTATE_ENABLED,
                                               TBSTYLE_CHECKGROUP, 0, 0},
    {VIEW_SORTDATE,   217, TBSTATE_ENABLED,
                                               TBSTYLE_CHECKGROUP, 0, 0}
};

// Array that stores the band configuration
COMMANDBANDSRESTOREINFO cbr[NUMBANDS];
INT nBandOrder[NUMBANDS];
//------------------------------------------------------------------------
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
```

```
    HWND hwndMain;
    MSG msg;
    int rc;

    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//------------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;
    INITCOMMONCONTROLSEX icex;

    // Register application main window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = MainWndProc;                   // Callback function
    wc.cbClsExtra = 0;                              // Extra class data
    wc.cbWndExtra = 0;                              // Extra window data
    wc.hInstance = hInstance;                       // Owner handle
    wc.hIcon = NULL;                                // Application icon
    wc.hCursor = NULL;                              // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                        // Menu name
    wc.lpszClassName = szAppName;                   // Window class name

    if (RegisterClass (&wc) == 0) return 1;

    // Load the command bar common control class.
    icex.dwSize = sizeof (INITCOMMONCONTROLSEX);
    icex.dwICC = ICC_COOL_CLASSES;
    InitCommonControlsEx (&icex);
```

(continued)

Figure 5-5. *continued*

```
    return 0;
}
//-----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow){
    HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.

    hWnd = CreateWindow (szAppName,              // Window class
                         TEXT ("CmdBand Demo"), // Window title
                         WS_VISIBLE,             // Style flags
                         CW_USEDEFAULT,          // x position
                         CW_USEDEFAULT,          // y position
                         CW_USEDEFAULT,          // Initial width
                         CW_USEDEFAULT,          // Initial height
                         NULL,                   // Parent
                         NULL,                   // Menu, must be null
                         hInstance,              // Application instance
                         NULL);                  // Pointer to create
                                                 // parameters
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//-----------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {
    return nDefRC;
}
//=======================================================================
// Message handling procedures for MainWindow
//-----------------------------------------------------------------------
// MainWndProc - Callback function for application window
```

```
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }

    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}

//------------------------------------------------------------------------
// DoCreateMain - Process WM_CREATE message for window.
//
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {

    CreateCommandBand (hWnd, TRUE);
    return 0;
}
//------------------------------------------------------------------------
// DoPaintMain - Process WM_PAINT message for window.
//
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    PAINTSTRUCT ps;
    HWND hwndCB;
    RECT rect;
    HDC hdc;
    POINT ptArray[2];

    // Adjust the size of the client rect to take into account
    // the command bar or command bands height.
    GetClientRect (hWnd, &rect);
    if (hwndCB = GetDlgItem (hWnd, IDC_CMDBAND))
        rect.top += CommandBands_Height (hwndCB);
    else
        rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    hdc = BeginPaint (hWnd, &ps);
```

(continued)

Figure 5-5. *continued*

```
    ptArray[0].x = rect.left;
    ptArray[0].y = rect.top;
    ptArray[1].x = rect.right;
    ptArray[1].y = rect.bottom;
    Polyline (hdc, ptArray, 2);

    ptArray[0].x = rect.right;
    ptArray[1].x = rect.left;
    Polyline (hdc, ptArray, 2);

    EndPaint (hWnd, &ps);
    return 0;
}
//-----------------------------------------------------------------
// DoCommandMain - Process WM_COMMAND message for window.
//
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)
            return (*MainCommandItems[i].Fxn)(hWnd, idItem, hwndCtl,
                                              wNotifyCode);
    }
    return 0;
}
//-----------------------------------------------------------------
// DoNotifyMain - Process WM_NOTIFY message for window.
//
LRESULT DoNotifyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    LPNMHDR pnmh;

    // Parse the parameters.
    pnmh = (LPNMHDR)lParam;
```

```
    if (pnmh->code == RBN_HEIGHTCHANGE) {
        InvalidateRect (hWnd, NULL, TRUE);

    }

    return 0;
}
//--------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;

}
//====================================================================
// Command handler routines
//--------------------------------------------------------------------
// DoMainCommandExit - Process Program Exit command.
//
LPARAM DoMainCommandExit (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {

    SendMessage (hWnd, WM_CLOSE, 0, 0);
    return 0;
}
//--------------------------------------------------------------------
// DoMainCommandVCmdBarStd - Process View | Std Command bar command.
//
LPARAM DoMainCommandViewCmdBar (HWND hWnd, WORD idItem, HWND hwndCtl,
                                WORD wNotifyCode) {
    HWND hwndCB;

    hwndCB = GetDlgItem (hWnd, IDC_CMDBAND);
    if (hwndCB)
        DestroyCommandBand (hWnd);
    else
        return 0;

    // Create a minimal command bar that has only a menu and
    // an exit button.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);

    // Insert the menu.
    CommandBar_InsertMenubar (hwndCB, hInst, ID_MENU, 0);
```

(continued)

Figure 5-5. *continued*

```
    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    InvalidateRect (hWnd, NULL, TRUE);
    return 0;
}
//------------------------------------------------------------------------
// DoMainCommandVCmdBand - Process View | Command band command.
//
LPARAM DoMainCommandVCmdBand (HWND hWnd, WORD idItem, HWND hwndCtl,
                              WORD wNotifyCode) {
    HWND hwndCB;
    hwndCB = GetDlgItem (hWnd, IDC_CMDBAR);
    if (hwndCB)
        CommandBar_Destroy (hwndCB);
    else
        return 0;

    CreateCommandBand (hWnd, FALSE);
    InvalidateRect (hWnd, NULL, TRUE);
    return 0;
}
//------------------------------------------------------------------------
// DoMainCommandAbout - Process the Help | About menu command.
//
LPARAM DoMainCommandAbout(HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {

    // Use DialogBox to create modal dialog box.
    DialogBox (hInst, TEXT ("aboutbox"), hWnd, AboutDlgProc);
    return 0;
}
//========================================================================
// About Dialog procedure
//
BOOL CALLBACK AboutDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                            LPARAM lParam) {
    switch (wMsg) {
        case WM_COMMAND:
            switch (LOWORD (wParam)) {
                case IDOK:
                case IDCANCEL:
                    EndDialog (hWnd, 0);
                    return TRUE;
            }
        break;
    }
```

```
    return FALSE;
}
//--------------------------------------------------------------------
// DestroyCommandBand - Destroy command band control after saving
// the current configuration.
//
int DestroyCommandBand (HWND hWnd) {
    HWND hwndCB;
    INT i, nBand, nMaxBand = 0;

    hwndCB = GetDlgItem (hWnd, IDC_CMDBAND);
    for (i = 0; i < NUMBANDS; i++) {

        // Get band index from ID value.
        nBand = SendMessage (hwndCB, RB_IDTOINDEX, IDB_CMDBAND+i, 0);

        // Save the band number to save order of bands.
        nBandOrder[i] = nBand;

        // Get the restore information.
        cbr[i].cbSize = sizeof (COMMANDBANDSRESTOREINFO);
        CommandBands_GetRestoreInformation (hwndCB, nBand, &cbr[i]);
    }
    DestroyWindow (hwndCB);
    return 0;
}
//--------------------------------------------------------------------
// CreateCommandBand - Create a formatted command band control.
//
int CreateCommandBand (HWND hWnd, BOOL fFirst) {
    HWND hwndCB, hwndBand, hwndChild;
    INT i, nBand, nBtnIndex, nEditIndex;
    LONG lStyle;
    HBITMAP hBmp;
    HIMAGELIST himl;
    REBARBANDINFO rbi[NUMBANDS];

    // Create image list control for bitmaps for minimized bands.
    himl = ImageList_Create (16, 16, ILC_COLOR, 3, 0);
    // Load first two images from one bitmap.
    hBmp = LoadBitmap (hInst, TEXT ("CmdBarBmps"));
    ImageList_Add (himl, hBmp, NULL);
    DeleteObject (hBmp);
    // Load third image as a single bitmap.
    hBmp = LoadBitmap (hInst, TEXT ("CmdBarEditBmp"));
    ImageList_Add (himl, hBmp, NULL);
    DeleteObject (hBmp);
```

(continued)

Figure 5-5. *continued*

```
// Create a command band.
hwndCB = CommandBands_Create (hInst, hWnd, IDC_CMDBAND,
                              RBS_SMARTLABELS |
                              RBS_AUTOSIZE | RBS_VARHEIGHT, himl);

// Load bitmap used as background for command bar.
hBmp = LoadBitmap (hInst, TEXT ("CmdBarBack"));
// Initialize common REBARBANDINFO structure fields.
for (i = 0; i < dim(rbi); i++) {
    rbi[i].cbSize = sizeof (REBARBANDINFO);
    rbi[i].fMask = RBBIM_ID | RBBIM_IMAGE | RBBIM_SIZE |
                   RBBIM_BACKGROUND | RBBIM_STYLE;
    rbi[i].wID = IDB_CMDBAND+i;
    rbi[i].hbmBack = hBmp;
}

// If first time, initialize the restore structure since it is
// used to initialize the band size and style fields.
if (fFirst) {
    nBtnIndex = 1;
    nEditIndex = 2;
    cbr[0].cxRestored = 130;
    cbr[1].cxRestored = 210;
    cbr[1].fStyle = RBBS_FIXEDBMP;
    cbr[2].cxRestored = 130;
    cbr[2].fStyle = RBBS_FIXEDBMP | RBBS_CHILDEDGE;
} else {
    // If not first time, set order of bands depending on
    // the last order.
    if (nBandOrder[1] < nBandOrder[2]) {
        nBtnIndex = 1;
        nEditIndex = 2;
    } else {
        nBtnIndex = 2;
        nEditIndex = 1;
    }
}
// Initialize REBARBANDINFO structure for each band.
// 1. Menu band
rbi[0].fStyle = RBBS_FIXEDBMP | RBBS_NOGRIPPER;
rbi[0].cx = cbr[0].cxRestored;
rbi[0].iImage = 0;

// 2. Standard button band
rbi[nBtnIndex].fMask |= RBBIM_TEXT;
rbi[nBtnIndex].iImage = 1;
rbi[nBtnIndex].lpText = TEXT ("Std Btns");
```

```
// The next two parameters are initialized from saved data.
rbi[nBtnIndex].cx = cbr[1].cxRestored;
rbi[nBtnIndex].fStyle = cbr[1].fStyle;

// 3. Edit control band
hwndChild = CreateWindow (TEXT ("edit"), TEXT ("edit ctl"),
              WS_VISIBLE | WS_CHILD | ES_MULTILINE | WS_BORDER,
              0, 0, 10, 5, hWnd, (HMENU)IDC_EDITCTL, hInst, NULL);

rbi[nEditIndex].fMask |= RBBIM_TEXT | RBBIM_STYLE |
                         RBBIM_CHILDSIZE | RBBIM_CHILD;
rbi[nEditIndex].hwndChild = hwndChild;
rbi[nEditIndex].cxMinChild = 0;
rbi[nEditIndex].cyMinChild = 23;
rbi[nEditIndex].cyChild = 55;
rbi[nEditIndex].iImage = 2;
rbi[nEditIndex].lpText = TEXT ("Edit field");
// The next two parameters are initialized from saved data.
rbi[nEditIndex].cx = cbr[2].cxRestored;
rbi[nEditIndex].fStyle = cbr[2].fStyle;

// Add bands.
CommandBands_AddBands (hwndCB, hInst, 3, rbi);

// Add menu to first band.
hwndBand = CommandBands_GetCommandBar (hwndCB, 0);
CommandBar_InsertMenubar (hwndBand, hInst, ID_MENU, 0);

// Add standard buttons to second band.
hwndBand = CommandBands_GetCommandBar (hwndCB, nBtnIndex);
// Insert buttons
CommandBar_AddBitmap (hwndBand, HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
                      16, 0, 0);
CommandBar_AddButtons (hwndBand, dim(tbCBStdBtns), tbCBStdBtns);

// Modify the style flags of each command bar to make transparent.
for (i = 0; i < NUMBANDS; i++) {
    hwndBand = CommandBands_GetCommandBar (hwndCB, i);
    lStyle = SendMessage (hwndBand, TB_GETSTYLE, 0, 0);
    lStyle |= TBSTYLE_TRANSPARENT;
    SendMessage (hwndBand, TB_SETSTYLE, 0, lStyle);
}

// If not the first time the command band has been created, restore
// the user's last configuration.
```

(continued)

Figure 5-5. *continued*

```
    if (!fFirst) {
        for (i = 0; i < NUMBANDS; i++) {
            if (cbr[i].fMaximized) {
                nBand = SendMessage (hwndCB, RB_IDTOINDEX,
                                     cbr[i].wID, 0);
                SendMessage (hwndCB, RB_MAXIMIZEBAND, nBand, TRUE);
            }
        }
    }
    // Add exit button to command band.
    CommandBands_AddAdornments (hwndCB, hInst, 0, NULL);
    return 0;
}
```

CmdBand creates the command band in the *CreateCommandBand* routine. This routine is initially called in *OnCreateMain* and later in the *DoMainCommand-VCmdBand* menu handler. The program creates the command bands control using the RBS_SMARTLABELS style along with an image list and text labels to identify each band when it's minimized and when it's restored or maximized. An image list is created and initialized with the bitmaps that are used when the bands are minimized.

The array of REBARBANDINFO structures is initialized to define each of the three bands. If the control had previously been destroyed, data from the COMMANDBANDSRESTOREINFO structure is used to initialize the *style* and *cx* fields. The *CreateCommandBand* routine also makes a guess at the order of the button and edit bands by looking at the band indexes saved when the control was last destroyed. While this method isn't completely reliable for determining the previous order of the bands, it gives you a good estimate.

When the command bands control is created, the command bars in each band are also modified to set the TBS_TRANSPARENT style. This process, along with a background bitmap defined for each band, demonstrates how you can use a background bitmap to make the command bands control have just the right look.

When CmdBand replaces the command bands control with a command bar, the application first calls the *DestroyCommandBand* function to save the current configuration and then destroy the command bands control. This function uses the *CommandBands_GetRestoreInformation* to query the size and style of each of the bands. The function also saves the band index for each band to supply the data for the guess on the current order of the button and edit bands. The first band, the menu band, is fixed with the RBBS_NOGRIPPER style, so there's no issue as to its position.

This completes the discussion of the command bar and command bands controls. I talk about these two controls at length because you'll need one or the other for almost every Windows CE application.

For the remainder of the chapter, I'll cover the highlights of some of the other controls. These other controls aren't very different from their counterparts under Windows 98 and Windows NT. I'll spend more time on the controls I think you'll need when writing a Windows CE application. I'll start with the month calendar and the time and date picker controls. These controls are rather new to the common control set and have a direct application to the PIM-like applications that are appropriate for many Windows CE systems. I'll also spend some time covering the list view control, concentrating on features of use to Windows CE developers. The remainder of the common controls, I'll cover just briefly.

The Month Calendar Control

The month calendar control gives you a handy month-view calendar that can be manipulated by users to look up any month, week, or day as far back as the adoption of the Gregorian calendar in September 1752. The control can display as many months as will fit into the size of the control. The days of the month can be highlighted to indicate appointments. The weeks can indicate the current week into the year. Users can spin through the months by tapping on the name of the month or change years by tapping on the year displayed.

Before using the month calendar control, you must initialize the common control library either by calling *InitCommonControls* or by calling *InitCommonControlsEx* with the ICC_DATE_CLASSES flag. You create the control by calling *CreateWindow* with the MONTHCAL_CLASS flag. The style flags for the control are shown here:

- *MCS_MULTISELECT* The control allows multiple selection of days.
- *MCS_NOTODAY* The control won't display today's date under the calendar.
- *MCS_NOTODAYCIRCLE* The control won't circle today's date.
- *MCS_WEEKNUMBERS* The control displays the week number (1 through 52) to the left of each week in the calendar.
- *MCS_DAYSTATE* The control sends notification messages to the parent requesting the days of the month that should be displayed in bold. You use this style to indicate which days have appointments or events scheduled.

Initializing the control

In addition to the styles I just described, you can use a number of messages or their corresponding wrapper macros to configure the month calendar control. You can use an MCM_SETFIRSTDAYOFWEEK message to display a different starting day of the week. You can also use the MCM_SETRANGE message to display dates within a given range in the control You can configure date selection to allow the user to choose only

single dates or to set a limit to the range of dates that a user can select at any one time. The single/multiple date selection ability is defined by the MCS_MULTISELECT style. If you set this style, you use the MCM_SETMAXSELCOUNT message to set the maximum number of days that can be selected at any one time.

You can set the background and text colors of the control by using the MCM_SETCOLOR message. This message can individually set colors for the different regions within the controls, including the calendar text and background, the header text and background, and the color of the days that precede and follow the days of the month being displayed. This message takes a flag indicating what part of the control to set and a COLORREF value to specify the color.

The month calendar control is designed to display months on an integral basis. That is, if the control is big enough for one and a half months, it displays only one month, centered in the control. You can use the MCM_GETMINREQRECT message to compute the minimum size necessary to display one month. Because the control must first be created before the MCM_GETMINREQRECT can be sent, properly sizing the control is a round-about process. You must create the control, send the MCM_GETMINREQRECT message, and then resize the control using the data returned from the message.

Month calendar notifications

The month calendar control has only three notification messages to send to its parent. Of these, the MCN_GETDAYSTATE notification is the most important. This notification is sent when the control needs to know what days of a month to display in bold. This is done by querying the parent for a series of bit field values encoded in a MONTHDAYSTATE variable. This value is nothing more than a 32-bit value with bits 1 through 31 representing the days 1 through 31 of the month.

When the control needs to display a month, it sends a MCN_GETDAYSTATE notification with a pointer to an NMDAYSTATE structure defined as the following:

```
typedef struct {
    NMHDR nmhdr;
    SYSTEMTIME stStart;
    int cDayState;
    LPMONTHDAYSTATE prgDayState;
} NMDAYSTATE;
```

The *nmbhdr* field is simply the NMHDR structure that's passed with every WM_NOTIFY message. The *stStart* field contains the starting date for which the control is requesting information. This date is encoded in a standard SYSTEMTIME structure used by all versions of Windows. It's detailed on the facing page.

```
typedef struct {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME;
```

For this notification, only the *wMonth*, *wDay*, and *wYear* fields are significant.

The *cDayState* field contains the number of entries in an array of MONTHDAYSTATE values. Even if a month calendar control is displaying only one month, it could request information about the previous and following months if days of those months are needed to fill in the top or bottom lines of the calendar.

The month calendar control sends an MCN_SELCHANGE notification when the user changes the days that are selected in the control. The structure passed with this notification, NMSELCHANGE, contains the newly highlighted starting and ending days. The MCN_SELECT notification is sent when the user double-taps on a day. The same NMSELCHANGE structure is passed with this notification to indicate the days that have been selected.

The Date and Time Picker Control

The date and time picker control looks deceptively simple but is a great tool for any application that needs to ask the user to specify a date. Any programmer that has had to parse, validate, and translate a string into a valid system date or time will appreciate this control.

When used to select a date, the control resembles a combo box, which is an edit field with a down arrow button on the right side. Clicking on the arrow, however, displays a month calendar control showing the current month. Selecting a day in the month dismisses the month calendar control and fills the date and time picker control with that date. When you configure it to query for a time, the date and time picker control resembles an edit field with a spin button on the right end of the control.

The date and time picker control has three default formats: two for displaying the date and one for displaying the time. The control also allows you to provide a formatting string so that users can completely customize the fields in the control. The control even lets you insert application-defined fields in the control.

Creating a date and time picker control

Before you can create the date and time picker control, the common control library must be initialized. If *InitCommonControlsEx* is used, it must be passed a

ICC_DATE_CLASSES flag. The control is created by using *CreateWindow* with a class of DATETIMEPICK_CLASS. The control defines the following styles:

- *DTS_LONGDATEFORMAT* The control displays a date in long format, as in Saturday, September 19, 1998. The actual long date format is defined in the system registry.
- *DTS_SHORTDATEFORMAT* The control displays a date in short format, as in 9/19/98. The actual short date format is defined in the system registry.
- *DTS_TIMEFORMAT* The control displays the time in a format such as 5:50:28 PM. The actual time format is defined in the system registry.
- *DTS_SHOWNONE* The control has a check box to indicate that the date is valid.
- *DTS_UPDOWN* An up-down control replaces the drop-down button that displays a month calendar control in date view.
- *DTS_APPCANPARSE* Allows the user to directly enter text into the control. The control sends a DTN_USERSTRING notification when the user is finished.

The first three styles simply specify a default format string. These formats are based on the regional settings in the registry. Since these formats can change if the user picks different regional settings in the Control Panel, the date and time picker control needs to know when these formats change. The system informs top-level windows of these types of changes by sending a WM_SETTINGCHANGE message. An application that uses the date and time picker control and uses one of these default fonts should forward the WM_SETTINGCHANGE message to the control if one is sent. This causes the control to reconfigure the default formats for the new regional settings.

The DTS_APPCANPARSE style enables the user to directly edit the text in the control. If this isn't set, the allowable keys are limited to the cursor keys and the numbers. When a field, such as a month, is highlighted in the edit field and the user presses the 6 key, the month changes to June. With the DTS_APPCANPARSE style, the user can directly type any character into the edit field of the control. When the user has finished, the control sends a DTN_USERSTRING notification to the parent window so that the text can be verified.

Customizing the format

To customize the display format, all you need to do is create a format string and send it to the control using a DTM_SETFORMAT message. The format string can be made up of any of the following codes:

```
String          Description
fragment

"d"      One- or two-digit day.
"dd"     Two-digit day. Single digits have a leading zero.
"ddd"    The three-character weekday abbreviation. As in Sun, Mon...
"dddd"   The full weekday name.

"h"      One- or two-digit hour (12-hour format).
"hh"     Two-digit hour (12-hour format) Single digits have a leading zero.
"H"      One- or two-digit hour (24-hour format).
"HH"     Two-digit hour (24-hour format) Single digits have a leading zero.

"m"      One- or two-digit minute.
"mm"     Two-digit minute. Single digits have a leading zero.

"M"      One- or two-digit month.
"MM"     Two-digit month. Single digits have a leading zero.

"MMM"    Three-character month abbreviation.
"MMMM"   Full month name.

"t"      The one-letter AM/PM abbreviation. As in A or P.
"tt"     The two-letter AM/PM abbreviation. As in AM or PM.

"X"      Specifies a callback field that must be parsed by the application.

"y"      One-digit year. As in 8 for 1998.
"yy"     Two-digit year. As in 98 for 1998.
"yyy"    Full four-digit year. As in 1998.
```

Literal strings can be included in the format string by enclosing them in single quotes. For example, to display the string *Today is: Saturday, December 5, 1998* the format string would be

```
'Today is: 'dddd', 'MMMM' 'd', 'yyy
```

The single quotes enclose the strings that aren't parsed. That includes the *Today is:* as well as all the separator characters, such as spaces and commas.

The callback field, designated by a series of X characters, provides for the application the greatest degree of flexibility for configuring the display of the date. When the control detects an *X* field in the format string, it sends a series of notification messages to its owner asking what to display in that field. A format string can have any number of *X* fields. For example the following string has two *X* fields.

```
'Today 'XX' is: ' dddd', 'MMMM' 'd', 'yyy' and is 'XXX' birthday'
```

The number of X characters is used by the application only to differentiate the application-defined fields; it doesn't indicate the number of characters that should

be displayed in the fields. When the control sends a notification asking for information about an *X* field, it includes a pointer to the *X* string so that the application can determine which field is being referenced.

When the date and time picker control needs to display an application-defined *X* field, it sends two notifications: DTN_FORMATQUERY and DTN_FORMAT. The DTN_FORMATQUERY notification is sent to get the maximum size of the text to be displayed. The DTN_FORMAT notification is then sent to get the actual text for the field. A third notification, DTN_WMKEYDOWN is sent when the user highlights an application-defined field and presses a key. The application is responsible for determining which keys are valid and modifying the date if an appropriate key is pressed.

The List View Control

The list view control is arguably the most complex of the common controls. It displays a list of items in one of four modes: large icon, small icon, list, and report. The Windows CE version of the list view control supports many, but not all, of the valuable new features recently added for Internet Explorer 4.0. Some of these new functions are a great help in the memory-constrained environment of Windows CE. These new features include the ability to manage virtual lists of almost any size, headers that can have images and be rearranged using drag and drop, the ability to indent an entry, and new styles for report mode. The list view control also supports the new custom draw interface, which allows a fairly easy way of changing the appearance of the control.

You register the list view control by calling either *InitCommonControls* or *InitCommonControls* using a ICC_LISTVIEW_CLASSES flag. You create the control by calling *CreateWindow* using the class filled with WC_LISTVIEW. Under Windows CE, the list view control supports all the styles supported by other versions of Windows, including the new LVS_OWNERDATA style that designates the control as a virtual list view control.

New styles in report mode

In addition to the standard list view styles that you can use when creating the list view, the list view control supports a number of *extended styles*. This rather unfortunate term doesn't refer to the extended styles field in the *CreateWindowsEx* function. Instead, two messages, LVM_GETEXTENDEDLISTVIEWSTYLE and LVM_SETEXTENDEDLISTVIEWSTYLE, are used to get and set these extended list view styles. The extended styles supported by Windows CE are listed below.

- *LVS_EX_CHECKBOXES* The control places check boxes next to each item in the control.
- *LVS_EX_HEADERDRAGDROP* Allows headers to be rearranged by the user using drag and drop.

- *LVS_EX_GRIDLINES* The control draws grid lines around the items in report mode.
- *LVS_EX_SUBITEMIMAGES* The control displays images in the subitem columns in report mode.
- *LVS_EX_FULLROWSELECT* The control highlights the item's entire row in report mode when that item is selected.

Aside from the LVS_EX_CHECKBOXES extended style, which works in all display modes, these new styles all affect the actions of the list view when in report mode. The effort here has clearly been to make the list view control an excellent control for displaying large lists of data.

Note that the list view control under Windows CE doesn't support other extended list view styles, such as LVS_EX_INFOTIP, LVS_EX_ONECLICKACTIVATE, LVS_EX_TWOCLICKACTIVATE, LVS_EX_TRACKSELECT, LVS_EX_REGIONAL, or LVS_EX_FLATSB, supported in some versions of the common control library.

Virtual list view

The virtual list view mode of the list view control is a huge help for Windows CE devices. In this mode, the list view control tracks only the selection and focus state of the items. The application maintains all the other data for the items in the control. This mode is handy for two reasons. First, virtual list view controls are fast. The initialization of the control is almost instantaneous because all that's required is that you set the number of items in the control. The list view control also gives you hints about what items it will be looking for in the near term. This allows applications to cache necessary data in RAM and leave the remainder of the data in a database or file. Without a virtual list view, an application would have to load an entire database or list of items in the list view when it's initialized. With the virtual list view, the application loads only what the control requires to display at any one time.

The second advantage of the virtual list view is RAM savings. Because the virtual list view control maintains little information on each item, the control doesn't keep a huge data array in RAM to support the data. The application manages what data is in RAM with some help from the virtual list view's cache hint mechanism.

The virtual list view has some limitations. The LVS_OWNERDATA style that designates a virtual list view can't be set or cleared after the control has been created. Also, virtual list views don't support drag and drop in large icon or small icon mode. A virtual list view defaults to LVS_AUTOARRANGE style and the LVM_SETITEMPOSITION message isn't supported. Also, the sort styles LVS_SORTASCENDING and LVS_SORTDESCENDING aren't supported. Even so, the ability to store large lists of items is handy.

To implement a virtual list view, an application needs to create a list view control with an LVS_OWNERDATA style and handle three notifications—LVN_GETDISPINFO, LVN_ODCACHEHINT, and LVN_ODFINDITEM. The LVN_GETDISPINFO notification

should be familiar to those of you who have programmed list view controls before. It has always been sent when the list view control needed information to display an item. In the virtual list view, it's used in a similar manner but the notification is sent to gather all the information about every item in the control.

The virtual list view lets you know what data items it needs using the LVN_ODCACHEHINT notification. This notification passes the starting and ending index of items that the control expects to make use of in the near term. An application can take its cue from this set of numbers to load a cache of those items so that they can be quickly accessed. The hints tend to be requests for the items about to be displayed in the control. Because the number of items can change from view to view in the control, it's helpful that the control tracks this instead of having the application guess which items are going to be needed. Because the control often also needs information about the first and last pages of items, it also helps to cache them so that the frequent requests for those items don't clear the main cache of items that will be needed again soon.

The final notification necessary to manage a virtual list view is the LVN_ODFINDITEM notification. This is sent by the control when it needs to locate an item in response to a key press or in response to an LVM_FINDITEM message.

The LView Example Program

The LView program demonstrates a virtual list view control. The program creates a list view control that displays the contents of a fictional database. A picture of the LView window is shown in Figure 5-6 while the LView code is shown in Figure 5-7.

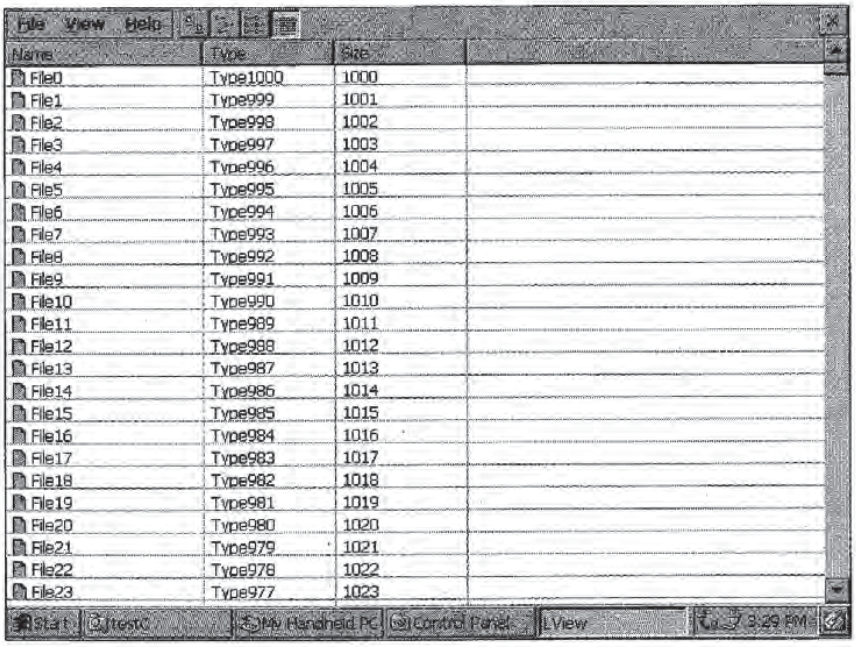

Figure 5-6. *The LView window.*

LView.rc

```
//=============================================================================
// Resource file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//=============================================================================
#include "windows.h"
#include "LView.h"                      // Program-specific stuff

//-----------------------------------------------------------------------------
// Icons and bitmaps
//
ID_ICON         ICON    "lview.ico"         // Program icon
docicon         ICON    "docicon.ico"       // Document icon
//-----------------------------------------------------------------------------
// Menu
//
ID_MENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit",                       IDM_EXIT
    END
    POPUP "&View"
    BEGIN
        MENUITEM "&Lar&ge Icons",               IDC_LICON
        MENUITEM "&S&mall Icons",               IDC_SICON
        MENUITEM "&List",                       IDC_LIST
        MENUITEM "&Details",                    IDC_RPT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About",                      IDM_ABOUT
    END
END
//-----------------------------------------------------------------------------
// About box dialog template
//
aboutbox DIALOG discardable 10, 10, 160, 40
STYLE  WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU | DS_CENTER |
       DS_MODALFRAME
CAPTION "About"
BEGIN
    ICON  ID_ICON,                  -1,   5,   5,  10,  10
```

Figure 5-7. *The LView program.* *(continued)*

Figure 5-7. *continued*

```
    LTEXT "LView - Written for the book Programming Windows \
           CE Copyright 1998 Douglas Boling"
                                        -1,  40,   5, 110,  30
END
```

LView.h

```
//------------------------------------------------------------------------
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//------------------------------------------------------------------------
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))

//------------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                                 // Structure associates
    UINT Code;                                      // messages
                                                    // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                                  // Structure associates
    UINT Code;                                      // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);         // function.
};

//------------------------------------------------------------------------
// Generic defines used by application
#define  IDC_CMDBAR   1                             // Command bar ID
#define  IDC_LISTVIEW 2                             // ID for report list box

#define  ID_ICON              10                    // Icon resource ID
#define  ID_MENU              11                    // Main menu resource ID

// Menu item and Command bar IDs
#define  IDM_EXIT             101

#define  IDC_LICON            111
#define  IDC_SICON            112
#define  IDC_LIST             113
#define  IDC_RPT              114
```

```
#define  IDM_ABOUT              120
#define  VIEW_BMPS              (VIEW_NEWFOLDER+1) // Number of BMPS in
                                                   // view list
//------------------------------------------------------------------------
// Program-specific structures
//

// Defines for simulated database
typedef struct {
    TCHAR szName[32];
    TCHAR szType[32];
    INT nSize;
    INT nImage;
    INT nState;
} LVDATAITEM;
typedef LVDATAITEM *PLVDATAITEM;

//------------------------------------------------------------------------
// Function prototypes
//
// Cache functions
PLVDATAITEM GetItemData (INT nItem);
void InitDatabase (void);
void FlushMainCache (void);
void FlushEndCaches (void);
INT LoadTopCache (void);
INT LoadBotCache (void);
INT LoadMainCache (INT nStart, INT nEnd);

// Database functions
void InitDatabase (void);
PLVDATAITEM GetDatabaseItem (INT nItem);
INT SetDatabaseItem (INT nItem, PLVDATAITEM pIn);
PLVDATAITEM GetItemData (INT nItem);
INT AddItem (HWND, INT, LPTSTR, LPTSTR, INT);

int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Listview compare callback
int CALLBACK CompareLV (LPARAM, LPARAM, LPARAM);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);
```

(continued)

Figure 5-7. *continued*

```
// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoSizeMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoNotifyMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoCommandMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);

// Command functions
LPARAM DoMainCommandExit (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandChView (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandAbout (HWND, WORD, HWND, WORD);

// Dialog procedures
BOOL CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM);
```

LView.c

```
//=====================================================================
// LView - ListView control demonstration
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//=====================================================================
#include <windows.h>                    // For all that Windows stuff
#include <commctrl.h>                   // Command bar includes
#include "LView.h"                      // Program-specific stuff
//---------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("LView");
HINSTANCE hInst;                        // Program instance handle
HWND hMain;

//
// Data for simulated database
//
#define LVCNT 2000
LVDATAITEM lvdatabase[LVCNT];

// Defines and data for list view control cache
#define  CACHESIZE      100
#define  TOPCACHESIZE   100
#define  BOTCACHESIZE   100
```

```
INT nCacheItemStart = 0, nCacheSize = 0;
LVDATAITEM lvdiCache[CACHESIZE];
LVDATAITEM lvdiTopCache[TOPCACHESIZE];
LVDATAITEM lvdiBotCache[BOTCACHESIZE];

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_SIZE, DoSizeMain,
    WM_NOTIFY, DoNotifyMain,
    WM_COMMAND, DoCommandMain,
    WM_DESTROY, DoDestroyMain,
};
// Command message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDM_EXIT, DoMainCommandExit,
    IDC_LICON, DoMainCommandChView,
    IDC_SICON, DoMainCommandChView,
    IDC_LIST, DoMainCommandChView,
    IDC_RPT, DoMainCommandChView,
    IDM_ABOUT, DoMainCommandAbout,
};
// Standard file bar button structure
const TBBUTTON tbCBCmboBtns[] = {
//  BitmapIndex      Command    State        Style        UserData String
    {0,              0,         0,           TBSTYLE_SEP,        0,  0},
    {VIEW_LARGEICONS, IDC_LICON, TBSTATE_ENABLED,
                                             TBSTYLE_CHECKGROUP, 0,  0},
    {VIEW_SMALLICONS, IDC_SICON, TBSTATE_ENABLED,
                                             TBSTYLE_CHECKGROUP, 0,  0},
    {VIEW_LIST,      IDC_LIST,  TBSTATE_ENABLED,
                                             TBSTYLE_CHECKGROUP, 0,  0},
    {VIEW_DETAILS,   IDC_RPT,   TBSTATE_ENABLED | TBSTATE_CHECKED,
                                             TBSTYLE_CHECKGROUP, 0,  0}
};
//=========================================================================
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    HWND hwndMain;
    int rc = 0;
```

(continued)

Figure 5-7. *continued*

```
    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

    hMain = hwndMain;
    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//------------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS     wc;
    INITCOMMONCONTROLSEX icex;

    // Register application main window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = MainWndProc;                   // Callback function
    wc.cbClsExtra = 0;                              // Extra class data
    wc.cbWndExtra = 0;                              // Extra window data
    wc.hInstance = hInstance;                       // Owner handle
    wc.hIcon = NULL;                                // Application icon
    wc.hCursor = NULL;                              // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                        // Menu name
    wc.lpszClassName = szAppName;                   // Window class name

    if (RegisterClass (&wc) == 0) return 1;

    // Load the command bar common control class.
    icex.dwSize = sizeof (INITCOMMONCONTROLSEX);
    icex.dwICC = ICC_LISTVIEW_CLASSES;
    InitCommonControlsEx (&icex);
```

```
    // Initialize the fictional database.
    InitDatabase ();
    return 0;
}
//-----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow){
    HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.
    hWnd = CreateWindow (szAppName,             // Window class
                         TEXT ("LView"),        // Window title
                         WS_VISIBLE,            // Style flags
                         CW_USEDEFAULT,         // x position
                         CW_USEDEFAULT,         // y position
                         CW_USEDEFAULT,         // Initial width
                         CW_USEDEFAULT,         // Initial height
                         NULL,                  // Parent
                         NULL,                  // Menu, must be null
                         hInstance,             // Application instance
                         NULL);                 // Pointer to create
                                                // parameters
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//-----------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {

    // Flush caches used with list view control.
    FlushMainCache ();
    FlushEndCaches ();
    return nDefRC;
```

(continued)

Figure 5-7. *continued*

```
}
//------------------------------------------------------------------------
// Message-handling procedures for MainWindow
//------------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT wMsg, WPARAM wParam,
                             LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc(hWnd, wMsg, wParam, lParam);
}
//------------------------------------------------------------------------
// DoCreateMain - Process WM_CREATE message for window.
//
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB, hwndLV;
    INT i, nHeight;
    LPCREATESTRUCT lpcs;
    HIMAGELIST himlLarge, himlSmall;
    HICON hIcon;

    // Convert lParam into pointer to create structure.
    lpcs = (LPCREATESTRUCT) lParam;

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);

    // Insert a menu.
    CommandBar_InsertMenubar (hwndCB, hInst, ID_MENU, 0);

    // Add bitmap list followed by buttons.
    CommandBar_AddBitmap (hwndCB, HINST_COMMCTRL, IDB_VIEW_SMALL_COLOR,
                          VIEW_BMPS, 0, 0);
    CommandBar_AddButtons (hwndCB, dim(tbCBCmboBtns), tbCBCmboBtns);
```

```
// Add exit button to command bar.
CommandBar_AddAdornments (hwndCB, 0, 0);
nHeight = CommandBar_Height (hwndCB);
//
// Create the list view control.
//
hwndLV = CreateWindowEx (0, WC_LISTVIEW, TEXT (""),
                         LVS_REPORT | LVS_SINGLESEL |
                         LVS_OWNERDATA | WS_VISIBLE | WS_CHILD |
                         WS_VSCROLL, 0, nHeight, lpcs->cx,
                         lpcs->cy - nHeight, hWnd,
                         (HMENU)IDC_LISTVIEW,
                         lpcs->hInstance, NULL);
// Destroy frame if window not created.
if (!IsWindow (hwndLV)) {
    DestroyWindow (hWnd);
    return 0;
}
// Add columns.
{
    LVCOLUMN lvc;

    lvc.mask = LVCF_TEXT | LVCF_WIDTH | LVCF_FMT | LVCF_SUBITEM;
    lvc.fmt = LVCFMT_LEFT;
    lvc.cx = 150;
    lvc.pszText = TEXT ("Name");
    lvc.iSubItem = 0;
    SendMessage (hwndLV, LVM_INSERTCOLUMN, 0, (LPARAM)&lvc);

    lvc.mask |= LVCF_SUBITEM;
    lvc.pszText = TEXT ("Type");
    lvc.cx = 100;
    lvc.iSubItem = 1;
    SendMessage (hwndLV, LVM_INSERTCOLUMN, 1, (LPARAM)&lvc);

    lvc.mask |= LVCF_SUBITEM;
    lvc.pszText = TEXT ("Size");
    lvc.cx = 100;
    lvc.iSubItem = 2;
    SendMessage (hwndLV, LVM_INSERTCOLUMN, 2, (LPARAM)&lvc);
}
// Add items.
ListView_SetItemCount (hwndLV, LVCNI);
LoadTopCache ();
LoadBotCache ();
```

(continued)

Figure 5-7. *continued*

```
    // Create image list control for bitmaps for minimized bands.
    i = GetSystemMetrics (SM_CXICON);
    himlLarge = ImageList_Create(i, i, ILC_COLOR, 2, 0);
    i = GetSystemMetrics (SM_CXSMICON);
    himlSmall = ImageList_Create(i, i, ILC_COLOR, 2, 0);

    // Load large and small icons into their respective image lists.
    hIcon = LoadIcon (hInst, TEXT ("DocIcon"));
    i = ImageList_AddIcon (himlLarge, hIcon);

    hIcon = LoadImage (hInst, TEXT ("DocIcon"), IMAGE_ICON, 16, 16,
                       LR_DEFAULTCOLOR);
    ImageList_AddIcon (himlSmall, hIcon);

    ListView_SetImageList (hwndLV, himlLarge, LVSIL_NORMAL);
    ListView_SetImageList (hwndLV, himlSmall, LVSIL_SMALL);

    // Set cool new styles.
    ListView_SetExtendedListViewStyle (hwndLV, LVS_EX_GRIDLINES |
                                       LVS_EX_HEADERDRAGDROP |
                                       LVS_EX_FULLROWSELECT);
    return 0;
}
//------------------------------------------------------------------------
// DoSizeMain - Process WM_SIZE message for window.
//
LRESULT DoSizeMain (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam){
    HWND hwndLV;
    RECT rect;

    hwndLV = GetDlgItem (hWnd, IDC_LISTVIEW);

    // Adjust the size of the client rect to take into account
    // the command bar height.
    GetClientRect (hWnd, &rect);
    rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    SetWindowPos (hwndLV, NULL, rect.left, rect.top,
                  rect.right - rect.left, rect.bottom - rect.top,
                  SWP_NOZORDER);
    return 0;
}
//------------------------------------------------------------------------
// DoNotifyMain - Process WM_NOTIFY message for window.
//
```

```
LRESULT DoNotifyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    int idItem;
    LPNMHDR    pnmh;
    LPNMLISTVIEW pnmlv;
    NMLVDISPINFO *pLVdi;
    PLVDATAITEM pdi;     // Pointer to data
    LPNMLVCACHEHINT pLVch;
    HWND hwndLV;

    // Parse the parameters.
    idItem = (int) wParam;
    pnmh = (LPNMHDR)lParam;
    hwndLV = pnmh->hwndFrom;

    if (idItem == IDC_LISTVIEW) {
        pnmlv = (LPNMLISTVIEW)lParam;

        switch (pnmh->code) {
        case LVN_GETDISPINFO:
            pLVdi = (NMLVDISPINFO *)lParam;

            // Get a pointer to the data either from the cache
            // or from the actual database.
            pdi = GetItemData (pLVdi->item.iItem);

            if (pLVdi->item.mask & LVIF_IMAGE)
                pLVdi->item.iImage = pdi->nImage;

            if (pLVdi->item.mask & LVIF_PARAM)
                pLVdi->item.lParam = 0;

            if (pLVdi->item.mask & LVIF_STATE)
                pLVdi->item.state = pdi->nState;

            if (pLVdi->item.mask & LVIF_TEXT) {
                switch (pLVdi->item.iSubItem) {
                case 0:
                    lstrcpy (pLVdi->item.pszText, pdi->szName);
                    break;
                case 1:
                    lstrcpy (pLVdi->item.pszText, pdi->szType);
                    break;
                case 2:
                    wsprintf (pLVdi->item.pszText, TEXT ("%d"),
                              pdi->nSize);
```

(continued)

Figure 5-7. *continued*

```
                    break;
                }
            }
            break;

        case LVN_ODCACHEHINT:
            pLVch = (LPNMLVCACHEHINT)lParam;
            LoadMainCache (pLVch->iFrom, pLVch->iTo);
            break;

        case LVN_ODFINDITEM:
            // We should do a reverse look up here to see if
            // an item exists for the text passed.
            return -1;
        }
    }
    return 0;
}
//-----------------------------------------------------------------------
// DoCommandMain - Process WM_COMMAND message for window.
//
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)
            return (*MainCommandItems[i].Fxn)(hWnd, idItem, hwndCtl,
                                              wNotifyCode);
    }
    return 0;
}
//-----------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
```

```
}
//=============================================================================
// Command handler routines
//-----------------------------------------------------------------------------
// DoMainCommandExit - Process Program Exit command.
//
LPARAM DoMainCommandExit (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {

    SendMessage (hWnd, WM_CLOSE, 0, 0);
    return 0;
}
//-----------------------------------------------------------------------------
// DoMainCommandChView - Process View xxx command.
//
LPARAM DoMainCommandChView (HWND hWnd, WORD idItem, HWND hwndCtl,
                            WORD wNotifyCode) {
    HWND hwndLV;
    LONG lStyle;

    hwndLV = GetDlgItem (hWnd, IDC_LISTVIEW);

    lStyle = GetWindowLong (hwndLV, GWL_STYLE);
    lStyle &= ~LVS_TYPEMASK;

    switch (idItem) {
    case IDC_LICON:
        lStyle |= LVS_ICON;
        break;
    case IDC_SICON:
        lStyle |= LVS_SMALLICON;
        break;
    case IDC_LIST:
        lStyle |= LVS_LIST;
        break;
    case IDC_RPT:
        lStyle |= LVS_REPORT;
        break;
    }
    SetWindowLong (hwndLV, GWL_STYLE, lStyle);
    return 0;
}
//-----------------------------------------------------------------------------
// DoMainCommandAbout - Process the Help | About menu command.
//
LPARAM DoMainCommandAbout(HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {
```

(continued)

Figure 5-7. *continued*

```
    // Use DialogBox to create modal dialog box.
    DialogBox (hInst, TEXT ("aboutbox"), hWnd, AboutDlgProc);
    return 0;
}
//=======================================================================
// About Dialog procedure
//
BOOL CALLBACK AboutDlgProc(HWND hWnd, UINT wMsg, WPARAM wParam,
                           LPARAM lParam) {

    switch (wMsg) {
        case WM_COMMAND:
            switch (LOWORD (wParam)) {
                case IDOK:
                case IDCANCEL:
                    EndDialog (hWnd, 0);
                    return TRUE;
            }
        break;
    }
    return FALSE;
}
//=======================================================================
// Helper routines for list view control management
//-----------------------------------------------------------------------
// AddItem - Add an item to the list view control.
//
INT AddItem (HWND hwndCtl, INT nItem, LPTSTR pszName, LPTSTR pszType,
             INT nSize) {
    LVITEM lvi;
    TCHAR szTmp[40];

    lvi.mask = LVIF_TEXT | LVIF_IMAGE | LVIF_PARAM;
    lvi.iItem = nItem;
    lvi.iSubItem = 0;
    lvi.pszText = pszName;
    lvi.iImage = 0;
    lvi.lParam = nItem;
    SendMessage (hwndCtl, LVM_INSERTITEM, 0, (LPARAM)&lvi);

    lvi.mask = LVIF_TEXT;
    lvi.iItem = nItem;
    lvi.iSubItem = 1;
    lvi.pszText = pszType;
    SendMessage (hwndCtl, LVM_SETITEM, 0, (LPARAM)&lvi);
```

```
    wsprintf (szTmp, TEXT ("%d"), nSize);
    lvi.mask = LVIF_TEXT;
    lvi.iItem = nItem;
    lvi.iSubItem = 2;
    lvi.pszText = szTmp;
    SendMessage (hwndCtl, LVM_SETITEM, 0, (LPARAM)&lvi);

    return 0;
}
//-----------------------------------------------------------------------
// GetItemData - This routine returns a pointer to the data.  It
// first checks the caches before calling directly to the database.
//
PLVDATAITEM GetItemData (INT nItem) {
    INT nCacheIndex;
    PLVDATAITEM pdi;

    // See if it's in the top cache.
    if (nItem < TOPCACHESIZE) {

        nCacheIndex = nItem;
        pdi = &lvdiTopCache[nCacheIndex];
    }

    // See if it's in the bottom cache.
    else if (nItem > LVCNT - BOTCACHESIZE) {

        nCacheIndex = nItem - (LVCNT - BOTCACHESIZE);
        pdi = &lvdiBotCache[nCacheIndex];
    }
    // See if item's in the main cache.
    else if ((nItem >= nCacheItemStart) &&
             (nItem < nCacheItemStart + nCacheSize)) {

        nCacheIndex = nItem - nCacheItemStart;
        pdi = &lvdiCache[nCacheIndex];
    }

    // Otherwise it's not in any cache.
    else
        pdi = GetDatabaseItem (nItem);

    return pdi;
}
```

(continued)

Figure 5-7. *continued*

```
//-----------------------------------------------------------------------
INT LoadACache (PLVDATAITEM pCache, INT nStart, INT nSize) {
    PLVDATAITEM pdi;
    INT i;

    for (i = 0; i < nSize; i++) {
        // Get a pointer to the data.
        pdi = GetDatabaseItem (nStart+i);

        // Save the data in the cache.
        lstrcpy (pCache[i].szName, pdi->szName);
        lstrcpy (pCache[i].szType, pdi->szType);
        pCache[i].nSize = pdi->nSize;
        pCache[i].nImage = pdi->nImage;
        pCache[i].nState = pdi->nState;
    }
    return 0;
}
//-----------------------------------------------------------------------
// LoadMainCache - This routine loads the hint cache.  If the
// recommended range is already in the top or bottom caches, the range
// is adjusted to grab items outside the end caches.
//
// The logic expects the total number of items to be greater than the
// size of the start and end caches.
//
INT LoadMainCache (INT nStart, INT nEnd) {
    INT nOverlap;

    // Size the hint range to fit the cache.
    if (nEnd - nStart > CACHESIZE)
        nEnd = nStart + CACHESIZE;

    // See if end of hint in bottom cache.
    if (nEnd > LVCNT - BOTCACHESIZE) {

        // If completely in bottom cache, keep old data.
        if (nStart > LVCNT - BOTCACHESIZE)
            return 0;

        // If partial overlap, adjust end points to get data just
        // above the bottom cache.
        nOverlap = nEnd - (LVCNT - BOTCACHESIZE);
        nEnd = LVCNT - BOTCACHESIZE - 1;
        if (nStart - nOverlap < TOPCACHESIZE)
            nStart = TOPCACHESIZE;
```

```
        else
            nStart -= nOverlap;
    }
    // See if start of hint in top cache.
    if (nStart < TOPCACHESIZE) {

        // If completely in top cache, keep old data.
        if (nEnd < TOPCACHESIZE)
            return 0;

        // Adjust the starting value to just beyond top cache end.
        nOverlap = TOPCACHESIZE - nStart;
        nStart = TOPCACHESIZE;
        if (nOverlap + nEnd > (LVCNT - BOTCACHESIZE))
            nEnd = LVCNT - BOTCACHESIZE;
        else
            nEnd += nOverlap;
    }
    // If hint already completely contained in the cache, exit.
    if ((nStart >= nCacheItemStart) &&
        (nEnd < nCacheItemStart + nCacheSize))
        return 0;

    // Flush old data in cache.  We should really be smart here to
    // see whether part of the data is already in the cache.
    FlushMainCache ();

    // Load the new data.
    nCacheSize = nEnd - nStart;
    nCacheItemStart = nStart;
    LoadACache (lvdiCache, nStart, nCacheSize);
    return 0;
}
//------------------------------------------------------------------------
INT LoadTopCache (void) {

    LoadACache (lvdiTopCache, 0, TOPCACHESIZE);
    return 0;
}
//------------------------------------------------------------------------
INT LoadBotCache (void) {

    LoadACache (lvdiBotCache, LVCNT - BOTCACHESIZE, BOTCACHESIZE);
    return 0;
}
```

(continued)

Figure 5-7. *continued*

```
//-----------------------------------------------------------------------
void FlushMainCache (void) {
    INT i;

    // Send the data back to the database.
    for (i = 0; i < nCacheSize; i++) {
        SetDatabaseItem (nCacheItemStart+i, &lvdiCache[i]);
    }
    return;
}
//-----------------------------------------------------------------------
void FlushEndCaches (void) {
    INT i;

    // Flush the top cache.
    for (i = 0; i < TOPCACHESIZE; i++) {
        SetDatabaseItem (i, &lvdiCache[i]);
    }
    // Flush the bottom cache.
    for (i = 0; i < BOTCACHESIZE; i++) {
        SetDatabaseItem (LVCNT - BOTCACHESIZE + i, &lvdiBotCache[i]);
    }
    return;
}
//=======================================================================
// Code for fictional database to be displayed in the list view control
//
//-----------------------------------------------------------------------
// InitDatabaseItem - Copy an item into the database.
//
INT InitDatabaseItem (INT nItem, LPTSTR pszName, LPTSTR pszType,
                      INT nSize) {

    lstrcpy (lvdatabase[nItem].szName, pszName);
    lstrcpy (lvdatabase[nItem].szType, pszType);
    lvdatabase[nItem].nSize = nSize;
    lvdatabase[nItem].nImage = 0;
    lvdatabase[nItem].nState = 0;
    return 0;
}
//-----------------------------------------------------------------------
// InitDatabase - Create fictional data for fictional database.
//
```

```
void InitDatabase (void) {
    TCHAR szName[64];
    TCHAR szType[64];
    HCURSOR hOldCur;
    INT i;

    hOldCur = SetCursor (LoadCursor (NULL, IDC_WAIT));

    for (i = 0; i < LVCNT; i++) {
        wsprintf (szName, TEXT ("File%d"), i);
        wsprintf (szType, TEXT ("Type%d"), 1000 - i);
        InitDatabaseItem (i, szName, szType, i+1000);
    }
    SetCursor (hOldCur);
    return;
}
//-----------------------------------------------------------------------
// GetDatabaseItem - Return a pointer to data in the database.
//
PLVDATAITEM GetDatabaseItem (INT nItem) {

    // Normally, this would be more work. But since
    // we have only a simulated data store, the
    // code is trivial.
    return &lvdatabase[nItem];
}
//-----------------------------------------------------------------------
// SetDatabaseItem - Copy data from list view control back into database.
//
INT SetDatabaseItem (INT nItem, PLVDATAITEM pIn) {

    lstrcpy (lvdatabase[nItem].szName, pIn->szName);
    lstrcpy (lvdatabase[nItem].szType, pIn->szType);
    lvdatabase[nItem].nSize = pIn->nSize;
    lvdatabase[nItem].nImage = pIn->nImage;
    lvdatabase[nItem].nState = pIn->nState;
    return 0;
}
```

Notice that the size for the database is set to 2000 items by default. Even with this large number, the performance of the list view control is quite acceptable. Most of the brief application startup time is taken up not by initializing the list view control, but just by filling in the dummy database. Support for the virtual list view is centered on the *OnNotifyMain* routine.

Data for each item is supplied to the list view control through responses to the LVN_GETDISPINFO notification. The flags in the mask field of the LVDISPINFO determine exactly what element of the item is being requested. The code that handles the notification simply requests the item data from the cache and fills in the requested fields.

The cache implemented by LView uses three separate buffers. Two of the buffers are initialized with the first and last 100 items from the database. The third 100-item cache, referred to as the main cache, is loaded using the hints passed by the list view control.

The routine that reads the data from the cache is located in the *GetItemData* routine. That routine uses the index value of the requested item to see whether the data is in the top or bottom caches, and if not, whether it's in the main cache. If the data isn't in one of the caches, a call to *GetDatabaseItem* is made to read the data directly from the dummy database.

The routine that handles the cache hints from the list view control is *LoadMainCache*. This routine is called when the program receives a LVN_ODCACHEHINT notification. The routine takes two parameters, the starting and ending values of the hint passed by the notification. The routine first checks to see if the range of items in the hint lies in the two end caches that store data from the top and bottom of the database. If the range does lie in one of the end caches, the hint is ignored and the main cache is left unchanged. If the hint range isn't in either end cache and isn't already in the current main cache, the main cache is flushed to send any updated information back into the database. The cache is then loaded with data from the database from the range of items indicated by the hint.

The cache hint notifications sent by the list view control aren't necessarily intelligent. The control sends a request for a range of one item if that item is double-clicked by the user. The cache management code should always check to see whether the requested data is already in the cache before flushing and reloading the cache based on a single hint. The cache strategy you use, and the effort you must make to optimize it, of course depends on the access speed of the real data.

OTHER COMMON CONTROLS

Windows CE supports a number of other common controls available under Windows 98 and Windows NT. Most of these controls are supported completely within the limits of the capability of Windows CE. For example, while the tab control supports vertical tabs, Windows CE supports vertical text only on systems that support TrueType fonts. For other systems, including the Palm-size PC, the text in the tabs must be manually generated by the Windows CE application by rotating bitmap images of each letter. Frankly, it's probably much easier to devise a dialog box that doesn't need vertical tabs. Short descriptions of the other supported common controls follow.

The status bar control

The status bar is carried over unchanged from the desktop versions of Windows. The only difference is that under Windows CE, the SBARS_SIZEGRIP style that created a gripper area on the right end of the status bar has no meaning because users can't size Windows CE windows.

The tab control

The tab control is fully supported, the above-mentioned vertical text limitation not withstanding. But because the stylus can't hover over a tab, the TCS_HOTTRACK style that highlighted tabs under the cursor isn't supported. The TCS_EX_REGISTERDROP extended style is also not supported.

The trackbar control

The trackbar control gains the capacity for two "buddy" controls that are automatically updated with the trackbar value. The trackbar also supports the custom draw service, providing separate item drawing indications for the channel, the thumb, and the tic marks.

The progress bar control

The progress bar includes the latest support for vertical progress bars and 32-bit ranges. This control also supports the new smooth progression instead of moving the progress indicator in discrete chunks.

The up-down control

The up-down control under Windows CE only supports edit controls for its buddy control.

The toolbar control

The Windows CE toolbar supports tooltips differently from the way tool tips are supported by the desktop versions of this control. You add toolbar support for tool tips in Windows CE the same way you do for the command bar, by passing a pointer to a permanently allocated array of strings. The toolbar also supports the transparent and flat styles that are supported by the command bar.

The tree view control

The tree view control supports two new styles recently added to the tree view common control: TVS_CHECKBOXES and TVS_SINGLESEL. The TVS_CHECKBOXES style places a check box adjacent to each item in the control. The TVS_SINGLESEL style causes a previously expanded item to close up when a new item is selected. The tree view control also supports the custom draw service. The tree view control doesn't support the TVS_TRACKSELECT style, which allows you to highlight an item when the cursor hovers over it.

UNSUPPORTED COMMON CONTROLS

Windows CE doesn't support four common controls seen under other versions of Windows. The animation control, the drag list control, the hot key control, and, sadly, the rich edit control are all unsupported. Animation would be hard to support given the slower processors often seen running Windows CE. The hot key control is problematic in that keyboard layouts and key labels, standardized on the PC, vary dramatically on the different hardware that runs Windows CE. And the drag list control isn't that big a loss, given the improved power of the report style of the list view control.

The rich edit control is another story. The lack of an edit control that can contain multiple fonts and paragraph formatting is a noticeable gap in the Windows CE shell. Applications needing this functionality are forced to implement independent, and mutually incompatible, solutions. Let's hope the rich edit control is supported under future versions of Windows CE.

Windows CE supports fairly completely the common control library seen under other versions of Windows. The date and time picker, month calendar, and command bar are a great help given the target audience of Windows CE devices.

I've spent a fair amount of time in the past few chapters looking at the building blocks of applications. Now it's time to turn to the operating system itself. Over the next three chapters, I'll cover memory management, files and databases, and processes and threads. These chapters are aimed at the core of the Windows CE operating system.

Chapter 6

Memory Management

If you have an overriding concern when you're writing a Microsoft Windows CE program, it should be dealing with memory. A Windows CE machine might have only 1 or 2 MB of RAM. This is a tiny amount compared to that of a standard personal computer, which can range somewhere between 16 and 64 MB of RAM. In fact, memory on a Windows CE machine is so scarce that it's often necessary to write programs that conserve memory even to the point of sacrificing the overall performance of the application.

Fortunately, although the amount of memory is small in a Windows CE system, the functions available for managing that memory are fairly complete. Windows CE implements almost the full Win32 memory management API available under Microsoft Windows NT and Microsoft Windows 98. Windows CE supports virtual memory allocations, local and separate heaps, and even memory-mapped files.

Like Windows NT, Windows CE supports a 32-bit flat address space with memory protection between applications. But because Windows CE was designed for different environments, its underlying memory architecture is different from that for Windows NT. These differences can affect how you design a Windows CE application. In this chapter, I'll cover the basic memory architecture of Windows CE. I'll also cover the different types of memory allocation available to Windows CE programs and how to use each memory type to minimize your application's memory footprint.

MEMORY BASICS

As with all computers, systems running Windows CE have both ROM (read only memory) and RAM (random access memory). Under Windows CE, however, both ROM and RAM are used somewhat differently than they are in a standard personal computer.

About RAM

The RAM in a Windows CE system is divided into two areas: *program memory* and *object store*. The object store can be considered something like a permanent virtual RAM disk. Unlike the old virtual RAM disks on a PC, the object store retains the files stored in it even if the system is turned off.[1] This is the reason Windows CE systems such as the Handheld PC and the Palm-size PC each have a battery and a backup battery. When the user replaces the main batteries, the backup battery's job is to provide power to the RAM to retain the files in the object store. Even when the user hits the reset button, the Windows CE kernel starts up looking for a previously created object store in RAM and uses that store if it finds one.

The other area of the RAM is devoted to the program memory. Program memory is used like the RAM in personal computers. It stores the heaps and stacks for the applications that are running. The boundary between the object store and the program RAM is movable. The user can move the dividing line between object store and program RAM using the System control panel applet. Under low-memory conditions, the system will ask the user for permission to take some object store RAM to use as program RAM to satisfy an application's demand for more RAM.

About ROM

In a personal computer, the ROM is used to store the BIOS (basic input output system) and is typically 64–128 KB. In a Windows CE system, the ROM can range from 4 to 16 MB and stores the entire operating system, as well as the applications that are bundled with the system. In this sense, the ROM in a Windows CE system is like a small, read-only hard disk.

In a Windows CE system, ROM-based programs can be designated as Execute in Place (XIP). That is, they're executed directly from the ROM instead of being loaded into program RAM and then executed. This is a huge advantage for small systems in two ways. The fact that the code is executed directly from ROM means that the program code doesn't take up valuable program RAM. Also, since the program doesn't

1. On mobile systems like the H/PC and the Palm-size PC, the system is never really off. When the user presses the Off button, the system enters a very low power suspended state.

have to be copied into RAM before it's launched, it takes less time to start an application. Programs that aren't in ROM but are contained in the object store or on a Flash memory storage card aren't executed in place; they're copied into the RAM and executed.

About Virtual Memory

Windows CE implements a virtual memory management system. In a virtual memory system, applications deal with virtual memory, which is a separate, imaginary address space that might not relate to the physical memory address space that's implemented by the hardware. The operating system uses the memory management unit of the microprocessor to translate virtual addresses to physical addresses in real time.

The key advantage of a virtual memory system can be seen in the complexity of the MS-DOS address space. Once demand for RAM exceeded the 640-KB limit of the original PC design, programmers had to deal with schemes such as *expanded* and *extended* memory to increase the available RAM. OS/2 1.*x* and Windows 3.0 replaced these schemes with a segment-based virtual memory system. Applications using virtual memory have no idea (nor should they care) where the actual physical memory resides, only that the memory is available. In these systems, the virtual memory was implemented in segments, resizable blocks of memory that ranged from 16 bytes to 64 KB in size. The 64-KB limit wasn't due to the segments themselves, but to the 16-bit nature of the Intel 80286 that was the basis for the segmented virtual memory system in Windows 3.*x* and OS/2 1.*x*.

Paged memory

The Intel 80386 supported segments larger than 64 KB, but when Microsoft and IBM began the design for OS/2 2.0, they chose to use a different virtual memory system, also supported by the 386, known as a *paged virtual memory system*. In a paged memory system, the smallest unit of memory the microprocessor manages is the *page*. For Windows NT and OS/2 2.0, the pages were set to 386's default page size of 4096 bytes. When an application accesses a page, the microprocessor translates the virtual address of the page to a physical page in ROM or RAM. A page can also be tagged so that accessing the page causes an exception. The operating system then determines whether the virtual page is valid and, if so, maps a physical page of memory to the virtual page.

Windows CE implements a paged virtual memory management system similar to the other Win32 operating systems, Windows NT and Windows 98. Under Windows CE, a page is either 1024 or 4096 bytes, depending on the microprocessor, with the 1-KB page size preferred by the Windows CE architects. This is a change from Windows NT, where page sizes are 4096 bytes for Intel microprocessors and 8192

bytes for the DEC Alpha. For the CPUs currently supported by Windows CE, the NEC 4100 series and the Hitachi SH3 use 1024-byte pages and the 486, the Phillips 3910, and Power PC 821 use 4096-byte pages.

Virtual pages can be in one of three states: *free*, *reserved*, or *committed*. A free page is, as it sounds, free and available to be allocated. A reserved page is a page that has been reserved so that its virtual address can't be allocated by the operating system or another thread in the process. A reserved page can't be used elsewhere, but it also can't be used by the application because it isn't mapped to physical memory. To be mapped, a page must be committed. A committed page has been reserved by an application and has been directly mapped to a physical address.

All that I've just explained is old hat to experienced Win32 programmers. The important thing for the Windows CE programmer is to learn how Windows CE changes the equation. While Windows CE implements most of the same memory API set of its bigger Win32 cousins, the underlying architecture of Windows CE does impact programs. To better understand how the API is affected, it helps to look at how Windows CE uses memory under the covers.

The Windows CE Address Space

In OS circles, much is made of the extent to which the operating system goes to protect one application's memory from other applications. Microsoft Windows 95 used a single address space that provided minimal protection between applications and the Windows operating system code. Windows NT, on the other hand, implements completely separate address spaces for each Win32 application, although old 16-bit applications under Windows NT do share a single address space.

Windows CE implements a single, 2-GB virtual address space for all applications, but the memory space of an application is protected so that it can't be accessed by another application. A diagram of the Windows CE virtual address space is shown in Figure 6-1. A little over half of the virtual address space is divided into thirty-three 32-MB *slots*. Each slot is assigned to a currently running process, with the lowest slot, slot 0, assigned to the active process. As Windows CE switches between processes, it remaps the address space to move the old process out of slot 0 and the new process into slot 0. This task is quickly accomplished by the OS by manipulating the page translation tables of the microprocessor.

The region of the address space above the 33 slots is reserved for the operating system and for mapping memory-mapped files. Like Windows NT, Windows CE also reserves the lowest 64-KB block of the address space from access by any process.

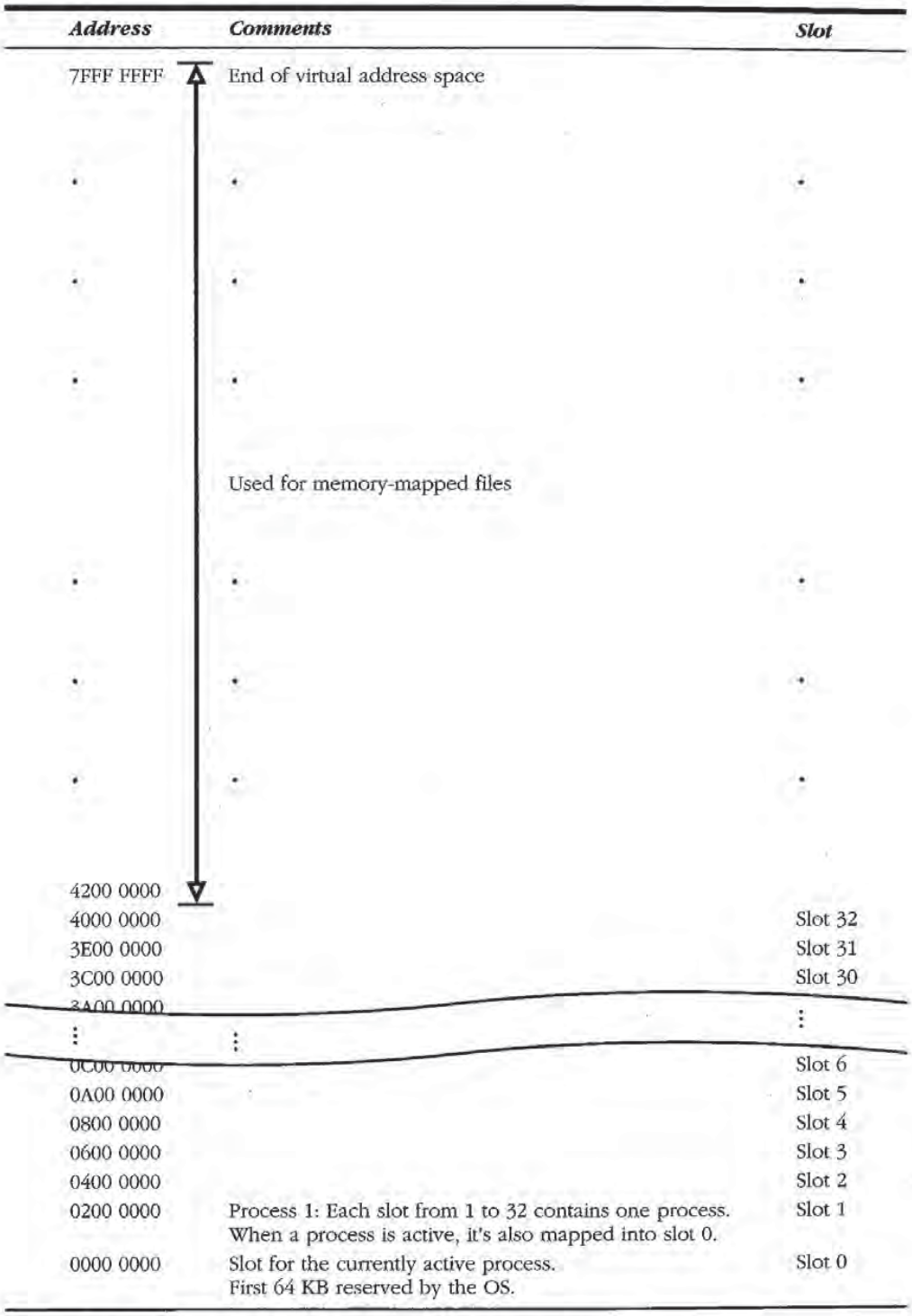

Figure 6-1. *A diagram of the Windows CE memory map.*

Querying the system memory

If an application knows the current memory state of the system, it can better manage the available resources. Windows CE implements both the Win32 *GetSystemInfo* and *GlobalMemoryStatus* functions. The *GetSystemInfo* function is prototyped below:

```
VOID GetSystemInfo (LPSYSTEM_INFO lpSystemInfo);
```

It's passed a pointer to a SYSTEM_INFO structure defined as

```
typedef struct {
    WORD wProcessorArchitecture;
    WORD wReserved;
    DWORD  dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD  dwActiveProcessorMask;
    DWORD  dwNumberOfProcessors;
    DWORD  dwProcessorType;
    DWORD  dwAllocationGranularity;
    WORD  wProcessorLevel;
    WORD  wProcessorRevision;
} SYSTEM_INFO;
```

The *wProcessorArchitecture* field identifies the type of microprocessor in the system. The value should be compared to the known constants defined in Winnt.h, such as PROCESSOR_ARCHITECTURE_INTEL. Windows CE has extended these constants to include PROCESSOR_ARCHITECTURE_ARM, PROCESSOR_ARCHITECTURE_SHx and others. Additional processor constants are added as net CPUs are supported by any of the Win32 operating systems. Skipping a few fields, the *dwProcessorType* field further narrows the microprocessor from a family to a specific microprocessor. Constants for the Hitachi SHx architecture include PROCESSOR_HITACHI_SH3 and PROCESSOR_HITACHI_SH4. The last two fields, *wProcessorLevel* and *wProcessorRevision*, further refine the CPU type. The *wProcessorLevel* field is similar to the *dwProcessorType* field in that it defines the specific microprocessor within a family. The *dwProcessorRevision* field tells you the model and the stepping level of the chip.

The *dwPageSize* field specifies the page size, in bytes, of the microprocessor. Knowing this value comes in handy when you're dealing directly with the virtual memory API, which I talk about shortly. The *lpMinimumApplicationAddress* and *lpMaximumApplicationAddress* fields specify the minimum and maximum virtual address available to the application. The *dwActiveProcessorMask* and *dwNumberOfProcessors* fields are used in Windows NT for systems that support more than one microprocessor. Since Windows CE supports only one microprocessor, you can ignore these fields. The *dwAllocationGranularity* field specifies the boundaries to which virtual memory regions are rounded. Like Windows NT, Windows CE rounds virtual regions to 64-KB boundaries.

A second handy function for determining the system memory state is this:

```
void GlobalMemoryStatus(LPMEMORYSTATUS lpmst);
```

which returns a MEMORYSTATUS structure defined as

```
typedef struct {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    DWORD dwTotalPhys;
    DWORD dwAvailPhys;
    DWORD dwTotalPageFile;
    DWORD dwAvailPageFile;
    DWORD dwTotalVirtual;
    DWORD dwAvailVirtual;
} MEMORYSTATUS;
```

The *dwLength* field must be initialized by the application before the call is made to *GlobalMemoryStatus*. The *dwMemoryLoad* field is of dubious value; it makes available a general loading parameter that's supposed to indicate the current memory use in the system. The *dwTotalPhys* and *dwAvailPhys* fields indicate how many pages of RAM are assigned to the program RAM and how many are available. These values don't include RAM assigned to the object store.

The *dwTotalPageFile* and *dwAvailPageFile* fields are used under Windows NT and Windows 98 to indicate the current status of the paging file. Because paging files aren't supported under Windows CE, these fields are always 0. The *dwTotalVirtual* and *dwAvailVirtual* fields indicate the total and available number of virtual memory pages accessible to the application.

The information returned by *GlobalMemoryStatus* provides confirmation of the memory architecture of Windows CE. Making this call on an HP 360 H/PC with 8 MB of RAM returned the following values:

```
dwMemoryLoad        0x18          (24)
dwTotalPhys         0x00555400    (5,592,064)
dwAvailPhys         0x00415C00    (4,283,392)
dwTotalPageFile     0
dwAvailPageFile     0
dwTotalVirtual      0x02000000    (33,554,432)
dwAvailVirtual      0x01EF0000    (32,440,320)
```

The *dwTotalPhys* field indicates that of the 8 MB of RAM in the system, I have dedicated 5.5 MB to the program RAM, of which 4.2 MB is still free. Note that there's no way for an application, using this call, to know that another 3 MB of RAM has been dedicated to the object store. To determine the amount of RAM dedicated to the object store, use the function *GetStoreInformation*.

The *dwTotalPageFile* and *dwAvailPageFile* fields are 0, indicating no support for a paging file under Windows CE. The *dwTotalVirtual* field is interesting because it shows the 32-MB limit on virtual memory that Windows CE enforces on an

application. Meanwhile, the *dwAvailVirtual* field indicates that in this application little of that 32 MB of virtual memory is being used.

An Application's Address Space

Although it's always interesting to look at the global memory map for an operating system, the fact is an application should be interested only in its own memory space, not the global address space. Nevertheless, the design of the Windows CE address space does have an impact on applications. Under Windows CE, an application is limited to the virtual memory space available in its 32-MB slot. While 32 MB might seem like a fair amount of space available to an application that might run on a system with only 4 MB of RAM, Win32 application programmers, used to a 2-GB virtual address space, need to keep in mind the limited virtual address space available to a Windows CE application.

Figure 6-2 shows the layout of an application's 32-MB virtual address space. Each line of the figure represents a block of virtual memory made up of one or more pages. The address of the blocks are offsets into the application's slot in the system address space. The Page status is free, reserved, private, or image. While I've just explained the terms *free* and *reserved*, *private* and *image* merit an explanation. *Image* indicates pages that have been committed and mapped to the image of an executable file in ROM or RAM. *Private* simply means the pages have been committed for use by the application. The size field indicates the size of the block, which is always a multiple of the page size. The access rights field displays the access rights for the block.

This memory map was captured on a Casio H/PC that has a SH3 processor with a 1024-byte page size. The application used in this example was stored in the object store and then launched. This allowed Windows CE to demand page only parts of the EXE image into RAM, as they're needed. If the application had been launched from an external storage device that didn't support demand paging, Windows CE would have loaded the entire application into memory when it was launched.

Address	*Page Status*	*Size*	*Access Rights*	*Comments*
0000 0000	Reserved	65,536		**EXE image**
0001 0000	Reserved	4,096		**Code**
0001 1000	Image	2,048	Execute, Read only	**Code**
0001 1800	Reserved	1,024		**Code**
0001 1C00	Image	1,024	Execute, Read only	**Code**
0001 2000	Reserved	2,048		**Code**
0001 2800	Image	8,192	Execute, Read only	**Code**
0001 4800	Reserved	2,048		**Code**
0001 5000	Image	1,024	Execute, Read only	**Code**

Figure 6-2. *Memory map of a Windows CE Application.*

Address	*Page Status*	*Size*	*Access Rights*	*Comments*
0001 5400	Reserved	11,264		
0001 8000	Image	3,072	Read only	**Read only static data**
0001 8C00	Reserved	1,024		
0001 9000	Image	1,024	Read/Write	**Read/Write static data**
0001 9400	Reserved	1,024		**Read/Write static data**
0001 9800	Image	7,168	Read/Write	**Read/Write static data**
0001 B400	Reserved	7,168		
0001 D000	Image	2,048	Read only	**Resource data segment**
0001 D800	Reserved	2,048		**Resource data segment**
0001 E000	Free	8,192		
0002 0000	Reserved	54,272		**Stack**
0002 D400	Private	7,168	Read/Write	
0002 F000	Free	4,096		
0003 0000	Private	1,024	Read/Write	**Local heap**
0003 0400	Reserved	92,192		
0009 0000	Free	30,408,704		**Free**
01D9 0000	Reserved	1,024		**COMMCTRL image**
01D9 0400	Image	237,568	Execute, Read only	
01DC A400	Image	2,048	Read/Write	
01DC AC00	Reserved	7,168		
01DC C800	Image	7,168	Read only	
01DC E400	Reserved	13,312		
01DD 1800	Free	2,091,008		**Free**
01FD 0000	Reserved	1,024		**COREDLL image**
01FD 0400	Image	119,808	Execute, Read only	
01FE D800	Image	1,024	Read/Write	
01FE DC00	Reserved	8,192		
01FE FC00	Image	1,024	Read only	
01FF 0000	Reserved	5,120		
01FF 1400	Free	60,416		

Notice that the application is mapped as a 64-KB region starting at 0x10000. Remember, the lowest 64 KB of the address space for any application is reserved by Windows CE. The image of the file contains the code along with the static data segments and the resource segments. Although it appears that the program code is broken into a number of disjointed pages from 0x10000 to 0x15400, this is actually the result of demand paging. What's happening is that only the pages containing executed code are mapped into the address space. The reserved pages within the code segment will be mapped into the space only when they're executed.

The read-only static data segment is mapped at 0x18000 and takes three pages. The read/write static data is mapped from 0x19000 to 0x1B3FF. Like the code, the read/write data segment is committed to RAM only as it's written to by the application. Any static data that was initialized by the loader is already committed, as is the static variables written before this capture of the address space was made. The resources for the application are mapped starting at 0x1D000. The resources are read only and are paged into the RAM only as they're accessed by the application.

Starting at 0x20000, the application's stack is mapped. The stack segment is easily recognized because the committed pages are at the end of the reserved section, indicative of a stack that grows from higher addresses down. If this application had more than one thread, more than one stack segment would be reserved in the application's address space.

Following the stack is the local heap. The heap has only a few blocks currently allocated, requiring only one page of RAM. The loader reserves another 392,192 bytes, or 383 pages, for the heap to grow. The over-30 MB of address space from the end of the reserved pages for the local heap to the start of the DLLs mapped into the address space is free to be reserved and, if RAM permits, committed by the application.

This application accesses two dynamic-link libraries. Coredll.dll is the DLL that contains the entry points to the Windows CE operating system. In Windows CE, the function entry points are combined into one DLL, unlike in Windows NT or Windows 98, where the core functions are distributed across Kernel, User, and GDI. The other DLL is the common control DLL, commctrl.dll. As with the executable image, these DLLs are mapped into the address space as linear images. However, unlike the EXE, these DLLs are in ROM and directly mapped into the virtual address space of the application; therefore, they don't take up any RAM.

THE DIFFERENT KINDS OF MEMORY ALLOCATION

A Windows CE application has a number of different methods for allocating memory. At the bottom of the memory-management food chain are the *Virtual*xxx functions that directly reserve, commit, and free virtual memory pages. Next comes the heap API.

Heaps are regions of reserved memory space managed by the system for the application. Heaps come in two flavors: the default local heap automatically allocated when an application is started, and separate heaps that can be manually created by the application. After the heap API is static data—data blocks defined by the compiler and that are allocated automatically by the loader. Finally, we come to the stack, where an application stores variables local to a function.

The one area of the Win32 memory API that Windows CE doesn't support is the global heap. The global heap API, which includes calls such as *GlobalAlloc*, *GlobalFree*, and *GlobalRealloc*, are therefore not present in Windows CE. The global heap is really just a holdover from the Win16 days of Windows 3.*x*. In Win32, the global and local heaps are quite similar. One unique use of global memory, allocating memory for data in the clipboard, is handled by using the local heap under Windows CE.

The key to minimizing memory use in Windows CE is choosing the proper memory-allocation strategy that matches the memory-use patterns for a given block of memory. I'll review each of these memory types and then describe strategies for minimizing memory use in Windows CE applications.

Virtual Memory

Virtual memory is the most basic of the memory types. The system uses calls to the virtual memory API to allocate memory for the other types of memory, including heaps and stacks. The virtual memory API, including the *VirtualAlloc*, *VirtualFree*, and *VirtualReSize* functions directly manipulate virtual memory pages in the application's virtual memory space. Pages can be reserved, committed to physical memory, and freed using these functions.

Allocating virtual memory

Allocating and reserving virtual memory is accomplished using this function:

```
LPVOID VirtualAlloc (LPVOID lpAddress, DWORD dwSize,
                     DWORD flAllocationType,
                     DWORD flProtect);
```

The first parameter to *VirtualAlloc* is the virtual address of the region of memory to allocate. The lpAddress parameter is used to identify the previously reserved memory block when you use VirtualAlloc to commit a block of memory previously reserved. If this parameter is NULL, the system determines where to allocate the memory region, rounded to a 64-KB boundary. The second parameter is *dwSize*, the size of the region to allocate or reserve. While this parameter is specified in bytes, not pages, the system rounds the requested size up to the next page boundary.

The *flAllocationType* parameter specifies the type of allocation. You can specify a combination of the following flags: MEM_COMMIT, MEM_AUTO_COMMIT, MEM_RESERVE, and MEM_TOP_DOWN. The MEM_COMMIT flag allocates the memory to be used by the program. MEM_RESERVE reserves the virtual address space to be later committed. Reserved pages can't be accessed until another call is made to *VirtualAlloc* specifying the region and using the MEM_COMMIT flag. The third flag, MEM_TOP_DOWN, tells the system to map the memory at the highest permissible virtual address for the application.

The MEM_AUTO_COMMIT flag is unique to Windows CE and is quite handy. When this flag is specified the block of memory is reserved immediately, but each page in the block will automatically be committed by the system when it's accessed for the first time. This allows you to allocate large blocks of virtual memory without burdening the system with the actual RAM allocation until the instant each page is first used. The drawback to auto-commit memory is that the physical RAM needed to back up a page might not be available when the page is first accessed. In this case, the system will generate an exception.

VirtualAlloc can be used to reserve a large region of memory with subsequent calls committing parts of the region or the entire region. Multiple calls to commit the same region won't fail. This allows an application to reserve memory and then blindly commit a page before it's written to. While this method isn't particularly efficient, it does free the application from having to check the state of a reserved page to see whether it's already committed before making the call to commit the page.

The *flProtect* parameter specifies the access protection for the region being allocated. The different flags available for this parameter are summarized in the following list.

- *PAGE_READONLY* The region can be read. If an application attempts to write to the pages in the region, an access violation will occur.
- *PAGE_READWRITE* The region can be read from or written to by the application.
- *PAGE_EXECUTE* The region contains code that can be executed by the system. Attempts to read from or write to the region will result in an access violation.
- *PAGE_EXECUTE_READ* The region can contain executable code and applications can also read from the region.
- *PAGE_EXECUTE_READWRITE* The region can contain executable code and applications can read from and write to the region.

- *PAGE_GUARD* The first access to this region results in a STATUS_GUARD_PAGE exception. This flag should be combined with the other protection flags to indicate the access rights of the region after the first access.
- *PAGE_NOACCESS* Any access to the region results in an access violation.
- *PAGE_NOCACHE* The RAM pages mapped to this region won't be cached by the microprocessor.

The PAGE_GUARD and PAGE_NOCHACHE flags can be combined with the other flags to further define the characteristics of a page. The PAGE_GUARD flag specifies a guard page, a page that generates a one-shot exception when it's first accessed and then takes on the access rights that were specified when the page was committed. The PAGE_NOCACHE flag prevents the memory that's mapped to the virtual page from being cached by the microprocessor. This flag is handy for device drivers that share memory blocks with devices using direct memory access (DMA).

Regions vs. pages

Before I go on to talk about the virtual memory API, I need to make a somewhat subtle distinction. Virtual memory is reserved in regions that must align on 64-KB boundaries. Pages within a region can then be committed page by page. You can directly commit a page or a series of pages without first reserving a region of pages, but the page, or series of pages, directly committed will be aligned on a 64-KB boundary. For this reason, it's best to reserve blocks of virtual memory in 64-KB chunks and then commit that page within the region as needed.

With the limit of a 32-MB virtual memory space per process, this leaves a maximum of 32 MB / 64 KB – 1= 511 virtual memory regions that can be reserved before the system reports that it's out of memory. Take, for example, the following code fragment:

```
#define PAGESIZE 1024   // Assume we're on a 1-KB page machine
for (i = 0; i < 512; i++)
    pMem[i] = VirtualAlloc (NULL, PAGESIZE, MEM_RESERVE | MEM_COMMIT,
                            PAGE_READWRITE);
```

This code attempts to allocate 512 one-page blocks of virtual memory. Even if you have half a megabyte of RAM available in the system, *VirtualAlloc* will fail before the loop completes because it will run out of virtual address space for the application. This happens because each 1-KB block is allocated on a 64-KB boundary. Since the code, stack, and local heap for an application must also be mapped into the same, 32-MB virtual address space, available virtual allocation regions usually top out at about 490.

A better way to make 512 distinct virtual allocations is to do something like this:

```
#define PAGESIZE 1024   // Assume we're on a 1-KB page machine.

// Reserve a region first.
pMemBase = VirtualAlloc (NULL, PAGESIZE * 512, MEM_RESERVE,
                         PAGE_NOACCESS);

for (i = 0; i < 512; i++)
    pMem[i] = VirtualAlloc (pMemBase + (i*PAGESIZE), PAGESIZE,
                            MEM_COMMIT, PAGE_READWRITE);
```

This code first reserves a region; the pages are committed later. Because the region was first reserved, the committed pages aren't rounded to 64-KB boundaries, and so, if you have 512 KB of available memory in the system, the allocations will succeed.

Although the code I just showed you is a contrived example (there are better ways to allocate 1-KB blocks than directly allocating virtual memory), it does demonstrate a major difference (from other Windows systems) in the way memory allocation works in Windows CE. In Windows NT, applications have a full 2-GB virtual address space with which to work. In Windows CE however, a programmer should remain aware of the relatively small 32-MB virtual address per application.

Freeing virtual memory

You can decommit or free virtual memory by calling *VirtualFree*. Decommitting a page unmaps the page from a physical page of RAM but keeps the page or pages reserved. The function is prototyped as

```
BOOL VirtualFree (LPVOID lpAddress, DWORD dwSize,
                  DWORD dwFreeType);
```

The *lpAddress* parameter should contain a pointer to the virtual memory region that's to be freed or decommitted. The *dwSize* parameter contains the size, in bytes, of the region if the region is to be decommitted. If the region is to be freed, this value must be 0. The *dwFreeType* parameter contains the flags that specify the type of operation. The MEM_DECOMMIT flag specifies that the region will be decommited but will remain reserved. The MEM_RELEASE flag both decommits the region if the pages are committed and also frees the region.

All the pages in a region being freed by means of *VirtualFree* must be in the same state. That is, all the pages in the region to be freed must either be committed or reserved. *VirtualFree* fails if some of the pages in the region are reserved while some are committed. To free a region with pages that are both reserved and committed, the committed pages should be decommitted first, and then the entire region can be freed.

Changing and querying access rights

You can modify the access rights of a region of virtual memory, initially specified in VirtualAlloc, by calling *VirtualProtect*. This function can change the access rights only on committed pages. The function is prototyped as

```
BOOL VirtualProtect (LPVOID lpAddress, DWORD dwSize,
                     DWORD flNewProtect, PDWORD lpflOldProtect);
```

The first two parameters, *lpAddress* and *dwSize*, specify the block and the size of the region that the function acts on. The *flNewProtect* parameter contains the new protection flags for the region. These flags are the same ones I mentioned when I explained the *VirtualAlloc* function. The *lpflOldProtect* parameter should point to a DWORD that will receive the old protection flags of the first page in the region.

The current protection rights of a region can be queried with a call to

```
DWORD VirtualQuery (LPCVOID lpAddress,
                    PMEMORY_BASIC_INFORMATION lpBuffer,
                    DWORD dwLength);
```

The *lpAddress* parameter contains the starting address of the region being queried. The *lpBuffer* pointer points to a PMEMORY_BASIC_INFORMATION structure that I'll talk about soon. The third parameter, *dwLength*, must contain the size of the PMEMORY_BASIC_INFORMATION structure.

The PMEMORY_BASIC_INFORMATION structure is defined as

```
typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    DWORD RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION;
```

The first field of MEMORY_BASIC_INFORMATION, *BaseAddress*, is the address passed to the *VirtualQuery* function. The *AllocationBase* field contains the base address of the region when it was allocated using a *VirtualAlloc* function. The *AllocationProtect* field contains the protection attributes for the region when it was originally allocated. The *RegionSize* field contains the number of bytes from the pointer passed to *VirtualQuery* to the end of series of pages that have the same attributes. The *State* field contains the state—free, reserved, or committed—of the pages in the region. The *Protect* field contains the current protection flags for the region. Finally, the *Type* field contains the type of memory in the region. This field

can contain the flags MEM_PRIVATE, indicating that the region contains private data for the application; MEM_MAPPED, indicating that the region is mapped to a memory-mapped file; or MEM_IMAGE, indicating that the region is mapped to an EXE or DLL module.

The best way to understand the values returned by *VirtualQuery* is to look at an example. Say an application uses *VirtualAlloc* to reserve 16,384 bytes (16 pages on a 1-KB page-size machine). The system reserves this 16-KB block at address 0xA0000. Later, the application commits 9216 bytes (9 pages) starting 2048 bytes (2 pages) into the initial region. Figure 6-3 shows a diagram of this scenario.

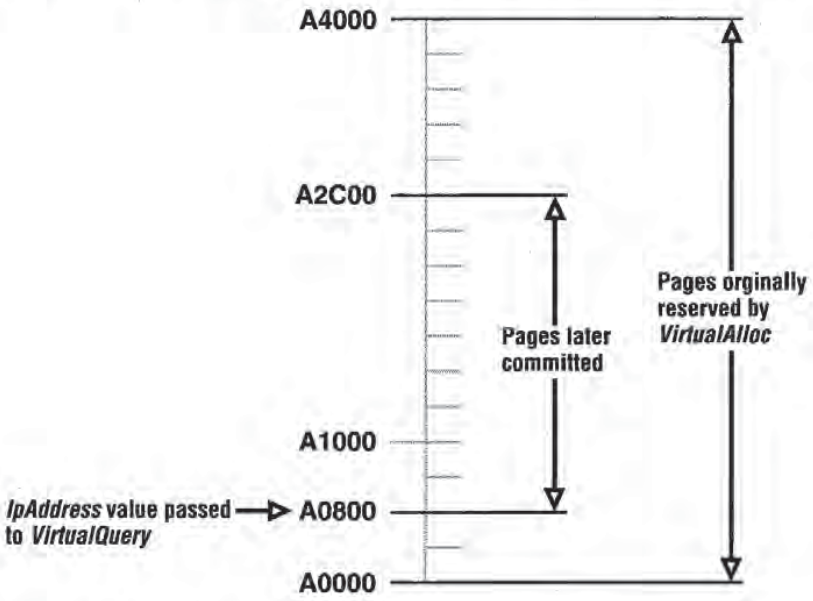

Figure 6-3. *A region of reserved virtual memory that has nine pages committed.*

If a call is made to *VirtualQuery* with the *lpAddress* pointer pointing 4 pages into the initial region (address 0xA1000), the returned values would be the following:

```
BaseAddress          0xA1000
AllocationBase       0xA0000
AllocationProtect    PAGE_NOACCESS
RegionSize           0x1C00    (7,168 bytes or 7 pages)
State                MEM_COMMIT
Protect              PAGE_READWRITE
Type                 MEM_PRIVATE
```

The *BaseAddress* field contains the address passed to *VirtualQuery*, 0xA1000, 4096 bytes into the initial region. The *AllocationBase* field contains the base address of the original region while *AllocationProtect* contains PAGE_NOACCESS, indicating that

the region was originally reserved, not directly committed. The *RegionSize* field contains the number of bytes from the pointer passed to *VirtualQuery*, 0xA1000 to the end of the committed pages at 0xA2C00. The *State* and *Protect* fields contain the flags indicating the current state of the pages. The *Type* field indicates that the region was allocated by the application for its own use.

Heaps

Clearly, allocating memory on a page basis is inefficient for most applications. To optimize memory use, an application needs to be able to allocate and free memory on a per byte, or at least a per 4-byte, basis. The system enables allocations of this size through heaps. Using heaps also protects an application from having to deal with the differing page sizes of the microprocessors that support Windows CE. An application can simply allocate a block in a heap and the system deals with the number of pages necessary for the allocation.

As I mentioned before, heaps are regions of reserved virtual memory space managed by the system for the application. The system gives you a number of functions that allow you to allocate and free blocks within the heap with a granularity much smaller than a page. As memory is allocated by the application within a heap, the system automatically grows the size of the heap to fill the request. As blocks in the heap are freed, the system looks to see if an entire page is freed. If so, that page is decommitted.

Unlike Windows NT or Windows 98, Windows CE supports the allocation of only fixed blocks in the heap. This simplifies the handling of blocks in the heap, but it can lead to the heaps becoming fragmented over time as blocks are allocated and freed. The result can be a heap being fairly empty but still requiring a large number of virtual pages because the system can't reclaim a page from the heap unless it's completely free.

Each application has a default, or local, heap created by the system when the application is launched. Blocks of memory in the local heap can be allocated, freed, and resized using the *LocalAlloc*, *LocalFree*, and *LocalRealloc* functions. An application can also create any number of separate heaps. These heaps have the same properties as the local heap but are managed through a separate set of *Heap*xxxx functions.

The Local Heap

By default, Windows CE initially reserves 384 pages, or 393,216 bytes, for the local heap but only commits the pages as they are allocated. If the application allocates more than the 384 KB in the local heap, the system allocates more space for the local heap. Growing the heap might require a separate, disjointed address space reserved

for the additional space on the heap. Applications shouldn't assume that the local heap is contained in one block of virtual address space. Because Windows CE heaps support only fixed blocks, Windows CE implements only the subset of the Win32 local heap functions necessary to allocate, resize, and free fixed blocks on the local heap.

Allocating memory on the local heap

You allocate a block of memory on the local heap by calling

```
HLOCAL LocalAlloc (UINT uFlags, UINT uBytes);
```

The call returns a value cast as an HLOCAL, which is a handle to a local memory block, but since the block allocated is always fixed, the return value can simply be recast as a pointer to the block.

The *uFlags* parameter describes the characteristics of the block. The flags supported under Windows CE are limited to those that apply to fixed allocations. They are the following:

- *LMEM_FIXED* Allocates a fixed block in the local heap. Since all local heap allocations are fixed, this flag is redundant.
- *LMEM_ZEROINIT* Initializes memory contents to 0.
- *LPTR* Combines the LMEM_FIXED and LMEM_ZEROINIT flags.

The *uBytes* parameter specifies the size of the block to allocate in bytes. The size of the block is rounded up, but only to the next DWORD (4 byte) boundary.

Freeing memory on the local heap

You can free a block by calling

```
HLOCAL LocalFree (HLOCAL hMem);
```

The function takes the handle to the local memory block and returns NULL if successful. If the function fails, it returns the original handle to the block.

Resizing and querying the size of local heap memory

You can resize blocks on the local heap by calling

```
HLOCAL LocalReAlloc (HLOCAL hMem, UINT uBytes, UINT uFlag);
```

The *hMem* parameter is the pointer (handle) returned by *LocalAlloc*. The *uBytes* parameter is the new size of the block. The *uFlag* parameter contains the flags for the new block. Under Windows CE, two flags are relevant, LMEM_ZEROINIT and LMEM_MOVEABLE. LMEM_ZEROINIT causes the contents of the new area of the block to be set to 0 if the block is grown as a result of this call. The LMEM_MOVEABLE flag

tells Windows that it can move the block if the block is being grown and there's not enough room immediately above the current block. Without this flag, if you don't have enough space immediately above the block to satisfy the request, *LocalRealloc* will fail with an out-of-memory error. If you specify the LMEM_MOVEABLE flag, the handle (really the pointer to the block of memory) might change as a result of the call.

The size of the block can be queried by calling

```
UINT LocalSize (HLOCAL hMem);
```

The size returned will be at least as great as the requested size for the block. As I mentioned earlier, Windows CE rounds the size of a local heap allocation up to the next 4-byte boundary.

Separate Heaps

To avoid fragmenting the local heap, it's better to create a separate heap if you need a series of blocks of memory that will be used for a set amount of time. An example of this would be a text editor that might manage a file by creating a separate heap for each file it's editing. As files are opened and closed, the heaps would be created and destroyed.

Heaps under Windows CE have the same API as those under Windows NT or Windows 98. The only noticeable difference is the lack of support for the HEAP_GENERATE_EXCEPTIONS flag. Under Windows NT, this flag causes the system to generate an exception if an allocation request can't be accommodated.

A subtle, but more important difference to the programmer is how Windows CE manages heaps. While the heap API looks like the standard Win32 heap API, Windows CE doesn't implement the functions as you might expect. For example, the *HeapCreate* function has parameters that allow a program to specify how much memory to allocate and reserve for a heap. Windows CE ignores these values. In fact, simply creating a heap doesn't allocate or reserve any memory. Memory is reserved and committed only when the first block of the heap is allocated.

Under most conditions, going through the details about when heap memory is reserved and committed would seem like nitpicking. But if you've used up the 32-MB virtual address space for other uses, a heap might not have the virtual address space available for the allocation even if you thought you had reserved enough using the *HeapCreate* call. On the other hand, Windows CE doesn't use the reserved parameter in the *HeapCreate* call as a hard-coded limit on the size of the heap. Windows CE accommodates almost any heap allocation request if the memory is available. Well, enough editorializing: on to the heap API.

Creating a separate heap

You create heaps by calling

```
HANDLE HeapCreate (DWORD flOptions, DWORD dwInitialSize,
                   DWORD dwMaximumSize);
```

Under Windows CE, the first parameter, *flOptions*, can be NULL, or it can contain the HEAP_NO_SERIALIZE flag. By default, Windows heap management routines prevent two threads in a process from accessing the heap at the same time. This serialization prevents the heap pointers that the system uses to track the allocated blocks in the heap from being corrupted. In other versions of Windows the HEAP_NO_SERIALIZE flag can be used if you don't want this type of protection. Under Windows CE however, this flag is only provided for compatibility and all heap accesses are serialized.

The other two parameters, *dwInitialSize* and *dwMaximumSize*, specify the initial size and expected maximum size of the heap. Windows NT and Windows 98 use the *dwMaximumSize* value to determine how many pages in the virtual address space to reserve for the heap. You can set this parameter to 0 if you want to defer to Windows' determination of how many pages to reserve. The *dwInitialSize* parameter is then used to determine how many of those initially reserved pages will be immediately committed. As I mentioned, while these two size parameters are documented exactly the same way as their counterparts under Windows NT and 98, the current version of Windows CE doesn't actually use them. You should, however, use valid numbers to retain compatibility with future versions of Windows CE that might use these parameters.

Allocating memory in a separate heap

You allocate memory on the heap using

```
LPVOID HeapAlloc (HANDLE hHeap, DWORD dwFlags, DWORD dwBytes);
```

Notice that the return value is a pointer, not a handle as in the *LocalAlloc* function. Separate heaps always allocate fixed blocks, even under Windows NT and Windows 98. The first parameter is the handle to the heap returned by the *HeapCreate* call. The *dwFlags* parameter can be one of two self-explanatory values, HEAP_NO_SERIALIZE and HEAP_ZERO_MEMORY. The final parameter, *dwBytes*, specifies the number of bytes in the block to allocate. The size is rounded up to the next DWORD.

Freeing memory in a separate heap

You can free a block in a heap by calling

```
BOOL HeapFree (HANDLE hHeap, DWORD dwFlags, LPVOID lpMem);
```

The only flag allowable in the *dwFlags* parameter is HEAP_NO_SERIALIZE. The *lpMem* parameter points to the block to free, while *hHeap* contains the handle to the heap.

Resizing and querying the size of memory in a separate heap

You can resize heap allocations by calling

```
LPVOID HeapReAlloc (HANDLE hHeap, DWORD dwFlags, LPVOID lpMem,
                    DWORD dwBytes);
```

The *dwFlags* parameter can be any combination of three flags: HEAP_NO_SERIALIZE, HEAP_REALLOC_IN_PLACE_ONLY, and HEAP_ZERO_MEMORY. The only new flag here is HEAP_REALLOC_IN_PLACE_ONLY, which tells the heap manager to fail the reallocation if the space can't be found for the block without relocating it. This flag is handy if you already have a number of pointers pointing to data in the block and you aren't interested in updating them. The *lpMem* parameter is the pointer to the block being resized, and the *dwBytes* parameter is the requested new size of the block. Notice that the function of the HEAP_REALLOC_IN_PLACE_ONLY flag in *HeapReAlloc* provides the opposite function from the one that the LMEM_MOVEABLE flag provides for *LocalReAlloc*. HEAP_REALLOC_IN_PLACE_ONLY prevents a block that would be moved by default in a separate heap while LMEM_MOVEABLE enables a block to be moved that by default would not be moved in the local heap. *HeapReAlloc* returns a pointer to the block if the reallocation was successful, and returns NULL otherwise. Unless you specified that the block not be relocated, the returned pointer might be different from the pointer passed in if the block had to be relocated to find enough space in the heap.

To determine the actual size of a block, you can call

```
DWORD HeapSize (HANDLE hHeap, DWORD dwFlags, LPCVOID lpMem);
```

The parameters are as you expect: the handle of the heap, the single, optional flag, HEAP_NO_SERIALIZE, and the pointer to the block of memory being checked.

Destroying a separate heap

You can completely free a heap by calling

```
BOOL HeapDestroy (HANDLE hHeap);
```

Individual blocks within the heap don't have to be freed before you destroy the heap.

One final heap function is valuable when writing DLLs. The function

```
HANDLE GetProcessHeap (VOID);
```

returns the handle to the local heap of the process calling the DLL. This allows a DLL to allocate memory within the calling process's local heap. All the other heap calls, with the exception of *HeapDestroy*, can be used with the handle returned by *GetProcessHeap*.

The Stack

The stack is the easiest to use (the most self-managing) of the different types of memory under Windows CE. The stack under Windows CE, as in any operating system, is the storage place for temporary variables that are referenced within a function. The operating system also uses the stack to store return addresses for functions and the state of the microprocessor registers during exception handling.

Windows CE manages a separate stack for every thread in the system. Under all versions of the operating system before Windows CE 2.1, each stack in the system is limited to fewer than 58 KB. Separate threads within one process can each grow its stack up to the 58-KB limit. This limit has to do with how Windows CE manages the stack. When a thread is created, Windows CE reserves a 60-KB region for the thread's stack. It then commits virtual pages from the top down as the stack grows. As the stack shrinks, the system will, under low-memory conditions, reclaim the unused but still committed pages below the stack. The limit of 58 KB comes from the size of the 64-KB region dedicated to the stack minus the number of pages necessary to guard the stack against overflow and underflow.

Starting with Windows CE 2.1, the size of the stack can be specified by a linker switch when an application is linked. The same guard pages are applied, but the stack size can be specified up to 1 MB. Note that the size defined for the default stack is also the size used for all the separate thread stacks. That is, if you specify the main stack to be 128 KB, all other threads in the application have a stack size limit of 128 KB.

One other consideration must be made when you're planning how to use the stack in an application. When an application calls a function that needs stack space, Windows CE attempts to commit the pages immediately below the current stack pointer to satisfy the request. If no physical RAM is available, the thread needing the stack space is briefly suspended. If the request can't be granted within a short period of time, an exception is raised. Windows CE goes to great lengths to free the required pages, but if this can't happen the system raises an exception. I'll cover low-memory situations shortly, but for now just remember that you shouldn't try to use large amounts of stack space in low-memory situations.

Static Data

C and C++ applications have predefined blocks of memory that are automatically allocated when the application is loaded. These blocks hold statically allocated strings, buffers, and global variables as well as buffers necessary for the library functions that were statically linked with the application. None of this is new to the C programmer, but under Windows CE, these spaces are handy for squeezing the last useful bytes out of RAM.

Windows CE allocates two blocks of RAM for the static data of an application, one for the read/write data and one for the read-only data. Because these areas are allocated on a per-page basis, you can typically find some space left over from the static data up to the next page boundary. The finely tuned Windows CE application should be written to ensure that it has little or no extra space left over. If you have space in the static data area, sometimes it's better to move a buffer or two into the static data area instead of allocating those buffers dynamically.

Another consideration is that if you're writing a ROM-based application, you should move as much data as possible to the read-only static data area. Windows CE doesn't allocate RAM to the read-only area for ROM-based applications. Instead, the ROM pages are mapped directly into the virtual address space. This essentially gives you unlimited read-only space with no impact on the RAM requirements of the application.

The best place to determine the size of the static data areas is to look in the map file that's optionally generated by the linker. The map file is chiefly used to determine the locations of functions and data for debugging purposes, but it also shows the size of the static data, if you know where to look. Figure 6-4 shows a portion of an example map file generated by Visual C++.

```
memtest

Timestamp is 34ce4088 (Tue Jan 27 12:16:08 1998)

Preferred load address is 00010000

Start         Length     Name                   Class
0001:00000000 00006100H .text                  CODE
0002:00000000 00000310H .rdata                 DATA
0002:00000310 00000014H .xdata                 DATA
0002:00000324 00000028H .idata$2               DATA
0002:0000034c 00000014H .idata$3               DATA
0002:00000360 000000f4H .idata$4               DATA
0002:00000454 000003eeH .idata$6               DATA
0002:00000842 00000000H .edata                 DATA
0003:00000000 000000f4H .idata$5               DATA
0003:000000f4 00000004H .CRT$XCA               DATA
0003:000000f8 00000004H .CRT$XCZ               DATA
0003:000000fc 00000004H .CRT$XIA               DATA
0003:00000100 00000004H .CRT$XIZ               DATA
0003:00000104 00000004H .CRT$XPA               DATA
0003:00000108 00000004H .CRT$XPZ               DATA
0003:0000010c 00000004H .CRT$XTA               DATA
```

(continued)

```
0003:00000110 00000004H .CRT$XTZ                   DATA
0003:00000114 000011e8H .data                      DATA
0003:000012fc 0000108cH .bss                       DATA
0004:00000000 000003e8H .pdata                     DATA
0005:00000000 000000f0H .rsrc$01                   DATA
0005:000000f0 00000334H .rsrc$02                   DATA

 Address         Publics by Value              Rva+Base     Lib:Object

0001:00000000       _WinMain                   00011000 f   memtest.obj
0001:0000007c       _InitApp                   0001107c f   memtest.obj
0001:000000d4       _InitInstance              000110d4 f   memtest.obj
0001:00000164       _TermInstance              00011164 f   memtest.obj
0001:00000248       _MainWndProc               00011248 f   memtest.obj
0001:000002b0       _GetFixedEquiv             000112b0 f   memtest.obj
0001:00000350       _DoCreateMain              00011350 f   memtest.obj.
:
```

Figure 6-4. *The top portion of a map file showing the size of the data segments in an application.*

The map file in Figure 6-4 indicates that the EXE has five sections. Section *0001* is the text segment containing the executable code of the program. Section *0002* contains the read-only static data. Section *0003* contains the read/write static data. Section *0004* contains the fix-up table to support calls to other DLLs. Finally, section *0005* is the resource section containing the application's resources, such as menu and dialog box templates.

Let's examine the *.data*, *.bss*, and *.rdata* lines. The *.data* section contains the initialized read/write data. If you initialized a global variable as in

```
static HINST g_hLoadlib = NULL;
```

the *g_loadlib* variable would end up in the *.data* segment. The *.bss* segment contains the uninitialized read/write data. A buffer defined as

```
static BYTE g_ucItems[256];
```

would end up in the *.bss* segment. The final segment, *.rdata*, contains the read-only data. Static data that you've defined using the *const* keyword ends up in the *.rdata* segment. An example of this would be the structures I use for my message look-up tables, as in the following:

```
// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_SIZE, DoSizeMain,
    WM_COMMAND, DoCommandMain,
    WM_DESTROY, DoDestroyMain,
};
```

The *.data* and *.bss* blocks are folded into the *0003* section which, if you add the size of all blocks in the third section, has a total size of 0x2274, or 8820, bytes. Rounded up to the next page size, the read/write section ends up taking nine pages, with 396 bytes not used. So, in this example, placing a buffer or two in the static data section of the application would be essentially free. The read-only segment, section *0002*, including *.rdata*, ends up being 0x0842, or 2114, bytes, which takes up three pages with 958 bytes, almost an entire page, wasted. In this case, moving 75 bytes of constant data from the read-only segment to the read /write segment saves a page of RAM when the application is loaded.

String Resources

One often forgotten area for read-only data is the resource segment of your application. While I mentioned a new, Windows CE–specific feature of the *LoadString* function in Chapter 3, it's worth repeating here. If you call *LoadString* with 0 in place of the pointer to the buffer, the function returns a pointer to the string in the resource segment. An example would be

```
LPCTSTR pString;

pString  = (LPCTSTR)LoadString (hInst, ID_STRING, NULL, 0)
```

The string returned is read only, but it does allow you to reference the string without having to allocate a buffer to hold the string.

Selecting the Proper Memory Type

Now that we've looked at the different types of memory, it's time to consider the best use of each. For large blocks of memory, directly allocating virtual memory is best. An application can reserve as much address space (up to the 32-MB limit of the application) but can commit only the pages necessary at any one time. While directly allocated virtual memory is the most flexible memory allocation type, it shifts to us the burden of worrying about page granularity as well as keeping track of the reserved versus committed pages.

The local heap is always handy. It doesn't need to be created and will grow as necessary to satisfy a request. Fragmentation is the issue here. Consider that applications on an H/PC might run for weeks or even months at a time. There's no Off button on an H/PC or a Palm-size PC—just a Suspend command. So, when you're thinking about memory fragmentation, don't assume that a user will open the application, change one item, and then close it. A user is likely to start an application and keep it running so that the application is just a quick click away.

The advantage of separate heaps is that you can destroy them when their time is up, nipping the fragmentation problem in the bud. A minor disadvantage of separate heaps is the need to manually create and destroy them. Another thing to remember

about separate heaps is that Windows CE doesn't reserve virtual address space when a heap is created, which can become an issue if your application uses much of the virtual address space available to the application.

The static data area is a great place to slip in a buffer or two essentially for free because the page is going to be allocated anyway. The key to managing the static data is to make the size of the static data segments close to, but over the page size of, your target processor. For applications written for the H/PC or Palm-size PC, consider the 1024-byte page size of the NEC MIPS 4100 and Hitachi SH3 processors as the default. Sometimes it's better to move constant data from the read-only segment to the read/write segment if it saves a page in the read-only segment. The only time you wouldn't do this is if the application is to be burned into ROM. Then, the more constant data, the better, because it doesn't take up RAM.

The stack is, well, the stack—simple to use and always around. The only considerations are the maximum size of the stack and the problems of enlarging the stack in a low memory condition. Make sure your application doesn't require large amounts of stack space to shut down. If the system suspends a thread in your application while it's being shut down, the user will more than likely lose data. That won't help customer satisfaction.

Managing Low-Memory Conditions

Even for applications that have been fine-tuned to minimize their memory use, there are going to be times when the system runs very low on RAM. Windows CE applications operate in an almost perpetual low-memory environment. The Palm-size PC is designed intentionally to run in a low-memory situation Applications on the Palm-size PC don't have a Close button—the shell automatically closes them when the system needs additional memory. Because of this, Windows CE offers a number of methods to distribute the scarce memory in the system among the running applications.

The WM_HIBERNATE message

The first and most obvious addition to Windows CE is the WM_HIBERNATE message. Windows CE sends this message to all top-level windows that have the WS_OVERLAPPED style (that is, have neither the WS_POPUP nor the WS_CHILD style) and have the WS_VISIBLE style. These qualifications should allow most applications to have at least one window that receives a WM_HIBERNATE message. An exception to this would be an application that doesn't really terminate, but simply hides all its windows. This arrangement allows an application a quick start because it only has to show its window, but this situation also means that the application is taking up RAM even when the user thinks it's closed. While this is exactly the kind of application design that should *not* be used under Windows CE, those that are designed this way must act as if they're always in hibernate mode when hidden because they'll never receive a WM_HIBERNATE message.

Windows CE sends WM_HIBERNATE messages to the top-level windows in reverse Z-order until enough memory is freed to push the available memory above a preset threshold. When an application receives a WM_HIBERNATE message, it should reduce its memory footprint as much as possible. This can involve releasing cached data; freeing any GDI objects such as fonts, bitmaps, and brushes; and destroying any window controls. In essence, the application should reduce its memory use to the smallest possible footprint that's necessary to retain its internal state.

If sending WM_HIBERNATE messages to the applications in the background doesn't free enough memory to move the system out of a limited-memory state, a WM_HIBERNATE message is sent to the application in the foreground. If part of your hibernation routine is to destroy controls on your window, you should be sure that you aren't the foreground application. Disappearing controls don't give the user a warm and fuzzy feeling.

Memory thresholds

Windows CE monitors the free RAM in the system and responds differently as less and less RAM is available. As less memory is available, Windows CE first sends WM_HIBERNATE messages and then begins limiting the size of allocations possible. The two figures below show the free-memory levels used by the Handheld PC and the Palm-size PC to trigger low-memory events in the system. Windows CE defines four memory states: normal, limited, low, and critical. The memory state of the system depends on how much free memory is available to the system as a whole. These limits are higher for 4-KB page systems because those systems have less granularity in allocations.

Event	*Free Memory 1024-Page Size*	*Free Memory 4096-Page Size*	*Comments*
Limited-memory state	128 KB	160 KB	Send MWM_HIBERNATE messages to applications in reverse Z-order. Free stack space reclaimed as needed.
Low-memory state	64 KB	96 KB	Limit virtual allocs to 16 KB. Low-memory dialog displayed.
Critical-memory state	16 KB	48 KB	Limit virtual allocs to 8 KB.

Figure 6-5. *Memory thresholds for the Handheld PC.*

Event	*Free Memory 1024-Page Size*	*Free Memory 4096-Page Size*	*Comments*
Hibernate threshold	200 KB	224 KB	Send WM_HIBERNATE messages to applications in reverse Z-order.
Limited-memory state	128 KB	160 KB	Begin to close applications in reverse Z-order. Free stack space reclaimed as needed.
Low-memory state	64 KB	96 KB	Limit virtual allocs to 16 KB.
Critical-memory state	16 KB	48 KB	Limit virtual allocs to 8 KB.

Figure 6-6. *Memory thresholds for the Palm-size PC.*

The effect of these memory states is to share the remaining wealth. First, WM_HIBERNATE messages are sent to the applications to ask them to reduce their memory footprint. After an application is sent a WM_HIBERNATE message, the system memory levels are checked to see whether the available memory is now above the threshold that caused the WM_HIBERNATE messages to be sent. If not, a WM_HIBERNATE message is sent to the next application. This continues until all applications have been sent a WM_HIBERNATE message.

The low-memory strategies of the Handheld PC and the Palm-size PC diverge at this point. If the memory level drops below the next threshold, limited for the Palm-size PC and Low for the H/PC, the system starts shutting down applications. On the H/PC, the system displays the OOM, the out-of-memory dialog, and requests that the user either select an application to close or reallocate some RAM dedicated to the object store to the program memory. If, after the selected application has been shut down or memory has been moved into program RAM, you still don't have enough memory, the out-of-memory dialog is displayed again. This process is repeated until there's enough memory to lift the H/PC above the threshold.

For the Palm-size PC, the actions are somewhat different. The Palm-size PC shell automatically starts shutting down applications in least recently used order without asking the user. If there still isn't enough memory after all applications except the foreground application and the shell are closed, the system uses its other techniques of scavenging free pages from stacks and limiting any allocations of virtual memory.

If, on either system, an application is requested to shut down and it doesn't, the system will purge the application after waiting approximately 8 seconds. This is the reason an application shouldn't allocate large amounts of stack space. If the application is shutting down due to low-memory conditions, it's quite possible that the

stack space can't be allocated and the application will be suspended. If this happens after the system has requested that the application close, it could be purged from memory without properly saving its state.

In the low- and critical-memory states, applications are limited in the amount of memory they can allocate. In these states, a request for virtual memory larger than what's allowed is refused even if there's memory available to satisfy the request. Remember that it isn't just virtual memory allocations that are limited; allocations on the heap and stack are rejected if, to satisfy the request, those allocations require virtual memory allocations above the allowable limits.

I should point out that sending WM_HIBERNATE messages and automatically closing down applications is performed by the shell of the H/PC and Palm-size PC. The embedded version of Windows CE uses a much simpler shell that doesn't support these memory management techniques. On these embedded systems, you'll have to devise your own strategy for managing low-memory situations.

It should go without saying that applications should check the return codes of any memory allocation call, but since some still don't, I'll say it. *Check the return codes from calls that allocate memory.* There's a much better chance of a memory allocation failing under Windows CE than under Windows NT or Windows 98. Applications must be written to react gracefully to rejected memory allocations.

The Win32 memory management API isn't fully supported by Windows CE, but there's clearly enough support for you to use the limited memory of a Windows CE device to the fullest. A great source for learning about the intricacies of the Win32 memory management API is Jeff Richter's *Advanced Windows* (Microsoft Press, 1997). Jeff spends five chapters on memory management while I have summarized the same topic in one.

We've looked at the program RAM, the part of RAM that is available to applications. Now it's time, in the next chapter, to look at the other part of the RAM, the object store. The object store supports more than a file system. It also supports the registry API as well as a database API unique to Windows CE.

Chapter 7

Files, Databases, and the Registry

One of the areas where Windows CE diverges the farthest from its larger cousins, Windows NT and Windows 98, is in the area of file storage. Instead of relying on ferromagnetic storage media such as floppy disks or hard disk drives, Windows CE implements a unique, RAM-based file system known as the *object store*. In implementation, the object store more closely resembles a database than it does a file allocation system for a disk. In the object store resides the files as well as the registry for the system and any Windows CE databases. Fortunately for the programmer, most of the unique implementation of the object store is hidden behind standard Win32 functions.

The Windows CE file API is taken directly from Win32. Aside from the lack of functions that directly reference volumes, the API is fairly complete. Windows CE implements the standard registry API, albeit without the vast levels of security found in Windows NT. The database API, however, is unique to Windows CE. The database functions provide a simple tool for managing and organizing data. They aren't to be confused with the powerful, multilevel SQL databases found on other computers. Even with its modest functionality, the database API is convenient for storing and organizing simple groups of data, such as address lists or mail folders.

Some differences in the object store do expose themselves to the programmer. Execute-in-place files, stored in ROM, appear as files in the object store but these functions can't be opened and read as standard files. Some of the ROM-based applications are also statically linked to other ROM-based dynamic-link libraries (DLLs).

This means that some ROM-based DLLs can't be replaced by copying an identically named file into the object store.

The concept of the *current directory*, so important in other versions of Windows, isn't present in Windows CE. Files are specified by their complete path. DLLs must be in the Windows directory, the root directory of the object store, or in the root directory of an attached file storage device, such as a PC Card.

As a general rule, Windows CE doesn't support the deep application-level security available under Windows NT. However, because the generic Win32 API was originally based on Windows NT, a number of the functions for file and registry operations have one or more parameters that deal with security rights. Under Windows CE, these values should be set to their default, not security state. This means you should almost always pass NULL in the security parameters for functions that request security information.

In this rather long chapter, I'll first explain the file system and the file API. Then I'll give you an overview of the database API. Finally, we'll do a tour of the registry API. The database API is one of the areas that has experienced a fair amount of change as Windows CE has evolved. Essentially, functionality has been added to later versions of Windows CE. Where appropriate, I'll cover the differences between the different versions and present workarounds, where possible, for maintaining a common code base.

THE WINDOWS CE FILE SYSTEM

The default file system, supported on all Windows CE platforms, is the object store. The object store is equivalent to the hard disk on a Windows CE device. It's a subtly complex file storage system incorporating compressed RAM storage for read/write files and seamless integration with ROM-based files. A user sees no difference between a file in RAM in the object store and those files based in ROM. Files in RAM and ROM can reside in the same directory, and document files in ROM can be opened (although not modified) by the user. In short, the object store integrates the default files provided in ROM with the user-generated files stored in RAM.

In addition to the object store, Windows CE supports multiple, installable file systems that can support up to 256 different storage devices or partitions on storage devices. (The limit is 10 storage devices for WIndows CE 2.0 and earlier.) The interface to these devices is the installable file system (IFS) API. Most Windows CE platforms include an IFS driver for the FAT file system for files stored on ATA flash cards or hard disks. In addition, under Windows CE 2.1 and later, third party manufacturers can write an IFS driver to support other file systems.

Windows CE doesn't use drive letters as is the practice on PCs. Instead, every storage device is simply a directory off the root directory. Under Windows CE 1.0, an

application can count on the name of the directory of the external drive being *PC Card*. If more than one PC Card was inserted, the additional ones are numbered, as in *PC Card 1* and *PC Card 2*, up to *PC Card 99* for the 100th card.[1] Under Windows CE 2.0, the default name was changed from *PC Card* to *Storage Card*, but the numbering concept stayed the same. For Windows CE 2.1, Windows CE doesn't assume a name. Instead it asks the driver what it wants to call the directory.[2] Later in this chapter, I'll demonstrate a method for determining which directories in the root are directories and which are actually storage devices.

As should be expected for a Win32-compatible operating system, the filename format for Windows CE is the same as its larger counterparts. Windows CE supports long filenames. Filenames and their complete path can be up to MAX_PATH in length, which is currently defined at 260 bytes. Filenames have the same *name.ext* format as they do in other Windows operating systems. The extension is the three characters following the last period in the filename and defines the type of file. The file type is used by the shell when determining the difference between executable files and different documents. Allowable characters in filenames are the same as for Windows NT and Windows 98.

Windows CE files support most of the same attribute flags as Windows 98 with a few additions. Attribute flags include the standard read-only, system, hidden, compressed, and archive flags. A few additional flags have been included to support the special RAM/ROM mix of files in the object store.

The Object Store vs. Other Storage Media

To the programmer, the difference between files in the RAM part of the object store and the files based in ROM are subtle. The files in ROM can be detected by a special, in-ROM file attribute flag. However, files in the RAM part of the object store that are always compressed don't have the compressed file attribute as might be expected. The reason is that the compressed attribute is used to indicate when a file or directory is in a compressed state relative to the other files on the drive. In the object store, all files are compressed, which makes the compressed attribute redundant.

The object store in Windows CE has some basic limitations. First, the size of the object store is currently limited to 16 MB of RAM. Given the compression features of the object store, this means that the amount of data that the object store can contain is somewhere around 32 MB. Individual files in the object store are limited to 4 MB under Windows CE 2.0 and earlier. Files under Windows CE 2.1 and later are limited only by the size of the object store's 16-MB limit. These file size limits don't apply to files on secondary storage such as hard disks, PC Cards, or Compact Flash Cards.

---

1. This limit is 10 cards for Windows CE 2.0 and earlier.
2. The Handheld PC Pro uses Storage Card as its default name.

Standard File I/O

Windows CE supports the most of the same file I/O functions found on Windows NT and Windows 98. The same Win32 API calls, such as *CreateFile*, *ReadFile*, *WriteFile* and *CloseFile*, are all supported. A Windows CE programmer must be aware of a few differences, however. First of all, the standard C file I/O functions, such as *fopen*, *fread*, and *fprintf*, aren't supported under Windows CE. Likewise, the old Win16 standards, *_lread*, *_lwrite*, and *_llseek*, aren't supported. This isn't really a huge problem because all of these functions can easily be implemented by wrapping the Windows CE file functions with a small amount of code. Windows CE 2.1 does support basic console library functions such as *printf* for console applications.

Windows CE doesn't support the overlapped I/O that's supported under Windows NT. Files or devices can't be opened with the FILE_FLAG_OVERLAPPED flag nor can reads or writes use the overlapped mode of asynchronous calls and returns.

File operations in Windows CE follow the traditional handle-based methodology used since the days of MS-DOS. Files are opened by means of a function that returns a handle. Read and write functions are passed the handle to indicate the file to act on. Data is read from or written to the offset in the file indicated by a system-maintained file pointer. Finally, when the reading and writing have been completed, the application indicates this by closing the file handle. Now on to the specifics.

Creating and Opening Files

Creating a file or opening an existing file or device is accomplished by means of the standard Win32 function:

```
HANDLE CreateFile (LPCTSTR lpFileName, DWORD dwDesiredAccess,
                   DWORD dwShareMode,
                   LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                   DWORD dwCreationDistribution,
                   DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);
```

The first parameter is the filename of the file to be opened or created. The name of the file should have a fully specified path. Filenames with no path information are assumed to be in the root directory of the object store.

The *dwDesiredAccess* parameter indicate the requested access rights. The allowable flags are GENERIC_READ to request read access to the file and GENERIC_WRITE for write access. Both flags must be passed to get read/write access. You can open a file with neither read nor write permissions. This is handy if you just want to get the attributes of a device. The *dwShareMode* parameter specifies the access rights that can be granted to other processes. This parameter can be FILE_SHARE_READ and/or FILE_SHARE_WRITE. The *lpSecurityAttributes* parameter is ignored by Windows CE and should be set to NULL.

The *dwCreationDistribution* parameter tells *CreateFile* how to open or create the file. The following flags are allowed:

- *CREATE_NEW* Creates a new file. If the file already exists, the function fails.
- *CREATE_ALWAYS* Creates a new file or truncates an existing file.
- *OPEN_EXISTING* Opens a file only if it already exists.
- *OPEN_ALWAYS* Opens a file or creates a file if it doesn't exist. This differs from CREATE_ALWAYS because it doesn't truncate the file to 0 bytes if the file exists.
- *TRUNCATE_EXISTING* Opens a file and truncates it to 0 bytes. The function fails if the file doesn't already exist.

The *dwFlagsAndAttributes* parameter defines the attribute flags for the file if it's being created in addition to flags in order to tailor the operations on the file. The following flags are allowed under Windows CE:

- *FILE_ATTRIBUTE_NORMAL* This is the default attribute. It's overridden by any of the other file attribute flags.
- *FILE_ATTRIBUTE_READONLY* Sets the read-only attribute bit for the file. Subsequent attempts to open the file with write access will fail.
- *FILE_ATTRIBUTE_ARCHIVE* Sets the archive bit for the file.
- *FILE_ATTRIBUTE_SYSTEM* Sets the system bit for the file indicating that the file is critical to the operation of the system.
- *FILE_ATTRIBUTE_HIDDEN* Sets the hidden bit. The file will be visible only to users who have the View All Files option set in the Explorer.
- *FILE_FLAG_WRITE_THROUGH* Write operations to the file won't be lazily cached in memory.
- *FILE_FLAG_RANDOM_ACCESS* Indicates to the system that the file will be randomly accessed instead of sequentially accessed. This flag can help the system determine the proper caching strategy for the file.

Windows CE doesn't support a number of file attributes and file flags that are supported under Windows 98 and Windows NT. The unsupported flags include but aren't limited to the following: FILE_ATTRIBUTE_OFFLINE, FILE_FLAG_OVERLAPPED, FILE_FLAG_NO_BUFFERING, FILE_FLAG_SEQUENTIAL_SCAN, FILE_FLAG_DELETE_ON_CLOSE, FILE_FLAG_BACKUP_SEMANTICS, and FILE_FLAG_POSIX_SEMANTICS.

Under Windows NT and Windows 98, the flag FILE_ATTRIBUTE_TEMPORARY is used to indicate a temporary file, but as we'll see below, it's used by Windows CE to indicate a directory that is in reality a separate drive or network share.

The final parameter in *CreateFile*, *hTemplate*, is ignored by Windows CE and should be set to 0. *CreateFile* returns a handle to the opened file if the function was successful. If the function fails, it returns INVALID_HANDLE_VALUE. To determine why the function failed, call *GetLastError*. If the *dwCreationDistribution* flags included CREATE_ALWAYS or OPEN_ALWAYS, you can determine whether the file previously existed by calling *GetLastError* to see if it returns ERROR_ALREADY_EXISTS. *CreateFile* will set this error code even though the function succeeded.

Reading and Writing

Windows CE supports the standard Win32 functions *ReadFile* and *WriteFile*. Reading a file is as simple as calling the following:

```
BOOL ReadFile (HANDLE hFile, LPVOID lpBuffer,
               DWORD nNumberOfBytesToRead,
               LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);
```

The parameters are fairly self-explanatory. The first parameter is the handle of the opened file to read followed by a pointer to the buffer that will receive the data and the number of bytes to read. The fourth parameter is a pointer to a DWORD that will receive the number of bytes that was actually read. Finally, the *lpOverlapped* parameter must be set to NULL because Windows CE doesn't support overlapped file operations. As an aside, Windows CE does support multiple reads and writes pending on a device; it just doesn't support the ability to return from the function before the operation completes.

Data is read from the file starting at the file offset indicated by the file pointer. After the read has completed, the file pointer is adjusted by the number of bytes read.

ReadFile won't read beyond the end of a file. If a call to *ReadFile* asks for more bytes than remains in the file, the read will succeed, but only the number of bytes remaining in the file will be returned. This is why you must check the variable pointed to by *lpNumberOfBytesRead* after a read completes to learn how many bytes were actually read. A call to *ReadFile* with the file pointer pointing to the end of the file results in the read being successful, but the number of read bytes is set to 0.

Writing to a file is accomplished with this:

```
BOOL WriteFile (HANDLE hFile, LPCVOID lpBuffer,
                DWORD nNumberOfBytesToWrite,
                LPDWORD lpNumberOfBytesWritten,
                LPOVERLAPPED lpOverlapped);
```

The parameters are similar to *ReadFile* with the obvious exception that *lpBuffer* now points to the data that will be written to the file. As in *ReadFile*, the *lpOverlapped* parameter must be NULL. The data is written to the file offset indicated by the file pointer, which is updated after the write so that it points to the byte immediately beyond the data written.

Moving the file pointer

The file pointer can be adjusted manually with a call to the following:

```
DWORD SetFilePointer (HANDLE hFile, LONG lDistanceToMove,
                      PLONG lpDistanceToMoveHigh, DWORD dwMoveMethod);
```

The parameters for *SetFilePointer* are the handle of the file; a signed offset distance to move the file pointer; a second, upper 32-bit offset parameter; and *dwMoveMethod*, a parameter indicating how to interpret the offset. While *lDistanceToMove* is a signed 32-bit value, *lpDistanceToMoveHigh* is a *pointer* to a signed 32-bit value. For file pointer moves of greater than 4 GB, *lpDistanceToMoveHigh* should point to a LONG that contains the upper 32-bit offset of the move. This variable will receive the high 32 bits of the resulting file pointer. For moves of less than 4 GB, simply set *lpDistanceToMoveHigh* to NULL. Clearly, under Windows CE, the *lpDistanceToMoveHigh* parameter is a bit excessive, but having the function the same format as its Windows NT counterpart aids in portability across platforms.

The offset value is interpreted as being from the start of the file if *dwMoveMethod* contains the flag FILE_BEGIN. To base the offset on the current position of the file pointer, use FILE_CURRENT. To base the offset from the end of the file, use FILE_END in *dwMoveMethod*.

SetFilePointer returns the file pointer at its new position after the move has been accomplished. To query the current file position without changing the file pointer, simply call *SetFilePointer* with a zero offset and relative to the current position in the file, as shown here:

```
nCurrFilePtr = SetFilePointer (hFile, 0, NULL, FILE_CURRENT);
```

Closing a file

Closing a file handle is a simple as calling

```
BOOL CloseHandle (HANDLE hObject);
```

This generic call, used to close a number of handles, is also used to close file handles. The function returns TRUE if it succeeds. If the function fails, a call to *GetLastError* will return the reason for the failure.

Truncating a file

When you have finished writing the data to a file, you can close it with a call to *CloseHandle* and you're done. Sometimes, however, you must truncate a file to make it smaller than it currently is. In the days of MS-DOS, the way to set the end of a file was to make a call to write zero bytes to a file. The file was then truncated at the current file pointer. This won't work in Windows CE. To set the end of a file, move the file pointer to the location in the file where you want the file to end and call:

```
BOOL SetEndOfFile (HANDLE hFile);
```

Of course, for this call to succeed, you need write access to the file. The function returns TRUE if it succeeds.

To insure that all the data has been written to a storage device and isn't just sitting around in a cache, you can call this function:

```
WINBASEAPI BOOL WINAPI FlushFileBuffers (HANDLE hFile);
```

The only parameter is the handle to the file you want to flush to the disk, or more likely in Windows CE a PC Card.

Getting file information

A number of calls allow you to query information about a file or directory. To quickly get the attributes knowing only the file or directory name, you can use this function:

```
DWORD GetFileAttributes (LPCTSTR lpFileName);
```

In general, the attributes returned by this function are the same ones that I covered for *CreateFile*, with the addition of the attributes listed below:

- *FILE_ATTRIBUTE_COMPRESSED* The file is compressed.
- *FILE_ATTRIBUTE_INROM* The file is in ROM.
- *FILE_ATTRIBUTE_ROMMODULE* The file is an executable module in ROM formatted for execute-in-place loading. These files can't be opened with *CreateFile*.
- *FILE_ATTRIBUTE_DIRECTORY* The name specifies a directory, not a file.
- *FILE_ATTRIBUTE_TEMPORARY* When this flag is set in combination with FILE_ATTRIBUTE_DIRECTORY, the directory is the root of a secondary storage device, such as a PC Card or a hard disk.

The attribute FILE_ATTRIBUTE_COMPRESSED is somewhat misleading on a Windows CE device. Files in the RAM-based object store are always compressed, but this flag isn't set for those files. On the other hand, the flag does accurately reflect

whether a file in ROM is compressed. Compressed ROM files have the advantage of taking up less space but the disadvantage of not being execute-in-place files.

An application can change the basic file attributes, such as read only, hidden, system, and attribute by calling this function:

```
BOOL SetFileAttributes (LPCTSTR lpFileName, DWORD dwFileAttributes);
```

This function simply takes the name of the file and the new attributes. Note that you can't compress a file by attempting to set its compressed attribute. Under other Windows systems that do support selective compression of files, the way to compress a file is to make a call directly to the file system driver.

A number of other informational functions are supported by Windows CE. All of these functions, however, require a file handle instead of a filename, so the file must have been previously opened by means of a call to *CreateFile*.

File times

The standard Win32 API supports three file times: the time the file was created, the time the file was last accessed (that is, the time it was last read, written, or executed), and the last time the file was written to. That being said, the Windows CE object store keeps track of only one time, the time the file was last written to. One of the ways to query the file times for a file is to call this function:

```
BOOL GetFileTime (HANDLE hFile, LPFILETIME lpCreationTime,
                  LPFILETIME lpLastAccessTime,
                  LPFILETIME lpLastWriteTime);
```

The function takes a handle to the file being queried and pointers to three FILETIME values that will receive the file times. If you're interested in only one of the three values, the other pointers can be set to NULL.

When the file times are queried for a file in the object store, Windows CE copies the last write time into all FILETIME structures. This goes against Win32 documentation, which states that any unsupported time fields should be set to 0. For the FAT file system used on storage cards, two times are maintained: the file creation time and the last write time. When *GetFileTime* is called on a file on a storage card, the file creation and last write times are returned and the last access time is set to 0.

The FILETIME structures returned by *GetFileTime* and other functions can be converted to something readable by calling

```
BOOL FileTimeToSystemTime (const FILETIME *lpFileTime,
                           LPSYSTEMTIME lpSystemTime);
```

This function translates the FILETIME structure into a SYSTEMTIME structure that has documented day, date, and time fields that can be used. One large caveat is that file times are stored in coordinated universal time format (UTC), also known as Greenwich

Mean Time. This doesn't make much difference as long as you're using unreadable FILETIME structures but when you're translating a file time into something readable, a call to

```
BOOL FileTimeToLocalFileTime (const FILETIME *lpFileTime,
                              LPFILETIME lpLocalFileTime);
```

before translating the file time into system time provides the proper time zone translation to the user.

You can manually set the file times of a file by calling

```
BOOL SetFileTime (HANDLE hFile, const FILETIME *lpCreationTime,
                  const FILETIME *lpLastAccessTime,
                  const FILETIME *lpLastWriteTime);
```

The function takes a handle to a file and three times each in FILETIME format. If you want to set only one or two of the times, the remaining parameters can be set to NULL. Remember that file times must be in UTC time, not local time.

For files in the Windows CE object store, setting any one of the time fields results in all three being updated to that time. If you set multiple fields to different times and attempt to set the times for an object store file, the *lpLastWriteTime* takes precedence. Files on storage cards maintain separate creation and last-write times. You must open the file with write access for *SetFileTime* to work.

File size and other information

You can query a file's size by calling

```
DWORD GetFileSize (HANDLE hFile, LPDWORD lpFileSizeHigh);
```

The function takes the handle to the file and an optional pointer to a DWORD that's set to the high 32 bits of the file size. This second parameter can be set to NULL if you don't expect to be dealing with files over 4 GB. *GetFileSize* returns the low 32 bits of the file size.

I've been talking about these last few functions separately, but an additional function, *GetFileInformationByHandle*, returns all this information and more. The function prototyped as

```
BOOL GetFileInformationByHandle (HANDLE hFile,
                        LPBY_HANDLE_FILE_INFORMATION lpFileInformation);
```

takes the handle of an opened file and a pointer to a BY_HANDLE_FILE_INFORMATION structure. The function returns TRUE if it was successful.

The BY_HANDLE_FILE_INFORMATION structure is defined this way:

```
typedef struct _BY_HANDLE_FILE_INFORMATION {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
```

```
    FILETIME ftLastWriteTime;
    DWORD dwVolumeSerialNumber;
    DWORD nFileSizeHigh;
    DWORD nFileSizeLow;
    DWORD nNumberOfLinks;
    DWORD nFileIndexHigh;
    DWORD nFileIndexLow;
    DWORD dwOID;
} BY_HANDLE_FILE_INFORMATION;
```

As you can see, the structure returns data in a number of fields that separate functions return. I'll talk about only the new fields here.

The *dwVolumeSerialNumber* field is filled with the serial number of the volume in which the file resides. The *volume* is what's considered a disk or partition under Windows 98 or Windows NT. Under Windows CE, the volume refers to the object store, a storage card, or a disk on a local area network. For files in the object store, the volume serial number is 0.

The *nNumberOfLinks* field is used by Windows NT's NTFS file system and can be ignored under Windows CE. The *nFileIndexHigh* and *nFileIndexLow* fields contain a systemwide unique identifier number for the file. This number can be checked to see whether two different file handles point to the same file. The File Index value is used under Windows NT and Windows 98, but Windows CE has a more useful value, the *object ID* of the file, which is returned in the *dwOID* field. I'll explain the object ID later in the chapter; for now I'll just mention that it's a universal identifier that can be used to reference directories, files, databases, and individual database records. Handy stuff.

The FileView Sample Program

FileView is an example program that displays the contents of a file in a window. It displays the data in hexadecimal format instead of text, which makes it different from simply opening the file in Microsoft Pocket Word or another editor. FileView is simply a file *viewer*; it doesn't allow you to modify the file. The code for FileView is shown in Figure 7-1.

FileView.rc

```
//=====================================================================
// Resource file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//=====================================================================
```

Figure 7-1. *The Viewer program.* *(continued)*

Figure 7-1. *continued*

```
#include "windows.h"
#include "FileView.h"                   // Program-specific stuff

//------------------------------------------------------------------------
// Icons and bitmaps
ID_ICON ICON   "fileview.ico"           // Program icon

//------------------------------------------------------------------------
// Menu
ID_MENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&Open...",                    IDM_OPEN
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                       IDM_EXIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...",                   IDM_ABOUT
    END
END
//------------------------------------------------------------------------
// About box dialog template
aboutbox DIALOG discardable 10, 10, 160, 40
STYLE  WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU | DS_CENTER |
       DS_MODALFRAME
CAPTION "About"
BEGIN
    ICON  ID_ICON,                      -1,   5,   5,  10,  10
    LTEXT "FileView - Written for the book Programming Windows \
           CE Copyright 1998 Douglas Boling"
                                        -1,  40,   5, 110,  30
END
```

FileView.h

```
//------------------------------------------------------------------------
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//------------------------------------------------------------------------
// Returns number of elements.
```

```
#define dim(x) (sizeof(x) / sizeof(x[0]))

//------------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                                 // Structure associates
    UINT Code;                                      // messages
                                                    // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                                  // Structure associates
    UINT Code;                                      // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);         // function.
};

//------------------------------------------------------------------------
// Generic defines used by application
#define  ID_ICON                1                  // Application icon
                                                    // Resource ID
#define  IDC_CMDBAR             2                  // Command band ID
#define  ID_MENU                3                  // Main menu resource ID
#define  ID_VIEWER              4                  // View control ID

// Menu item IDs
#define  IDM_OPEN               101                // File menu
#define  IDM_EXIT               102
#define  IDM_ABOUT              120                // Help menu

//------------------------------------------------------------------------
// Function prototypes
//
INT MyGetFileName (HWND hWnd, LPTSTR szFileName, INT nMax);

int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoSizeMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoCommandMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);
```

(continued)

Figure 7-1. *continued*

```
// Command functions
LPARAM DoMainCommandOpen (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandExit (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandAbout (HWND, WORD, HWND, WORD);

// Dialog procedures
BOOL CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM);
```

FileView.c

```
//------------------------------------------------------------------------
// FileView - A Windows CE file viewer
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//------------------------------------------------------------------------
#include <windows.h>                    // For all that Windows stuff
#include <commctrl.h>                   // Command bar includes
#include <commdlg.h>                    // Common dialog includes

#include "FileView.h"                   // Program-specific stuff
#include "Viewer.h"                     // Program-specific stuff
//------------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("FileView");
extern TCHAR szViewerCls[];
HINSTANCE hInst;                        // Program instance handle

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_SIZE, DoSizeMain,
    WM_COMMAND, DoCommandMain,
    WM_DESTROY, DoDestroyMain,
};

// Command message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDM_OPEN, DoMainCommandOpen,
    IDM_EXIT, DoMainCommandExit,
    IDM_ABOUT, DoMainCommandAbout,
};
```

```
//-------------------------------------------------------------------
//
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    HWND hwndMain;
    MSG msg;
    int rc = 0;

    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//-------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;
    INITCOMMONCONTROLSEX icex;

    // Register application main window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = MainWndProc;                   // Callback function
    wc.cbClsExtra = 0;                              // Extra class data
    wc.cbWndExtra = 0;                              // Extra window data
    wc.hInstance = hInstance;                       // Owner handle
    wc.hIcon = NULL;                                // Application icon
    wc.hCursor = NULL;                              // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                        // Menu name
    wc.lpszClassName = szAppName;                   // Window class name
```

(continued)

Figure 7-1. *continued*

```
    if (RegisterClass (&wc) == 0) return 1;

    RegisterCtl (hInstance);                    // Register viewer window.

    // Load the command bar common control class.
    icex.dwSize = sizeof (INITCOMMONCONTROLSEX);
    icex.dwICC = ICC_BAR_CLASSES;
    InitCommonControlsEx (&icex);
    return 0;
}
//------------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow){
    HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;
    // Create main window.
    hWnd = CreateWindow (szAppName, TEXT ("FileView"),
                         WS_VISIBLE, CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL,
                         hInstance, NULL);

    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//------------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {
    TermViewer (hInstance, nDefRC);
    return nDefRC;
}
//========================================================================
// Message handling procedures for MainWindow
//------------------------------------------------------------------------
// MainWndProc - Callback function for application window.
//
```

```
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call function.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//---------------------------------------------------------------------
// DoCreateMain - Process WM_CREATE message for window.
//
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB, hwndChild;
    INT nHeight, nCnt;
    RECT rect;
    LPCREATESTRUCT lpcs;

    // Convert lParam into pointer to create structure.
    lpcs = (LPCREATESTRUCT) lParam;

    // Create a minimal command bar that only has a menu and an
    // exit button.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);
    // Insert the menu.
    CommandBar_InsertMenubar (hwndCB, hInst, ID_MENU, 0);
    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    nHeight = CommandBar_Height (hwndCB);

    SetRect (&rect, 0, nHeight, lpcs->cx, lpcs->cy - nHeight);
    hwndChild = CreateViewer (hWnd, &rect, ID_VIEWER);

    // Destroy frame if window not created.
    if (!IsWindow (hwndChild)) {
        DestroyWindow (hWnd);
        return 0;
    }
    ListView_SetItemCount (hwndChild, nCnt);
    return 0;
}
```

(continued)

Figure 7-1. *continued*

```
//-----------------------------------------------------------------
// DoSizeMain - Process WM_SIZE message for window.
//
LRESULT DoSizeMain (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam){
    HWND hwndViewer;
    RECT rect;

    hwndViewer = GetDlgItem (hWnd, ID_VIEWER);

    // Adjust the size of the client rect to take into account
    // the command bar height.
    GetClientRect (hWnd, &rect);
    rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    SetWindowPos (hwndViewer, NULL, rect.left, rect.top,
                  (rect.right - rect.left), rect.bottom - rect.top,
                  SWP_NOZORDER);
    return 0;
}
//-----------------------------------------------------------------
// DoCommandMain - Process WM_COMMAND message for window.
//
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)
            return (*MainCommandItems[i].Fxn)(hWnd, idItem, hwndCtl,
                                              wNotifyCode);
    }
    return 0;
}
//-----------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
```

```
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
//====================================================================
// Command handler routines
//--------------------------------------------------------------------
// DoMainCommandOpen - Process File Open command.
//
LPARAM DoMainCommandOpen (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {
    TCHAR szFileName[MAX_PATH], szText[64];
    HWND hwndViewer;
    INT rc;

    hwndViewer = GetDlgItem (hWnd, ID_VIEWER);

    if (MyGetFileName (hWnd, szFileName, dim(szFileName)) == 0)
        return 0;
    // Tell the viewer control to open the file.
    rc = SendMessage (hwndViewer, VM_OPEN, 0, (LPARAM)szFileName);

    if (rc) {
        wsprintf (szText, TEXT ("File open failed.  rc: %d ") ,rc);
        MessageBox (hWnd, szText, szAppName, MB_OK);
        return 0;
    }
    return 0;
}
//--------------------------------------------------------------------
// DoMainCommandExit - Process Program Exit command.
//
LPARAM DoMainCommandExit (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {

    SendMessage (hWnd, WM_CLOSE, 0, 0);
    return 0;
}
//--------------------------------------------------------------------
// DoMainCommandVText - Process the View Text command.
//
LPARAM DoMainCommandVText (HWND hWnd, WORD idItem, HWND hwndCtl,
                           WORD wNotifyCode) {
    return 0;
}
```

(continued)

Figure 7-1. *continued*

```
//-----------------------------------------------------------------
// DoMainCommandVHex - Process the View Hex command.
//
LPARAM DoMainCommandVHex (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {
    return 0;
}
//-----------------------------------------------------------------
// DoMainCommandAbout - Process the Help | About menu command.
//
LPARAM DoMainCommandAbout(HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {

    // Use DialogBox to create a modal dialog.
    DialogBox (hInst, TEXT ("aboutbox"), hWnd, AboutDlgProc);
    return 0;
}
//=================================================================
// About Dialog procedure
//
BOOL CALLBACK AboutDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                            LPARAM lParam) {
    switch (wMsg) {
        case WM_COMMAND:
            switch (LOWORD (wParam)) {
                case IDOK:
                case IDCANCEL:
                    EndDialog (hWnd, 0);
                    return TRUE;
            }
        break;
    }
    return FALSE;
}

//-----------------------------------------------------------------
// MyGetFileName - Returns a filename using the common dialog.
//
INT MyGetFileName (HWND hWnd, LPTSTR szFileName, INT nMax) {
    OPENFILENAME of;
    const LPTSTR pszOpenFilter = TEXT ("All Documents (*.*)\0*.*\0\0");

    szFileName[0] = '\0';                   // Initial filename
    memset (&of, 0, sizeof (of));           // Initial file open structure
```

```
    of.lStructSize = sizeof (of);
    of.hwndOwner = hWnd;
    of.lpstrFile = szFileName;
    of.nMaxFile = nMax;
    of.lpstrFilter = pszOpenFilter;
    of.Flags = 0;

    if (GetOpenFileName (&of))
        return lstrlen (szFileName);
    else
        return 0;
}
```

Viewer.h

```
//------------------------------------------------------------------------
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//------------------------------------------------------------------------

#define  VM_OPEN                (WM_USER+100)

//------------------------------------------------------------------------
// Function prototypes
//
int RegisterCtl (HINSTANCE hInstance);
HWND CreateViewer (HWND hParent, RECT *prect, int nID);
int TermViewer (HINSTANCE hInstance, int nDefRC);
```

Viewer.c

```
//------------------------------------------------------------------------
// Viewer - A file view control
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//------------------------------------------------------------------------
#include <windows.h>                    // For all that Windows stuff

#include "fileview.h"                   // Program-specific stuff
#include "viewer.h"                     // Control-specific stuff
```

(continued)

Figure 7-1. *continued*

```
//-----------------------------------------------------------------
// Internal function prototypes
LRESULT CALLBACK ViewerWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateViewer (HWND, UINT, WPARAM, LPARAM);
LRESULT DoSizeViewer (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintViewer (HWND, UINT, WPARAM, LPARAM);
LRESULT DoVScrollViewer (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyViewer (HWND, UINT, WPARAM, LPARAM);
LRESULT DoOpenViewer (HWND, UINT, WPARAM, LPARAM);

HFONT GetFixedEquiv (HWND hWnd, HFONT hFontIn);

#define BUFFSIZE   4096
//-----------------------------------------------------------------
// Global data
extern HINSTANCE hInst;              // Program instance handle
HANDLE g_hFile = 0;                  // Handle to the opened file
LONG g_lFileSize;                    // Size of the file
PBYTE g_pBuff = 0;                   // Pointer to file data buffer
LONG g_lFilePtr = 0;                 // Pointer to current offset
                                     // into file
LONG g_lBuffBase = 0;                // Offset into file of buffer data
INT g_nBuffLen = 0;                  // Size of data in file buffer
HFONT g_hFont = 0;                   // Fixed pitch font used for text
INT g_nPageLen = 0;                  // Number of bytes displayed / page

const TCHAR szViewerCls[] = TEXT ("Viewer");

// Message dispatch table for ViewerWindowProc
const struct decodeUINT ViewerMessages[] = {
    WM_CREATE, DoCreateViewer,
    WM_PAINT, DoPaintViewer,
    WM_SIZE, DoSizeViewer,
    WM_VSCROLL, DoVScrollViewer,
    WM_DESTROY, DoDestroyViewer,
    VM_OPEN, DoOpenViewer,
};

//-----------------------------------------------------------------
// RegisterCtl - Register the viewer control.
//
int RegisterCtl (HINSTANCE hInstance) {
    WNDCLASS    wc;
```

```
    // Register application viewer window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = ViewerWndProc;                 // Callback function
    wc.cbClsExtra = 0;                              // Extra class data
    wc.cbWndExtra = 0;                              // Extra window data
    wc.hInstance = hInstance;                       // Owner handle
    wc.hIcon = NULL;                                // Application icon
    wc.hCursor = NULL;                              // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                        // Menu name
    wc.lpszClassName = szViewerCls;                 // Window class name

    if (RegisterClass (&wc) == 0) return 1;

    return 0;
}
//---------------------------------------------------------------------
// CreateViewer - Create a viewer control.
//
HWND CreateViewer (HWND hParent, RECT *prect, int nID) {
    HWND hwndCtl;

    // Create viewer control.
    hwndCtl = CreateWindowEx (0, szViewerCls, TEXT (""),
                              WS_VISIBLE | WS_CHILD | WS_VSCROLL |
                              WS_BORDER, prect->left, prect->top,
                              prect->right - prect->left,
                              prect->bottom - prect->top,
                              hParent, (HMENU)nID, hInst, NULL);
    return hwndCtl;
}
//---------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermViewer (HINSTANCE hInstance, int nDefRC) {

    if (g_hFile)
        CloseHandle (g_hFile);                      // Close the opened file.

    if (g_pBuff)
        LocalFree (g_pBuff);                        // Free buffer.
    if (g_hFont)
        DeleteObject (g_hFont);
    return nDefRC;
}
```

(continued)

Figure 7-1. *continued*

```
//=======================================================================
// Message handling procedures for ViewerWindow
//-----------------------------------------------------------------------
// ViewerWndProc - Callback function for viewer window
//
LRESULT CALLBACK ViewerWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                                LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(ViewerMessages); i++) {
        if (wMsg == ViewerMessages[i].Code)
            return (*ViewerMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//-----------------------------------------------------------------------
// DoCreateViewer - Process WM_CREATE message for window.
//
LRESULT DoCreateViewer (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    LPCREATESTRUCT lpcs;

    // Convert lParam into pointer to create struct.
    lpcs = (LPCREATESTRUCT) lParam;

    // Allocate a buffer.
    g_pBuff = LocalAlloc (LMEM_FIXED, BUFFSIZE);
    if (!g_pBuff) {
        MessageBox (NULL, TEXT ("Not enough memory"),
                    TEXT ("Error"), MB_OK);
        return 0;
    }
    // Create a fixed-pitch font.
    g_hFont = GetFixedEquiv (hWnd, 0);
    return 0;
}
//-----------------------------------------------------------------------
// DoSizeViewer - Process WM_SIZE message for window.
//
LRESULT DoSizeViewer (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam){
    return 0;
}
```

```
//-------------------------------------------------------------------
// ComposeLine - Converts hex buff to unicode string
//
int ComposeLine (INT nOffset, LPTSTR szOut) {
    INT i, nLen, nBuffOffset;
    TCHAR szTmp[16];
    LPBYTE pPtr;
    DWORD cBytes;

    szOut[0] = TEXT ('\0');
    if (g_hFile == 0)                   // If no file open, no text
        return 0;
    // Make sure we have enough bytes in buffer for dump.
    if ((nOffset + 16 > g_lBuffBase + g_nBuffLen) ||
        (nOffset < g_lBuffBase)) {

        // Move file pointer to new place and read data.
        SetFilePointer (g_hFile, nOffset, NULL, FILE_BEGIN);
        if (!ReadFile (g_hFile, g_pBuff, BUFFSIZE, &cBytes, NULL))
            return 0;
        g_lBuffBase = nOffset;
        g_nBuffLen = cBytes;
    }
    nBuffOffset = nOffset - g_lBuffBase;
    if (nBuffOffset > g_nBuffLen)
        return 0;

    // Now create the text for the line.
    wsprintf (szOut, TEXT ("%08X   "), nOffset);

    pPtr = g_pBuff + nBuffOffset;
    nLen = g_nBuffLen - nBuffOffset;
    if (nLen > 16)
        nLen = 16;
    for (i = 0; i < nLen; i++) {
        wsprintf (szTmp, TEXT ("%02X"), *pPtr++);
        lstrcat (szOut, szTmp);
        if (i == 7)
            lstrcat (szOut, TEXT ("-"));
        else
            lstrcat (szOut, TEXT (" "));
    }
    return nLen;
}
```

(continued)

Figure 7-1. *continued*

```
//-----------------------------------------------------------------
// DoPaintViewer - Process WM_PAINT message for window.
//
LRESULT DoPaintViewer (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    TCHAR szOut[128];
    INT nFontHeight;
    INT i, yCurrent;
    TEXTMETRIC tm;
    PAINTSTRUCT ps;
    HFONT hOldFont;
    RECT rect;
    HDC hdc;

    hdc = BeginPaint (hWnd, &ps);
    GetClientRect (hWnd, &rect);

    hOldFont = SelectObject (hdc, g_hFont);

    // Get the height of the default font.
    GetTextMetrics (hdc, &tm);
    nFontHeight = tm.tmHeight + tm.tmExternalLeading;

    i = 0;
    yCurrent = rect.top;
    while (yCurrent < rect.bottom) {
        i += ComposeLine (g_lFilePtr+i, szOut);
        ExtTextOut (hdc, 5, yCurrent, 0, NULL,
                    szOut, lstrlen (szOut),    NULL);

        // Update new draw point.
        yCurrent += nFontHeight;
    }
    SelectObject (hdc, hOldFont);
    EndPaint (hWnd, &ps);
    g_nPageLen = i;
    return 0;
}
//-----------------------------------------------------------------
// DoVScrollViewer - Process WM_VSCROLL message for window.
//
LRESULT DoVScrollViewer (HWND hWnd, UINT wMsg, WPARAM wParam,
                         LPARAM lParam) {
    RECT rect;
    SCROLLINFO si;
    INT sOldPos = g_lFilePtr;
```

```
    GetClientRect (hWnd, &rect);

    switch (LOWORD (wParam)) {
    case SB_LINEUP:
        g_lFilePtr -= 16;
        break;

    case SB_LINEDOWN:
        g_lFilePtr += 16;
        break;

    case SB_PAGEUP:
        g_lFilePtr -= g_nPageLen;
        break;

    case SB_PAGEDOWN:
        g_lFilePtr += g_nPageLen;
        break;

    case SB_THUMBPOSITION:
        g_lFilePtr = HIWORD (wParam);
        break;
    }
    // Check range.
    if (g_lFilePtr < 0)
        g_lFilePtr = 0;
    if (g_lFilePtr > g_lFileSize-16)
        g_lFilePtr = (g_lFileSize - 16) & 0xfffffff0;

    // If scroll position changed, update scrollbar and
    // force redraw of window.
    if (g_lFilePtr != sOldPos) {
        si.cbSize = sizeof (si);
        si.nPos = g_lFilePtr;
        si.fMask = SIF_POS;
        SetScrollInfo (hWnd, SB_VERT, &si, TRUE);

        InvalidateRect (hWnd, NULL, TRUE);
    }
    return 0;
}
//------------------------------------------------------------------------
// DoDestroyViewer - Process WM_DESTROY message for window.
//
```

(continued)

Figure 7-1. *continued*

```
LRESULT DoDestroyViewer (HWND hWnd, UINT wMsg, WPARAM wParam,
                         LPARAM lParam) {
    if (g_hFile)
        CloseHandle (g_hFile);
    g_hFile = 0;
    return 0;
}
//----------------------------------------------------------------------
// DoOpenViewer - Process VM_OPEN message for window.
//
LRESULT DoOpenViewer (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam){
    SCROLLINFO si;

    if (g_hFile)
        CloseHandle (g_hFile);

    // Open the file.
    g_hFile = CreateFile ((LPTSTR)lParam, GENERIC_READ | GENERIC_WRITE,
                          FILE_SHARE_READ, NULL, OPEN_EXISTING,
                          FILE_ATTRIBUTE_NORMAL, NULL);

    if (g_hFile == INVALID_HANDLE_VALUE) {
        g_hFile = 0;
        return GetLastError();
    }
    g_lFilePtr = wParam;
    g_lFileSize = GetFileSize (g_hFile, NULL);

    si.cbSize = sizeof (si);
    si.nMin = 0;
    si.nMax = g_lFileSize;
    si.nPos = g_lFilePtr;
    si.fMask = SIF_POS | SIF_RANGE;
    SetScrollInfo (hWnd, SB_VERT, &si, TRUE);

    InvalidateRect (hWnd, NULL, TRUE);
    return 0;
}
//----------------------------------------------------------------------
HFONT GetFixedEquiv (HWND hWnd, HFONT hFontIn) {
    HDC hdc;
    TEXTMETRIC tm;
    LOGFONT lf;
    HFONT hOldFont;
```

```
    hdc = GetDC (hWnd);
    if (hFontIn == 0)
        hFontIn = GetStockObject (SYSTEM_FONT);
    hOldFont = SelectObject (hdc, hFontIn);
    GetTextMetrics (hdc, &tm);
    SelectObject (hdc, hOldFont);
    ReleaseDC (hWnd, hdc);

    memset (&lf, 0, sizeof (lf));

    lf.lfHeight = -(tm.tmHeight);
    lf.lfWeight    = tm.tmWeight;
    lf.lfItalic    = tm.tmItalic;
    lf.lfUnderline = tm.tmUnderlined;
    lf.lfStrikeOut = tm.tmStruckOut;
    lf.lfCharSet   = tm.tmCharSet;
    lf.lfOutPrecision = OUT_DEFAULT_PRECIS;
    lf.lfClipPrecision = CLIP_DEFAULT_PRECIS;
    lf.lfQuality = DEFAULT_QUALITY;
    lf.lfPitchAndFamily = (tm.tmPitchAndFamily & 0xf0) | TMPF_FIXED_PITCH;
    lf.lfFaceName[0] = TEXT ('\0');

    // Create the font from the LOGFONT structure passed.
    return CreateFontIndirect (&lf);
}
```

The C source code is divided into two files, FileView.c and Viewer.c. FileView.c contains the standard windows functions and the menu command handlers. In Viewer.c, you find the source code for a child window that opens the file and displays its contents. The routines of interest are *DoOpenViewer*, where the file is opened, and *ComposeLine*, where the file data is read. Both of these routines are in Viewer.c. *DoOpenViewer* uses *CreateFile* to open the file with read only access. If the function succeeds, it calls *GetFileSize* to query the size of the file being viewed. This is used to initialize the range of the view window scrollbar. The window is then invalidated to force a WM_PAINT message to be sent.

In the WM_PAINT handler, *OnPaintViewer*, a fixed pitch font is selected into the device context, and data from the file, starting at the current scroll location, is displayed in the window after the application calls the *ComposeLine* function. This routine is responsible for reading the file data into a 4096-byte buffer. The data is then read out of the buffer 16 bytes at a time as each line is displayed. If the data for the line isn't in the file buffer, *ComposeLine* refills the buffer with the proper data from the file by calling *SetFilePointer* and then *ReadFile*.

Memory-Mapped Files and Objects

Memory-mapped files give you a completely different method for reading and writing files. With the standard file I/O functions, files are read as streams of data. To access bytes in different parts of a file, the file pointer must be moved to the first byte, the data read, the file pointer moved to the other byte, and then the file read again.

With memory-mapped files, the file is mapped to a region of memory. Then, instead of using *FileRead* and *FileWrite*, you simply read and write the region of memory that's mapped to the file. Updates of the memory are automatically reflected back to the file itself. Setting up a memory-mapped file is a somewhat more complex process than making a simple call to *CreateFile*, but once a file is mapped, reading and writing the file is trivial.

Memory-mapped files

Windows CE uses a slightly different procedure from Windows NT or Windows 98 to access a memory-mapped file. To open a file for memory-mapped access, a new function, unique to Windows CE, is used; it's named *CreateFileForMapping*. The prototype for this function is the following:

```
HANDLE CreateFileForMapping (LPCTSTR lpFileName, DWORD dwDesiredAccess,
                             DWORD dwShareMode,
                             LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                             DWORD dwCreationDisposition,
                             DWORD dwFlagsAndAttributes,
                             HANDLE hTemplateFile);
```

The parameters for this function are similar to those for *CreateFile*. The filename is the name of the file to read. The *dwDesiredAccess* parameter, specifying the access rights to the file, must be a combination of GENERIC_READ and GENERIC_WRITE, or it must be 0. The security attributes must be NULL, while the *hTemplateFile* parameter is ignored by Windows CE. Note that Windows CE 2.1 is the first version of Windows CE to support write access to memory-mapped files. If you try to use this function in versions earlier than 2.1, it will fail if the *dwDesiredAccess* parameter contains the GENERIC_WRITE flag.

The handle returned by *CreateFileForMapping* can then be passed to

```
HANDLE CreateFileMapping (HANDLE hFile,
                          LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
                          DWORD flProtect, DWORD dwMaximumSizeHigh,
                          DWORD dwMaximumSizeLow, LPCTSTR lpName);
```

This function creates a file mapping object and ties the opened file to it. The first parameter for this function is the handle to the opened file. The security attributes parameter must be set to NULL under Windows CE. The *flProtect* parameter should be loaded with the protection flags for the virtual pages that will contain the file data.

The maximum size parameters should be set to the expected maximum size of the object, or they can be set to 0 if the object should be the same size as the file being mapped. The *lpName* parameter allows you to specify a name for the object. This is handy when you're using a memory-mapped file to share information across different processes. Calling *CreateFileMapping* with the name of an already-opened file-mapping object returns a handle to the object already opened instead of creating a new one.

Once a mapping object has been created, a view into the object is created by calling

```
LPVOID MapViewOfFile (HANDLE hFileMappingObject, DWORD dwDesiredAccess,
                      DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow,
                      DWORD dwNumberOfBytesToMap);
```

MapViewOfFile returns a pointer to memory that's mapped to the file. The function takes as its parameters the handle of the mapping object just opened as well as the access rights, which can be FILE_MAP_READ, FILE_MAP_WRITE, or FILE_MAP_ALL_ACCESS. The offset parameters let you specify the starting point within the file that the view starts, while the *dwNumberOfBytesToMap* parameter specifies the size of the view window.

These last three parameters are useful when you're mapping large objects. Instead of attempting to map the file as one large object, you can specify a smaller view that starts at the point of interest in the file. This reduces the memory required because only the view of the object, not the object itself, is backed up by physical RAM.

When you're finished with the memory-mapped file, a little cleanup is required. First a call to

```
BOOL UnmapViewOfFile (LPCVOID lpBaseAddress);
```

unmaps the view to the object. The only parameter is the pointer to the base address of the view.

Next, a call should be made to close the mapping object and the file itself. Both these actions are accomplished by means of calls to *CloseHandle*. The first call should be to close the memory-mapped object, and then *CloseHandle* should be called to close the file.

The code fragment that follows shows the entire process of opening a file for memory mapping, creating the file-mapping object, mapping the view, then cleaning up. The routine is written to open the file in read-only mode. This allows the code to run under all versions of Windows CE.

```
HANDLE hFile, hFileMap;
PBYTE pFileMem;
TCHAR szFileName[MAX_PATH];
```

(continued)

```
// Get the filename.

hFile = CreateFileForMapping (szFileName, GENERIC_READ,
                              FILE_SHARE_READ, NULL,
                              OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL |
                              FILE_FLAG_RANDOM_ACCESS,0);

if (hFile != INVALID_HANDLE_VALUE) {

    hFileMap = CreateFileMapping (hFile, NULL, PAGE_READONLY, 0, 0, 0);
    if (hFileMap) {
        pFileMem = MapViewOfFile (hFileMap, FILE_MAP_READ, 0, 0, 0);
        if (pFileMem) {
            //
            // Use the data in the file.
            //

            // Start cleanup by unmapping view.
            UnmapViewOfFile (pFileMem);
        }
        CloseHandle (hFileMap);
    }
    CloseHandle (hFile);
}
```

Memory-mapped objects

One of the more popular uses for memory-mapped objects is for interprocess communication. For this purpose, you don't need to have an actual file; it's the shared memory that's important. Windows CE supports entities referred to as *unnamed memory-mapped objects*. These objects are memory-mapped objects that, under Windows NT and Windows 98, are backed up by the paging file but under Windows CE are simply areas of virtual memory with only program RAM to back up the object. Without the paging file, these objects can't be as big as they would be under Windows NT or Windows 98 but Windows CE does have a way of minimizing the RAM required to back up the memory-mapped object.

You create such a memory-mapped object by eliminating the call to *CreateFileForMapping* and passing a –1 in the handle field of *CreateFileMapping*. Since no file is specified, you must specify the size of the memory-mapped region in the maximum size fields of *CreateFileMapping*. The following routine creates a 16-MB region using a memory-mapped file:

```
// Create a 16-MB memory mapped object.
hNFileMap = CreateFileMapping ((HANDLE)-1, NULL, PAGE_READWRITE,
                               0, 0x1000000, NULL);
```

```
if (hNFileMap)
    // Map in the object.
    pNFileMem = MapViewOfFile (hNFileMap,
                               FILE_MAP_WRITE, 0, 0, 0);
```

The memory object created by the code above doesn't actually commit 16 MB of RAM. Instead, only the address space is reserved. Pages are autocommitted as they're accessed. This process allows an application to create a huge, sparse array of pages that takes up only as much physical RAM as is needed to hold the data. At some point, however, if you start reading or writing to a greater number of pages, you'll run out of memory. When this happens, the system generates an exception. I'll talk about how to deal with exceptions in the next chapter. The important thing to remember is that if you really need RAM to be committed to a memory-mapped object, you need to read each of the pages so that the system will commit physical RAM to that object. Of course, don't be too greedy with RAM; commit only the pages you absolutely require.

Naming a memory-mapped object

A memory-mapped object can be named by passing a string to *CreateFileMapping*. This isn't the name of a file being mapped. Instead the name identifies the mapping object being created. In the previous example, the region was unnamed. The following code creates a named memory-mapped object named *Bob*. This name is global so that if another process opens a mapping object with the same name, the two processes will share the same memory mapped object.

```
// Create a 16-MB memory mapped object.
hNFileMap = CreateFileMapping ((HANDLE)-1, NULL, PAGE_READWRITE,
                               0, 0x1000000, TEXT ("Bob"));
if (hNFileMap)
    // Map in the object.
    pNFileMem = MapViewOfFile (hNFileMap,
                               FILE_MAP_WRITE, 0, 0, 0);
```

The difference between named and unnamed file mapping objects is that a named object is allocated only once in the system. Subsequent calls to *CreateFileMapping* that attempt to create a region with the same name will succeed, but the function will return a handle to the original mapping object instead of creating a new one. For unnamed objects, the system creates a new object each time *CreateFileMapping* is called.

When using a memory-mapped object for interprocess communication, processes should create a named object and pass the name of the region to the second process, not a pointer. While the first process can simply pass a pointer to the mapping region to the other process, this isn't advisable. If the first process frees the memory-mapped

file region while the second process is still accessing the file, an exception will occur. Instead, the second process should create a memory-mapped object with the same name as the initial process. Windows knows to pass a pointer to the same region that was opened by the first process. The system also increments a use count to track the number of opens. A named memory-mapped object won't be destroyed until all processes have closed the object. This assures a process that the object will remain at least until it closes the object itself. The XTALK example in Chapter 8 provides an example of how to use a named memory mapped object for interprocess communication.

Navigating the File System

Now that we've seen how files are read and written, let's take a look at how the files themselves are managed in the file system. Windows CE supports most of the convenient file and directory management APIs, such as *CopyFile*, *MoveFile*, and *CreateDirectory*.

File and directory management

Windows CE supports a number of functions useful in file and directory management. You can move files using *MoveFile*, copy them using *CopyFile*, and delete them using *DeleteFile*. You can create directories using *CreateDirectory* and delete them using *RemoveDirectory*. While most of these functions are straightforward, I should cover a few intricacies here.

To copy a file, call

```
BOOL CopyFile (LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName,
               BOOL bFailIfExists);
```

The parameters are the name of the file to copy and the name of the destination directory. The third parameter indicates whether the function should overwrite the destination file if one already exists before the copy is made.

Files and directories can be moved and renamed using

```
BOOL MoveFile (LPCTSTR lpExistingFileName, LPCTSTR lpNewFileNam);
```

To move a file, simply indicate the source and destination names for the file. The destination file must not already exist. File moves can be made within the object store, from the object store to an external drive, or from an external drive to the object store. *MoveFile* can also be used to rename a file. In this case, the source and target directories remain the same; only the name of the file changes.

MoveFile can also be used in the same manner to move or rename directories. The only exception is that *MoveFile* can't move a directory from one volume to another. Under Windows CE, *MoveFile* moves a directory and all its subdirectories and

files to a different location within the object store or different locations within another volume.

Deleting a file is as simple as calling

```
BOOL DeleteFile (LPCTSTR lpFileName);
```

You pass the name of the file to delete. For the delete to be successful, the file must not be currently open.

You can create and destroy directories using the following two functions:

```
BOOL CreateDirectory (LPCTSTR lpPathName,
                      LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

and

```
BOOL RemoveDirectory (LPCTSTR lpPathName);
```

CreateDirectory takes the name of the directory to create and a security parameter that should be NULL under Windows CE. *RemoveDirectory* deletes a directory. The directory must be empty for the function to be successful.

Finding files

Windows CE supports the basic *FindFirstFile*, *FindNextFile*, *FindClose* procedure for enumerating files as is supported under Windows NT or Windows 98. Searching is accomplished on a per-directory basis using template filenames with wild card characters in the template.

Searching a directory involves first passing a filename template to *FindFirstFile*, which is prototyped in this way:

```
HANDLE FindFirstFile (LPCTSTR lpFileName,
                      LPWIN32_FIND_DATA lpFindFileData);
```

The first parameter is the template filename used in the search. This filename can contain a fully specified path if you want to search a directory other than the root. Windows CE has no concept of *Current Directory* built into it; if no path is specified in the search string, the root directory of the object store is searched.

As would be expected, the wildcards for the filename template are ? and *. The question mark (?) indicates that any single character can replace the question mark. The asterisk (*) indicates that any number of characters can replace the asterisk. For example, the search string *\windows\alarm?.wav* would return the files \windows\alarm1.wav, \windows\alarm2.wav, and \windows\alarm3.wav. On the other hand, a search string of *\windows*.wav* would return all files in the windows directory that have a wav extension.

The second parameter of *FindFirstFile* is a pointer to a WIN32_FIND_DATA structure as defined at the top of the following page.

```
typedef struct _WIN32_FIND_DATA {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
    DWORD nFileSizeHigh;
    DWORD nFileSizeLow;
    DWORD dwOID;
    WCHAR cFileName[ MAX_PATH ];
} WIN32_FIND_DATA;
```

This structure is filled with the file data for the first file found in the search. The fields shown are similar to what we've seen.

If *FindFirstFile* finds no files or directories that match the template filename, it returns INVALID_HANDLE_VALUE. If at least one file is found, *FindFirstFile* fills in the WIN32_FIND_DATA structure with the specific data for the found file and returns a handle value that you use to track the current search.

To find the next file in the search, call this function:

```
BOOL FindNextFile (HANDLE hFindFile,
                   LPWIN32_FIND_DATA lpFindFileData);
```

The two parameters are the handle returned by *FindFirstFile* and a pointer to a find data structure. *FindNextFile* returns TRUE if a file matching the template passed to *FindFirstFile* is found and fills in the appropriate file data in the WIN32_FIND_DATA structure. If no file is found, *FindNextFile* returns FALSE.

When you've finished searching either because *FindNextFile* returned FALSE or because you simply don't want to continue searching, you must call this function:

```
BOOL FindClose (HANDLE hFindFile);
```

This function accepts the handle returned by *FindFirstFile*. If *FindFirstFile* returned INVALID_ HANDLE_VALUE, you shouldn't call *FindClose*.

The following short code fragment encompasses the entire file search process. This code computes the total size of all files in the Windows directory.

```
WIN32_FIND_DATA fd;
HANDLE hFind;
INT nTotalSize = 0;

// Start search for all files in the windows directory.
hFind = FindFirstFile (TEXT ("\\windows\\*.*"), &fd);

// If a file was found, hFind will be valid.
if (hFind != INVALID_HANDLE_VALUE) {
```

```
    // Loop through found files.  Be sure to process file
    // found with FindFirstFile before calling FindNextFile.
    do {
        // If found file is not a directory, add its size to
        // the total.  (Assume that the total size of all files
        // is less than 2 GB.)
        if (!(fd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY))
            nTotalSize += fd.nFileSizeLow;

    // See if another file exists.
    } while (FindNextFile (hFind, &fd));

    // Clean up by closing file search handle.
    FindClose (hFind);
}
```

In this example, the windows directory is searched for all files. If the found "file" isn't a directory, that is, if it's a true file, its size is added to the total. Notice that the return handle from *FindFirstFile* must be checked, not only so that you know whether a file was found but also to prevent *FindClose* from being called if the handle is invalid.

Determining drives from directories

As I mentioned at the beginning of this chapter, Windows CE doesn't support the concept of drive letters so familiar to MS-DOS and Windows users. Instead, file storage devices such as PC Cards or even hard disks are shown as directories in the root directory. That leads to the question, "How can you tell a directory from a drive?" The newer versions of Windows CE, starting with version 2.1, don't have a predefined name for these other storage devices. Using a predefined name is shaky at best, anyway, given that the name was originally *PC Card* and then changed to *Storage Card*. Instead, you need to look at the file attributes for the directory. Directories that are actually secondary storage devices—that is, they store files in a place other than the object store—have the file attribute flag FILE_ATTRIBUTE_TEMPORARY set. So, finding storage devices on any version of Windows CE is fairly easy as is shown in the following code fragment:

```
WIN32_FIND_DATA fd;
HANDLE hFind;
TCHAR szPath[MAX_PATH];
ULARGE_INTEGER lnTotal, lnFree;

lstrcpy (szPath, TEXT ("\\*.*"));
hFind = FindFirstFile (szPath, &fd);

if (hFind != INVALID_HANDLE_VALUE) {
```

(continued)

```
        do {
            if ((fd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) &&
                (fd.dwFileAttributes & FILE_ATTRIBUTE_TEMPORARY)) {

                // Get the disk space statistics for drive.
                GetDiskFreeSpaceEx (fd.cFileName, NULL, &lnTotal,
                                    &lnFree);
            }
        } while (FindNextFile (hFind, &fd));
        FindClose (hFind);
    }
```

This code uses the find first/find next functions to search the root directory for all directories with the FILE_ATTRIBUTE_TEMPORARY attribute set.

Notice in the code I just showed you, the call to this function:

```
BOOL GetDiskFreeSpaceEx (LPCWSTR lpDirectoryName,
                         PULARGE_INTEGER lpFreeBytesAvailableToCaller,
                         PULARGE_INTEGER lpTotalNumberOfBytes,
                         PULARGE_INTEGER lpTotalNumberOfFreeBytes);
```

This function provides information about the total size of the drive, and amount of free space it contains. The first parameter is the name of any directory on the drive in question. This doesn't have to be the root directory of the drive. *GetDiskFreeSpaceEx* returns three values: the free bytes available to the caller, the total size of the drive, and the total free space on the drive. These values are returned in three ULARGE_INTEGER structures. These structures contain two DWORD fields named *LowPart* and *HighPart*. This allows *GetDiskFreeSpaceEx* to return 64-bit values. Those 64-bit values can come in handy on Windows NT and Windows 98, where the drives can be large. If you aren't interested in one or more of the fields, you can pass a NULL in place of the pointer for that parameter. You can also use *GetDiskFreeSpaceEx* to determine the size of the object store.

Another function that can be used to determine the size of the object store is

```
BOOL GetStoreInformation (LPSTORE_INFORMATION lpsi);
```

GetStoreInformation takes one parameter a pointer to a STORE_INFORMATION structure defined as

```
typedef struct STORE_INFORMATION {
    DWORD dwStoreSize;
    DWORD dwFreeSize;
} STORE_INFORMATION, *LPSTORE_INFORMATION;
```

As you can see, this structure simply returns the total size and amount of free space in the object store. Why would you use *GetStoreInformation* when *GetDiskFreeSpaceEx* is available and more general? Because *GetDiskFreeSpaceEx* wasn't available under Windows CE 1.0 but *GetStoreInformation* was.

That covers the Windows CE file API. As you can see, very little Windows CE–unique code is necessary when you're working with the object store. Now let's look at an entirely new set of functions, the database API.

DATABASES

Windows CE gives you an entirely unique set of database APIs not available under the other versions of Windows. The database implemented by Windows CE is simple, with only one level and a maximum of four sort indexes, but it serves as an effective tool for organizing uncomplicated data, such as address lists or to-do lists.

Under the first two versions of Windows CE, databases could reside only in the object store, not on external media such as PC Cards. Starting with the release of Windows CE 2.1 however, Windows CE can now work with databases on PC Cards or other storage devices. This new feature required changes to the database API, effectively doubling the number of functions with xxx*Ex* database functions now shadowing the original database API. While the newer versions of Windows CE still support the original database functions, those functions can be used only with databases stored in the object store.

Basic Definitions

A Windows CE database is composed of a series of records. Records can contain any number of properties. These properties can be one of the data types shown in Figure 7-2.

Data Type	*Description*
iVal	2-byte signed integer
uiVal	2-byte unsigned integer
lVal	4-byte signed integer
ulVal	4-byte unsigned integer
FILETIME	A time and date structure
LPWSTR	0-terminated Unicode string
CEBLOB	A collection of bytes
BOOL*	Boolean
Double*	8-byte signed value

* This data type supported only under Windows CE 2.1 and later

Figure 7-2. *Database data types supported by Windows CE.*

Records can't contain other records. Also, records can reside on only one database. Windows CE databases can't be locked. However, Windows CE does provide a method of notifying a process that another thread has modified a database.

A Windows CE database can have up to four sort indices. These indices are defined when the database is created but can be redefined later, although the restructuring of a database takes a large amount of time. Each sort index by itself results in a fair amount of overhead, so you should limit the number of sort indices to what you really need.

In short, Windows CE gives you a basic database functionality that helps applications organize simple data structures. The pocket series of Windows CE applications provided by Microsoft with the H/PC, H/PC Pro, and the Palm-size PC use the database API to manage the address book, the task list, and e-mail messages. So, if you have a collection of data, this database API might just be the best method of managing that data.

Designing a database

Before you can jump in with a call to *CeCreateDatabase*, you need to think carefully about how the database will be used. While the basic limitations of the Windows CE database structure rule out complex databases, the structure is quite handy for managing collections of related data on a small personal device, which, after all, is one of the target markets for Windows CE.

Each record in a database can have as many properties as you need as long as they don't exceed the basic limits of the database structure. The limits are fairly loose. An individual property can't exceed the constant CEDB_MAXPROPDATASIZE, which is set to 65,471. A single record can't exceed CEDB_MAXRECORDSIZE, currently defined as 131,072.

Database volumes

Starting with Windows CE 2.1, database files can now be stored in volumes instead of directly in the object store. A database volume is nothing more than a specially formatted file where Windows CE databases can be located. Because database volumes can be stored on file systems other than the object store, database information can be stored on PC Cards or similar external storage devices. The most immediate disadvantage of working with database volumes is that they must be first *mounted* and then *unmounted* after you close the databases within the volume. Essentially, mounting the database creates or opens the file that contains one or more databases along with the transaction data for those databases.

There are disadvantages to database volumes aside from the overhead of mounting and unmounting the volumes. Database volumes are actual files and therefore can be deleted by means of standard file operations. The volumes are, by default, marked as hidden, but that wouldn't deter the intrepid user from finding and

deleting a volume in a desperate search for more space on the device. Databases created directly within the object store aren't files and therefore are much more difficult for the user to accidentally delete.

The Database API

Once you have planned your database, given the restrictions and considerations necessary to it, the programming can begin.

Mounting a database volume

To mount a database volume, call

```
BOOL CeMountDBVol (PCEGUID pguid, LPWSTR lpszVol, DWORD dwFlags);
```

This function performs a dual purpose: it can create a new volume or open an existing volume. The first parameter is a pointer to a guid. *CeMountDBVol* returns a guid that's used by many of the *Ex* database functions to identify the location of the database file. You shouldn't confuse the CEGUID-type guid parameter in the database functions with the GUID type that is used by OLE and parts of the Windows shell. A CEGUID is simply a handle that tracks the opened database volume.

The second parameter in *CeMountDBVol* is the name of the volume to mount. This isn't a database name, but the name of a file that will contain one or more databases. Since the parameter is a filename, you should define it in \path\name.ext format. The standard extension should be cdb.

The last parameter, *dwFlags*, should be loaded with flags that define how this function acts. The possible flags are the following:

- *CREATE_NEW* Creates a new database volume. If the volume already exists, the function fails.
- *CREATE_ALWAYS* Creates a new database volume. If the volume already exists, it overwrites the old volume.
- *OPEN_EXISTING* Opens a database volume. If the volume doesn't exist, the function fails.
- *OPEN_ALWAYS* Opens a database volume. If the volume doesn't exist, a new database volume is created.
- *TRUNCATE_EXISTING* Opens a database volume and truncates it to 0 bytes. If the volume already exists, the function fails.

If the flags resemble the action flags for *CreateFile*, they should. The actions of *CeMountDBVol* essentially mirror *CreateFile* except that instead of creating or

opening a generic file, *CeMountDBVol* creates or opens a file especially designed to hold databases.

If the function succeeds, it returns TRUE and the guid is set to a value that is then passed to the other database functions. If the function fails, a call to *GetLastError* returns an error code indicating the reason for the failure.

Database volumes can be open by more than one process at a time. The system maintains a reference count for the volume. As the last process unmounts a database volume, the system unmounts the volume.

Enumerated mounted database volumes

You can determine what database volumes are currently mounted by repeatedly calling this function:

```
BOOL CeEnumDBVolumes (PCEGUID pguid, LPWSTR lpBuf, DWORD dwSize);
```

The first time you call *CeEnumDBVolumes*, set the guid pointed to by *pguid* to be invalid. You use the CREATE_INVALIDGUID macro to accomplish this. *CeEnumDBVolumes* returns TRUE if a mounted volume is found and returns the guid and name of that volume in the variables pointed to by *pguid* and *lpBuff*. The *dwSize* parameter should be loaded with the size of the buffer pointed to by *lpBuff*. To enumerate the next volume, pass the guid returned by the previous call to the function. Repeat this process until *CeEnumDBVolumes* returns FALSE. The code below demonstrates this process:

```
CEGUID guid;
TCHAR szVolume[MAX_PATH];
INT nCnt = 0;

CREATE_INVALIDGUID (&guid);
while (CeEnumDBVolumes (&guid, szVolume, sizeof (szVolume))) {
    // guid contains the guid of the mounted volume.
    // szVolume contains the name of the volume.
    nCnt++;   // Count the number of mounted volumes.
}
```

Unmounting a database volume

When you have completed using the volume, you should unmount it by calling this function:

```
BOOL CeUnmountDBVol (PCEGUID pguid);
```

The function's only parameter is the guid of a mounted database volume. Calling this function is necessary when you no longer need a database volume and you want to free system resources. Database volumes are only unmounted when all applications that have mounted the volume have called *CeUnmountDBVol*.

Using the object store as a database volume

If you're writing an application for Windows CE 2.1 or later, you still might want to use the new *Ex* database functions but not want to use a separate database volume. Because most of the new *Ex* functions require a CEGUID that identifies a database volume, you need a CEGUID that references the system object store. Fortunately, one can be created using this macro:

```
CREATE_SYSTEMGUID (PCEGUID pguid);
```

The parameter is, of course, a pointer to a CEGUID. The value set in the CEGUID by this macro can then be passed to any of the *Ex* database functions as a placeholder for a separate volume CEGUID. Databases created within this system CEGUID are actually created directly in the object store as if you were using the old non-*Ex* database functions.

Creating a database

Creating a database is accomplished by calling one of two functions, *CeCreateDatabase* or *CeCreateDatabaseEx*. The newer function is *CeCreateDatabaseEx* and works only for Windows CE 2.1 and later. *CeCreateDatabase* is the proper function to use on Windows CE 2.0. First, I'm going to talk about *CeCreateDatabase*, then I'll talk about the expanded functionality of *CeCreateDatabaseEx*.

CeCreateDatabase is prototyped as

```
CEOID CeCreateDatabase (LPWSTR lpszName, DWORD dwDbaseType,
                        WORD wNumSortOrder,
                        SORTORDERSPEC * rgSortSpecs);
```

The first parameter of the function is the name of the new database. Unlike filenames, the database name is limited to 32 characters, including the terminating zero. The *deDbaseType* parameter is a user-defined parameter that can be employed to differentiate families of databases. For example, you might want to use a common type value for all databases that your application creates. This allows them to be easily enumerated. At this point, there are no rules for what type values to use. Some example type values used by the Microsoft Pocket suite are listed in Figure 7-3.

Database	*Value*
Contacts	24 (18 hex)
Appointments	25 (19 hex)
Tasks	26 (1A hex)
Categories	27 (1B hex)

Figure 7-3. *Predefined database types.*

The values listed in Figure 7-3 aren't guaranteed to remain constant; I simply wanted to show some typical values. If you use a 4-byte value, it shouldn't be too hard to find a unique database type for your application although there's no reason another application couldn't use the same type.

The final two parameters specify the sort specification for the database. The parameter *wNumSortOrder* specifies the number of sort specifications, up to a maximum of 4, while the *rgSortSpecs* parameter points to an array of SORTORDERSPEC structures defined as

```
typedef struct _SORTORDERSPEC {
    PEGPROPID propid;
    DWORD dwFlags;
} SORTORDERSPEC;
```

The first field in the SORTORDERSPEC structure is a property ID or PEGPROPID. A property ID is nothing more than a unique identifier for a property in the database. Remember that a property is one field within a database record. The property ID is a DWORD value with the low 16 bits containing the data type and the upper 16 bits containing an application-defined value. These values are defined as constants and are used by various database functions to identify a property. For example, a property that contained the name of a contact might be defined as

```
#define PID_NAME        MAKELONG (CEVT_LPWSTR, 1)
```

The MAKELONG macro simply combines two 16-bit values into a DWORD or LONG. The first parameter is the low word or the result, while the second parameter becomes the high word. In this case, the CEVT_LPWSTR constant indicates that the property contains a string while the second parameter is simply a value that uniquely identifies the *Name* property, distinguishing it from other string properties in the record.

The second field in the SORTORDERSPEC, *dwFlags*, contains flags that define how the sort is to be accomplished. The following flags are defined for this field:

- *CEDB_SORT_DESCENDING* The sort is to be in descending order. By default, properties are sorted in ascending order.
- *CEDB_SORT_CASEINSENSITIVE* The sort should ignore the case of the letters in the string.
- *CEDB_SORT_UNKNOWNFIRST* Records without this property are to be placed at the start of the sort order. By default, these records are placed last.

A typical database might have three or four sort orders defined. After a database is created, these sort orders can be changed by calling *CeSetDatabaseInfo*. However,

this function is quite resource intensive and can take from seconds up to minutes to execute on large databases.

If you want to open a database outside of the object store, you can use the following function:

```
CEOID CeCreateDatabaseEx (PCEGUID pguid, CEDBASEINFO *pInfo);
```

This function takes a *pguid* parameter that identifies the mounted database volume where the database is located. The second parameter is a pointer to a CEDBASEINFO structure defined as

```
typedef struct _CEDBASEINFO {
    DWORD    dwFlags;
    WCHAR    szDbaseName[CEDB_MAXDBASENAMELEN];
    DWORD    dwDbaseType;
    WORD     wNumRecords;
    WORD     wNumSortOrder;
    DWORD    dwSize;
    FILETIME ftLastModified;
    SORTORDERSPEC rgSortSpecs[CEDB_MAXSORTORDER];
} CEDBASEINFO;
```

As you can see, this structure contains a number of the same parameters passed individually to *CeCreateDatabase*. The *szDatabaseName*, *dwDbaseType*, *wNumSortOrder*, and *rgSortSpecs* fields must be initialized in the same manner as they are when you call *CeCreateDatabase*.

The *dwFlags* parameter has two uses. First, it contains flags indicating which fields in the structure are valid. The possible values for the *dwFlags* field are: CEDB_VALIDNAME, CEDB_VALIDTYPE, CEDB_VALIDSORTSPEC, and CEDB_VALIDDBFLAGS. When you're creating a database, it's easier to simply set the *dwFlags* field to CEDB_VALIDCREATE, which is a combination of the flags I just listed. An additional flag, CEDB_VALIDMODTIME, is used when this structure is used by *CeOidGetInfo*.

The other use for the *dwFlags* parameter is to specify the properties of the database. The only flag currently defined is CEDB_NOCOMPRESS. This flag can be specified if you don't want the database you're creating to be compressed. By default, all databases are compressed, which saves storage space at the expense of speed. By specifing the CEDB_NOCOMPRESS flag, the database will be larger but you will be able to read and write the database faster.

You can use *CeCreateDatabaseEx* but create a database within the object store instead of within a separate database volume. The advantage of this strategy is that the database itself isn't created within a file and is therefore safer from a user who might delete the database volume.

The value returned by either *CeCreateDatabase* or *CeCreateDatabaseEx* is a CEOID. We have seen this kind of value a couple of times so far in this chapter. It's

an ID value that uniquely identifies the newly created database, not just among other databases, but also among all files, directories, and even database records in the file system. If the value is 0, an error occurred while you were trying to create the database. You can call *GetLastError* to diagnose the reason the database creation failed.

Opening a database

In contrast to what happens when you create a file, creating a database doesn't also open the database. To do that, you must make an additional call to

```
HANDLE CeOpenDatabase(PCEOID poid, LPWSTR lpszName, CEPROPID propid,
                      DWORD dwFlags, HWND hwndNotify);
```

A database can be opened either by referencing its CEOID value or by referencing its name. To open the database by using its name, set the value pointed to by the *poid* parameter to 0 and specify the name of the database using the *lpszName* parameter. If you already know the CEOID of the database, simply put that value in the parameter pointed to by *poid*. If the CEOID value isn't 0, the functions ignore the *lpszName* parameter.

The *propid* parameter specifies which of the sort order specifications should be used to sort the database while it's opened. A Windows CE database can have only one active sort order. To use a different sort order, you can open a database again, specifying a different sort order.

The *dwFlags* parameter can contain either 0 or CEDB_AUTOINCREMENT. If CEDB_AUTOINCREMENT is specified, each read of a record in the database results in the database pointer being moved to the next record in the sort order. Opening a database without this flag means that the record pointer must be manually moved to the next record to be read. This flag is helpful if you plan to read the database records in sequential order.

The final parameter is the handle of a window that's to be notified when another process or thread modifies the database. This message-based notification allows you to monitor changes to the database while you have it opened. When a database is opened with *CeOpenDatabase*, Windows CE sends the following three messages to notify you of changes.

- *DB_CEOID_CREATED* A record has been created in the database.
- *DB_CEOID_CHANGED* A record has been changed.
- *DB_CEOID_RECORD_DELETED* A record has been deleted.

These messages are encoded as WM_USER+1, WM_USER+3, and WM_USER+6 respectively, so be careful not to use these low WM_USER messages for your own purposes if you want to have that window monitor database changes.

If the function is successful, it returns a handle to the opened database. This handle is then used by the other database functions to reference this opened database. If the handle returned is 0, the function failed for some reason and you can use *GetLastError* to identify the problem.

If you're running under Windows CE 2.1 or later you can use the function:

```
HANDLE CeOpenDatabaseEx (PCEGUID pguid,
                         PCEOID poid, LPWSTR lpszName, CEPROPID propid,
                         DWORD dwFlags, CENOTIFYREQUEST *pRequest);
```

With a couple of exceptions, the parameters for *CeOpenDatabaseEx* are the same as for *CeOpenDatabase*. The first difference between the two functions is the extra pointer to a guid that identifies the volume in which the database resides.

The other difference is the method Windows CE uses to notify you of a change to the database. Instead of passing a handle to a window that will receive one of three WM_USER based messages, you pass a pointer to a CENOTIFYREQUEST structure that you have previously filled in. This structure is defined as

```
typedef struct _CENOTIFYREQUEST {
    DWORD dwSize;
    HWND hWnd;
    DWORD dwFlags;
    HANDLE hHeap;
    DWORD dwParam;
} CENOTIFYREQUEST;
```

The first field must be initialized to the size of the structure. The *hWnd* field should be set to the window that will receive the change notifications. The *dwFlags* field specifies how you want to be notified. If you put 0 in this field, you'll receive the same DB_CEIOD_*xxx* messages that are sent if you'd opened the database with *CeOpenDatabase*. If you put CEDB_EXNOTIFICATION in the *dwFlags* field, your window will receive an entirely new and more detailed notification method.

Instead of receiving the three DB_CEIOD_ messages, your window receives a WM_ DBNOTIFICATION message. When your window receives this message, the *lParam* parameter points to a CENOTIFICATION structure defined as

```
typedef struct _CENOTIFICATION {
    DWORD dwSize
    DWORD dwParam;
    UINT uType;
    CEGUID guid;
    CEOID oid;
    CEOID oidParent;
} CENOTIFICATION;
```

As expected, the *dwSize* field fills with the size of the structure. The *dwParam* field contains the value passed in the *dwParam* field in the CENOTIFYREQUEST structure. This is an application-defined value that can be used for any purpose.

The *uType* field indicates why the WM_DBNOTIFICATION message was sent. It will be set to one of the following values:

- *DB_CEOID_CREATED* A new file system object was created.
- *DB_CEOID_DATABASE_DELETED* The database was deleted from a volume.
- *DB_CEOID_RECORD_DELETED* A record was deleted in a database.
- *DB_CEOID_CHANGED* An object was modified.

The *guid* field contains the guid for the database volume that the message relates to while the *oid* field contains the relevant database record oid. Finally, the *oidParent* field contains the oid of the parent of the oid that the message references.

When you receive a WM_DBNOTIFICATION message, the CENOTIFICATION structure is placed in a memory block that you must free. If you specified a handle to a heap in the *hHeap* field of CENOTIFYREQUEST, the notification structure will be placed in that heap; otherwise, the system defined where the structure is placed. Regardless of its location, you are responsible for freeing the memory that contains the CENOTIFICATION structure. You do this with a call to

```
BOOL CeFreeNotification(PCENOTIFYREQUEST pRequest,
                        PCENOTIFICATION pNotify);
```

The function's two parameters are a pointer to the original CENOTIFYREQUEST structure and a pointer to the CENOTIFICATION structure to free. You must free the CENOTIFICATION structure each time you receive a WM_DBNOTIFICATION message.

Seeking (or searching for) a record

Now that the database is opened, you can read and write the records. But before you can read or write a record, you must *seek* to that record. That is, you must move the database pointer to the record you want to read or write. You accomplish this using

```
CEOID CeSeekDatabase (HANDLE hDatabase, DWORD dwSeekType, DWORD dwValue,
                      LPDWORD lpdwIndex);
```

The first parameter for this function is the handle to the opened database. The *dwSeekType* parameter describes how the seek is to be accomplished. The parameter can have one of the following values:

- *CEDB_SEEK_CEOID* Seek a specific record identified by its object ID. The object ID is specified in the *dwValue* parameter. This type of seek is particularly efficient in Windows CE databases.
- *CEDB_SEEK_BEGINNING* Seek the n^{th} record in the database. The index is contained in the *dwValue* parameter.
- *CEDB_SEEK_CURRENT* Seek from the current position *n* records forward or backward in the database. The offset is contained in the *dwValue* parameter. Even though *dwValue* is typed as a unsigned value, for this seek it's interpreted as a signed value.
- *CEDB_SEEK_END* Seek backward from the end of the database *n* records. The number of records to seek backward from the end is specified in the *dwValue* parameter.
- *CEDB_SEEK_VALUESMALLER* Seek from the current location until a record is found that contains a property that is the closest to, but not equal to or over the value specified. The value is specified by a CEPROPVAL structure pointed to by *dwValue*.
- *CEDB_SEEK_VALUEFIRSTEQUAL* Starting with the current location, seek until a record is found that contains the property that's equal to the value specified. The value is specified by a CEPROPVAL structure pointed to by *dwValue*. The location returned can be the current record.
- *CEDB_SEEK_VALUENEXTEQUAL* Starting with the next location, seek until a record is found that contains a property that's equal to the value specified. The value is specified by a CEPROPVAL structure pointed to by *dwValue*.
- *CEDB_SEEK_VALUEGREATER* Seek from the current location until a record is found that contains a property that is equal to, or the closest to, the value specified. The value is specified by a CEPROPVAL structure pointed to by *dwValue*.

As you can see from the available flags, seeking in the database is more than just moving a pointer; it also allows you to search the database for a particular record.

As I just mentioned in the descriptions of the seek flags, the *dwValue* parameter can either be loaded with an offset value for the seeks or point to a property value for the searches. The property value is described in a CEPROPVAL structure defined as

```
typedef struct _CEPROPVAL {
    CEPROPID propid;
```

(continued)

```
    WORD wLenData;
    WORD wFlags;
    CEVALUNION val;
} CEPROPVAL;
```

The *propid* field should contain one of the property ID values you defined for the properties in your database. Remember that the property ID is a combination of a data type identifier along with an application specific ID value that uniquely identifies a property in the database. This field identifies the property to examine when seeking. The *wLenData* field is ignored. None of the defined flags for the *wFlags* field is used by *CeSeekDatabase*, so this field should be set to 0. The *val* field is actually a union of the different data types supported in the database.

Following is a short code fragment that demonstrates seeking to the third record in the database.

```
DWORD dwIndex;
CEOID oid;

// Seek to the third record.
oid = CeSeekDatabase (g_hDB, CEDB_SEEK_BEGINNING, 3, &dwIndex);
if (oid == 0) {
    // There is no third item in the database.
}
```

Now say we want to find the first record in the database that has a height property of greater than 100. For this example, assume the size property type is a signed long value.

```
// Define pid for height property as a signed long with ID of one.
#define PID_HEIGHT      MAKELONG (CEVT_I4, 1)

CEOID oid;
DWORD dwIndex;
CEPROPVAL Property;

// First seek to the start of the database.
oid = CeSeekDatabase (g_hDB, CEDB_SEEK_BEGINNING, 0, &dwIndex);

// Seek the record with height > 100.
Property.propid = PID_HEIGHT;           // Set property to search.
Property.wLenData = 0;                  // Not used but clear anyway.
Property.wFlags = 0;                    // No flags to set
Property.val.lVal = 100;                // Data for property

oid = CeSeekDatabase (g_hDB, CEDB_SEEK_VALUEGREATER, &Property,
                      &dwIndex);
```

```
if (oid == 0) {
    // No matching property found, db pointer now points to end of db.
} else {
    // oid contains the object ID for the record,
    // dwIndex contains the offset from the start of the database
    // of the matching record.
}
```

Because the search for the property starts at the current location of the database pointer, you first need to seek to the start of the database if you want to find the first record in the database that has the matching property.

Changing the sort order

I talked earlier about how *CeDatabaseSeek* depends on the sort order of the opened database. If you want to choose one of the predefined sort orders instead, you must close the database and then reopen it specifying the predefined sort order. But what if you need a sort order that isn't one of the four sort orders that were defined when the database was created? You can redefine the sort orders using this function:

```
BOOL CeSetDatabaseInfo (CEOID oidDbase, CEDBASEINFO *pNewInfo);
```

or, under Windows CE 2.1 or later, this function:

```
BOOL CeSetDatabaseInfoEx (PCEGUID pguid,
                          CEOID oidDbase, CEDBASEINFO *pNewInfo);
```

Both these functions take the object ID of the database you want to redefine and a pointer to a CEDBASEINFO structure. This structure is the same one used by *CeCreateDatabaseEx*. You can use these functions to rename the database, change its type, or redefine the four sort orders. You shouldn't redefine the sort orders casually. When the database sort orders are redefined, the system has to iterate through every record in the database to rebuild the sort indexes. This can take minutes for large databases. If you must redefine the sort order of a database, you should inform the user of the massive amount of time it might take to perform the operation.

Reading a record

Once you have the database pointer at the record you're interested in, you can read or write that record. You can read a record in a database by calling the following function:

```
CEOID CeReadRecordProps (HANDLE hDbase, DWORD dwFlags, LPWORD lpcPropID,
                         CEPROPID *rgPropID, LPBYTE *lplpBuffer,
                         LPDWORD lpcbBuffer);
```

or, if you're running under Windows CE 2.1 or later, by calling the function you see at the top of the next page.

```
CEOID CeReadRecordPropsEx (HANDLE hDbase, DWORD dwFlags,
                           LPWORD lpcPropID,
                           CEPROPID *rgPropID, LPBYTE *lplpBuffer,
                           LPDWORD lpcbBuffer,
                           HANDLE hHeap);
```

The differences between these two functions is the addition of the *hHeap* parameter in *CeReadRecordPropsEx*. I'll explain the significance of this parameter shortly.

The first parameter in these functions is the handle to the opened database. The *lpcPropID* parameter points to a variable that contains the number of CEPROPID structures pointed to by the next parameter *rgPropID*. These two parameters combine to tell the function which properties of the record you want to read. There are two ways to utilize the *lpcPropID* and *rgPropID* parameters. If you want only to read a selected few of the properties of a record, you can initialize the array of CEPROPID structures with the ID values of the properties you want and set the variable pointed to by *lpcPropID* with the number of these structures. When you call the function, the returned data will be inserted into the CEPROPID structures for data types such as integers. For strings and blobs, where the length of the data is variable, the data is returned in the buffer indirectly pointed to by *lplpBuffer*.

Since *CeReadRecordProps* and *CeReadRecordPropsEx* have a significant overhead to read a record, it is always best to read all the properties necessary for a record in one call. To do this, simply set *rgPropID* to NULL. When the function returns, the variable pointed to by *lpcPropID* will contain the count of properties returned and the function will return all the properties for that record in the buffer. The buffer will contain an array of CEPROPID structures created by the function immediately followed by the data for those properties such as blobs and strings where the data isn't stored directly in the CEPROPID array.

One very handy feature of *CeReadRecordProps* and *CeReadRecordPropsEx* is that if you set CEDB_ALLOWREALLOC in the *dwFlags* parameter, the function will enlarge, if necessary, the results buffer to fit the data being returned. Of course, for this to work, the buffer being passed to the function must not be on the stack or in the static data area. Instead, it must be an allocated buffer, in the local heap for *CeReadRecordProps* or in the case of *CeReadRecordPropsEx*, in the local heap or a separate heap. In fact, if you use the CEDB_ALLOWREALLOC flag, you don't even need to pass a buffer to the function, instead you can set the buffer pointer to 0. In this case, the function will allocate the buffer for you.

Notice that the buffer parameter isn't a pointer to a buffer but a pointer to a pointer to a buffer. There actually is a method to this pointer madness. Since the resulting buffer can be reallocated by the function, it might be moved if the buffer needs to be reallocated. So the pointer to the buffer must be modified by the function. You

must always use the pointer the buffer returned by the function because it might have changed. Also, you're responsible for freeing the buffer after you have used it. Even if the function failed for some reason, the buffer might have moved or even have been freed by the function. You must clean up after the read by freeing the buffer if the pointer returned isn't 0.

Now to the difference between *CeReadRecordProps* and *CeReadRecordPropsEx*. As you might have guessed by the above discussion, the extra *hHeap* parameter allows *CeReadRecordPropsEx* to use a heap different from the local heap when reallocating the buffer. When you use *CeReadRecordPropsEx* and you want to use the local heap, simply pass a 0 in the *hHeap* parameter.

The routine below reads all the properties for a record, then copies the data into a structure.

```
int ReadDBRecord (HANDLE hDB, DATASTRUCT *pData) {
    WORD wProps;
    CEOID oid;
    PCEPROPVAL pRecord;
    PBYTE pBuff;
    DWORD dwRecSize;
    int i;

    // Read all properties for the record.
    pBuff = 0;   // Let the function allocate the buffer.
    oid = CeReadRecordProps (hDB, CEDB_ALLOWREALLOC, &wProps, NULL,
                             &(LPBYTE)pBuff, &dwRecSize);
    // Failure on read.
    if (oid == 0)
        return 0;

    // Copy the data from the record to the structure.  The order
    // of the array is not defined.
    memset (pData, 0 , sizeof (DATASTRUCT));  // Zero return struct
    pRecord = (PCEPROPVAL)pBuff;              // Point to CEPROPVAL
                                              // array.
    for (i = 0; i < wProps; i++) {
        switch (pRecord->propid) {
        case PID_NAME:
            lstrcpy (pData->szName, pRecord->val.lpwstr);
            break;
        case PID_TYPE:
            lstrcpy (pData->szType, pRecord->val.lpwstr);
            break;
```

(continued)

```
            case PID_SIZE:
                pData->nSize = pRecord->val.iVal;
                break;
            }
            pRecord++;
        }
        LocalFree (pBuff);
        return i;
}
```

Since the function above reads all the properties for the record, *CeReadRecordProps* creates the array of CEPROPVAL structures. The order of these structures isn't defined so the function cycles through each one to look for the data to fill in the structure. After all the data has been read, a call to *LocalFree* is made to free the buffer that was returned by *CeReadRecordProps*.

There is no requirement for every record to contain all the same properties. You might encounter a situation where you request a specific property from a record by defining the CEPROPID array and that property doesn't exist in the record. When this happens, *CeReadRecordProps* will set the CEDB_PROPNOTFOUND flag in the *wFlags* field of the CEPROPID structure for that property. You should always check for this flag if you call *CeReadRecordProps* and you specify the properties to be read. In the example above, all properties were requested, so if a property didn't exist, no CEPROPID structure for that property would have been returned.

Writing a record

You can write a record to the database using this function:

```
CEOID CeWriteRecordProps (HANDLE hDbase, CEOID oidRecord, WORD cPropID,
                          CEPROPVAL * rgPropVal);
```

The first parameter is the obligatory handle to the opened database. The *oidRecord* parameter is the object ID of the record to be written. To create a new record instead of modifying a record in the database, set *oidRecord* to 0. The *cPropID* parameter should contain the number of items in the array of property ID structures pointed to by *rgPropVal*. The *rcPropVal* array specifies which of the properties in the record to modify and the data to write.

Deleting properties, records, and entire databases

You can delete individual properties in a record using *CeWriteRecordProps*. To do this, create a CEPROPVAL structure that identifies the property to delete and set CEDB_PROPDELETE in the *wFlags* field.

To delete an entire record in a database, call

```
BOOL CeDeleteRecord (HANDLE hDatabase, CEOID oidRecord);
```

The parameters are the handle to the database and the object ID of the record to delete.

You can delete an entire database using this function:

```
BOOL CeDeleteDatabase (CEOID oidDbase);
```

or, under Windows CE 2.1 or later, this function:

```
BOOL CeDeleteDatabaseEx (PCEGUID pguid, CEOID oid);
```

The database being deleted can't be currently open. The difference between the two functions is that *CeDeleteDatabaseEx* can delete databases outside the object store.

Enumerating databases

Sometimes you must search the system to determine what databases are on the system. Windows CE provides two sets of functions to enumerate the databases in a volume. The first set of these functions works only for databases directly within the object store. These functions are

```
HANDLE CeFindFirstDatabase (DWORD dwDbaseType);
```

and

```
CEOID CeFindNextDatabase (HANDLE hEnum);
```

These functions act like *FindFirstFile* and *FindNextFile* with the exception that *CeFindFirstDatabase* only opens the search, it doesn't return the first database found. With these functions the only way to limit the search is to specify the ID of a specific database type in the *dwDbaseType* parameter. If this parameter is set to 0, all databases are enumerated. *CeFindFirstDatabase* returns a handle that is then passed to *CeFindNextDatabase* to actually enumerate the databases.

Below is an example of how to enumerate the databases in the object store.

```
HANDLE hDBList;
CEOID oidDB;

SendDlgItemMessage (hWnd, IDC_RPTLIST, WM_SETREDRAW, FALSE, 0);

hDBList = CeFindFirstDatabase (0);
if (hDBList != INVALID_HANDLE_VALUE) {

    oidDB = CeFindNextDatabase (hDBList);
    while (oidDB) {
        // Enumerated database identified by object ID.
        MyDisplayDatabaseInfo (hCeDB);

        hCeDB = CeFindNextDatabase (hDBList);
    }
    CloseHandle (hDBList);
}
```

To enumerate databases within a separate database volume, use

```
HANDLE CeFindFirstDatabaseEx (PCEGUID pguid, DWORD dwClassID);
```

and

```
HANDLE CeFindFirstDatabaseEx (PCEGUID pguid, DWORD dwClassID);
```

For the most past, these two functions work identically to their non-*Ex* predecessors with the exception that they enumerate the different databases within a single database volume. The additional parameter in these functions is the *CEOID* of the mounted volume to search.

Querying object information

To query information about a database, use this function:

```
BOOL CeOidGetInfo (CEOID oid, CEOIDINFO *poidInfo);
```

or, if under Windows CE 2.1 or later, use this function:

```
BOOL CeOidGetInfoEx (PCEGUID pguid, CEOID oid, CEOIDINFO *oidInfo);
```

These functions return information about not just databases, but any object in the file system. This includes files and directories as well as databases and database records. The functions are passed the object ID of the item of interest and a pointer to an CEOIDINFO structure. Here is the definition of the CEIOIDINFO structure:

```
typedef struct _CEOIDINFO {
    WORD wObjType;
    WORD wPad;
    union {
        CEFILEINFO infFile;
        CEDIRINFO infDirectory;
        CEDBASEINFO infDatabase;
        CERECORDINFO infRecord;
    };
} CEOIDINFO;
```

This structure contains a word indicating the type of the item and a union of four different structures each detailing information on that type of object. The currently supported flags are: OBJTYPE_FILE, indicating that the object is a file, OBJTYPE_DIRECTORY for directory objects, OBJTYPE_DATABASE for database objects, and OBJTYPE_RECORD indicating that the object is a record inside a database. The structures in the union are specific to each object type.

The CEFILEINFO structure is defined as

```
typedef struct _CEFILEINFO {
    DWORD dwAttributes;
```

```
    CEOID oidParent;
    WCHAR szFileName[MAX_PATH];
    FILETIME ftLastChanged;
    DWORD dwLength;
} CEFILEINFO;
```

the CEDIRINFO structure is defined as

```
typedef struct _CEDIRINFO {
    DWORD dwAttributes;
    CEOID oidParent;
    WCHAR szDirName[MAX_PATH];
} CEDIRINFO;
```

and the CERECORDINFO structure is defined as

```
typedef struct _CERECORDINFO {
    CEOID oidParent;
} CERECORDINFO;
```

You have already seen the CEDBASEINFO structure used in *CeCreateDatabaseEx* and *CeSetDatabaseInfo*. As you can see from the above structures, *CeGetOidInfo* returns a wealth of information about each object. One of the more powerful bits of information is the object's parent oid, which will allow you to trace the chain of files and directories back to the root. These functions also allow you to convert an object ID into a name of a database, directory, or file.

The object ID method of tracking a file object should not be confused with the PID scheme used by the shell. Object IDs are maintained by the file system, and are independent of whatever shell is being used. This would be a minor point under other versions of Windows, but with the ability of Windows CE to be built as components and customized for different targets, it's important to know what parts of the operating system support which functions.

The AlbumDB Example Program

It's great to talk about the database functions; it's another experience to use them in an application. The example program that follows, AlbumDB, is a simple database that tracks record albums, the artist that recorded them, and the individual tracks on the albums. It has a simple interface because the goal of the program is to demonstrate the database functions, not the user interface. Figure 7-4 on the next page shows the AlbumDB window with a few albums entered in the database.

Figure 7-5 contains the code for the AlbumDB program. When the program is first launched, it attempts to open a database called AlbumDB. If one isn't found, a new one is created. This is accomplished in the *OpenCreateDB* function.

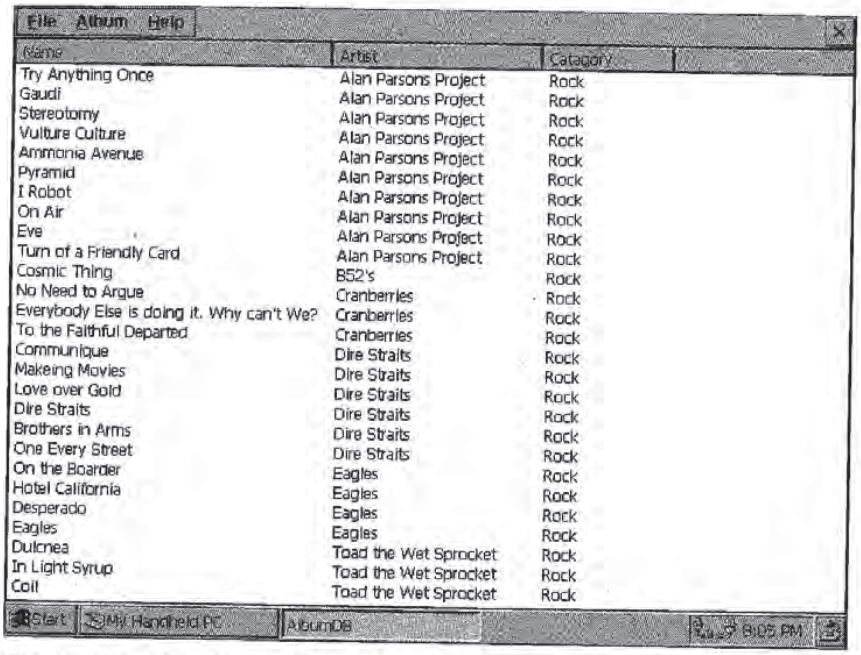

Figure 7-4. *The AlbumDB window.*

AlbumDB.rc

```
//=====================================================================
// Resource file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//=====================================================================

#include "windows.h"
#include "albumdb.h"                    // Program-specific stuff
//---------------------------------------------------------------------
// Icons and bitmaps
//
ID_ICON ICON    "albumdb.ico"           // Program icon

//---------------------------------------------------------------------
// Menu
//
ID_MENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
```

Figure 7-5. *The AlbumDB program.*

```
        MENUITEM "&Delete Database",            IDM_DELDB
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                       IDM_EXIT
    END
    POPUP "&Album"
    BEGIN
        MENUITEM "&New",                        IDM_NEW
        MENUITEM "&Edit",                       IDM_EDIT
        MENUITEM "&Delete",                     IDM_DELETE
        MENUITEM SEPARATOR
        MENUITEM "&Sort Name",                  IDM_SORTNAME
        MENUITEM "Sort &Artist",                IDM_SORTARTIST
        MENUITEM "Sort &Category",              IDM_SORTCATEGORY
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...",                   IDM_ABOUT
    END
END
//------------------------------------------------------------------------
// New/Edit Track dialog template
//
EditTrackDlg DIALOG discardable 10, 10, 165, 40
STYLE  WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU | DS_CENTER |
       DS_MODALFRAME
EXSTYLE WS_EX_CAPTIONOKBTN
CAPTION "Edit Track"
BEGIN
    LTEXT "Track Name"              -1,   5,   5,  50,  12
    EDITTEXT                  IDD_TRACK,  60,   5, 100,  12, WS_TABSTOP

    LTEXT "Time"                    -1,   5,  20,  50,  12
    EDITTEXT                  IDD_TIME,  60,  20,  50,  12, WS_TABSTOP
END
//------------------------------------------------------------------------
// New/Edit Album data dialog template
//
EditAlbumDlg DIALOG discardable 10, 10, 200, 100
STYLE  WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU | DS_CENTER |
       DS_MODALFRAME
EXSTYLE WS_EX_CAPTIONOKBTN
CAPTION "Edit Album"
BEGIN
    LTEXT "Album Name"              -1,   5,   5,  50,  12
    EDITTEXT                  IDD_NAME,  60,   5, 135,  12, WS_TABSTOP
```

(continued)

Figure 7-5. *continued*

```
    LTEXT "Artist"                  -1,   5,  20,  50,  12
    EDITTEXT                    IDD_ARTIST,  60,  20, 135,  12,
                                    WS_TABSTOP

    LTEXT "Category"                -1,   5,  35,  50,  12
    COMBOBOX                    IDD_CATEGORY,  60,  35, 135,  60,
                                    WS_TABSTOP | CBS_DROPDOWN
    LISTBOX                     IDD_TRACKS,  60,  50, 135,  45,
                                    LBS_USETABSTOPS

    PUSHBUTTON "&New Track...",
                                IDD_NEWTRACK,   5,  50,  50,  12,
                                    WS_TABSTOP
    PUSHBUTTON "&Edit Track...",
                                IDD_EDITTRACK,   5,  65,  50,  12,
                                    WS_TABSTOP
    PUSHBUTTON "&Del Track",
                                IDD_DELTRACK,   5,  80,  50,  12,
                                    WS_TABSTOP
END
//----------------------------------------------------------------------
// About box dialog template
//
aboutbox DIALOG discardable 10, 10, 160, 40
STYLE  WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU | DS_CENTER |
       DS_MODALFRAME
CAPTION "About"
BEGIN
    ICON  ID_ICON,                  -1,   5,   5,  10,  10
    LTEXT "AlbumDB - Written for the book Programming Windows \
           CE Copyright 1998 Douglas Boling"
                                    -1,  40,   5, 110,  30
END
```

AlbumDB.h

```
//=====================================================================
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//=====================================================================
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))
```

```
//---------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                             // Structure associates
    UINT Code;                                  // messages
                                                // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                              // Structure associates
    UINT Code;                                  // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);     // function.
};

//---------------------------------------------------------------------
// Generic defines used by application
#define  ID_ICON            1                   // App icon resource ID
#define  IDC_CMDBAR         2                   // Command band ID
#define  ID_MENU            3                   // Main menu resource ID
#define  ID_LISTV           5                   // List view control ID

// Menu item IDs
#define  IDM_DELDB          101                 // File menu
#define  IDM_EXIT           102

#define  IDM_NEW            110                 // Album menu
#define  IDM_EDIT           111
#define  IDM_DELETE         112

#define  IDM_SORTNAME       120                 // Sort IDs must be
#define  IDM_SORTARTIST     121                 // consecutive.
#define  IDM_SORTCATEGORY   122

#define  IDM_ABOUT          150                 // Help menu

// IDs for dialog box controls
#define  IDD_NAME           100                 // Edit album dialog.
#define  IDD_ARTIST         101
#define  IDD_NUMTRACKS      102
#define  IDD_CATEGORY       103
#define  IDD_TRACKS         104
#define  IDD_NEWTRACK       105
#define  IDD_EDITTRACK      106
#define  IDD_DELTRACK       107

#define  IDD_TRACK          200                 // Edit track dialog.
#define  IDD_TIME           201
```

(continued)

Figure 7-5. *continued*

```
//------------------------------------------------------------------
// Program-specific structures
//
// Structure used by New/Edit Album dlg proc
#define MAX_NAMELEN          64
#define MAX_ARTISTLEN        64
#define MAX_TRACKNAMELEN     512
typedef struct {
    TCHAR szName[MAX_NAMELEN];
    TCHAR szArtist[MAX_ARTISTLEN];
    INT nDateRel;
    SHORT sCategory;
    SHORT sNumTracks;
    INT nTrackDataLen;
    TCHAR szTracks[MAX_TRACKNAMELEN];
} ALBUMINFO, *LPALBUMINFO;

// Structure used by Add/Edit album track
typedef struct {
    TCHAR szTrack[64];
    TCHAR szTime[16];
} TRACKINFO, *LPTRACKINFO;

// Structure used by GetItemData
typedef struct {
    int nItem;
    ALBUMINFO Album;
} LVCACHEDATA, *PLVCACHEDATA;

// Database property identifiers
#define PID_NAME        MAKELONG (CEVT_LPWSTR, 1)
#define PID_ARTIST      MAKELONG (CEVT_LPWSTR, 2)
#define PID_RELDATE     MAKELONG (CEVT_I2, 3)
#define PID_CATEGORY    MAKELONG (CEVT_I2, 4)
#define PID_NUMTRACKS   MAKELONG (CEVT_I2, 5)
#define PID_TRACKS      MAKELONG (CEVT_BLOB, 6)
#define NUM_DB_PROPS    6

//------------------------------------------------------------------
// Function prototypes
//
int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

HANDLE OpenCreateDB (HWND, int *);
void ReopenDatabase (HWND, INT);
```

```
int GetItemData (int, PLVCACHEDATA);
HWND CreateLV (HWND, RECT *);
void ClearCache (void);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoSizeMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoCommandMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoNotifyMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDbNotifyMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);

// Command functions
LPARAM DoMainCommandDelDB (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandExit (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandNew (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandEdit (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandDelete (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandSort (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandAbout (HWND, WORD, HWND, WORD);

// Dialog procedures
BOOL CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK EditAlbumDlgProc (HWND, UINT, WPARAM, LPARAM);
```

AlbumDB.c

```
//=======================================================================
// AlbumDB - A Windows CE database
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//=======================================================================
#include <windows.h>                    // For all that Windows stuff
#include <windowsx.h>                   // For Window Controls macros
#include <commctrl.h>                   // Command bar includes

#include "AlbumDB.h"                    // Program-specific stuff

//-----------------------------------------------------------------------
// Global data
//
```

(continued)

Figure 7-5. *continued*

```
const TCHAR szAppName[] = TEXT ("AlbumDB");
HINSTANCE hInst;                    // Program instance handle
HANDLE g_hDB = 0;                   // Handle to album database
CEOID g_oidDB = 0;                  // Handle to album database
INT g_nLastSort = PID_NAME;         // Last sort order used

// These two variables represent a one item cache for
// the list view control.
int g_nLastItem = -1;
LPBYTE g_pLastRecord = 0;

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_SIZE, DoSizeMain,
    WM_COMMAND, DoCommandMain,
    WM_NOTIFY, DoNotifyMain,
    WM_DESTROY, DoDestroyMain,
    DB_CEOID_CHANGED, DoDbNotifyMain,
    DB_CEOID_CREATED, DoDbNotifyMain,
    DB_CEOID_RECORD_DELETED, DoDbNotifyMain,
};

// Command message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDM_DELDB, DoMainCommandDelDB,
    IDM_EXIT, DoMainCommandExit,
    IDM_NEW, DoMainCommandNew,
    IDM_EDIT, DoMainCommandEdit,
    IDM_DELETE, DoMainCommandDelete,
    IDM_SORTNAME, DoMainCommandSort,
    IDM_SORTARTIST, DoMainCommandSort,
    IDM_SORTCATEGORY, DoMainCommandSort,
    IDM_ABOUT, DoMainCommandAbout,
};
// Album category strings; must be alphabetical.
const TCHAR *pszCategories[] = {TEXT ("Classical"), TEXT ("Country"),
                                TEXT ("New Age"), TEXT ("Rock")};
//------------------------------------------------------------------------
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    HWND hwndMain;
    MSG msg;
    int rc = 0;
```

```
    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//------------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;
    INITCOMMONCONTROLSEX icex;

    // Register application main window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = MainWndProc;                   // Callback function
    wc.cbClsExtra = 0;                              // Extra class data
    wc.cbWndExtra = 0;                              // Extra window data
    wc.hInstance = hInstance;                       // Owner handle
    wc.hIcon = NULL;                                // Application icon
    wc.hCursor = NULL;                              // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                        // Menu name
    wc.lpszClassName = szAppName;                   // Window class name

    if (RegisterClass (&wc) == 0) return 1;

    // Load the command bar common control class.
    icex.dwSize = sizeof (INITCOMMONCONTROLSEX);
    icex.dwICC = ICC_BAR_CLASSES | ICC_TREEVIEW_CLASSES |
                 ICC_LISTVIEW_CLASSES;
    InitCommonControlsEx (&icex);
    return 0;
}
```

(continued)

Figure 7-5. *continued*

```
//-----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow){
    HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.
    hWnd = CreateWindow (szAppName, TEXT ("AlbumDB"), WS_VISIBLE,
                         CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, NULL, NULL, hInstance, NULL);

    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//-----------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {
    // Close the opened database.
    if (g_hDB)
        CloseHandle (g_hDB);
    // Free the last db query if saved.
    ClearCache ();

    return nDefRC;
}
//-----------------------------------------------------------------------
// Message handling procedures for MainWindow
//-----------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
```

```
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//------------------------------------------------------------------------
// DoCreateMain - Process WM_CREATE message for window.
//
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB, hwndChild;
    INT  nHeight, nCnt;
    RECT rect;
    LPCREATESTRUCT lpcs;

    // Convert lParam into pointer to create structure.
    lpcs = (LPCREATESTRUCT) lParam;

    // Create a minimal command bar that only has a menu and an
    // exit button.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);
    // Insert the menu.
    CommandBar_InsertMenubar (hwndCB, hInst, ID_MENU, 0);
    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    nHeight = CommandBar_Height (hwndCB);

    // Open the album database.  If one doesn't exist, create it.
    g_hDB = OpenCreateDB (hWnd, &nCnt);
    if (g_hDB == INVALID_HANDLE_VALUE) {
        MessageBox (hWnd, TEXT ("Could not open database."), szAppName,
                    MB_OK);
        DestroyWindow (hWnd);
        return 0;
    }
    // Create the list view control in right pane.
    SetRect (&rect, 0, nHeight, lpcs->cx, lpcs->cy - nHeight);
    hwndChild = CreateLV (hWnd, &rect);

    // Destroy frame if window not created.
    if (!IsWindow (hwndChild)) {
        DestroyWindow (hWnd);
        return 0;
    }
```

(continued)

Figure 7-5. *continued*

```
    ListView_SetItemCount (hwndChild, nCnt);
    return 0;
}
//--------------------------------------------------------------------
// DoSizeMain - Process WM_SIZE message for window.
//
LRESULT DoSizeMain (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam){
    HWND hwndLV;
    RECT rect;

    hwndLV = GetDlgItem (hWnd, ID_LISTV);

    // Adjust the size of the client rect to take into account
    // the command bar height.
    GetClientRect (hWnd, &rect);
    rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    SetWindowPos (hwndLV, NULL, rect.left, rect.top,
                  (rect.right - rect.left), rect.bottom - rect.top,
                  SWP_NOZORDER);
    return 0;
}
//--------------------------------------------------------------------
// DoCommandMain - Process WM_COMMAND message for window.
//
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)
            return (*MainCommandItems[i].Fxn)(hWnd, idItem, hwndCtl,
                                              wNotifyCode);
    }
    return 0;
}
//--------------------------------------------------------------------
// DoNotifyMain - Process DB_CEOID_xxx messages for window.
```

```
//
LRESULT DoDbNotifyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    switch (wMsg) {
    case DB_CEOID_CHANGED:
        InvalidateRect (GetDlgItem (hWnd, ID_LISTV), NULL, TRUE);
        break;
    case DB_CEOID_CREATED:
        ReopenDatabase (hWnd, -1);
        break;
    case DB_CEOID_RECORD_DELETED:
        ReopenDatabase (hWnd, -1);
        break;
    }
    return 0;
}
//------------------------------------------------------------------------
// DoNotifyMain - Process WM_NOTIFY message for window.
//
LRESULT DoNotifyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    int idItem, i;
    LPNMHDR pnmh;
    LPNMLISTVIEW pnmlv;
    NMLVDISPINFO *pLVdi;
    LVCACHEDATA data;
    HWND hwndLV;

    // Parse the parameters.
    idItem = (int) wParam;
    pnmh = (LPNMHDR)lParam;
    hwndLV = pnmh->hwndFrom;

    if (idItem == ID_LISTV) {
        pnmlv = (LPNMLISTVIEW)lParam;

        switch (pnmh->code) {
        case LVN_GETDISPINFO:
            pLVdi = (NMLVDISPINFO *)lParam;

            // Get a pointer to the data either from the cache
            // or from the actual database.
            GetItemData (pLVdi->item.iItem, &data);

            if (pLVdi->item.mask & LVIF_IMAGE)
                pLVdi->item.iImage = 0;
```

(continued)

Figure 7-5. *continued*

```
            if (pLVdi->item.mask & LVIF_PARAM)
                pLVdi->item.lParam = 0;

            if (pLVdi->item.mask & LVIF_STATE)
                pLVdi->item.state = 0;

            if (pLVdi->item.mask & LVIF_TEXT) {
                switch (pLVdi->item.iSubItem) {
                case 0:
                    lstrcpy (pLVdi->item.pszText, data.Album.szName);
                    break;
                case 1:
                    lstrcpy (pLVdi->item.pszText, data.Album.szArtist);
                    break;
                case 2:
                    lstrcpy (pLVdi->item.pszText,
                            pszCategories[data.Album.sCategory]);
                    break;
                }
            }
            break;

        // Ignore cache hinting for db example.
        case LVN_COLUMNCLICK:
            i = ((NM_LISTVIEW *)lParam)->iSubItem + IDM_SORTNAME;
            PostMessage (hWnd, WM_COMMAND, MAKELPARAM (i, 0), 0);
            break;

        // Ignore cache hinting for db example.
        case NM_DBLCLK:
            PostMessage (hWnd, WM_COMMAND, MAKELPARAM (IDM_EDIT, 0), 0);
            break;

        // Ignore cache hinting for db example.
        case LVN_ODCACHEHINT:
            break;

        case LVN_ODFINDITEM:
            // We should do a reverse look up here to see if
            // an item exists for the text passed.
            return -1;
        }
    }
    return 0;
}
```

```
//------------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
//========================================================================
// Command handler routines
//------------------------------------------------------------------------
// DoMainCommandDelDB - Process Program Delete command.
//
LPARAM DoMainCommandDelDB (HWND hWnd, WORD idItem, HWND hwndCtl,
                           WORD wNotifyCode) {
    int i, rc;

    i = MessageBox (hWnd, TEXT ("Delete the entire database?"),
                    TEXT ("Delete"), MB_YESNO);
    if (i != IDYES)
        return 0;
    if (g_oidDB) {
        CloseHandle (g_hDB);
        rc = CeDeleteDatabase (g_oidDB);
        if (rc == 0) {
            TCHAR szDbg[128];
            rc = GetLastError();
            wsprintf (szDbg, TEXT ("Couldn\'t delete db. rc=%d"), rc);
            MessageBox (hWnd, szDbg, szAppName, MB_OK);
            g_hDB = CeOpenDatabase (&g_oidDB, NULL, g_nLastSort,
                                    0, hWnd);
            return 0;
        }
        g_hDB = 0;
        g_oidDB = 0;
    }
    ListView_SetItemCount (GetDlgItem (hWnd, ID_LISTV), 0);
    return 0;
}
//------------------------------------------------------------------------
// DoMainCommandExit - Process Program Exit command.
//
LPARAM DoMainCommandExit (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {
```

(continued)

Figure 7-5. *continued*

```
    SendMessage (hWnd, WM_CLOSE, 0, 0);
    return 0;
}
//--------------------------------------------------------------------
// DoMainCommandNew - Process Program New command.
//
LPARAM DoMainCommandNew (HWND hWnd, WORD idItem, HWND hwndCtl,
                         WORD wNotifyCode) {
    PCEPROPVAL pcepv;
    INT i, rc;
    CEOID oid;
    HWND hwndLV = GetDlgItem (hWnd, ID_LISTV);

    // Display the new/edit dialog.
    pcepv = 0;
    rc = DialogBoxParam (hInst, TEXT ("EditAlbumDlg"), hWnd,
                         EditAlbumDlgProc, (LPARAM)&pcepv);
    if (rc == 0)
        return 0;

    // Write the record.
    oid = CeWriteRecordProps(g_hDB, 0, NUM_DB_PROPS, pcepv);
    if (!oid) {
        TCHAR szText[64];
        rc = GetLastError ();
        wsprintf (szText, TEXT ("Write Rec fail, Error %d (%x)"),
                  rc, rc);
        MessageBox (hWnd, szText, TEXT ("Error"), MB_OK);
    }
    ClearCache ();                              // Clear the lv cache.

    i = ListView_GetItemCount (hwndLV) + 1;     // Increment list view
                                                // count.
    ListView_SetItemCount (hwndLV, i);
    InvalidateRect (hwndLV, NULL, TRUE);        // Force list view
                                                // redraw.
    return 0;
}
//--------------------------------------------------------------------
// DoMainCommandEdit - Process Program Edit command.
//
LPARAM DoMainCommandEdit (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {
    PCEPROPVAL pcepv = 0;
    INT nSel, rc;
    WORD wProps = 0;
```

```
DWORD dwRecSize, dwIndex;
CEOID oid;
HWND hwndLV = GetDlgItem (hWnd, ID_LISTV);

nSel = ListView_GetSelectionMark (hwndLV);
if (nSel == -1)
    return 0;

// Seek to the necessary record.
oid = CeSeekDatabase (g_hDB, CEDB_SEEK_BEGINNING, nSel, &dwIndex);
if (oid == 0) {
    TCHAR szTxt[64];
    INT rc = GetLastError();
    wsprintf (szTxt, TEXT ("Db item not found. rc = %d (%x)"),
              rc, rc);
    MessageBox (NULL, szTxt, TEXT ("err"), MB_OK);
    return 0;
}
// Read all properties for the record.  Have the system
// allocate the buffer containing the data.
oid = CeReadRecordProps (g_hDB, CEDB_ALLOWREALLOC, &wProps, NULL,
                         &(LPBYTE)pcepv, &dwRecSize);
if (oid == 0) {
    TCHAR szTxt[64];
    INT rc = GetLastError();
    wsprintf (szTxt, TEXT ("Db item not read. rc = %d (%x)"),
              rc, rc);
    MessageBox (NULL, szTxt, TEXT ("err"), MB_OK);
    return 0;
}
// Display the edit dialog.
rc = DialogBoxParam (hInst, TEXT ("EditAlbumDlg"), hWnd,
                     EditAlbumDlgProc, (LPARAM)&pcepv);
if (rc == 0)
    return 0;

// Write the record.
oid = CeWriteRecordProps(g_hDB, oid, NUM_DB_PROPS, pcepv);
if (!oid) {
    TCHAR szText[64];
    rc = GetLastError ();
    wsprintf (szText, TEXT ("Write Rec fail. Error %d (%x)"),
              rc, rc);
    MessageBox (hWnd, szText, TEXT ("Error"), MB_OK);
}
```

(continued)

Figure 7-5. *continued*

```
    LocalFree ((LPBYTE)pcepv);
    ClearCache ();                                  // Clear the lv cache.

    InvalidateRect (hwndLV, NULL, TRUE);            // Force list view
                                                    // redraw.
    return 0;
}
//-----------------------------------------------------------------------
// DoMainCommandDelete - Process Program Delete command.
//
LPARAM DoMainCommandDelete (HWND hWnd, WORD idItem, HWND hwndCtl,
                            WORD wNotifyCode) {
    HWND hwndLV;
    TCHAR szText[64];
    DWORD dwIndex;
    int i;
    CEOID oid;
    int nSel;

    hwndLV = GetDlgItem (hWnd, ID_LISTV);
    nSel = ListView_GetSelectionMark (hwndLV);
    if (nSel != -1) {

        wsprintf (szText, TEXT ("Delete this item?"));
        i = MessageBox (hWnd, szText, TEXT ("Delete"), MB_YESNO);
        if (i != IDYES)
            return 0;

        // Seek to the necessary record.
        oid = CeSeekDatabase (g_hDB, CEDB_SEEK_BEGINNING, nSel, &dwIndex);
        CeDeleteRecord (g_hDB, oid);

        // Reduce the list view count by one and force redraw.
        i = ListView_GetItemCount (hwndLV) - 1;
        ListView_SetItemCount (hwndLV, i);
        ClearCache ();                              // Clear the lv cache.
        InvalidateRect (hwndLV, NULL, TRUE);
    }
    return 0;
}
//-----------------------------------------------------------------------
// DoMainCommandSort - Process the Sort commands.
//
LPARAM DoMainCommandSort(HWND hWnd, WORD idItem, HWND hwndCtl,
                         WORD wNotifyCode) {
    int nSort;
```

```
    switch (idItem) {
    case IDM_SORTNAME:
        nSort = PID_NAME;
        break;
    case IDM_SORTARTIST:
        nSort = PID_ARTIST;
        break;
    case IDM_SORTCATEGORY:
        nSort = PID_CATEGORY;
        break;
    }
    if (nSort == g_nLastSort)
        return 0;

    ReopenDatabase (hWnd, nSort);      // Close and reopen the database.
    return 0;
}
//---------------------------------------------------------------------
// DoMainCommandAbout - Process the Help | About menu command.
//
LPARAM DoMainCommandAbout(HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {
    // Use DialogBox to create modal dialog.
    DialogBox (hInst, TEXT ("aboutbox"), hWnd, AboutDlgProc);
    return 0;
}
//---------------------------------------------------------------------
// CreateLV - Creates the list view control
//
HWND CreateLV (HWND hWnd, RECT *prect) {
    HWND hwndLV;
    LVCOLUMN lvc;

    // Create album list window.
    hwndLV = CreateWindowEx (0, WC_LISTVIEW, TEXT (""),
                        WS_VISIBLE | WS_CHILD | WS_VSCROLL |
                        LVS_OWNERDATA | WS_BORDER | LVS_REPORT,
                        prect->left, prect->top,
                        prect->right - prect->left,
                        prect->bottom - prect->top,
                        hWnd, (HMENU)ID_LISTV,
                        hInst, NULL);

    // Add columns.
    if (hwndLV) {
        lvc.mask = LVCF_TEXT | LVCF_WIDTH | LVCF_FMT | LVCF_SUBITEM;
```

(continued)

Figure 7-5. *continued*

```
        lvc.fmt = LVCFMT_LEFT;
        lvc.cx = 150;
        lvc.pszText = TEXT ("Name");
        lvc.iSubItem = 0;
        SendMessage (hwndLV, LVM_INSERTCOLUMN, 0, (LPARAM)&lvc);

        lvc.mask |= LVCF_SUBITEM;
        lvc.pszText = TEXT ("Artist");
        lvc.cx = 100;
        lvc.iSubItem = 1;
        SendMessage (hwndLV, LVM_INSERTCOLUMN, 1, (LPARAM)&lvc);

        lvc.mask |= LVCF_SUBITEM;
        lvc.pszText = TEXT ("Category");
        lvc.cx = 100;
        lvc.iSubItem = 2;
        SendMessage (hwndLV, LVM_INSERTCOLUMN, 2, (LPARAM)&lvc);
    }

    return hwndLV;
}
//------------------------------------------------------------------------
// OpenCreateDB - Open database, create if necessary.
//
HANDLE OpenCreateDB (HWND hWnd, int *pnRecords) {
    INT i, rc;
    CEOIDINFO oidinfo;
    SORTORDERSPEC sos[4];

    g_oidDB = 0;
    g_hDB = CeOpenDatabase (&g_oidDB, TEXT ("\\Albums"),
                            g_nLastSort, 0, hWnd);
    if (g_hDB == INVALID_HANDLE_VALUE) {
        rc = GetLastError();
        if (rc == ERROR_FILE_NOT_FOUND) {
            i = 0;
            sos[i].propid = PID_NAME;
            sos[i++].dwFlags = 0;

            sos[i].propid = PID_ARTIST;
            sos[i++].dwFlags = 0;

            sos[i].propid = PID_CATEGORY;
            sos[i++].dwFlags = 0;
```

```
            g_oidDB = CeCreateDatabase (TEXT ("\\Albums"), 0, 3,
                                        sos);
            if (g_oidDB == 0) {
                TCHAR szErr[128];
                wsprintf (szErr, TEXT ("Database create failed. \
                          rc %d"), GetLastError());
                MessageBox (hWnd, szErr, szAppName, MB_OK);
                return 0;
            }
            g_hDB = CeOpenDatabase(&g_oidDB,NULL, g_nLastSort, 0, hWnd);
        }
    }
    CeOidGetInfo (g_oidDB, &oidinfo);
    *pnRecords = oidinfo.infDatabase.wNumRecords;
    return g_hDB;
}
//--------------------------------------------------------------------
// ClearCache - Clears the one item cache for the list view control
//
void ClearCache (void) {

    if (g_pLastRecord)
        LocalFree (g_pLastRecord);
    g_pLastRecord = 0;
    g_nLastItem = -1;
    return;
}
//--------------------------------------------------------------------
// ReopenDatabase - Closes and reopens the database
//
void ReopenDatabase (HWND hWnd, INT nNewSort) {
    INT nCnt;

    if (nNewSort != -1)
        g_nLastSort = nNewSort;

    if (g_hDB)
        CloseHandle (g_hDB);
    ClearCache ();                          // Clear the lv cache.

    g_hDB = OpenCreateDB (hWnd, &nCnt);

    ListView_SetItemCount (GetDlgItem (hWnd, ID_LISTV), nCnt);
    InvalidateRect (GetDlgItem (hWnd, ID_LISTV), NULL, 0);
    return;
}
```

(continued)

Figure 7-5. *continued*

```
//-----------------------------------------------------------------------
// Get the album data from the database for the requested lv item.
//
int GetItemData (int nItem, PLVCACHEDATA pcd) {
    static WORD wProps;
    DWORD dwIndex;
    CEOID oid;
    PCEPROPVAL pRecord = NULL;
    DWORD dwRecSize;
    int i;

    // See if the item requested was the previous one.  If so,
    // just use the old data.
    if ((nItem == g_nLastItem) && (g_pLastRecord))
        pRecord = (PCEPROPVAL)g_pLastRecord;
    else {
        // Seek to the necessary record.
        oid = CeSeekDatabase (g_hDB, CEDB_SEEK_BEGINNING, nItem, &dwIndex);
        if (oid == 0) {
            TCHAR szTxt[64];
            INT rc = GetLastError();
            wsprintf (szTxt, TEXT ("Db item not found. rc = %d (%x)"),
                      rc, rc);
            MessageBox (NULL, szTxt, TEXT ("err"), MB_OK);
            return 0;
        }
        // Read all properties for the record.  Have the system
        // allocate the buffer containing the data.
        oid = CeReadRecordProps (g_hDB, CEDB_ALLOWREALLOC, &wProps, NULL,
                                 &(LPBYTE)pRecord, &dwRecSize);
        if (oid == 0) {
            TCHAR szTxt[64];
            INT rc = GetLastError();
            wsprintf (szTxt, TEXT ("Db item not read. rc = %d (%x)"),
                      rc, rc);
            MessageBox (NULL, szTxt, TEXT ("err"), MB_OK);
            return 0;
        }
        // Free old record and save the newly read one.
        if (g_pLastRecord)
            LocalFree (g_pLastRecord);
        g_nLastItem = nItem;
        g_pLastRecord = (LPBYTE)pRecord;

    }
```

```
    // Copy the data from the record to the album structure.
    for (i = 0; i < wProps; i++) {
        switch (pRecord->propid) {
        case PID_NAME:
            lstrcpy (pcd->Album.szName, pRecord->val.lpwstr);
            break;
        case PID_ARTIST:
            lstrcpy (pcd->Album.szArtist, pRecord->val.lpwstr);
            break;
        case PID_CATEGORY:
            pcd->Album.sCategory = pRecord->val.iVal;
            break;
        case PID_NUMTRACKS:
            pcd->Album.sNumTracks = pRecord->val.iVal;
            break;
        }
        pRecord++;
    }
    return 1;
}
//---------------------------------------------------------------------
// InsertLV - Add an item to the list view control.
//
INT InsertLV (HWND hWnd, INT nItem, LPTSTR pszName, LPTSTR pszType,
              INT nSize) {
    LVITEM lvi;
    HWND hwndLV = GetDlgItem (hWnd, ID_LISTV);

    lvi.mask = LVIF_TEXT | LVIF_IMAGE | LVIF_PARAM;
    lvi.iItem = nItem;
    lvi.iSubItem = 0;
    lvi.pszText = pszName;
    lvi.iImage = 0;
    lvi.lParam = nItem;
    SendMessage (hwndLV, LVM_INSERTITEM, 0, (LPARAM)&lvi);

    lvi.mask = LVIF_TEXT;
    lvi.iItem = nItem;
    lvi.iSubItem = 1;
    lvi.pszText = pszType;
    SendMessage (hwndLV, LVM_SETITEM, 0, (LPARAM)&lvi);

    return 0;
}
//---------------------------------------------------------------------
```

(continued)