Figure 3-12. *continued*

```
    // 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;

    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 ("TicTac2"),     // 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) {
    HWND hwndCB;
    HICON hIcon;

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);
    // Add the menu.
    CommandBar_InsertMenubar (hwndCB, hInst, ID_MENU, 0);
    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);

    hIcon = (HICON) SendMessage (hWnd, WM_GETICON, 0, 0);
    if (hIcon == 0) {
        hIcon = LoadImage (hInst, MAKEINTRESOURCE (ID_ICON),
                           IMAGE_ICON, 16, 16, 0);
```

(continued)

Figure 3-12. *continued*

```
        SendMessage (hWnd, WM_SETICON, FALSE, (LPARAM)hIcon);
    }

    // Initialize game.
    ResetGame ();
    return 0;
}
//----------------------------------------------------------------------
// DoSizeMain - Process WM_SIZE message for window.
//
LRESULT DoSizeMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                    LPARAM lParam) {
    RECT rect;

    // 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));

    // Define the playing board rect.
    rectBoard = rect;
    rectPrompt = rect;
    // Layout depends on portrait or landscape screen.
    if (rect.right - rect.left > rect.bottom - rect.top) {
        rectBoard.left += 20;
        rectBoard.top += 10;
        rectBoard.bottom -= 10;
        rectBoard.right = rectBoard.bottom - rectBoard.top + 10;

        rectPrompt.left = rectBoard.right + 10;

    } else {
        rectBoard.left += 20;
        rectBoard.right -= 20;
        rectBoard.top += 10;
        rectBoard.bottom = rectBoard.right - rectBoard.left + 10;

        rectPrompt.top = rectBoard.bottom + 10;
    }
    return 0;
}
//----------------------------------------------------------------------
// DoPaintMain - Process WM_PAINT message for window.
//
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    PAINTSTRUCT ps;
    RECT rect;
    HFONT hFont, hOldFont;
    TCHAR szPrompt[32];
    HDC hdc;

    // 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));

    hdc = BeginPaint (hWnd, &ps);

    // Draw the board.
    DrawBoard (hdc, &rectBoard);

    // Write the prompt to the screen.
    hFont = GetStockObject (SYSTEM_FONT);
    hOldFont = SelectObject (hdc, hFont);

    if (bTurn == 0)
        LoadString (hInst, IDS_XTURN, szPrompt, sizeof (szPrompt));
    else
        LoadString (hInst, IDS_OTURN, szPrompt, sizeof (szPrompt));

    DrawText (hdc, szPrompt, -1, &rectPrompt,
              DT_CENTER | DT_VCENTER | DT_SINGLELINE);

    SelectObject (hdc, hOldFont);
    EndPaint (hWnd, &ps);
    return 0;
}
//----------------------------------------------------------------------
// DoInitMenuPopMain - Process WM_INITMENUPOPUP message for window.
//
LRESULT DoInitMenuPopMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                           LPARAM lParam) {
    HMENU hMenu;

    hMenu = CommandBar_GetMenu (GetDlgItem (hWnd, IDC_CMDBAR), 0);

    if (bLastMove == -1)
        EnableMenuItem (hMenu, IDM_UNDO, MF_BYCOMMAND | MF_GRAYED);
    else
        EnableMenuItem (hMenu, IDM_UNDO,  MF_BYCOMMAND | MF_ENABLED);
    return 0;
}
```

(continued)

Figure 3-12. *continued*

```
//----------------------------------------------------------------------
// 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;
}
//----------------------------------------------------------------------
// DoLButtonUpMain - Process WM_LBUTTONUP message for window.
//
LRESULT DoLButtonUpMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                         LPARAM lParam) {
    POINT pt;
    INT cx, cy, nCell = 0;

    pt.x = LOWORD (lParam);
    pt.y = HIWORD (lParam);

    // See if pen on board.  If so, determine which cell.
    if (PtInRect (&rectBoard, pt)){
        // Normalize point to upper left corner of board.
        pt.x -= rectBoard.left;
        pt.y -= rectBoard.top;

        // Compute size of each cell.
        cx = (rectBoard.right - rectBoard.left)/3;
        cy = (rectBoard.bottom - rectBoard.top)/3;

        // Find column.
        nCell = (pt.x / cx);
        // Find row.
        nCell += (pt.y / cy) * 3;

        // If cell empty, fill it with mark.
        if (bBoard[nCell] == 0) {
            if (bTurn) {
                bBoard[nCell] = 2;
                bTurn = 0;
            } else {
                bBoard[nCell] = 1;
                bTurn = 1;
            }
            // Save the cell for the undo command.
            bLastMove = nCell;
            // Force the screen to be repainted.
            InvalidateRect (hWnd, NULL, FALSE);
        } else {
            // Inform the user of the filled cell.
            MessageBeep (0);
            return 0;
        }
    }
    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
//
//----------------------------------------------------------------------
// DoMainCommandNewGame - Process New Game command.
//
LPARAM DoMainCommandNewGame (HWND hWnd, WORD idItem, HWND hwndCtl,
                             WORD wNotifyCode) {
    INT i, j = 0, rc;

    // Count the number of used spaces.
    for (i = 0; i < 9; i++)
        if (bBoard[i])
            j++;
```

(continued)

Figure 3-12. *continued*

```
    // If not new game or complete game, ask user before clearing.
    if (j && (j != 9)) {
        rc = MessageBox (hWnd,
                         TEXT ("Are you sure you want to clear the board?"),
                         TEXT ("New Game"), MB_YESNO | MB_ICONQUESTION);
        if (rc == IDNO)
            return 0;
    }
    ResetGame ();
    InvalidateRect (hWnd, NULL, TRUE);
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandUndo - Process Undo Last Move command.
//
LPARAM DoMainCommandUndo (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {

    if (bLastMove != -1) {
        bBoard[bLastMove] = 0;
        if (bTurn) {
            bTurn = 0;
        } else {
            bTurn = 1;
        }
        // Only one level of undo
        bLastMove = -1;
        InvalidateRect (hWnd, NULL, TRUE);
    }
    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;
}
//======================================================================
// Game-specific routines
//
//----------------------------------------------------------------------
// ResetGame - Initialize the structures for a game.
//
```

```
void ResetGame (void) {
    INT i;

    // Initialize the board.
    for (i = 0; i < dim(bBoard); i++)
        bBoard[i] = 0;

    bTurn = 0;
    bLastMove = -1;
    return;
}
//----------------------------------------------------------------------
// DrawXO - Draw a single X or O in a square.
//
void DrawXO (HDC hdc, HPEN hPen, RECT *prect, INT nCell, INT nType) {
    POINT pt[2];
    INT cx, cy;
    RECT rect;

    cx = (prect->right - prect->left)/3;
    cy = (prect->bottom - prect->top)/3;

    // Compute the dimensions of the target cell.
    rect.left = (cx * (nCell % 3) + prect->left) + 10;
    rect.right = rect.left + cx - 20;
    rect.top = cy * (nCell / 3) + prect->top + 10;
    rect.bottom =  rect.top + cy - 20;

    // Draw an X?
    if (nType == 1) {
        pt[0].x = rect.left;
        pt[0].y = rect.top;
        pt[1].x = rect.right;
        pt[1].y = rect.bottom;
        Polyline (hdc, pt, 2);

        pt[0].x = rect.right;
        pt[1].x = rect.left;
        Polyline (hdc, pt, 2);
    // How about an O?
    } else if (nType == 2) {
        Ellipse (hdc, rect.left, rect.top, rect.right, rect.bottom);
    }
    return;
}
//----------------------------------------------------------------------
```

(continued)

Figure 3-12. *continued*

```
//  DrawBoard - Draw the tic-tac-toe board.
//
void DrawBoard (HDC hdc, RECT *prect) {
    HPEN hPen, hOldPen;
    POINT pt[2];
    LOGPEN lp;
    INT i, cx, cy;

    // Create a nice thick pen.
    lp.lopnStyle = PS_SOLID;
    lp.lopnWidth.x = 5;
    lp.lopnWidth.y = 5;
    lp.lopnColor = RGB (0, 0, 0);
    hPen = CreatePenIndirect (&lp);

    hOldPen = SelectObject (hdc, hPen);

    cx = (prect->right - prect->left)/3;
    cy = (prect->bottom - prect->top)/3;

    // Draw lines down.
    pt[0].x = cx + prect->left;
    pt[1].x = cx + prect->left;
    pt[0].y = prect->top;
    pt[1].y = prect->bottom;
    Polyline (hdc, pt, 2);

    pt[0].x += cx;
    pt[1].x += cx;
    Polyline (hdc, pt, 2);

    // Draw lines across.
    pt[0].x = prect->left;
    pt[1].x = prect->right;
    pt[0].y = cy + prect->top;
    pt[1].y = cy + prect->top;
    Polyline (hdc, pt, 2);

    pt[0].y += cy;
    pt[1].y += cy;
    Polyline (hdc, pt, 2);

    // Fill in Xs and Os.
    for (i = 0; i < dim (bBoard); i++)
        DrawXO (hdc, hPen, &rectBoard, i, bBoard[i]);
    SelectObject (hdc, hOldPen);
    DeleteObject (hPen);
    return;
}
```

The biggest change in TicTac2 is the addition of a WM_COMMAND handler in the form of the routine *OnCommandMain*. Because a program might end up handling a large number of different menu items and other controls, I extend the table-lookup design of the window procedure to another table lookup for command IDs from menus and accelerators. For TicTac2, I use three command handlers, one for each of the menu items. This results in another table of IDs and procedure pointers that associates menu IDs with handler procedures. Again, this way of using a table lookup instead of the standard switch statement isn't necessary or specific to Windows CE. It's simply my programming style.

The first menu handler, *OnCommandNewGame*, simply calls the reset game routine to clear the game structures. The routine itself returns 0, which is the default value for a WM_COMMAND handler.

The *OnCommandUndo* command handler is interesting in that it isn't always enabled. TicTac2 handles an additional message WM_INITMENUPOPUP, which is sent to a window immediately before the window menu is displayed. This gives the window a chance to initialize any of the menu items. In this case, the routine *OnInitMenuPopMain* looks to see whether the *bLastMove* field contains a valid cell value (0 through 8). If not, the routine disables the Undo menu item using *EnableMenuItem*. This action also disables the keyboard accelerator for that menu item as well.

The final command handler, *OnCommandExit*, sends a WM_CLOSE message to the main window. Closing the window eventually results in Windows sending a WM_DESTROY message, which results in a *PostQuitMessage* call that terminates the program. Sending a WM_CLOSE message is, by the way, the same action that results from clicking on the Close button on the command bar.

Other changes from the first TicTac example include modification of the message loop to provide for keyboard accelerators and the addition of code in the *OnCreateMain* routine to load and assign a window icon. Also, the string prompts for whose turn it is are loaded from the resource file.

Looking at the *OnCommandNewGame* handler introduces one last new function. If the game isn't complete, the program asks the players whether they really want to clear the game board. This query is accomplished by calling

```
int MessageBox (HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption,
                UINT uType);
```

This function displays a message box, a simple dialog box, with definable text and buttons. A message box can display a message along with a limited series of buttons. Message boxes are often used to query users for a simple response or to notify them of some event. The *uType* parameter allows the programmer to select different button configurations, such as Yes/No, OK/Cancel, Yes/No/Cancel, and simply OK. You can also select an icon to appear in the message box that signals the level of importance of the answer.

A message box is essentially a poor man's dialog box. It offers a simple method of querying the user but little flexibility in how the dialog box is configured. Now that we've introduced the subject of *dialog boxes*, it's time to take a closer look at them and other types of secondary and child windows.

Chapter 4

Windows, Controls, and Dialog Boxes

Understanding how windows work and relate to each other is the key to understanding the user interface of the Microsoft Windows operating system, whether it be Microsoft Windows 98, Microsoft Windows NT, or Microsoft Windows CE. Everything you see on a Windows display is a window. The desktop is a window, the taskbar is a window, even the Start button on the taskbar is a window. Windows are related to one another according to one relationship model or another; they may be in *parent/child, sibling,* or *owner/owned* relationships. Windows supports a number of predefined window classes, called *controls*. These controls simplify the work of programmers by providing a range of predefined user interface elements as simple as a button or as complex as a multiline text editor. Windows CE supports the same standard set of built-in controls as the other versions of Windows. These built-in controls shouldn't be confused with the complex controls provided by the common control library. I'll talk about those controls in Chapter 5.

Controls are usually contained in dialog boxes (sometimes simply referred to as *dialogs*). These dialog boxes constitute a method for a program to query users for information the program needs. A specialized form of dialog, named a *property sheet*, allows a program to display multiple but related dialog boxes in an overlapping style; each box or property sheet is equipped with an identifying tab. Property sheets are particularly valuable given the tiny screens associated with Windows CE devices.

Finally, Windows CE supports a subset of the common dialog library available under Windows NT and Windows 98. Specifically, Windows CE supports versions of the common dialog boxes File Open, File Save, Color, and Print. These dialogs are somewhat different on Windows CE. They're reformatted for the smaller screens and aren't as extensible as their desktop counterparts.

CHILD WINDOWS

Each window is connected via a parent/child relationship scheme. Applications create a main window with no parent, called a *top-level window*. That window might (or might not) contain windows, called *child* windows. A child window is clipped to its parent. That is, no part of a child window is visible beyond the edge of its parent. Child windows are automatically destroyed when their parent windows are destroyed. Also, when a parent window moves, its child windows move with it.

Child windows are programmatically identical to top-level windows. You use the *CreateWindow* or *CreateWindowEx* function to create them, each has a window procedure that handles the same messages as its top-level window, and each can, in turn, contain its own child windows. To create a child window, use the WS_CHILD window style in the *dwStyle* parameter of *CreateWindow* or *CreateWindowEx*. In addition, the *hMenu* parameter, unused in top-level Windows CE windows, passes an ID value that you can use to reference the window.

Under Windows CE, there's one other major difference between top-level windows and child windows. Windows sends WM_HIBERNATE messages only to top-level windows that have the WS_OVERLAPPED and WS_VISIBLE styles. (Window visibility in this case has nothing to do with what a user sees. A window can be "visible" to the system and still not be seen by the user if other windows are above it in the Z-order.) This means that child windows and most dialog boxes aren't sent WM_HIBERNATE messages. Top-level windows must either manually send a WM_HIBERNATE message to their child windows as necessary or perform all the necessary tasks themselves to reduce the application's memory footprint. On Windows CE systems, such as the H/PC that support application buttons on the taskbar, the rules for determining the target of WM_HIBERNATE messages are also used to determine what windows get buttons on the taskbar.

In addition to the parent/child relationship, windows also have an owner/owned relationship. Owned windows aren't clipped to their owners. However, they always appear "above" (in Z-order) the window that owns them. If the owner window is minimized, all windows it owns are hidden. Likewise, if a window is destroyed, all windows it owns are destroyed. Windows CE 1.0 supports window ownership only for dialog boxes, but from version 2.0 on, Windows CE provides full support for owned windows.

Window Management Functions

Given the windows-centric nature of Windows, it's not surprising that you can choose from a number of functions that enable a window to interrogate its environment so that it might determine its location in the window family tree. To find its parent, a window can call

```
HWND GetParent (HWND hWnd);
```

This function is passed a window handle and returns the handle of the calling window's parent window. If the window has no parent, the function returns NULL.

Enumerating windows

GetWindow, prototyped as

```
HWND GetWindow (HWND hWnd, UINT uCmd);
```

is an omnibus function that allows a window to query its children, owner, and siblings. The first parameter is the window's handle while the second is a constant that indicates the requested relationship. The GW_CHILD constant returns a handle to the first child window of a window. *GetWindow* returns windows in Z-order, so the first window in this case is the child window highest in the Z-order. If the window has no child windows, this function returns NULL. The two constants, GW_HWNDFIRST and GW_HWNDLAST, return the first and last windows in the Z-order. If the window handle passed is a top-level window, these constants return the first and last topmost windows in the Z-order. If the window passed is a child window, the GetWindow function returns the first and last sibling window. The GW_HWNDNEXT and GW_HWNDPREV constants return the next lower and next higher windows in the Z-order. These constants allow a window to iterate through all the sibling windows by getting the next window, then using that window handle with another call to *GetWindow* to get the next, and so on. Finally, the GW_OWNER constant returns the handle of the owner of a window.

Another way to iterate through a series of windows is

```
BOOL EnumWindows (WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

This function calls the callback function pointed to by *lpEnumFunc* once for each top-level window on the desktop, passing the the handle of each window in turn. The *lParam* value is an application-defined value, which is also passed to the enumeration function. This function is better than iterating through a *GetWindow* loop to find the top-level windows because it always returns valid window handles; it's possible that a *GetWindow* iteration loop will get a window handle whose window is destroyed before the next call to *GetWindow* can occur. However, since *EnumWindows* works only with top-level windows, *GetWindow* still has a place when iterating through a series of child windows.

Finding a window

To get the handle of a specific window, use the function

```
HWND FindWindow (LPCTSTR lpClassName, LPCTSTR lpWindowName);
```

This function can find a window either by means of its window class name or by means of a window's title text. This function is handy when an application is just starting up; it can determine whether another copy of the application is already running. All an application has to do is call *FindWindow* with the name of the window class for the main window of the application. Because an application almost always has a main window while it's running, a NULL returned by *FindWindow* indicates that the function can't locate another window with the specified window class—therefore, it's almost certain that another copy of the application isn't running.

Editing the window structure values

The pair of functions

```
LONG GetWindowLong (HWND hWnd, int nIndex);
```

and

```
LONG SetWindowLong (HWND hWnd, int nIndex, LONG dwNewLong);
```

allow an application to edit data in the window structure for a window. Remember the WNDCLASS structure passed to the *RegisterClass* function has a field, *cbWndExtra*, that controls the number of extra bytes that are to be allocated after the structure. If you allocated extra space in the window structure when the window class was registered, you can access those bytes using the *GetWindowLong* and *SetWindowLong* functions. Under Windows CE, the data must be allocated and referenced in 4-byte (integer sized and aligned) blocks. So, if a window class was registered with 12 in the *cbWndExtra* field, an application can access those bytes by calling *GetWindowLong* or *SetWindowLong* with the window handle and by setting values of 0, 4, and 8 in the *nIndex* parameter.

GetWindowLong and *SetWindowLong* support a set of predefined index values that allow an application access to some of the basic parameters of a window. Here is a list of the supported values for Windows CE.

- *GWL_STYLE* The style flags for the window
- *GWL_EXSTYLE* The extended style flags for the window
- *GWL_WNDPROC* The pointer to the window procedure for the window
- *GWL_ID* The ID value for the window
- *GWL_USERDATA* An application-usable 32-bit value

Dialog box windows support the following additional values:

- *DWL_DLGPROC* The pointer to the dialog procedure for the window
- *DWL_MSGRESULT* The value returned when the dialog box function returns
- *DWL_USER* An application-usable 32-bit value

Windows CE doesn't support the GWL_HINSTANCE and GWL_HWNDPARENT values supported by Windows NT and Windows 98.

Scroll Bars and the FontList2 Example Program

To demonstrate a handy use for a child window, we return to the FontList program from Chapter 2. As you might remember, the problem was that if a scroll bar were attached to the main window of the application, the scroll bar would extend upward, past the right side of the command bar. The reason for this is that a scroll bar attached to a window is actually placed in the nonclient area of that window. Because the command bar lies in the client space, we have no easy way to properly position the two controls in the same window.

An easy way to solve this problem is to use a child window. We place the child window so that it fills all of the client area of the top-level window not covered by the command bar. The scroll bar can then be attached to the child window so that it appears on the right side of the window but stops just beneath the command bar. Figure 4-1 shows the Fontlist2 window. Notice that the scroll bar now fits properly underneath the command bar. Also notice that the child window is completely undetectable by the user.

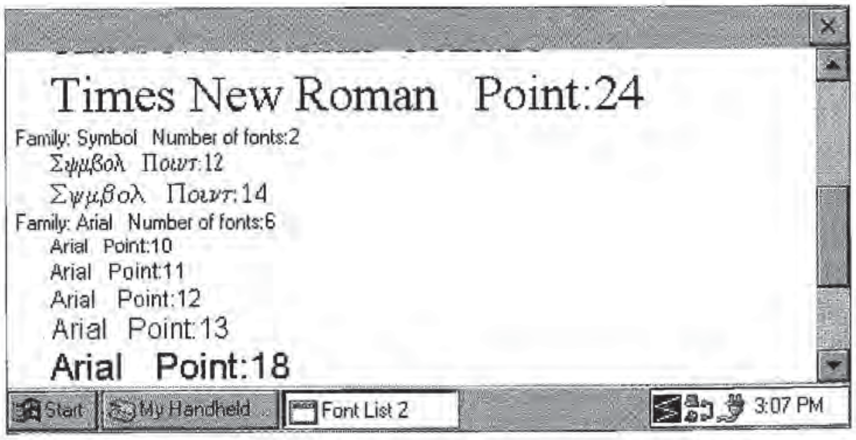

```
Times New Roman   Point:24
Family: Symbol  Number of fonts:2
    Σψμβολ  Ποιντ:12
    Σψμβολ  Ποιντ:14
Family: Arial  Number of fonts:6
    Arial  Point:10
    Arial  Point:11
    Arial  Point:12
    Arial  Point:13
    Arial  Point:18
```

Figure 4-1. *The FontList2 window with the scroll bar properly positioned just beneath the command bar.*

The code for this fix, which isn't that much more complex than the original FontList example, is shown in Figure 4-2. Instead of one window procedure, there are now two, one for the top-level window, which I have labeled the Frame window, and one for the child window. I separated the code for these two windows into two different source files, FontList2.c and ClientWnd.c. ClientWnd.c also contains a function, *InitClient*, which registers the client window class.

FontList2.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_CLIENT 2                          // Client window ID

//----------------------------------------------------------------------
// Window prototypes and defines
//
#define FAMILYMAX   24
typedef struct {
    int nNumFonts;
    TCHAR szFontFamily[LF_FACESIZE];
} FONTFAMSTRUCT;
```

Figure 4-2. *The FontList2 program.*

Page 00177

```
typedef FONTFAMSTRUCT *PFONTFAMSTRUCT;

typedef struct {
    INT yCurrent;
    HDC hdc;
} PAINTFONTINFO;
typedef PAINTFONTINFO *PPAINTFONTINFO;

#define CLIENTWINDOW    TEXT ("ClientWnd")

int InitClient (HINSTANCE);
int TermClient (HINSTANCE, int);

//----------------------------------------------------------------------
// Function prototypes
//
int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

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

// Message handlers
LRESULT DoCreateFrame (HWND, UINT, WPARAM, LPARAM);
LRESULT DoSizeFrame (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyFrame (HWND, UINT, WPARAM, LPARAM);

LRESULT DoCreateClient (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintClient (HWND, UINT, WPARAM, LPARAM);
LRESULT DoVScrollClient (HWND, UINT, WPARAM, LPARAM);
```

FontList2.c

```
//======================================================================
// FontList2 - Lists the available fonts in the system
//
// 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 "FontList2.h"               // Program-specific stuff
```

(continued)

Figure 4-2. *continued*

```
//----------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("FontList2");
HINSTANCE hInst;                     // Program instance handle

// Message dispatch table for FrameWindowProc
const struct decodeUINT FrameMessages[] = {
    WM_CREATE, DoCreateFrame,
    WM_SIZE, DoSizeFrame,
    WM_DESTROY, DoDestroyFrame,
};
//======================================================================
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;
    HWND hwndFrame;

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

    // Initialize this instance.
    hwndFrame = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndFrame == 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;

    // Register application frame window class.
    wc.style = 0;                               // Window style
    wc.lpfnWndProc = FrameWndProc;              // 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;

    // Initialize client window class.
    if (InitClient (hInstance) != 0) return 2;
    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 frame window.
    hWnd = CreateWindow (szAppName,             // Window class
                         TEXT ("Font List 2"),  // 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;
}
```

(continued)

Figure 4-2. *continued*

```
//----------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {

    return nDefRC;
}
//======================================================================
// Message handling procedures for FrameWindow
//----------------------------------------------------------------------
// FrameWndProc - Callback function for application window
//
LRESULT CALLBACK FrameWndProc (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(FrameMessages); i++) {
        if (wMsg == FrameMessages[i].Code)
            return (*FrameMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
// DoCreateFrame - Process WM_CREATE message for window.
//
LRESULT DoCreateFrame (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    HWND hwndCB, hwndClient;
    INT sHeight;
    LPCREATESTRUCT lpcs;

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

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);
    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    sHeight = CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));
    //
    // Create client window.  Size it so that it fits under
    // the command bar and fills the remaining client area.
    //
    hwndClient = CreateWindow (CLIENTWINDOW, TEXT (""),
                               WS_VISIBLE | WS_CHILD | WS_VSCROLL,
                               lpcs->x, lpcs->y + sHeight,
                               lpcs->cx, lpcs->cy - sHeight,
                               hWnd, (HMENU)IDC_CLIENT,
                               lpcs->hInstance, NULL);

    // Destroy frame if client window not created.
    if (!IsWindow (hwndClient))
        DestroyWindow (hWnd);
    return 0;
}
//----------------------------------------------------------------------
// DoSizeFrame - Process WM_SIZE message for window.
//
LRESULT DoSizeFrame (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam) {
    RECT rect;
    INT i;

    GetClientRect (hWnd, &rect);
    i = CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));
    rect.top += i;

    SetWindowPos (GetDlgItem (hWnd, IDC_CLIENT), NULL, rect.left, rect.top,
                  rect.right - rect.left, rect.bottom - rect.top,
                  SWP_NOZORDER);
    return 0;
}
//----------------------------------------------------------------------
// DoDestroyFrame - Process WM_DESTROY message for window.
//
LRESULT DoDestroyFrame (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}MM
```

ClientWnd.c

```
//======================================================================
// ClientWnd - Client window code for FontList2
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
```

(continued)

Figure 4-2. *continued*

```
//======================================================================
#include <windows.h>                 // For all that Windows stuff
#include "FontList2.h"               // Program-specific stuff

extern HINSTANCE hInst;
BOOL fFirst = TRUE;

//----------------------------------------------------------------------
// Global data
//
FONTFAMSTRUCT ffs[FAMILYMAX];
INT sFamilyCnt = 0;
INT sVPos = 0;
INT sVMax = 0;

// Message dispatch table for ClientWindowProc
const struct decodeUINT ClientMessages[] = {
    WM_CREATE, DoCreateClient,
    WM_PAINT, DoPaintClient,
    WM_VSCROLL, DoVScrollClient,
};
//----------------------------------------------------------------------
// InitClient - Client window initialization
//
int InitClient (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application client window class.
    wc.style = 0;                             // Window style
    wc.lpfnWndProc = ClientWndProc;           // 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 = CLIENTWINDOW;          // Window class name

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

    return 0;
}
//----------------------------------------------------------------------
// TermClient - Client window cleanup
//
```

```
int TermClient (HINSTANCE hInstance, int nDefRC) {
    return nDefRC;
}
//======================================================================
// Font callback functions
//----------------------------------------------------------------------
// FontFamilyCallback - Callback function that enumerates the font
// families.
//
int CALLBACK FontFamilyCallback (CONST LOGFONT *lplf,
                                 CONST TEXTMETRIC *lpntm,
                                 DWORD nFontType, LPARAM lParam) {
    int rc = 1;

    // Stop enumeration if array filled.
    if (sFamilyCnt >= FAMILYMAX)
        return 0;
    // Copy face name of font.
    lstrcpy (ffs[sFamilyCnt++].szFontFamily, lplf->lfFaceName);

    return rc;
}
//----------------------------------------------------------------------
// EnumSingleFontFamily - Callback function that enumerates the font
// families
//
int CALLBACK EnumSingleFontFamily (CONST LOGFONT *lplf,
                                   CONST TEXTMETRIC *lpntm,
                                   DWORD nFontType, LPARAM lParam) {
    PFONTFAMSTRUCT pffs;

    pffs = (PFONTFAMSTRUCT) lParam;
    pffs->nNumFonts++;     // Increment count of fonts in family.
    return 1;
}
//----------------------------------------------------------------------
// PaintSingleFontFamily - Callback function that enumerates the font
// families.
//
int CALLBACK PaintSingleFontFamily (CONST LOGFONT *lplf,
                                    CONST TEXTMETRIC *lpntm,
                                    DWORD nFontType, LPARAM lParam) {
    PPAINTFONTINFO ppfi;
    TCHAR szOut[256];
    INT nFontHeight, nPointSize;
    TEXTMETRIC tm;
    HFONT hFont, hOldFont;
```

(continued)

Figure 4-2. *continued*

```
    ppfi = (PPAINTFONTINFO) lParam;  // Translate lParam into
                                     // structure pointer.

    // Create the font from the LOGFONT structure passed.
    hFont = CreateFontIndirect (lplf);

    // Select the font into the device context.
    hOldFont = SelectObject (ppfi->hdc, hFont);

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

    // Compute font size.
    nPointSize = (lplf->lfHeight * 72) /
                 GetDeviceCaps(ppfi->hdc,LOGPIXELSY);

    // Format string and paint on display.
    wsprintf (szOut, TEXT ("%s   Point:%d"), lplf->lfFaceName,
              nPointSize);
    ExtTextOut (ppfi->hdc, 25, ppfi->yCurrent, 0, NULL,
                szOut, lstrlen (szOut), NULL);

    // Update new draw point.
    ppfi->yCurrent += nFontHeight;
    // Deselect font and delete.
    SelectObject (ppfi->hdc, hOldFont);
    DeleteObject (hFont);
    return 1;
}
//======================================================================
// Message handling procedures for ClientWindow
//----------------------------------------------------------------------
// ClientWndProc - Callback function for application window
//
LRESULT CALLBACK ClientWndProc (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(ClientMessages); i++) {
        if (wMsg == ClientMessages[i].Code)
            return (*ClientMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
// DoCreateClient - Process WM_CREATE message for window.
//
LRESULT DoCreateClient (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    HDC hdc;
    INT i, rc;

    //Enumerate the available fonts.
    hdc = GetDC (hWnd);
    rc = EnumFontFamilies ((HDC)hdc, (LPTSTR)NULL, FontFamilyCallback, 0);

    for (i = 0; i < sFamilyCnt; i++) {
        ffs[i].nNumFonts = 0;
        rc = EnumFontFamilies ((HDC)hdc, ffs[i].szFontFamily,
                               EnumSingleFontFamily,
                               (LPARAM)(PFONTFAMSTRUCT)&ffs[i]);
    }
    ReleaseDC (hWnd, hdc);
    return 0;
}
//----------------------------------------------------------------------
// DoPaintClient - Process WM_PAINT message for window.
//
LRESULT DoPaintClient (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PAINTSTRUCT ps;
    RECT rect;
    HDC hdc;
    TEXTMETRIC tm;
    INT nFontHeight, i;
    TCHAR szOut[256];
    PAINTFONTINFO pfi;
    SCROLLINFO si;

    hdc = BeginPaint (hWnd, &ps);

    GetClientRect (hWnd, &rect);

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

(continued)

Figure 4-2. *continued*

```
    // Initialize struct that is passed to enumerate function.
    pfi.yCurrent = rect.top - sVPos;
    pfi.hdc = hdc;
    for (i = 0; i < sFamilyCnt; i++) {

        // Format output string and paint font family name.
        wsprintf (szOut, TEXT ("Family: %s   Number of fonts:%d"),
                  ffs[i].szFontFamily, ffs[i].nNumFonts);
        ExtTextOut (hdc, 5, pfi.yCurrent, 0, NULL,
                    szOut, lstrlen (szOut), NULL);
        pfi.yCurrent += nFontHeight;

        // Enumerate each family to draw a sample of that font.
        EnumFontFamilies ((HDC)hdc, ffs[i].szFontFamily,
                          PaintSingleFontFamily,
                          (LPARAM)&pfi);
    }
    // Compute the total height of the text in the window.
    if (fFirst) {
        sVPos = 0;
        sVMax = (pfi.yCurrent - rect.top) - (rect.bottom - rect.top);

        si.cbSize = sizeof (si);
        si.nMin = 0;
        si.nMax = pfi.yCurrent;
        si.nPage = rect.bottom - rect.top;
        si.nPos = sVPos;
        si.fMask = SIF_ALL;
        SetScrollInfo (hWnd, SB_VERT, &si, TRUE);
        fFirst = FALSE;
    }
    EndPaint (hWnd, &ps);
    return 0;
}
//----------------------------------------------------------------------
// DoVScrollClient - Process WM_VSCROLL message for window.
//
LRESULT DoVScrollClient (HWND hWnd, UINT wMsg, WPARAM wParam,
                         LPARAM lParam) {
    RECT rect;
    SCROLLINFO si;
    INT sOldPos = sVPos;

    GetClientRect (hWnd, &rect);
```

Page 00187

```
    switch (LOWORD (wParam)) {
    case SB_LINEUP:
        sVPos -= 10;
        break;

    case SB_LINEDOWN:
        sVPos += 10;
        break;

    case SB_PAGEUP:
        sVPos -= rect.bottom - rect.top;
        break;

    case SB_PAGEDOWN:
        sVPos += rect.bottom - rect.top;
        break;

    case SB_THUMBPOSITION:
        sVPos = HIWORD (wParam);
        break;
    }
    // Check range.
    if (sVPos < 0)
        sVPos = 0;
    if (sVPos > sVMax)
        sVPos = sVMax;

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

        InvalidateRect (hWnd, NULL, TRUE);
    }
    return 0;
}
```

The window procedure for the frame window is quite simple. Just as in the original FontList program in Chapter 2, the command bar is created in the WM_CREATE message handler, *DoCreateFrame*. Now, however, this procedure also calls *CreateWindow* to create the child window in the area underneath the command bar. The child window is created with three style flags: WS_VISIBLE, so that the window is initially visible; WS_CHILD, required because it will be a child window of the frame window; and WS_VSCROLL to add the vertical scroll bar to the child window.

The majority of the work for the program is handled in the client window procedure. Here the same font enumeration calls are made to query the fonts in the system. The WM_PAINT handler, *DoPaintClient*, has a new characteristic: it now bases what it paints on the new global variable *sVPos*, which provides vertical positioning. That variable is initialized to 0 in *DoCreateClient* and is changed in the handler for a new message, WM_VSCROLL.

Scroll bar messages

A WM_VSCROLL message is sent to the owner of a vertical scroll bar any time the user taps on the scroll bar to change its position. A complementary message, WM_HSCROLL, is identical to WM_VSCROLL but is sent when the user taps on a horizontal scroll bar. For both these messages, the *wParam* and *lParam* assignments are the same. The low word of the *wParam* parameter contains a code indicating why the message was sent. Figure 4-3 shows a diagram of horizontal and vertical scroll bars and how tapping on different parts of the scroll bars results in different messages. The high word of *wParam* is the position of the thumb, but this value is valid only while you're processing the SB_THUMBPOSITION and SB_THUMBTRACK codes, which I'll explain shortly. If the scroll bar sending the message is a stand-alone control and not attached to a window, the *lParam* parameter contains the window handle of the scroll bar.

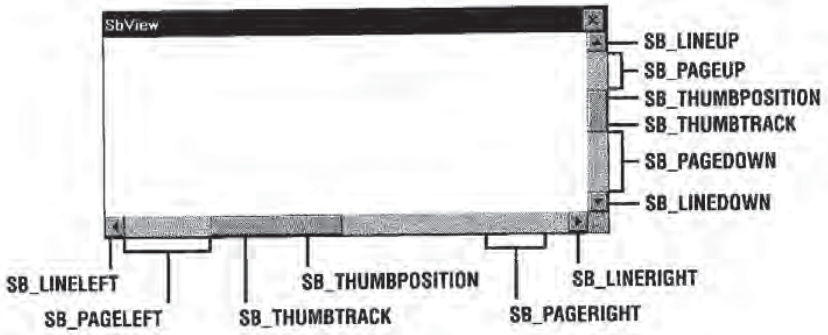


Figure 4-3. *Scroll bars and their hot spots.*

The scroll bar message codes sent by the scroll bar allow the program to react to all the different user actions allowable by a scroll bar. The response required by each code is listed in the following table, Figure 4-4.

The SB_LINE*xxx* and SB_PAGE*xxx* codes are pretty straightforward. You move the scroll position either a line or a page at a time. The SB_THUMBPOSITION and SB_THUMBTRACK codes can be processed in one of two ways. When the user drags the scroll bar thumb, the scroll bar sends SB_THUMBTRACK code so that a program can interactively track the dragging of the thumb. If your application is fast enough, you can simply process the SB_THUMBTRACK code and interactively update the display. If you field the SB_THUMBTRACK code, however, your application must be quick enough to redraw the display so that the thumb can be dragged without hesitation or jumping of the scroll bar. This is especially a problem on the slower devices that run Windows CE.

Codes	***Response***
For WS_VSCROLL	
SB_LINEUP	Program should scroll the screen up one line.
SB_LINEDOWN	Program should scroll the screen down one line.
SB_PAGEUP	Program should scroll the screen up one screen's worth of data.
SB_PAGEDOWN	Program should scroll the screen down one screen's worth of data.
For WS_HSCROLL	
SB_LINELEFT	Program should scroll the screen left one character.
SB_LINERIGHT	Program should scroll the screen right one character.
SB_PAGELEFT	Program should scroll the screen left one screen's worth of data.
SB_PAGERIGHT	Program should scroll the screen right one screen's worth of data.
For both WS_VSCROLL and WS_HSCROLL	
SB_THUMBTRACK	Programs with enough speed to keep up should update the display with the new scroll position.
SB_THUMBPOSITION	Programs that can't update the display fast enough to keep up with the SB_THUMBTRACK message should update the display with the new scroll position.
SB_ENDSCROLL	This code indicates that the scroll bar has completed the scroll event. No action is required by the program.
SB_TOP	Program should set the display to the top or left end of the data.
SB_BOTTOM	Program should set the display to the bottom or right end of the data.

Figure 4-4. *Scroll codes.*

If your application (or the system it's running on) is too slow to quickly update the display for every SB_THUMBTRACK code, you can ignore the SB_THUMBTRACK and wait for the SB_THUMBPOSITION code that's sent when the user drops the scroll bar thumb. Then you have to update the display only once, after the user has finished moving the scroll bar thumb.

Configuring a scroll bar

To use a scroll bar, an application should first set the minimum and maximum values—the range of the scroll bar, along with the initial position. Windows CE scroll bars, like their Win32 cousins, support proportional thumb sizes, which provide feedback to the user about the size of the current visible page compared to the entire scroll range. To set all these parameters, Windows CE applications should use the *SetScrollInfo* function, prototyped as

```
int SetScrollInfo (HWND hwnd, int fnBar, LPSCROLLINFO lpsi, BOOL fRedraw);
```

The first parameter is either the handle of the window that contains the scroll bar or the window handle of the scroll bar itself. The second parameter, *fnBar*, is a flag that determines the use of the window handle. The scroll bar flag can be one of three values: SB_HORZ for a window's standard horizontal scroll bar, SB_VERT for a window's standard vertical scroll bar, or SB_CTL if the scroll bar being set is a standalone control. Unless the scroll bar is a control, the window handle is the handle of the window containing the scroll bar. With SB_CTL, however, the handle is the window handle of the scroll bar control itself. The last parameter is *fRedraw*, a Boolean value that indicates whether the scroll bar should be redrawn after the call has been completed.

The third parameter is a pointer to a SCROLLINFO structure, which is defined as

```
typedef struct tagSCROLLINFO {
    UINT cbSize;
    UINT fMask;
    int   nMin;
    int   nMax;
    UINT nPage;
    int   nPos;
    int   nTrackPos;
} SCROLLINFO;
```

This structure allows you to completely specify the scroll bar parameters. The *cbSize* field must be set to the size of the SCROLLINFO structure. The *fMask* field contains flags indicating what other fields in the structure contain valid data. The *nMin* and *nMax* fields can contain the minimum and maximum scroll values the scroll bar can report. Windows looks at the values in these fields if the *fMask* parameter contains the SIF_RANGE flag. Likewise, the *nPos* field sets the position of the scroll bar within its predefined range if the *fMask* field contains the SIF_POS flag.

The *nPage* field allows a program to define the size of the currently viewable area of the screen in relation to the entire scrollable area. This allows a user to have a feel for how much of the entire scrolling range is currently visible. This field is used only if the *fMask* field contains the SIF_PAGE flag. The last member of the SCROLLINFO structure, *nTrackPos,* isn't used by the *SetScrollInfo* call and is ignored.

The *fMask* field can contain one last flag. Passing a SIF_DISABLENOSCROLL flag causes the scroll bar to be disabled, but still visible. This is handy when the entire scrolling range is visible within the viewable area and no scrolling is necessary. Disabling the scroll bar in this case is often preferable to simply removing the scroll bar completely.

Those with a sharp eye for detail will notice a problem with the width of the fields in the SCROLLINFO structure. The *nMin*, *nMax*, and *nPos* fields are integers and therefore in the world of Windows CE, are 32 bits wide. On the other hand, the WM_HSCROLL and WM_VSCROLL messages can return only a 16-bit position in the high word of the *wParam* parameter. If you're using scroll ranges greater than 65,535, use this function:

```
BOOL GetScrollInfo (HWND hwnd, int fnBar, LPSCROLLINFO lpsi);
```

As with *SetScrollInfo*, the flags in the *fnBar* field indicate the window handle that should be passed to the function. The SCROLLINFO structure is identical to the one used in *SetScrollInfo*; however, before it can be passed to *GetScrollInfo*, it must be initialized with the size of the structure in *cbSize*. An application must also indicate what data it wants the function to return by setting the appropriate flags in the *fMask* field. The flags used in *fMask* are the same as the ones used in *SetScrollInfo* with a couple of additions. Now a SIF_TRACKPOS flag can be passed to have the scroll bar return its current thumb position. When called during a WM_*x*SCROLL message, the *nTrackPos* field contains the real time position while the *nPos* field contains the scroll bar position at the start of the drag of the thumb.

The scroll bar is an unusual control in that it can be added easily to windows simply by specifying a window style flag. It's also unusual in that the control is placed outside the client area of the window. The reason for this assistance is that scroll bars are commonly needed by applications, so the Windows developers made it easy to attach scroll bars to windows. Now let's look at the other basic Windows controls.

WINDOWS CONTROLS

While scroll bars hold a special place because of their easy association with standard windows, there are a large number of other controls that Windows applications often use, including buttons, edit boxes, and list boxes. In short, controls are simply predefined window classes. Each has a custom window procedure supplied by Windows that gives each of these controls a tightly defined user and programming interface.

Since a control is just another window, it can be created with a call to *CreateWindow* or *CreateWindowEx*, or, as I will explain later in this chapter, automatically by the dialog manager during the creation of a dialog box. Like menus, controls notify the parent window of events via WM_COMMAND messages encoding events and the ID and window handle of the control encoded in the parameters of the message. Controls can also be configured and manipulated using predefined messages sent to the control. Among other things, applications can set the state of buttons, add or delete items to list boxes, and set the selection of text in edit boxes all by sending messages to the controls.

There are six predefined window control classes. They are

- *Button* A wide variety of buttons.
- *Edit* A window that can be used to enter or display text.
- *List* A window that contains a list of strings.
- *Combo* A combination edit box and list box.
- *Static* A window that displays text or graphics that a user can't change.
- *Scroll bar* A scroll bar not attached to a specific window.

Each of these controls has a wide range of function, far too much for me to cover completely in this chapter. But I'll quickly review these controls, mentioning at least the highlights. Afterward, I'll show you an example program, CtlView, to demonstrate these controls and their interactions with their parent windows.

Button Controls

Button controls enable several forms of input to the program. Buttons come in many styles, including push buttons, check boxes, and radio buttons. Each style is designed for a specific use—for example, push buttons are designed for receiving momentary input, check boxes are designed for on/off input, and radio buttons allow a user to select one of a number of choices.

Push buttons

In general, push buttons are used to invoke some action. When a user presses a push button using a stylus, the button sends a WM_COMMAND message with a BN_CLICKED (for button notification clicked) notify code in the high word of the *wParam* parameter.

Check boxes

Check boxes display a square box and a label that asks the user to specify a choice. A check box retains its state, either checked or unchecked, until the user clicks it again or the program forces the button to change state. In addition to the standard

BS_CHECKBOX style, check boxes can come in a 3-state style, BS_3STATE, that allows the button to be disabled and shown grayed out. Two additional styles, BS_AUTOCHECKBOX and BS_AUTO3STATE, automatically update the state and look of the control to reflect the checked, unchecked, and in the case of the 3-state check box, the disabled state.

As with push buttons, check boxes send a BN_CLICKED notification when the button is clicked. Unless the check box has one of the automatic styles, it's the responsibility of the application to manually change the state of the button. This can be done by sending a BM_SETCHECK message to the button with the *wParam* set to 0 to uncheck the button or 1 to check the button. The 3-state check boxes have a third, disabled state that can be set by means of the BM_SETCHECK message with the *wParam* value set to 2. An application can determine the current state using the BM_GETCHECK message.

Radio buttons

Radio buttons allow a user to select from a number of choices. Radio buttons are grouped in a set, with only one of the set ever being checked at a time. If it's using the standard BS_RADIOBUTTON style, the application is responsible for checking and unchecking the radio buttons so that only one is checked at a time. However, like check boxes, radio buttons have an alternative style, BS_AUTORADIOBUTTON, that automatically maintains the group of buttons so that only one is checked.

Group boxes

Strangely, the group box is also a type of button. A group box appears to the user as a hollow box with an integrated text label surrounding a set of controls that are naturally grouped together. Group boxes are merely an organizational device and have no programming interface other than the text of the box, which is specified in the window title text upon creation of the group box. Group boxes should be created after the controls within the box are created. This ensures that the group box will be "beneath" the controls it contains in the window Z-order.

You should also be careful when using group boxes on Windows CE devices. The problem isn't with the group box itself, but with the small size of the Windows CE screen. Group boxes take up valuable screen real estate that can be better used by functional controls. This is especially the case on the Palm-size PC with its very small screen. In many cases, a line drawn between sets of controls can visually group the controls as well as a group box can.

Customizing the appearance of a button

You can further customize the appearance of the buttons described so far by using a number of additional styles. The styles, BS_RIGHT, BS_LEFT, BS_BOTTOM, and BS_TOP, allow you to position the button text in a place other than the default center of the button. The BS_MULTILINE style allows you to specify more than one line of text in the button. The text is flowed to fit within the button. The newline character (\n) in the button text can be used to specifically define where line breaks occur. Windows CE doesn't support the BS_ICON and BS_BITMAP button styles supported by other versions of Windows.

Owner-draw buttons

You can totally control the look of a button by specifying the BS_OWNERDRAW style. When a button is specified as owner-draw, its owner window is entirely responsible for drawing the button for all the states in which it might occur. When a window contains an owner-draw button, it's sent a WM_DRAWITEM message to inform it that a button needs to be drawn. For this message, the *wParam* parameter contains the ID value for the button and the *lParam* parameter points to a DRAWITEMSTRUCT structure defined as

```
typedef struct tagDRAWITEMSTRUCT {
    UINT  CtlType;
    UINT  CtlID;
    UINT  itemID;
    UINT  itemAction;
    UINT  itemState;
    HWND  hwndItem;
    HDC   hDC;
    RECT  rcItem;
    DWORD itemData;
} DRAWITEMSTRUCT;
```

The *CtlType* field is set to ODT_BUTTON while the *CtlID* field, like the *wParam* parameter, contains the button's ID value. The *itemAction* field contains flags that indicate what needs to be drawn and why. The most significant of these fields is *itemState*, which contains the state (selected, disabled, and so forth) of the button. The *hDC* field contains the device context handle for the button window while the *rcItem* RECT contains the dimensions of the button. The *itemData* field is NULL for owner-draw buttons.

As you might expect, the WM_DRAWITEM handler contains a number of GDI calls to draw lines, rectangles, and whatever else is needed to render the button. An important aspect of drawing a button is matching the standard colors of the other windows in the system. Since these colors can change, they shouldn't be hard coded. You can query to find out which are the proper colors by using the function

```
DWORD GetSysColor (int nIndex);
```

This function returns an RGB color value for the colors defined for different aspects of windows and controls in the system. Among a number of predefined index values passed in the index parameter, an index of COLOR_BTNFACE returns the proper color for the face of a button while COLOR_BTNSHADOW returns the dark color for creating the three-dimensional look of a button.

The Edit Control

The edit control is a window that allows the user to enter and edit text. As you might imagine, the edit control is one of the handiest controls in the Windows control pantheon. The edit control is equipped with full editing capability, including cut, copy, and paste interaction with the system clipboard, all without assistance from the application. Edit controls display a single line, or by specifying the ES_MULTILINE style, multiple lines of text. The Notepad accessory, provided with the desktop versions of Windows, is simply a top-level window that contains a multiline edit control.

The edit control has a few other features that should be mentioned. An edit control with the ES_PASSWORD style displays an asterisk (*) character by default in the control for each character typed; the control saves the real character. The ES_READONLY style protects the text contained in the control so that it can be read, or copied into the clipboard, but not modified. The ES_LOWERCASE and ES_UPPERCASE styles force characters entered into the control to be changed to the specified case.

You can add text to an edit control by using the WM_SETTEXT message and retrieve text by using the WM_GETTEXT message. Selection can be controlled using the EM_SETSEL message. This message specifies the starting and ending characters in the selected area. Other messages allow the position of the caret (the marker that indicates the current entry point in an edit field) to be queried and set. Multiline edit controls contain a number of additional messages to control scrolling as well as to access characters by line and column position.

The List Box Control

The list box control displays a list of text items so that the user might select one or more of the items within the list. The list box stores the text, optionally sorts the items, and manages the display of the items, including scrolling. List boxes can be configured to allow selection of a single item or multiple items or to prevent any selection at all.

You add an item to a list box by sending an LB_ADDSTRING or LB_INSERTSTRING message to the control, passing a pointer to the string to add in the *lParam* parameter. The LB_ADDSTRING message places the newly added string at the end of the list of items while LB_INSERTSTRING can place the string anywhere within the list of items in the list box. The list box can be searched for a particular item using the LB_FIND message.

Selection status can be queried using the LB_GETCURSEL for single selection list boxes. For multiple selection list boxes, the LB_GETSELCOUNT and LB_GETSELITEMS can be used to retrieve the items currently selected. Items in the list box can be selected programmatically using the LB_SETCURSEL and LB_SETSEL messages.

Windows CE supports most of the list box functionality available in other versions of Windows with the exception of owner-draw list boxes, and the LB_DIR family of messages. A new style, LBS_EX_CONSTSTRINGDATA, is supported under Windows CE. A list box with this style doesn't store strings passed to it. Instead, the pointer to the string is stored and the application is responsible for maintaining the string. For large arrays of strings that might be loaded from a resource, this procedure can save RAM because the list box won't maintain a separate copy of the list of strings.

The Combo Box Control

The combo box is (as the name implies) a combination of controls—in this case, a single-line edit control and a list box. The combo box is a space-efficient control for selecting one item from a list of many or for providing an edit field with a list of predefined, suggested entries. Under Windows CE, the combo box comes in two styles: drop-down and drop-down list. (Simple combo boxes aren't supported.) The drop-down style combo box contains an edit field with a button at the right end. Clicking on the button displays a list box that might contain more selections. Clicking on one of the selections fills the edit field of the combo box with the selection. The drop-down list style replaces the edit box with a static text control. This allows the user to select from an item in the list but prevents the user from entering an item that's not in the list.

Since the combo box combines the edit and list controls, a list of the messages used to control the combo box strongly resembles a merged list of the messages for the two base controls. CB_ADDSTRING, CB_INSERTSTRING, and CB_FINDSTRING act like their list box cousins. Likewise the CB_SETEDITSELECT and CB_GETEDITSELECT messages set and query the selected characters in the edit box of a drop-down or a drop-down list combo box. To control the drop-down state of a drop-down or drop-down list combo box, the messages CB_SHOWDROPDOWN and CB_GETDROPPEDSTATE can be used.

As in the case of the list box, Windows CE doesn't support owner-draw combo boxes. However, the combo box supports the CBS_EX_CONSTSTRINGDATA extended style, which instructs the combo box to store a pointer to the string for an item instead of the string itself. As with the list box LBS_EX_CONSTSTRINGDATA style, this procedure can save RAM if an application has a large array of strings stored in ROM because the combo box won't maintain a separate copy of the list of strings.

Static Controls

Static controls are windows that display text, icons, or bitmaps not intended for user interaction. You can use static text controls to label other controls in a window. What a static control displays is defined by the text and the style for the control Under Windows CE, static controls support the following styles:

- *SS_LEFT* Displays a line of left-aligned text. The text is wrapped, if necessary, to fit inside the control.
- *SS_CENTER* Displays a line of text centered in the control. The text is wrapped, if necessary, to fit inside the control.
- *SS_RIGHT* Displays a line of text aligned with the right side of the control. The text is wrapped, if necessary, to fit inside the control.
- *SS_LEFTNOWORDWRAP* Displays a line of left-aligned text. The text isn't wrapped to multiple lines. Any text extending beyond the right side of the control is clipped.
- *SS_BITMAP* Displays a bitmap. Window text for the control specifies the name of the resource containing the bitmap.
- *SS_ICON* Displays an icon. Window text for the control specifies the name of the resource containing the icon.

Static controls with the SS_NOTIFY style send a WM_COMMAND message when the control is clicked, enabled, or disabled, although the Windows CE version of the static control doesn't send a notification when it's double-clicked. The SS_CENTERIMAGE style, used in combination with the SS_BITMAP or SS_ICON style, centers the image within the control. The SS_NOPREFIX style can be used in combination with the text styles. It prevents the ampersand (&) character from being interpreted as indicating the next character is an accelerator character.

Windows CE doesn't support static controls that display filled or hollow rectangles such as those drawn with the SS_WHITEFRAME or SS_BLACKRECT styles. Also, Windows CE doesn't support owner-draw static controls.

The Scroll Bar Control

The scroll bar control operates identically to the window scroll bars described previously' with the exception that the *fnBar* field used in *SetScrollInfo* and *GetScrollInfo* must be set to SB_CTL. The *hwnd* field then must be set to the handle of the scroll bar control, not to the window that owns the scroll bar. Like window scroll bars, the owner of the scroll bar is responsible for fielding the scroll messages WM_VSCROLL and WM_HSCROLL and setting the new position of the scroll bar in response to these messages.

The CtlView Example Program

The CtlView example program, shown in Figure 4-5, demonstrates all the controls I've just described. The example makes use of several application-defined child windows that contain various controls. You switch between the different child windows by clicking on one of five radio buttons displayed across the top of the main window. As each of the controls reports a notification through a WM_COMMAND message, that notification is displayed in a list box on the right side of the window. CtlView is handy for observing just what messages a control sends to its parent window and when they're sent. One problem with CtlView is that it's designed for an H/PC screen, not a Palm-size PC screen. If you run CtlView on a Palm-size PC, you'll see that the controls don't all fit onto the small Palm-size PC screen.

CtlView.rc

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

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

ID_ICON      ICON   "CtlView.ico"     // Program icon
TEXTICON     ICON   "btnicon.ico"     // Icon used in static window
STATICBMP    BITMAP "statbmp.bmp"     // Bitmap used in static window
```

CtlView.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);
```

Figure 4-5. *The CtlView program.*

```
};
struct decodeCMD {                                    // Structure associates
    UINT Code;                                        // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);           // function.
};

//----------------------------------------------------------------------
// Generic defines used by application
#define  IDI_BTNICON        20                        // Icon used on button

#define  ID_ICON            1                         // Icon ID
#define  IDC_CMDBAR         2                         // Command bar ID
#define  IDC_RPTLIST        3                         // Report window ID

// Client window IDs go from 5 through 9.
#define  IDC_WNDSEL         5                         // Starting client
                                                      // window IDs

// Radio button IDs go from 10 through 14.
#define  IDC_RADIOBTNS      10                        // Starting ID of
                                                      // radio buttons

// Button window defines
#define  IDC_PUSHBTN   100
#define  IDC_CHKBOX    101
#define  IDC_ACHKBOX   102
#define  IDC_A3STBOX   103
#define  IDC_RADIO1    104
#define  IDC_RADIO2    105
#define  IDC_OWNRDRAW  106

// Edit window defines
#define  IDC_SINGLELINE 100
#define  IDC_MULTILINE  101
#define  IDC_PASSBOX    102

// List box window defines
#define  IDC_COMBOBOX   100
#define  IDC_SNGLELIST  101
#define  IDC_MULTILIST  102

// Static control window defines
#define  IDC_LEFTTEXT   100
#define  IDC_RIGHTTEXT  101
#define  IDC_CENTERTEXT 102
#define  IDC_ICONCTL    103
#define  IDC_BITMAPCTL  104
```

(continued)

Figure 4-5. *continued*

```
// Scroll bar window defines
#define  IDC_LRSCROLL   100
#define  IDC_UDSCROLL   101

// User defined message to add a line to the window
#define MYMSG_ADDLINE   (WM_USER + 10)

typedef struct {
    TCHAR *szClass;
    INT   nID;
    TCHAR *szTitle;
    INT   x;
    INT   y;
    INT   cx;
    INT   cy;
    DWORD lStyle;
} CTLWNDSTRUCT, *PCTLWNDSTRUCT;

typedef struct {
    WORD wMsg;
    INT nID;
    WPARAM wParam;
    LPARAM lParam;
} CTLMSG, * PCTLMSG;

typedef struct {
    TCHAR *pszLabel;
    WORD wNotification;
} NOTELABELS, *PNOTELABELS;

//----------------------------------------------------------------------
// Function prototypes
//
int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

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

// Message handlers
LRESULT DoCreateFrame (HWND, UINT, WPARAM, LPARAM);
LRESULT DoCommandFrame (HWND, UINT, WPARAM, LPARAM);
LRESULT DoAddLineFrame (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyFrame (HWND, UINT, WPARAM, LPARAM);
```

```
//----------------------------------------------------------------------
// Window prototypes and defines for BtnWnd
//
#define BTNWND    TEXT ("ButtonWnd")
int InitBtnWnd (HINSTANCE);

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

LRESULT DoCreateBtnWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoCtlColorBtnWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoCommandBtnWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDrawItemBtnWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoMeasureItemBtnWnd (HWND, UINT, WPARAM, LPARAM);

//----------------------------------------------------------------------
// Window prototypes and defines for EditWnd
//
#define EDITWND    TEXT ("EditWnd")
int InitEditWnd (HINSTANCE);

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

LRESULT DoCreateEditWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoCommandEditWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDrawItemEditWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoMeasureItemEditWnd (HWND, UINT, WPARAM, LPARAM);

//----------------------------------------------------------------------
// Window prototypes and defines for ListWnd
//
#define LISTWND    TEXT ("ListWnd")
int InitListWnd (HINSTANCE);

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

LRESULT DoCreateListWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoCommandListWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDrawItemListWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoMeasureItemListWnd (HWND, UINT, WPARAM, LPARAM);

//----------------------------------------------------------------------
// Window prototypes and defines for StatWnd
//
```

(continued)

Figure 4-5. *continued*

```
#define STATWND     TEXT ("StaticWnd")
int InitStatWnd (HINSTANCE);

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

LRESULT DoCreateStatWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoCommandStatWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDrawItemStatWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoMeasureItemStatWnd (HWND, UINT, WPARAM, LPARAM);

//----------------------------------------------------------------------
// Window prototypes and defines ScrollWnd
//
#define SCROLLWND     TEXT ("ScrollWnd")
int InitScrollWnd (HINSTANCE);

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

LRESULT DoCreateScrollWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoVScrollScrollWnd (HWND, UINT, WPARAM, LPARAM);
LRESULT DoHScrollScrollWnd (HWND, UINT, WPARAM, LPARAM);
```

CtlView.c

```
//======================================================================
// CtlView - Lists the available fonts in the system.
//
// 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 "CtlView.h"                     // Program-specific stuff

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

// Message dispatch table for FrameWindowProc
const struct decodeUINT FrameMessages[] = {
    WM_CREATE, DoCreateFrame,
    WM_COMMAND, DoCommandFrame,
    MYMSG_ADDLINE, DoAddLineFrame,
    WM_DESTROY, DoDestroyFrame,
};

typedef struct {
    TCHAR *szTitle;
    INT   nID;
    TCHAR *szCtlWnds;
    HWND  hWndClient;
} RBTNDATA;

// Text for main window radio buttons
TCHAR *szBtnTitle[] = {TEXT ("Buttons"), TEXT ("Edit"), TEXT ("List"),
                       TEXT ("Static"), TEXT ("Scroll")};
// Class names for child windows containing controls
TCHAR *szCtlWnds[] = {BTNWND, EDITWND, LISTWND, STATWND, SCROLLWND};

INT nWndSel = 0;

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

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

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

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
```

(continued)

Figure 4-5. *continued*

```
//----------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application frame window class.
    wc.style = 0;                               // Window style
    wc.lpfnWndProc = FrameWndProc;              // 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) GetSysColorBrush (COLOR_STATIC);
    wc.lpszMenuName =  NULL;                    // Menu name
    wc.lpszClassName = szAppName;               // Window class name

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

    // Initialize client window classes
    if (InitBtnWnd (hInstance) != 0) return 2;
    if (InitEditWnd (hInstance) != 0) return 2;
    if (InitListWnd (hInstance) != 0) return 2;
    if (InitStatWnd (hInstance) != 0) return 2;
    if (InitScrollWnd (hInstance) != 0) return 2;
    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 frame window.
    hWnd = CreateWindow (szAppName,             // Window class
                         TEXT ("Control View"), // 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 FrameWindow
//
//----------------------------------------------------------------------
// FrameWndProc - Callback function for application window
//
LRESULT CALLBACK FrameWndProc (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(FrameMessages); i++) {
        if (wMsg == FrameMessages[i].Code)
            return (*FrameMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
// DoCreateFrame - Process WM_CREATE message for window.
//
LRESULT DoCreateFrame (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    LPCREATESTRUCT lpcs;
    HWND hwndCB, hwndChild;
    INT sHeight, i, x, y, cx, cy;
```

(continued)

Figure 4-5. *continued*

```
    // Convert lParam into pointer to create struct.
    lpcs = (LPCREATESTRUCT) lParam;
    x = lpcs->x;
    y = lpcs->y;
    cx = lpcs->cx;
    cy = lpcs->cy;
    nWndSel = 0;
    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);
    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    sHeight = CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    // Create the radio buttons.
    for (i = 0; i < dim(szBtnTitle); i++) {
        hwndChild = CreateWindow (TEXT ("BUTTON"),
                                  szBtnTitle[i], BS_AUTORADIOBUTTON |
                                  WS_VISIBLE | WS_CHILD,
                                  10 + (i * 85), sHeight,
                                  80, 23, hWnd, (HMENU)(IDC_RADIOBTNS+i),
                                  hInst, NULL);

        // Destroy frame if window not created.
        if (!IsWindow (hwndChild)) {
            DestroyWindow (hWnd);
            break;
        }
    }
    //
    // Create report window.  Size it so that it fits under
    // the command bar and fills the remaining client area.
    //
    hwndChild = CreateWindowEx (WS_EX_CLIENTEDGE, TEXT ("listbox"),
                         TEXT (""), WS_VISIBLE | WS_CHILD | WS_VSCROLL |
                         LBS_USETABSTOPS | LBS_NOINTEGRALHEIGHT,
                         cx/2, y + sHeight + 25,
                         cx/2, cy - sHeight - 25,
                         hWnd, (HMENU)IDC_RPTLIST,
                         hInst, NULL);

    // Destroy frame if window not created.
    if (!IsWindow (hwndChild)) {
        DestroyWindow (hWnd);
        return 0;
    }
    // Initialize tab stops for display list box.
    i = 25;
    SendMessage (hwndChild, LB_SETTABSTOPS, 1, (LPARAM)&i);

    //
    // Create the child windows.  Size them so that they fit under
    // the command bar and fill the left side of the child area.
    //
    for (i = 0; i < dim(szCtlWnds); i++) {
        hwndChild = CreateWindowEx (WS_EX_CLIENTEDGE,
                             szCtlWnds[i],
                             TEXT (""), WS_CHILD,
                             x, y + sHeight + 25,
                             cx/2, cy - sHeight - 25,
                             hWnd, (HMENU)(IDC_WNDSEL+i),
                             hInst, NULL);
        // Destroy frame if client window not created.
        if (!IsWindow (hwndChild)) {
            DestroyWindow (hWnd);
            return 0;
        }
    }
    // Check one of the auto radio buttons.
    SendDlgItemMessage (hWnd, IDC_RADIOBTNS+nWndSel, BM_SETCHECK, 1, 0);
    hwndChild = GetDlgItem (hWnd, IDC_WNDSEL+nWndSel);
    ShowWindow (hwndChild, SW_SHOW);
    return 0;
}
//----------------------------------------------------------------------
// DoCommandFrame - Process WM_COMMAND message for window.
//
LRESULT DoCommandFrame (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    HWND hwndTemp;
    int nBtn;
    // Don't look at list box messages.
    if (LOWORD (wParam) == IDC_RPTLIST)
        return 0;
    nBtn = LOWORD (wParam) - IDC_RADIOBTNS;
    if (nWndSel != nBtn) {

        // Hide the currently visible window.
        hwndTemp = GetDlgItem (hWnd, IDC_WNDSEL+nWndSel);
        ShowWindow (hwndTemp, SW_HIDE);

        // Save the current selection.
        nWndSel = nBtn;
```

(continued)

Figure 4-5. *continued*

```
        // Show the window selected via the radio button.
        hwndTemp = GetDlgItem (hWnd, IDC_WNDSEL+nWndSel);
        ShowWindow (hwndTemp, SW_SHOW);
    }
    return 0;
}
//----------------------------------------------------------------------
// DoAddLineFrame - Process MYMSG_ADDLINE message for window.
//
LRESULT DoAddLineFrame (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    TCHAR szOut[128];
    INT i;

    if (LOWORD (wParam) == 0xffff)
        wsprintf (szOut, TEXT ("      \t %s"), (LPTSTR)lParam);
    else
        wsprintf (szOut, TEXT ("id:%x \t %s"), LOWORD (wParam),
                  (LPTSTR)lParam);


    i = SendDlgItemMessage (hWnd, IDC_RPTLIST, LB_ADDSTRING, 0,
                            (LPARAM)(LPCTSTR)szOut);

    if (i != LB_ERR)
        SendDlgItemMessage (hWnd, IDC_RPTLIST, LB_SETTOPINDEX, i,
                            (LPARAM)(LPCTSTR)szOut);
    return 0;
}
//----------------------------------------------------------------------
// DoDestroyFrame - Process WM_DESTROY message for window.
//
LRESULT DoDestroyFrame (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
```

BtnWnd.c

```
//======================================================================
// BtnWnd - Button window code
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
```

```
//======================================================================
#include <windows.h>                 // For all that Windows stuff
#include "Ctlview.h"                 // Program-specific stuff

extern HINSTANCE hInst;

LRESULT DrawButton (HWND hWnd, LPDRAWITEMSTRUCT pdi);
//----------------------------------------------------------------------
// Global data
//

// Message dispatch table for BtnWndWindowProc
const struct decodeUINT BtnWndMessages[] = {
    WM_CREATE, DoCreateBtnWnd,
    WM_CTLCOLORSTATIC, DoCtlColorBtnWnd,
    WM_COMMAND, DoCommandBtnWnd,
    WM_DRAWITEM, DoDrawItemBtnWnd,
};

// Structure defining the controls in the window
CTLWNDSTRUCT  Btns [] = {
    {TEXT ("BUTTON"), IDC_PUSHBTN, TEXT ("Button"),
     10,  10, 120,  23, BS_PUSHBUTTON | BS_NOTIFY},
    {TEXT ("BUTTON"), IDC_CHKBOX, TEXT ("Check box"),
     10,  35, 120,  23, BS_CHECKBOX},
    {TEXT ("BUTTON"), IDC_ACHKBOX, TEXT ("Auto check box"),
     10,  60, 120,  23, BS_AUTOCHECKBOX},
    {TEXT ("BUTTON"), IDC_A3STBOX, TEXT ("Auto 3-state box"),
     10,  85, 120,  23, BS_AUTO3STATE},
    {TEXT ("BUTTON"), IDC_RADIO1, TEXT ("Auto radio button 1"),
     10, 110, 120,  23, BS_AUTORADIOBUTTON},
    {TEXT ("BUTTON"), IDC_RADIO2, TEXT ("Auto radio button 2"),
     10, 135, 120,  23, BS_AUTORADIOBUTTON},
    {TEXT ("BUTTON"), IDC_OWNRDRAW, TEXT ("OwnerDraw"),
     150,  10,  44,  44, BS_PUSHBUTTON | BS_OWNERDRAW},
};
// Structure labeling the button control WM_COMMAND notifications
NOTELABELS nlBtn[] = {{TEXT ("BN_CLICKED "),      0},
                      {TEXT ("BN_PAINT   "),      1},
                      {TEXT ("BN_HILITE  "),      2},
                      {TEXT ("BN_UNHILITE"),      3},
                      {TEXT ("BN_DISABLE "),      4},
                      {TEXT ("BN_DOUBLECLICKED"), 5},
                      {TEXT ("BN_SETFOCUS "),     6},
                      {TEXT ("BN_KILLFOCUS"),     7}
};
```

(continued)

Figure 4-5. *continued*

```
// Handle for icon used in owner-draw icon
HICON hIcon = 0;
//----------------------------------------------------------------------
// InitBtnWnd - BtnWnd window initialization
//
int InitBtnWnd (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application BtnWnd window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = BtnWndProc;                    // 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 = BTNWND;                      // Window class name

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

    return 0;
}

//======================================================================
// Message handling procedures for BtnWindow
//----------------------------------------------------------------------
// BtnWndWndProc - Callback function for application window
//
LRESULT CALLBACK BtnWndProc (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(BtnWndMessages); i++) {
        if (wMsg == BtnWndMessages[i].Code)
            return (*BtnWndMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
// DoCreateBtnWnd - Process WM_CREATE message for window.
//
```

Page 00211

```
LRESULT DoCreateBtnWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    INT i;

    for (i = 0; i < dim(Btns); i++) {

        CreateWindow (Btns[i].szClass, Btns[i].szTitle,
                      Btns[i].lStyle | WS_VISIBLE | WS_CHILD,
                      Btns[i].x, Btns[i].y, Btns[i].cx, Btns[i].cy,
                      hWnd, (HMENU) Btns[i].nID, hInst, NULL);
    }
    hIcon = LoadIcon (hInst, TEXT ("TEXTICON"));

    // We need to set the initial state of the radio buttons.
    CheckRadioButton (hWnd, IDC_RADIO1, IDC_RADIO2, IDC_RADIO1);
    return 0;
}
//----------------------------------------------------------------------
// DoCtlColorBtnWnd - process WM_CTLCOLORxx messages for window.
//
LRESULT DoCtlColorBtnWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                          LPARAM lParam) {
    return GetStockObject (WHITE_BRUSH);
}
//----------------------------------------------------------------------
// DoCommandBtnWnd - Process WM_COMMAND message for window.
//
LRESULT DoCommandBtnWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                         LPARAM lParam) {
    TCHAR szOut[128];
    INT i;

    // Since the Check Box button is not an auto check box, it
    // must be set manually.
    if ((LOWORD (wParam) == IDC_CHKBOX) &&
        (HIWORD (wParam) == BN_CLICKED)) {
        // Get the current state, complement, and set.
        i = SendDlgItemMessage (hWnd, IDC_CHKBOX, BM_GETCHECK, 0, 0);
        if (i == 0)
            SendDlgItemMessage (hWnd, IDC_CHKBOX, BM_SETCHECK, 1, 0);
        else
            SendDlgItemMessage (hWnd, IDC_CHKBOX, BM_SETCHECK, 0, 0);
    }

    // Report WM_COMMAND messages to main window.
```

(continued)

Figure 4-5. *continued*

```
    for (i = 0; i < dim(nlBtn); i++) {
        if (HIWORD (wParam) == nlBtn[i].wNotification) {
            lstrcpy (szOut, nlBtn[i].pszLabel);
            break;
        }
    }
    if (i == dim(nlBtn))
        wsprintf (szOut, TEXT ("notification: %x"), HIWORD (wParam));

    SendMessage (GetParent (hWnd), MYMSG_ADDLINE, wParam,
                 (LPARAM)szOut);
    return 0;
}
//----------------------------------------------------------------------
// DoDrawItemBtnWnd - Process WM_DRAWITEM message for window.
//
LRESULT DoDrawItemBtnWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                          LPARAM lParam) {

    return DrawButton (hWnd, (LPDRAWITEMSTRUCT)lParam);
}

//----------------------------------------------------------------------
// DrawButton - Draws an owner-draw button
//
LRESULT DrawButton (HWND hWnd, LPDRAWITEMSTRUCT pdi) {

    HPEN hPenShadow, hPenLight, hPenDkShadow, hOldPen;

    HBRUSH hBr, hOldBr;
    LOGPEN lpen;
    TCHAR szOut[128];
    POINT ptOut[3], ptIn[3];

    // Reflect the messages to the report window.
    wsprintf (szOut, TEXT ("WM_DRAWITEM  Action:%x  State:%x"),
              pdi->itemAction, pdi->itemState);
    SendMessage (GetParent (hWnd), MYMSG_ADDLINE, pdi->CtlID,
                 (LPARAM)szOut);

    // Create pens for drawing.
    lpen.lopnStyle = PS_SOLID;
    lpen.lopnWidth.x = 3;
    lpen.lopnWidth.y = 3;
    lpen.lopnColor = GetSysColor (COLOR_3DSHADOW);
    hPenShadow = CreatePenIndirect (&lpen);
```

```
lpen.lopnWidth.x = 1;
lpen.lopnWidth.y = 1;
lpen.lopnColor = GetSysColor (COLOR_3DLIGHT);
hPenLight = CreatePenIndirect (&lpen);

lpen.lopnColor = GetSysColor (COLOR_3DDKSHADOW);
hPenDkShadow = CreatePenIndirect (&lpen);

// Create a brush for the face of the button.
hBr = CreateSolidBrush (GetSysColor (COLOR_3DFACE));

// Draw a rectangle with a thick outside border to start the
// frame drawing.
hOldPen = SelectObject (pdi->hDC, hPenShadow);
hOldBr = SelectObject (pdi->hDC, hBr);
Rectangle (pdi->hDC, pdi->rcItem.left, pdi->rcItem.top,
           pdi->rcItem.right, pdi->rcItem.bottom);

// Draw the upper left inside line.
ptIn[0].x = pdi->rcItem.left + 1;
ptIn[0].y = pdi->rcItem.bottom - 2;
ptIn[1].x = pdi->rcItem.left + 1;
ptIn[1].y = pdi->rcItem.top + 1;
ptIn[2].x = pdi->rcItem.right - 2;
ptIn[2].y = pdi->rcItem.top+1;

// Select a pen to draw shadow or light side of button.
if (pdi->itemState & ODS_SELECTED) {
    SelectObject (pdi->hDC, hPenDkShadow);
} else {
    SelectObject (pdi->hDC, hPenLight);
}
Polyline (pdi->hDC, ptIn, 3);

// If selected, also draw a bright line inside the lower
// right corner.
if (pdi->itemState & ODS_SELECTED) {
    SelectObject (pdi->hDC, hPenLight);
    ptIn[1].x = pdi->rcItem.right- 2;
    ptIn[1].y = pdi->rcItem.bottom - 2;
    Polyline (pdi->hDC, ptIn, 3);
}

// Now draw the black outside line on either the upper left or lower
// right corner.
ptOut[0].x = pdi->rcItem.left;
ptOut[0].y = pdi->rcItem.bottom-1;
```

(continued)

Figure 4-5. *continued*

```
    ptOut[2].x = pdi->rcItem.right-1;
    ptOut[2].y = pdi->rcItem.top;

    SelectObject (pdi->hDC, hPenDkShadow);
    if (pdi->itemState & ODS_SELECTED) {
        ptOut[1].x = pdi->rcItem.left;
        ptOut[1].y = pdi->rcItem.top;
    }  else {
        ptOut[1].x = pdi->rcItem.right-1;
        ptOut[1].y = pdi->rcItem.bottom-1;
    }
    Polyline (pdi->hDC, ptOut, 3);

    // Draw the icon.
    if (hIcon) {
        ptIn[0].x = (pdi->rcItem.right - pdi->rcItem.left)/2 -
                     GetSystemMetrics (SM_CXICON)/2 - 2;
        ptIn[0].y = (pdi->rcItem.bottom - pdi->rcItem.top)/2 -
                     GetSystemMetrics (SM_CYICON)/2 - 2;
        // If pressed, shift image down one pel to simulate depress.
        if (pdi->itemState & ODS_SELECTED) {
            ptOut[1].x += 2;
            ptOut[1].y += 2;
        }
        DrawIcon (pdi->hDC, ptIn[0].x, ptIn[0].y, hIcon);
    }

    // If button has the focus, draw the dotted rect inside the button.
    if (pdi->itemState & ODS_FOCUS) {
        pdi->rcItem.left += 3;
        pdi->rcItem.top += 3;
        pdi->rcItem.right -= 4;
        pdi->rcItem.bottom -= 4;
        DrawFocusRect (pdi->hDC, &pdi->rcItem);
    }

    // Clean up. First select the original brush and pen into the DC.
    SelectObject (pdi->hDC, hOldBr);
    SelectObject (pdi->hDC, hOldPen);

    // Now delete the brushes and pens created.
    DeleteObject (hBr);
    DeleteObject (hPenShadow);
    DeleteObject (hPenDkShadow);
    DeleteObject (hPenLight);
    return 0;
}
```

EditWnd.c

```
//======================================================================
// EditWnd - Edit control window code
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                 // For all that Windows stuff
#include "Ctlview.h"                 // Program-specific stuff

extern HINSTANCE hInst;
//----------------------------------------------------------------------
// Global data
//
// Message dispatch table for EditWndWindowProc
const struct decodeUINT EditWndMessages[] = {
    WM_CREATE, DoCreateEditWnd,
    WM_COMMAND, DoCommandEditWnd,
};

// Structure defining the controls in the window
CTLWNDSTRUCT  Edits[] = {
    {TEXT ("edit"), IDC_SINGLELINE, TEXT ("Single line edit control"),
     10,  10, 130,  23, ES_AUTOHSCROLL},

    {TEXT ("edit"), IDC_MULTILINE, TEXT ("Multi line edit control"),
     10,  35, 130,  90, ES_MULTILINE | ES_AUTOVSCROLL},

    {TEXT ("edit"), IDC_PASSBOX, TEXT (""),
     10, 127, 130,  23, ES_PASSWORD},
};
// Structure labeling the edit control WM_COMMAND notifications
NOTELABELS nlEdit[] = {{TEXT ("EN_SETFOCUS "), 0x0100},
                       {TEXT ("EN_KILLFOCUS"), 0x0200},
                       {TEXT ("EN_CHANGE   "), 0x0300},
                       {TEXT ("EN_UPDATE   "), 0x0400},
                       {TEXT ("EN_ERRSPACE "), 0x0500},
                       {TEXT ("EN_MAXTEXT  "), 0x0501},
                       {TEXT ("EN_HSCROLL  "), 0x0601},
                       {TEXT ("EN_VSCROLL  "), 0x0602},
};
//----------------------------------------------------------------------
// InitEditWnd - EditWnd window initialization
//
```

(continued)

Figure 4-5. *continued*

```
int InitEditWnd (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application EditWnd window class.
    wc.style = 0;                                // Window style
    wc.lpfnWndProc = EditWndProc;                // 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 = EDITWND;                  // Window class name

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

    return 0;
}
//======================================================================
// Message handling procedures for EditWindow
//----------------------------------------------------------------------
// EditWndWndProc - Callback function for application window
//
LRESULT CALLBACK EditWndProc (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(EditWndMessages); i++) {
        if (wMsg == EditWndMessages[i].Code)
            return (*EditWndMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
// DoCreateEditWnd - Process WM_CREATE message for window.
//
LRESULT DoCreateEditWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                         LPARAM lParam) {
    INT i;

    for (i = 0; i < dim(Edits); i++) {
        CreateWindow (Edits[i].szClass, Edits[i].szTitle,
                      Edits[i].lStyle | WS_VISIBLE | WS_CHILD | WS_BORDER,
                      Edits[i].x, Edits[i].y, Edits[i].cx, Edits[i].cy,
                      hWnd, (HMENU) Edits[i].nID, hInst, NULL);
    }
    return 0;
}
//----------------------------------------------------------------------
// DoCommandEditWnd - Process WM_COMMAND message for window.
//
LRESULT DoCommandEditWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                          LPARAM lParam) {
    TCHAR szOut[128];
    INT i;

    for (i = 0; i < dim(nlEdit); i++) {
        if (HIWORD (wParam) == nlEdit[i].wNotification) {
            lstrcpy (szOut, nlEdit[i].pszLabel);
            break;
        }
    }

    if (i == dim(nlEdit))
        wsprintf (szOut, TEXT ("notification: %x"), HIWORD (wParam));

    SendMessage (GetParent (hWnd), MYMSG_ADDLINE, wParam,
                 (LPARAM)szOut);
    return 0;
}
```

ListWnd.c

```
//======================================================================
// ListWnd - List box control window code
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                 // For all that Windows stuff
#include "Ctlview.h"                 // Program-specific stuff

extern HINSTANCE hInst;
//----------------------------------------------------------------------
// Global data
//
```

(continued)

Figure 4-5. *continued*

```
// Message dispatch table for ListWndWindowProc
const struct decodeUINT ListWndMessages[] = {
    WM_CREATE, DoCreateListWnd,
    WM_COMMAND, DoCommandListWnd,
};

// Structure defining the controls in the window
CTLWNDSTRUCT  Lists[] = {
    {TEXT ("combobox"), IDC_COMBOBOX, TEXT (""), 10,  10, 170, 100,
     WS_VSCROLL},

    {TEXT ("Listbox"), IDC_SNGLELIST, TEXT (""),   10,  35, 100, 120,
     WS_VSCROLL | LBS_NOTIFY},

    {TEXT ("Listbox"), IDC_MULTILIST, TEXT (""), 115,  35, 100, 120,
     WS_VSCROLL | LBS_EXTENDEDSEL | LBS_NOTIFY}
};
// Structure labeling the list box control WM_COMMAND notifications
NOTELABELS nlList[] = {{TEXT ("LBN_ERRSPACE "), (-2)},
                       {TEXT ("LBN_SELCHANGE"), 1},
                       {TEXT ("LBN_DBLCLK   "), 2},
                       {TEXT ("LBN_SELCANCEL"), 3},
                       {TEXT ("LBN_SETFOCUS "), 4},
                       {TEXT ("LBN_KILLFOCUS"), 5},
};
// Structure labeling the combo box control WM_COMMAND notifications
NOTELABELS nlCombo[] = {{TEXT ("CBN_ERRSPACE    "), (-1)},
                        {TEXT ("CBN_SELCHANGE   "), 1},
                        {TEXT ("CBN_DBLCLK      "), 2},
                        {TEXT ("CBN_SETFOCUS    "), 3},
                        {TEXT ("CBN_KILLFOCUS   "), 4},
                        {TEXT ("CBN_EDITCHANGE  "), 5},
                        {TEXT ("CBN_EDITUPDATE  "), 6},
                        {TEXT ("CBN_DROPDOWN    "), 7},
                        {TEXT ("CBN_CLOSEUP     "), 8},
                        {TEXT ("CBN_SELENDOK    "), 9},
                        {TEXT ("CBN_SELENDCANCEL"), 10},
};
//----------------------------------------------------------------------
// InitListWnd - ListWnd window initialization
//
int InitListWnd (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application ListWnd window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = ListWndProc;               // 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 = LISTWND;                 // Window class name

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

    return 0;
}
//======================================================================
// Message handling procedures for ListWindow
//----------------------------------------------------------------------
// ListWndProc - Callback function for application window
//
LRESULT CALLBACK ListWndProc (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(ListWndMessages); i++) {
        if (wMsg == ListWndMessages[i].Code)
            return (*ListWndMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
// DoCreateListWnd - Process WM_CREATE message for window.
//
LRESULT DoCreateListWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                         LPARAM lParam) {
    INT i;
    TCHAR szOut[64];

    for (i = 0; i < dim(Lists); i++) {

        CreateWindow (Lists[i].szClass, Lists[i].szTitle,
                      Lists[i].lStyle | WS_VISIBLE | WS_CHILD | WS_BORDER,
                      Lists[i].x, Lists[i].y, Lists[i].cx, Lists[i].cy,
                      hWnd, (HMENU) Lists[i].nID, hInst, NULL);
    }
```

(continued)

Figure 4-5. *continued*

```
    for (i = 0; i < 20; i++) {
        wsprintf (szOut, TEXT ("Item %d"), i);
        SendDlgItemMessage (hWnd, IDC_SNGLELIST, LB_ADDSTRING, 0,
                            (LPARAM)szOut);

        SendDlgItemMessage (hWnd, IDC_MULTILIST, LB_ADDSTRING, 0,
                            (LPARAM)szOut);

        SendDlgItemMessage (hWnd, IDC_COMBOBOX, CB_ADDSTRING, 0,
                            (LPARAM)szOut);
    }
    // Set initial selection.
    SendDlgItemMessage (hWnd, IDC_COMBOBOX, CB_SETCURSEL, 0, 0);
    return 0;
}
//----------------------------------------------------------------------
// DoCommandListWnd - Process WM_COMMAND message for window.
//
LRESULT DoCommandListWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                          LPARAM lParam) {
    TCHAR szOut[128];
    INT i;

    if (LOWORD (wParam) == IDC_COMBOBOX) {
        for (i = 0; i < dim(nlCombo); i++) {
            if (HIWORD (wParam) == nlCombo[i].wNotification) {
                lstrcpy (szOut, nlCombo[i].pszLabel);
                break;
            }
        }
        if (i == dim(nlList))
            wsprintf (szOut, TEXT ("notification: %x"), HIWORD (wParam));
    } else {
        for (i = 0; i < dim(nlList); i++) {
            if (HIWORD (wParam) == nlList[i].wNotification) {
                lstrcpy (szOut, nlList[i].pszLabel);
                break;
            }
        }
        if (i == dim(nlList))
            wsprintf (szOut, TEXT ("notification: %x"), HIWORD (wParam));
    }
    SendMessage (GetParent (hWnd), MYMSG_ADDLINE, wParam,
                 (LPARAM)szOut);
    return 0;
}
```

StatWnd.c

```
//======================================================================
// StatWnd - Static control window code
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                  // For all that Windows stuff
#include "Ctlview.h"                  // Program-specific stuff

extern HINSTANCE hInst;
//----------------------------------------------------------------------
// Global data
//
// Message dispatch table for StatWndWindowProc
const struct decodeUINT StatWndMessages[] = {
    WM_CREATE, DoCreateStatWnd,
    WM_COMMAND, DoCommandStatWnd,
};

// Structure defining the controls in the window
CTLWNDSTRUCT  Stats [] = {
    {TEXT ("static"), IDC_LEFTTEXT, TEXT ("Left text"),
     10,  10, 120,  23, SS_LEFT | SS_NOTIFY},

    {TEXT ("static"), IDC_RIGHTTEXT, TEXT ("Right text"),
     10,  35, 120,  23, SS_RIGHT},

    {TEXT ("static"), IDC_CENTERTEXT, TEXT ("Center text"),
     10,  60, 120,  23, SS_CENTER | WS_BORDER},

    {TEXT ("static"), IDC_ICONCTL, TEXT ("TEXTICON"),
     10,  85, 120,  23, SS_ICON},

    {TEXT ("static"), IDC_BITMAPCTL, TEXT ("STATICBMP"),
     170,  10,  44,  44, SS_BITMAP | SS_NOTIFY},
};

// Structure labeling the static control WM_COMMAND notifications
NOTELABELS nlStatic[] = {{TEXT ("STN_CLICKED"), 0},
                         {TEXT ("STN_ENABLE "), 2},
                         {TEXT ("STN_DISABLE"), 3},
};
//----------------------------------------------------------------------
// InitStatWnd - StatWnd window initialization
```

(continued)

Figure 4-5. *continued*

```
//
int InitStatWnd (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application StatWnd window class.
    wc.style = 0;                                    // Window style
    wc.lpfnWndProc = StatWndProc;                    // 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 = STATWND;                      // Window class name

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

    return 0;
}
//======================================================================
// Message handling procedures for StatWindow
//----------------------------------------------------------------------
// StatWndProc - Callback function for application window
//
LRESULT CALLBACK StatWndProc (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(StatWndMessages); i++) {
        if (wMsg == StatWndMessages[i].Code)
            return (*StatWndMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
// DoCreateStatWnd - Process WM_CREATE message for window.
//
LRESULT DoCreateStatWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                         LPARAM lParam) {
    INT i;
    for (i = 0; i < dim(Stats); i++) {

        CreateWindow (Stats[i].szClass, Stats[i].szTitle,
                      Stats[i].lStyle | WS_VISIBLE | WS_CHILD,
                      Stats[i].x, Stats[i].y, Stats[i].cx, Stats[i].cy,
                      hWnd, (HMENU) Stats[i].nID, hInst, NULL);
    }
    return 0;
}
//----------------------------------------------------------------------
// DoCommandStatWnd - Process WM_COMMAND message for window.
//
LRESULT DoCommandStatWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                          LPARAM lParam) {
    TCHAR szOut[128];
    INT i;

    for (i = 0; i < dim(nlStatic); i++) {
        if (HIWORD (wParam) == nlStatic[i].wNotification) {
            lstrcpy (szOut, nlStatic[i].pszLabel);
            break;
        }
    }
    if (i == dim(nlStatic))
        wsprintf (szOut, TEXT ("notification: %x"), HIWORD (wParam));

    SendMessage (GetParent (hWnd), MYMSG_ADDLINE, wParam,
                 (LPARAM)szOut);
    return 0;
}
```

ScrollWnd.c

```
//======================================================================
// ScrollWnd - Scroll bar control window code
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                 // For all that Windows stuff
#include "Ctlview.h"                 // Program-specific stuff

extern HINSTANCE hInst;
//----------------------------------------------------------------------
// Global data
//
```

(continued)

Figure 4-5. *continued*

```
// Message dispatch table for ScrollWndWindowProc
const struct decodeUINT ScrollWndMessages[] = {
    WM_CREATE, DoCreateScrollWnd,
    WM_HSCROLL, DoVScrollScrollWnd,
    WM_VSCROLL, DoVScrollScrollWnd,
};

// Structure defining the controls in the window
CTLWNDSTRUCT  Scrolls [] = {
    {TEXT ("Scrollbar"), IDC_LRSCROLL, TEXT (""),
     10,  10, 150,  23, SBS_HORZ},

    {TEXT ("Scrollbar"), IDC_UDSCROLL, TEXT (""),
     180,  10,  23, 150, SBS_VERT},
};

// Structure labeling the scroll bar control scroll codes for WM_VSCROLL
NOTELABELS nlVScroll[] = {{TEXT ("SB_LINEUP        "), 0},
                          {TEXT ("SB_LINEDOWN      "), 1},
                          {TEXT ("SB_PAGEUP        "), 2},
                          {TEXT ("SB_PAGEDOWN      "), 3},
                          {TEXT ("SB_THUMBPOSITION"), 4},
                          {TEXT ("SB_THUMBTRACK    "), 5},
                          {TEXT ("SB_TOP           "), 6},
                          {TEXT ("SB_BOTTOM        "), 7},
                          {TEXT ("SB_ENDSCROLL     "), 8},
};
// Structure labeling the scroll bar control scroll codes for WM_HSCROLL
NOTELABELS nlHScroll[] = {{TEXT ("SB_LINELEFT      "), 0},
                          {TEXT ("SB_LINERIGHT     "), 1},
                          {TEXT ("SB_PAGELEFT      "), 2},
                          {TEXT ("SB_PAGERIGHT     "), 3},
                          {TEXT ("SB_THUMBPOSITION"), 4},
                          {TEXT ("SB_THUMBTRACK    "), 5},
                          {TEXT ("SB_LEFT          "), 6},
                          {TEXT ("SB_RIGHT         "), 7},
                          {TEXT ("SB_ENDSCROLL     "), 8},
};
//----------------------------------------------------------------------
// InitScrollWnd - ScrollWnd window initialization
//
int InitScrollWnd (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application ScrollWnd window class.
    wc.style = 0;                             // Window style
    wc.lpfnWndProc = ScrollWndProc;           // 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 = SCROLLWND;               // Window class name

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

    return 0;
}
//======================================================================
// Message handling procedures for ScrollWindow
//----------------------------------------------------------------------
// ScrollWndProc - Callback function for application window
//
LRESULT CALLBACK ScrollWndProc (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(ScrollWndMessages); i++) {
        if (wMsg == ScrollWndMessages[i].Code)
            return (*ScrollWndMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
// DoCreateScrollWnd - Process WM_CREATE message for window.
//
LRESULT DoCreateScrollWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                           LPARAM lParam) {
    INT i;

    for (i = 0; i < dim(Scrolls); i++) {
        CreateWindow (Scrolls[i].szClass, Scrolls[i].szTitle,
                      Scrolls[i].lStyle | WS_VISIBLE | WS_CHILD,
                      Scrolls[i].x, Scrolls[i].y, Scrolls[i].cx,
                      Scrolls[i].cy,
                      hWnd, (HMENU) Scrolls[i].nID, hInst, NULL);
    }
    return 0;
}
```

(continued)

Figure 4-5. *continued*

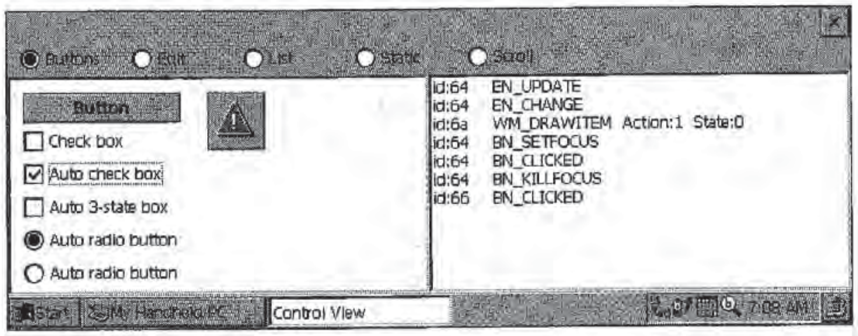
```
//----------------------------------------------------------------------
// DoVScrollScrollWnd - Process WM_VSCROLL message for window.
//
LRESULT DoVScrollScrollWnd (HWND hWnd, UINT wMsg, WPARAM wParam,
                            LPARAM lParam) {
    TCHAR szOut[128];
    SCROLLINFO si;
    INT i, sPos;

    // Update the report window.
    if (GetDlgItem (hWnd, 101) == (HWND)lParam) {

        for (i = 0; i < dim(nlVScroll); i++) {
            if (LOWORD (wParam) == nlVScroll[i].wNotification) {
                lstrcpy (szOut, nlVScroll[i].pszLabel);
                break;
            }
        }
        if (i == dim(nlVScroll))
            wsprintf (szOut, TEXT ("notification: %x"), HIWORD (wParam));
    } else {
        for (i = 0; i < dim(nlHScroll); i++) {
            if (LOWORD (wParam) == nlHScroll[i].wNotification) {
                lstrcpy (szOut, nlHScroll[i].pszLabel);
                break;
            }
        }
        if (i == dim(nlHScroll))
            wsprintf (szOut, TEXT ("notification: %x"), HIWORD (wParam));
    }
    SendMessage (GetParent (hWnd), MYMSG_ADDLINE, -1, (LPARAM)szOut);

    // Get scroll bar position.
    si.cbSize = sizeof (si);
    si.fMask = SIF_POS;
    GetScrollInfo ((HWND)lParam, SB_CTL, &si);
    sPos = si.nPos;

    // Act on the scroll code.
    switch (LOWORD (wParam)) {
    case SB_LINEUP:       // Also SB_LINELEFT
        sPos -= 2;
        break;
        case SB_LINEDOWN:      // Also SB_LINERIGHT
            sPos += 2;
            break;

        case SB_PAGEUP:        // Also SB_PAGELEFT
            sPos -= 10;
            break;

        case SB_PAGEDOWN:      // Also SB_PAGERIGHT
            sPos += 10;
            break;

        case SB_THUMBPOSITION:
            sPos = HIWORD (wParam);
            break;
        }
        // Check range.
        if (sPos < 0)
            sPos = 0;
        if (sPos > 100)
            sPos = 100;

        // Update scrollbar position.
        si.cbSize = sizeof (si);
        si.nPos = sPos;
        si.fMask = SIF_POS;
        SetScrollInfo ((HWND)lParam, SB_CTL, &si, TRUE);
        return 0;
    }
```

When the CtlView program starts, the WM_CREATE handler of the main window, *DoCreateFrame*, creates a row of radio buttons across the top of the window, a list box on the right side of the window, and five different child windows on the left side of the window. (The five child windows are all created without the WS_VISIBLE style, so they're initially hidden.) Each of the child windows in turn creates a number of controls. Before returning from the *DoCreateFrame*, CtlView checks one of the auto radio buttons and makes the BtnWnd child window (the window that contains the example button controls) visible using *ShowWindow*.

As each of the controls on the child windows are tapped, clicked, or selected, the control sends WM_COMMAND messages to its parent window. That window in turn sends the information from the WM_COMMAND message to its parent, the frame window, using the application-defined message MYMSG_ADDLINE. There the notification data is formatted and displayed in the list box on the right side of the frame window.

The other function of the frame window is to switch between the different child windows. The application accomplishes this by displaying only the child window that matches the selection of the radio buttons across the top of the frame window. The processing for this is done in the WM_COMMAND handler, *DoCommandFrame* in CtlView.c.

The best way to discover how and when these controls send notifications is to run the example program and use each of the controls. Figure 4-6 shows the CtlView window with the button controls displayed. As each of the buttons is clicked, a BN_CLICKED notification is sent to the parent window of the control. The parent window simply labels the notification and forwards it to the display list box. Because the Check Box button isn't an auto check box, CtlView must manually change the state of the check box when a user clicks it. The other check boxes and radio buttons, however, do automatically change state because they were created with the BS_AUTOCHECKBOX, BS_AUTO3STATE, and BS_AUTORADIOBUTTON styles. The square button with the exclamation mark inside a triangular icon is an owner-draw button.



```
id:64    EN_UPDATE
id:64    EN_CHANGE
id:6a    WM_DRAWITEM  Action:1  State:0
id:64    BN_SETFOCUS
id:64    BN_CLICKED
id:64    BN_KILLFOCUS
id:66    BN_CLICKED
```

Figure 4-6. *The CtlView window with the button child window displayed in the left pane.*

The source code for each child window is contained in a separate file. The source for the window containing the button controls is contained in BtnWnd.c. The file contains an initialization routine (*InitBtnWnd*) that registers the window and a window procedure (*BtnWndProc*) for the window itself. The button controls themselves are created during the WM_CREATE message using *CreateWindow*. The position, style, and other aspects of each control are contained in an array of structures named *Btns*. The *DoCreateBtnWnd* function cycles through each of the entries in the array, calling *CreateWindow* for each one. Each child window in CtlView uses a similar process to create its controls.

To support the owner-draw button, *BtnWndProc* must handle the WM_DRAWITEM message. The WM_DRAWITEM message is sent when the button needs to be drawn because it has changed state, gained or lost the focus, or because it has been uncovered. Although the *DrawButton* function (called each time a WM_DRAWITEM message is received) expends a great deal of effort to make the button look like a standard button, there's no reason a button can't have any look you want.

The other window procedures provide only basic support for their controls. The WM_COMMAND handlers simply reflect the notifications back to the main window. The ScrollWnd child window procedure, *ScrollWndProc*, handles WM_VSCROLL and WM_HSCROLL messages because that's how scroll bar controls communicate with their parent windows.

Controls and colors

Finally, a word about colors. A large number of Windows CE devices use a gray-scale display instead of a color display, including all of the first generation H/PC and Palm-size PC systems. This has made many Windows CE developers, including me, somewhat lazy in managing color in our Windows CE programs. Now that newer Windows CE systems sport color displays, we have to think a bit more.

In CtlView, the frame window class is registered in a subtly different way from the way I've registered it in previous programs. In the CtlView example, I set the background brush for the frame window using the line

```
wc.hbrBackground = (HBRUSH)GetSysColorBrush (COLOR_STATIC);
```

This sets the background color of the frame window to the same background color I used to draw the radio buttons. The function *GetSysColorBrush* returns a brush that matches the color used by the system to draw various objects in the system. In this case, the constant COLOR_STATIC is passed to *GetSysColorBrush*, which then returns the background color Windows uses when drawing static text and the text for check box and radio buttons. This makes the frame window background match the static text background.

In the window that contains the button controls, the check box and radio button background is changed to match the white background of the button window, by fielding the WM_CTLCOLORSTATIC message. This message is sent to the parent of a static control or a button control when the button is a check box or radio button to ask the parent which colors to use when drawing the control. In CtlView, the button window returns the handle to a white brush so that the control background will match the white background of the window. Modifying the color of a push button is done by fielding the WM_CTLCOLORBUTTON message. Other controls send different WM_CTLCOLOR*xxx* messages so that the colors used to draw them can be modified by the parent window.

DIALOG BOXES

The CtlView example program demonstrates a complex use of controls. While CtlView creates these controls for demonstration purposes, controls are generally used to query user input. As CtlView demonstrates, a fair amount of code is necessary for creating and placing the controls in the windows. Fortunately, you don't need this code because Windows provides a service for exactly this purpose: dialog boxes. Dialog boxes query data from the user or present data to the user, hence the term *dialog* box.

Dialog boxes are windows created by Windows using a template provided by an application. The template describes the type and placement of the controls in the window. The Dialog Manager—the part of Windows that creates and manages dialog boxes—also provides default functionality for switching focus between the controls using the Tab key as well as default actions for the Enter and Escape keys. In addition, Windows provides a default dialog box window class, freeing applications from the necessity of registering a window class for each of the dialog boxes it might create.

Dialog boxes come in two types: *modal* and *modeless*. A modal dialog prevents the user from using the application until the dialog box has been dismissed. For example, the File Open and the Print dialog boxes are modal. A modeless dialog box can be used interactively with the remainder of the application. The Find dialog box in Microsoft Pocket Word is modeless; the user doesn't need to dismiss it before typing in the main window.

Like other windows, dialog boxes have a window procedure, although the dialog box window procedure is constructed somewhat differently from standard windows procedures. Instead of passing unprocessed messages to *DefWindowProc* for default processing, a dialog box procedure returns TRUE if it processed the message and FALSE if it didn't process the message. Windows supplies a default procedure, *DefDialogProc*, for use in specific cases—that is, for specialized modeless dialog boxes that have their own window classes.

Dialog Box Resource Templates

Most of the time, the description for the size and placement of the dialog box and for the controls is provided via a resource called a *dialog template*. You can create a dialog template in memory, but unless a program has an overriding need to format the size and shape of the dialog box on the fly, loading a dialog template directly from a resource is a much better choice. As is the case for other resources such as menus, dialog templates are contained in the resource (RC) file. The template is referenced by the application using either its name or its resource ID. Here is a dialog template for a simple dialog box:

```
GetVal DIALOG discardable 10, 10, 75, 60
STYLE  WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU | DS_CENTER
EXSTYLE WS_EX_CAPTIONOKBTN
CAPTION "Enter line number"
BEGIN
    LTEXT "Enter &value:"  IDD_VALLABEL,   5,  10,  40,  12
    EDITTEXT                   IDD_VALUE,  50,  10,  20,  12, WS_TABSTOP
    AUTORADIOBUTTON "&Decimal", IDD_DEC,   5,  25,  60,  12,
                   WS_TABSTOP | WS_GROUP
    AUTORADIOBUTTON "&Hex",      IDD_HEX,   5,  40,  60,  12
END
```

The syntax for a dialog template follows a simple pattern similar to that for a menu resource. First is the name or ID of the resource followed by the keyword *DIALOG* identifying that what follows is a dialog template. The optional *discardable* keyword is followed by the position and size of the dialog box. The position specified is, by default, relative to the owner window of the dialog box.

The units of measurement in a dialog box aren't pixels but *dialog units*. A dialog unit is defined as one quarter of the average width of the characters in the system font for horizontal units and one eighth of the height of one character from the same font for vertical units. The goal is to create a unit of measurement independent of the display technology; in practice, dialog boxes still need to be tested in all display resolutions in which the box might be displayed. You can compute a pixel vs. dialog unit conversion using the *GetDialogBaseUnits* function but you'll rarely find it necessary. The visual tools that come with most compilers these days isolate a programmer from terms like *dialog units* but it's still a good idea to know just how dialog boxes are described in an RC file.

The *STYLE* line of code specifies the style flags for the dialog box. The styles include the standard window (WS_*xx*) style flags used for windows as well as a series of dialog (DS_*xx*) style flags specific to dialog boxes. Windows CE supports the following dialog box styles:

- *DS_ABSALIGN* Places the dialog box relative to the upper left corner of the screen instead of basing the position on the owner window.
- *DS_CENTER* Centers the dialog box vertically and horizontally on the screen.
- *DS_MODALFRAME* Creates a dialog box with a modal dialog box frame that can be combined with a title bar and System menu by specifying the WS_CAPTION and WS_SYSMENU styles.

- *DS_SETFONT* Tells Windows to use a nondefault font that is specified in the dialog template.
- *DS_SETFOREGROUND* Brings the dialog box to the foreground after it's created. If an application not in the foreground displays a dialog box, this style forces the dialog box to the top of the Z-order so that the user will see it.

Most dialog boxes are created with at least some combination of the WS_POPUP, WS_CAPTION, and WS_SYSMENU style flags. The WS_POPUP flag indicates the dialog box is a top-level window. The WS_CAPTION style gives the dialog box a title bar. A title bar allows the user to drag the dialog box around as well as serving as a site for title text for the dialog box. The WS_SYSMENU style causes the dialog box to have a Close button on the right end of the title bar, thus eliminating the need for a command bar control to provide the Close button. Note that Windows CE uses this flag differently from other versions of Windows, in which the flag indicates that a system menu is to be placed on the end of the title bar.

The *EXSTYLE* line of code specifies the extended style flags for the dialog box. For Windows CE, these flags are particularly important. The WS_EX_CAPTIONOKBTN flag tells the dialog manager to place an OK button on the title bar to the immediate left of the Close button. Having both OK and Close (or Cancel) buttons on the title bar saves precious space in dialog boxes that are displayed on the small screens typical of Windows CE devices. The WS_EX_CONTEXTHELP extended style places a Help button on the title bar to the immediate left of the OK button. Clicking on this button results in a WM_HELP message being sent to the dialog box procedure.

The *CAPTION* line of code specifies the title bar text of the dialog, providing that the WS_CAPTION style was specified so that the dialog box will have a title bar.

The lines describing the type and placement of the controls in the dialog box are enclosed in *BEGIN* and *END* keywords. Each control is specified either by a particular keyword, in the case of commonly used controls, or by the keyword *CONTROL,* which is a generic placeholder that can specify any window class to be placed in the dialog box. The *LTEXT* line of code on page 209 specifies a static left-justified text control. The keyword is followed by the default text for the control in quotes. The next parameter is the ID of the control, which must be unique for the dialog box. In this template, the ID is a constant defined in an include file that is included by both the resource script and the C or C++ file containing the dialog box procedure.

The next four values are the location and size of the control, in dialog units, relative to the upper left corner of the dialog box. Following that, any explicit style flags can be specified for the control. In the case of the *LTEXT* line, no style flags are necessary, but as you can see the *EDITTEXT* and first *AUTORADIOBUTTON* entries each have style flags specified. Each of the control keywords have subtly different syntax. For example, the *EDITTEXT* line doesn't have a field for default text. The style flags for the individual controls deserve notice. The edit control and the first of the two radio buttons have a WS_TABSTOP style. The dialog manager looks for controls with the WS_TABSTOP style to determine which control gets focus when the user presses the Tab. In this example, pressing the Tab key results in focus being switched between the edit control and the first radio button.

The WS_GROUP style on the first radio button starts a new group of controls. All the controls following the radio button are grouped together, up to the next control that has the WS_GROUP style. Grouping auto radio buttons allow only one radio button at a time to be selected.

Another benefit of grouping is that focus can be changed among the controls within a group by exploiting the cursor keys as well as the Tab key. The first member of a group should have a WS_TABSTOP style; this allows the user to tab to the group of controls and then use the cursor keys to switch the focus among the controls in the group.

The CONTROL statement isn't used in this example, but it's important and merits some explanation. It's a generic statement that allows inclusion of any window class in a dialog box. It has the following syntax:

```
CONTROL "text", id, class, style, x, y, width, height
    [, extended-style]
```

For this entry, the default text and control ID are similar to the other statements but the next field, *class*, is new. It specifies the window class of the control you want to place in the dialog box. The *class* field is followed by the *style* flags, then the location and size of your control. Finally, the CONTROL statement has a field for extended style flags. If you use Microsoft Developer Studio to create a dialog box and look at the resulting RC file using a text editor, you'll see that Developer Studio uses CONTROL statements instead of the more readable LTEXT, EDITTEXT, and BUTTON statements. There's no functional difference between an edit control created with a CONTROL statement and one created with an EDITTEXT statement. The CONTROL statement is a generic version of the more specific keywords. The CONTROL statement also allows inclusion of controls that don't have a special keyword associated with them.

Creating a Dialog Box

Creating and displaying a dialog box is simple; just use one of the many dialog box creation functions. The first two are these:

```
int DialogBox (HANDLE hInstance, LPCTSTR lpTemplate, HWND hWndOwner,
               DLGPROC lpDialogFunc);

int DialogBoxParam (HINSTANCE hInstance, LPCTSTR lpTemplate,
                    HWND hWndOwner, DLGPROC lpDialogFunc,
                    LPARAM dwInitParam);
```

These two functions differ only in *DialogBoxParam*'s additional *LPARAM* parameter, so I'll talk about them at the same time. The first parameter to these functions is the instance handle of the program. The second parameter specifies the name or ID of the resource containing the dialog template. As with other resources, to specify a resource ID instead of a name requires the use of the MAKEINTRESOURCE macro.

The third parameter is the handle of the window that will own the dialog box. The owning window isn't the parent of the dialog box because, were that true, the dialog box would be clipped to fit inside the parent. Ownership means instead that the dialog box will be hidden when the owner window is minimized and will always appear above the owner window in the Z-order.

The fourth parameter is a pointer to the dialog box procedure for the dialog box. I'll describe the dialog box procedure shortly. The *DialogBoxParam* function has a fifth parameter, which is a user-defined value that's passed to the dialog box procedure when the dialog box is to be initialized. This helpful value can be used to pass a pointer to a structure of data that can be referenced when your application is initializing the dialog box controls.

Two other dialog box creation functions create modal dialogs. They are the following:

```
int DialogBoxIndirect (HANDLE hInstance, LPDLGTEMPLATE lpTemplate,
                       HWND hWndParent, DLGPROC lpDialogFunc);

int DialogBoxIndirectParam (HINSTANCE hInstance,
                            LPCDLGTEMPLATE DialogTemplate, HWND hWndParent,
                            DLGPROC lpDialogFunc, LPARAM dwInitParam);
```

The difference between these two functions and the two previously described is that these two use a dialog box template in memory to define the dialog box rather than using a resource. This allows a program to dynamically create a dialog box template on the fly. The second parameter to these functions points to a DLGTEMPLATE structure, which describes the overall dialog box window, followed by an array of DLGITEMTEMPLATE structures defining the individual controls.

When any of these four functions are called, the dialog manager creates a modal dialog box using the template passed. The window that owns the dialog is disabled and the dialog manager then enters its own internal *GetMessage/DispatchMessage* message processing loop; this loop doesn't exit until the dialog box is destroyed. Because of this, these functions don't return to the caller until the dialog box has been destroyed. The WM_ENTERIDLE message that's sent to owner windows in other versions of Windows while the dialog box is displayed isn't supported under Windows CE.

If an application wanted to create a modal dialog box with the template shown above and pass a value to the dialog box procedure it might call this:

```
DialogBoxParam (hInstance, TEXT ("GetVal"), hWnd, GetValDlgProc,
                0x1234);
```

The *hInstance* and *hWnd* parameters would be the instance handle of the application and the handle of the owner window. The *GetVal* string is the name of the dialog box template while *GetValDlgProc* is the name of the dialog box procedure. Finally, *0x1234* is an application-defined value. In this case, it might be used to provide a default value in the dialog box.

Dialog Box Procedures

The final component necessary for a dialog box is the dialog box procedure. As in the case of a window procedure, the purpose of the dialog box procedure is to field messages sent to the window—in this case, a dialog box window—and perform the appropriate processing. In fact, a dialog box procedure is simply a special case of a window procedure, although we should pay attention to a few differences between the two.

The first difference, as mentioned in the previous section, is that a dialog box procedure doesn't pass unprocessed messages to *DefWindowProc*. Instead, the procedure returns TRUE for messages it processes and FALSE for messages that it doesn't process. The dialog manager uses this return value to determine whether the message needs to be passed to the default dialog box procedure.

The second difference from standard window procedures is the addition of a new message, WM_INITDIALOG. Dialog box procedures perform any initialization of the controls during the processing of this message. Also, if the dialog box was created with *DialogBoxParam* or *DialogBoxIndirectParam*, the *lParam* value is the generic parameter passed during the call that created the dialog box. While it might seem that the controls could be initialized during the WM_CREATE message, that doesn't work. The problem is that during the WM_CREATE message, the controls on the dialog box haven't yet been created, so they can't be initialized. The WM_INITDIALOG message is sent after the controls have been created and before the dialog box is made visible, which is the perfect time to initialize the controls.

Here are a few other minor differences between a window procedure and a dialog box procedure. Most dialog box procedures don't need to process the WM_PAINT message because any necessary painting is done by the controls or, in the case of owner-draw controls, in response to control requests. Most of the code in a dialog box procedure is responding to WM_COMMAND messages from the controls. As with menus, the WM_COMMAND messages are parsed by the control ID values. Two special predefined ID values that a dialog box has to deal with are IDOK and IDCANCEL. IDOK is assigned to the OK button on the title bar of the dialog box while IDCANCEL is assigned to the Close button. In response to a click of either button, a dialog box procedure should call

```
BOOL EndDialog (HWND hDlg, int nResult);
```

EndDialog closes the dialog box and returns control to the caller of whatever function created the dialog box. The *hDlg* parameter is the handle of the dialog box while the *nResult* parameter is the value that's passed back as the return value of the function that created the dialog box.

The difference, of course, between handling the IDOK and IDCANCEL buttons is that if the OK button is clicked, the dialog box procedure should collect any relevant data from the dialog box controls to return to the calling procedure before it calls *EndDialog*.

A dialog box procedure to handle the GetVal template previously described is shown here:

```
//======================================================================
// GetVal Dialog procedure
//
BOOL CALLBACK GetValDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                             LPARAM lParam) {
    TCHAR szText[64];
    INT nVal, nBase;

    switch (wMsg) {
        case WM_INITDIALOG:
            SetDlgItemInt (hWnd, IDD_VALUE, 0, TRUE);
            SendDlgItemMessage (hWnd, IDD_VALUE, EM_LIMITTEXT,
                                sizeof (szText)-1, 0);
            CheckRadioButton (hWnd, IDD_DEC, IDD_HEX, IDD_DEC);
            return TRUE;

        case WM_COMMAND:
            switch (LOWORD (wParam)) {

                case IDD_HEX:
                    // See if Hex already checked.
        if (SendDlgItemMessage (hWnd, IDD_HEX,
                BM_GETSTATE, 0, 0) == BST_CHECKED)
            return TRUE;

        // Get text from edit control.
        GetDlgItemText (hWnd, IDD_VALUE, szText,
                        sizeof (szText));
        // Convert value from decimal, then set as hex.
        if (ConvertValue (szText, 10, &nVal)) {
            // If conversion successful, set new value.
            wsprintf (szText, TEXT ("%X"), nVal);
            SetDlgItemText (hWnd, IDD_VALUE, szText);
            // Set radio button.
            CheckRadioButton (hWnd, IDD_DEC, IDD_HEX,
                              IDD_HEX);
        } else {
            MessageBox (hWnd,
                        TEXT ("Value not valid"),
                        TEXT ("Error"), MB_OK);
        }
        return TRUE;

    case IDD_DEC:
        // See if Dec already checked.
        if (SendDlgItemMessage (hWnd, IDD_DEC,
                BM_GETSTATE, 0, 0) == BST_CHECKED)
            return TRUE;

        // Get text from edit control.
        GetDlgItemText (hWnd, IDD_VALUE, szText,
                        sizeof (szText));
        // Convert value from hex, then set as decimal.
        if (ConvertValue (szText, 16, &nVal)) {
            // If conversion successful, set new value.
            wsprintf (szText, TEXT ("%d"), nVal);
            SetDlgItemText (hWnd, IDD_VALUE, szText);
            // Set radio button.
            CheckRadioButton (hWnd, IDD_DEC, IDD_HEX,
                              IDD_DEC);
        } else {
            // If bad conversion, tell user.
            MessageBox (hWnd,
                        TEXT ("Value not valid"),
                        TEXT ("Error"), MB_OK);
        }
        return TRUE;
```

(continued)

```
                case IDOK:
                    // Get the current text.
                    GetDlgItemText (hWnd, IDD_VALUE, szText,
                                    sizeof (szText));
                    // See which radio button checked.
                    if (SendDlgItemMessage (hWnd, IDD_DEC,
                            BM_GETSTATE, 0, 0) == BST_CHECKED)
                        nBase = 10;
                    else
                        nBase = 16;
                    // Convert the string to a number.
                    if (ConvertValue (szText, nBase, &nVal))
                        EndDialog (hWnd, nVal);
                    else
                        MessageBox (hWnd,
                                    TEXT ("Value not valid"),
                                    TEXT ("Error"), MB_OK);
                    break;

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

This is a typical example of a dialog box procedure for a simple dialog box. The only messages that are processed are the WM_INITDIALOG and WM_COMMAND messages. The WM_INITDIALOG message is used to initialize the edit control using a number passed, via *DialogBoxParam*, through to the *lParam* value. The radio button controls aren't auto radio buttons because the dialog box procedure needs to prevent the buttons from changing if the value in the entry field is invalid. The WM_COMMAND message is parsed by the control ID where the appropriate processing takes place. The IDOK and IDCANCEL buttons aren't in the dialog box template; as mentioned earlier, those buttons are placed by the dialog manager in the title bar of the dialog box.

Modeless Dialog Boxes

I've talked so far about modal dialog boxes that prevent the user from using other parts of the application before the dialog box is dismissed. Modeless dialog boxes, on the other hand, allow the user to work with other parts of the application while the dialog box is still open. Creating and using modeless dialog boxes requires a bit more work. For example, you create modeless dialog boxes using different functions than those for modal dialog boxes:

```
HWND CreateDialog (HINSTANCE hInstance, LPCTSTR lpTemplate,
                   HWND hWndOwner, DLGPROC lpDialogFunc);

HWND CreateDialogIndirect (HINSTANCE hInstance, LPCDLGTEMPLATE lpTemplate,
                           HWND hWndOwner, DLGPROC lpDialogFunc);

HWND CreateDialogIndirect (HINSTANCE hInstance,
                           LPCDLGTEMPLATE lpTemplate, HWND hWndOwner,
                           DLGPROC lpDialogFunc);
```

or

```
HWND CreateDialogIndirectParam (HINSTANCE hInstance,
                                LPCDLGTEMPLATE lpTemplate, HWND hWndOwner,
                                DLGPROC lpDialogFunc, LPARAM lParamInit);
```

The parameters in these functions mirror the creation functions for the modal dialog boxes with similar parameters. The difference is that these functions return immediately after creating the dialog boxes. Each function returns 0 if the create failed or returns the handle to the dialog box window if the create succeeded.

The handle returned after a successful creation is important because applications that use modeless dialog boxes must modify their message loop code to accommodate the dialog box. The new message loop should look similar to the following:

```
while (GetMessage (&msg, NULL, 0, 0)) {
    if ((hMlDlg == 0) || (!IsDialogMessage (hMlDlg, &msg))) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
}
```

The difference from a modal dialog box message loop is that if the modeless dialog box is being displayed, messages should be checked to see whether they're dialog messages. If they're not dialog messages, your application forwards them to *TranslateMessage* and *DispatchMessage*. The code shown above simply checks to see whether the dialog box exists by checking a global variable containing the handle to the modeless dialog box and, if it's not 0, calls *IsDialogMessage*. If *IsDialogMessage* doesn't translate and dispatch the message itself, the message is sent to the standard *TranslateMessage/DispatchMessage* body of the message loop. Of course, this code assumes that the handle returned by *CreateDialog* (or whatever function creates the dialog box) is saved in *hMlDlg* and that *hMlDlg* is set to 0 when the dialog box is closed.

Another difference between modal and modeless dialog boxes is in the dialog box procedure. Instead of using *EndDialog* to close the dialog box, you must call *DestroyWindow* instead. This is because *EndDialog* is designed to work only with the internal message loop processing that's performed with a modal dialog box. Finally, an application usually won't want more than one instance of a modeless dialog box displayed at a time. An easy way to prevent this is to check the global copy of the window handle to see whether it's nonzero before calling *CreateDialog*. To do this, the dialog box procedure must set the global handle to 0 after it calls *DestroyWindow*.

Property Sheets

To the user, a property sheet is a dialog box with one or more tabs across the top that allow the user to switch among different "pages" of the dialog box. To the programmer, a property sheet is a series of stacked dialog boxes. Only the top dialog box is visible; the dialog manager is responsible for displaying the dialog box associated with the tab on which the user clicks. However you approach property sheets, they're invaluable given the limited screen size of Windows CE devices.

Each page of the property sheet, named appropriately enough a *property page*, is a dialog box template, either loaded from a resource or created dynamically in memory. Each property page has its own dialog box procedure. The frame around the property sheets is maintained by the dialog manager, so the advantages of property sheets come with little overhead to the programmer. Unlike the property sheets supported in other versions of Windows, the property sheets in Windows CE don't support the Apply button. Also, the OK and Cancel buttons for the property sheet are contained in the title bar, not positioned below the pages.

Creating a property sheet

Instead of using the dialog box creation functions to create a property sheet, a new function is used:

```
int PropertySheet (LPCPROPSHEETHEADER lppsph);
```

The *PropertySheet* function creates the property sheet according to the information contained in the PROPSHEETHEADER structure which is defined as the following:

```
typedef struct _PROPSHEETHEADER {
    DWORD dwSize;
    DWORD dwFlags;
    HWND hwndOwner;
    HINSTANCE hInstance;
    union {
        HICON hIcon;
        LPCWSTR pszIcon;
    };
    LPCWSTR pszCaption;
    UINT nPages;
    union {
        UINT nStartPage;
        LPCWSTR pStartPage;
    };
    union {
        LPCPROPSHEETPAGE ppsp;
        HPROPSHEETPAGE FAR *phpage;
    };
    PFNPROPSHEETCALLBACK pfnCallback;
} PROPSHEETHEADER;
```

Filling in this convoluted structure isn't as imposing a task as it might look. The *dwSize* field is the standard size field that must be initialized with the size of the structure. The *dwFlags* field contains the creation flags that define how the property sheet is created, which fields of the structure are valid, and how the property sheet behaves. Some of the flags indicate which fields in the structure are used. (I'll talk about those flags when I describe the other fields.) Two other flags set the behavior of the property sheet. The PSH_PROPTITLE flag appends the string "Properties" to the end of the caption specified in the *pszCaption* field. The PSH_MODELESS flag causes the *PropertySheet* function to create a modeless property sheet and immediately return. A modeless property sheet is like a modeless dialog box; it allows the user to switch back to the original window while the property sheet is still being displayed.

The next two fields are the handle of the owner window and the instance handle of the application. Neither the *hIcon* or *pszIcon* fields are used in Windows CE so they should be set to 0. The *pszCaption* field should point to the title bar text for the property sheet. The *nStartPage/pStartPage* union should be set to indicate the page that should be initially displayed. This can be selected either by number or by title if the PSH_USEPSTARTPAGE flag is set in the *dwFlags* field.

The *ppsp/phpage* union points to either an array of PROPSHEETPAGE structures describing each of the property pages or handles to previously created property pages. For either of these, the *nPages* field must be set to the number of entries of the array of structures or page handles. To indicate that the pointer points to an array of PROPSHEETPAGE structures, set the PSH_PROPSHEETPAGE flag in the *dwFlags* field. I'll describe both the structure and how to create individual pages shortly.

The *pfnCallBack* field is an optional pointer to a procedure that's called twice—when the property sheet is about to be created and again when it's about to be initialized. The callback function allows applications to fine-tune the appearance of the property sheet for the rare times when it's necessary. This field is ignored unless the PSP_USECALLBACK flag is set in the *dwFlags* field.

Creating a property page

As I mentioned earlier, individual property pages can be specified by an array of PROPSHEETPAGE structures or an array of handles to existing property pages. Creating a property page is accomplished with a call to the following:

```
HPROPSHEETPAGE CreatePropertySheetPage (LPCPROPSHEETPAGE lppsp);
```

This function is passed a pointer to the same PROPSHEETPAGE structure and returns a handle to a property page. PROPSHEETPAGE is defined as this:

```
typedef struct _PROPSHEETPAGE {
    DWORD dwSize;
    DWORD dwFlags;
    HINSTANCE hInstance;
    union {
        LPCSTR pszTemplate;
        LPCDLGTEMPLATE pResource;
    };
    union {
        HICON hIcon;
        LPCSTR pszIcon;
    };
    LPCSTR pszTitle;
    DLGPROC pfnDlgProc;
    LPARAM lParam;
    LPFNPSPCALLBACK pfnCallback;
    UINT FAR * pcRefParent;
} PROPSHEETPAGE;
```

The structure looks similar to the PROPSHEETHEADER structure, leading with a *dwSize* and *dwFlags* field followed by an *hInstance* field. In this structure, *hInstance* is the handle of the module from which the resources will be loaded. The *dwFlags* field again specifies which fields of the structure are used and how they're used, as well as a few flags specifying the characteristics of the page itself.

The *pszTemplate/pResource* union specifies the dialog box template used to define the page. If the PSP_DLGINDIRECT flag is set in the *dwFlags* field, the union points to a dialog box template in memory. Otherwise, the field specifies the name of a dialog box resource. The *hIcon/pszIcon* union isn't used in Windows CE and should be set to 0. If the *dwFlags* field contains a PSP_USETITLE flag, the *pszTitle* field points to the text used on the tab for the page. Otherwise, the tab text is taken from the caption field in the dialog box template. The *pfnDlgProc* field points to the dialog box procedure for this specific page and the *lParam* field is an application-defined parameter that can be used to pass data to the dialog box procedure. The *pfnCallback* field can point to a callback procedure that's called twice— when the page is about to be created and when it's about to be destroyed. Again, like the callback for the property sheet, the property page callback allows applications to fine-tune the page characteristics. This field is ignored unless the *dwFlags* field contains the PSP_USECALLBACK flag. Finally, the *pcRefCount* field can contain a pointer to an integer that will store a reference count for the page. This field is ignored unless the flags field contains the PSP_USEREFPARENT flag.

Windows CE supports a new flag for property pages, PSP_PREMATURE. This flag causes a property page to be created when the property sheet that owns it is created. Normally, a property page isn't created until the first time it's shown. This has an impact on property pages that communicate and cooperate with each other. Without the PSP_PREMATURE flag, the only property page that's automatically created when the property sheet is created is the page that is displayed first. So, at that moment, that first page has no sibling pages to communicate with. Using the PSP_PREMATURE flag, you can ensure that a page is created when the property sheet is created even though it isn't the first page in the sheet. While it's easy to get overwhelmed with all these structures, simply using the default values and not using the optional fields results in a powerful and easily maintainable property sheet that's also as easy to construct as a set of individual dialog boxes.

Once a property sheet has been created, the application can add and delete pages. The application adds a page by sending a PSM_ADDPAGE message to the property sheet window. The message must contain the handle of a previously created property page in *lParam*; *wParam* isn't used. Likewise, the application can remove a page by sending a PSM_REMOVEPAGE message to the property sheet window. The application specifies a page for deletion either by setting *wParam* to the zero-based index of the page selected for removal or by passing the handle to that page in *lParam*.

The code below creates a simple property sheet with three pages. Each of the pages references a dialog box template resource. As you can see, most of the initialization of the structures can be performed in a fairly mechanical fashion.

```
PROPSHEETHEADER psh;
PROPSHEETPAGE psp[3];
INT i;
```

(continued)

```
// Init page structures with generic information.
memset (&psp, 0, sizeof (psp));      // Zero out all unused values.
for (i = 0; i < dim(psp); i++) {
    psp[i].dwSize = sizeof (PROPSHEETPAGE);
    psp[i].dwFlags = PSP_DEFAULT;     // No special processing needed
    psp[i].hInstance = hInst;         // Instance handle where the
}                                     // dialog templates are located
// Now do the page specific stuff.
psp[0].pszTemplate = TEXT ("Page1"); // Name of dialog resource for page 1
psp[0].pfnDlgProc = Page1DlgProc;    // Pointer to dialog proc for page 1

psp[1].pszTemplate = TEXT ("Page2"); // Name of dialog resource for page 2
psp[1].pfnDlgProc = Page2DlgProc;    // Pointer to dialog proc for page 2

psp[2].pszTemplate = TEXT ("Page3"); // Name of dialog resource for page 3
psp[2].pfnDlgProc = Page3DlgProc;    // Pointer to dialog proc for page 3

// Init property sheet header structure.
psh.dwSize = sizeof (PROPSHEETHEADER);
psh.dwFlags = PSH_PROPSHEETPAGE;    // We are using templates not handles.
psh.hwndParent = hWnd;              // Handle of the owner window
psh.hInstance = hInst;              // Instance handle of the application
psh.pszCaption = TEXT ("Property sheet title");
psh.nPages = dim(psp);              // Number of pages
psh.nStartPage = 0;                 // Index of page to be shown first
psh.ppsp = psp;                     // Pointer to page structures
psh.pfnCallback = 0;                // We don't need a callback procedure.

// Create property sheet.  This returns when the user dismisses the sheet
// by tapping OK or the Close button.
i = PropertySheet (&psh);
```

While this fragment has a fair amount of structure filling, it's boilerplate code. Everything not defined, such as the page dialog box resource templates and the page dialog box procedures, are required for dialog boxes as well as property sheets. So, aside from the boilerplate stuff, property sheets require little, if any, work beyond simple dialog boxes.

Property page procedures

The procedures that back up each of the property pages have only a few differences from standard dialog box procedures. First, as I mentioned previously, unless the PSP_PREMATURE flag is used, pages aren't created immediately when the property sheet is created. Instead, each page is created and WM_INITDIALOG messages are sent only when the page is initially shown. Also, the *lParam* parameter doesn't point to a user-defined parameter; instead, it points to the PROPSHEETPAGE structure that defined the page. Of course, that structure contains a user-definable value that can be used to pass data to the dialog box procedure.

Also, a property sheet procedure doesn't field the IDOK and IDCANCEL control IDs for the OK and Close buttons on a standard dialog box. These buttons instead are handled by the system-provided property sheet procedure that coordinates the display and management of each page. When the OK or Close button is tapped, the property sheet sends a WM_NOTIFY message to each sheet notifying them that one of the two buttons has been tapped and that they should acknowledge that it's okay to close the property sheet.

WM_NOTIFY

While this is the first time I've mentioned the WM_NOTIFY message, it has become a mainstay of the new common controls added to Windows over the last few years. The WM_NOTIFY message is essentially a redefined WM_COMMAND message, which instead of encoding the reason for the message in one of the parameters passes a pointer to an extensible structure instead. This has allowed the WM_NOTIFY message to be extended and adapted for each of the controls that use it. In the case of property sheets, the WM_NOTIFY message is sent under a number of conditions; when the user taps the OK button, when the user taps the Close button, when the page gains or loses focus from or to another page, or when the user requests help.

At a minimum, the WM_NOTIFY message is sent with *lParam* pointing to an NMHDR structure defined as the following:

```
typedef struct tagNMHDR {
    HWND hwndFrom;
    UINT idFrom;
    UINT code;
} NMHDR;
```

The *hwndFrom* field contains the handle of the window that sent the notify message. For property sheets, this is the property sheet window. The *idFrom* field contains the ID of the control if a control is sending the notification. Finally, the *code* field contains the notification code. While this basic structure doesn't contain any more information than the WM_COMMAND message, often this structure is extended with additional fields appended to the structure. The notification code then indicates what, if any, additional fields are appended to the notification structure.

Switching pages

When a user switches from one page to the next, the Dialog Manager sends a WM_NOTIFY message with the code PSN_KILLACTIVE to the page currently being displayed. The dialog box procedure should then validate the data on the page. If it's permissible for the user to change the page, the dialog box procedure should then set the return value of the window structure of the page to PSNRET_NOERROR and return TRUE. You set the PSNRET_NOERROR return field by calling *SetWindowLong* with DWL_MSGRESULT as in the following line of code:

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

where *hwndPage* is the handle of the property sheet page. A page can keep focus by returning PSNRET_INVALID_NOCHANGEPAGE in the return field. Assuming a page has indicated that it's okay to lose focus, the page being switched to receives a PSN_SETACTIVE notification via a WM_NOTIFY message. The page can then accept the focus or specify another page that should receive the focus.

Closing a property sheet

When the user taps on the OK button, the property sheet procedure sends a WM_NOTIFY with the notification code PSN_KILLACTIVE to the page currently being displayed followed by a WM_NOTIFY with the notification code PSN_APPLY to each of the pages that has been created. Each page procedure should save any data from the page controls when it receives the PSN_APPLY notification code.

When the user clicks the Close button, a PSN_QUERYCANCEL notification is sent to the page procedure of the page currently being displayed. All this notification requires is that the page procedure return TRUE to prevent the close or FALSE to allow the close. A further notification, PSN_RESET, is then sent to all the pages that have been created, indicating that the property sheet is about to be destroyed.

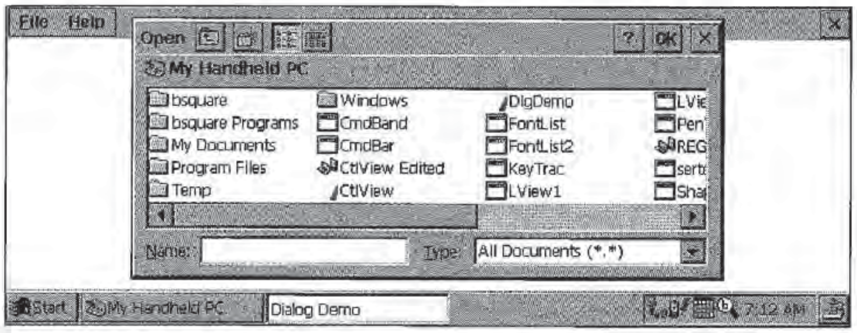
Common Dialogs

In the early days of Windows, it was a rite of passage for a Windows developer to write his or her own File Open dialog box. A File Open dialog box is complex—it must display a list of the possible files from a specific directory, allow file navigation, and return a fully justified filename back to the application. While it was great for programmers to swap stories about how they struggled with their unique implementation of a File Open dialog, it was hard on the users. Users had to learn a different file open interface for every Windows application.

Windows now provides a set of common dialog boxes that perform typical functions, such as selecting a filename to open or save or picking a color. These standard dialog boxes (called *common dialogs*) serve two purposes. First, common dialogs lift from developers the burden of having to create these dialog boxes from scratch. Second, and just as important, common dialogs provide a common interface to the user across different applications. (These days, Windows programmers swap horror stories about learning COM.)

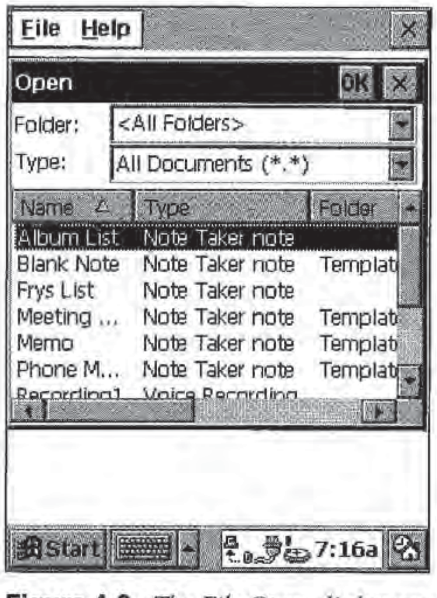
Windows CE 2.0 provides four common dialogs: File Open, Save As, Print, and Choose Color. Common dialogs, such as Find, Choose Font, and Page Setup, that are available under other versions of Windows aren't supported under Windows CE. Applications developed for Windows CE 1.0 or for the first release of the Palm-size PC must also do without the Print and Color common dialogs, but this isn't much of a sacrifice because neither color screens nor printing is supported on those systems.

The other advantage of the common dialogs is that they have a customized look for each platform while retaining the same programming interface. This makes it easy to use, say, the File Open dialog on both the H/PC and the Palm-size PC because the dialog box has the same interface on both systems even though the look of the dialog box is vastly different on the two platforms. Figure 4-7 shows the File Open dialog on the H/PC; Figure 4-8 shows the File Open dialog box on the Palm-size PC.



Name:
Type: All Documents (*.*)

Figure 4-7. *The File Open dialog on a Handheld PC.*



Folder: <All Folders>
Type: All Documents (*.*)

```
Name        Type              Folder
Album List  Note Taker note
Blank Note  Note Taker note   Templat
Frys List   Note Taker note
Meeting ... Note Taker note   Templat
Memo        Note Taker note   Templat
Phone M...  Note Taker note   Templat
Recording1  Voice Recording
```

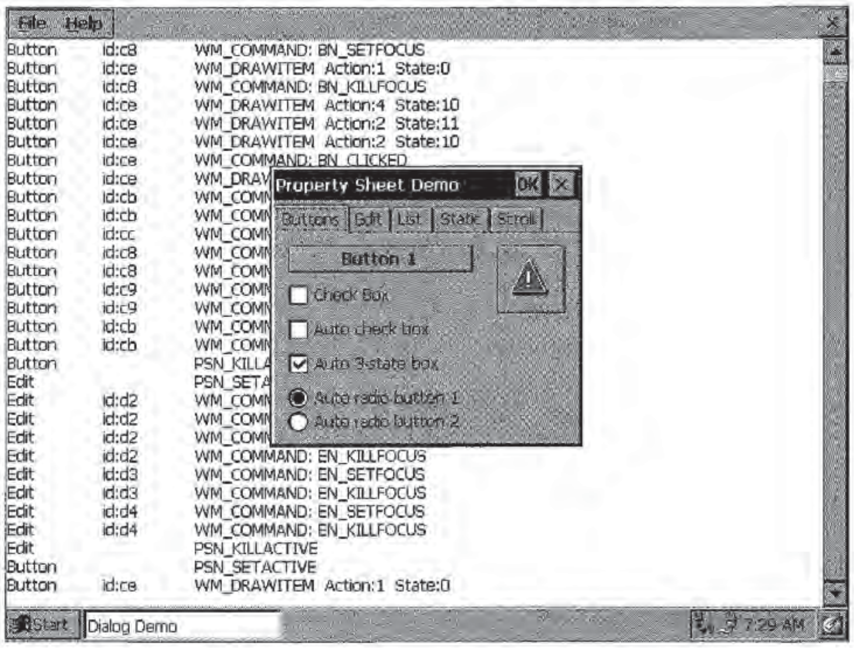
Figure 4-8. *The File Open dialog on a Palm-size PC.*

Page 00248

Instead of showing you how to use the common dialogs here, I'll let the next example program, DlgDemo, show you. That program demonstrates all four supported common dialog boxes.

The DlgDemo Example Program

The DlgDemo program demonstrates basic dialog boxes, modeless dialog boxes, property sheets, and common dialogs. When you start DlgDemo, it displays a window that shows the WM_COMMAND and WM_NOTIFY messages sent by the various controls in the dialogs, similar to the right side of the CtlView window. The different dialogs can be opened using the various menu items. Figure 4-9 shows the Dialog Demo window with the property sheet dialog displayed.



```
Button      id:c8      WM_COMMAND: BN_SETFOCUS
Button      id:ce      WM_DRAWITEM  Action:1  State:0
Button      id:c8      WM_COMMAND: BN_KILLFOCUS
Button      id:ce      WM_DRAWITEM  Action:4  State:10
Button      id:ce      WM_DRAWITEM  Action:2  State:11
Button      id:ce      WM_DRAWITEM  Action:2  State:10
Button      id:ce      WM_COMMAND: BN_CLICKED
Button      id:ce      WM_DRAW
Button      id:cb      WM_COMM
Button      id:cb      WM_COMM
Button      id:cc      WM_COMM
Button      id:c8      WM_COMM
Button      id:c8      WM_COMM
Button      id:c9      WM_COMM
Button      id:c9      WM_COMM
Button      id:cb      WM_COMM
Button      id:cb      WM_COMM
Button                 PSN_KILLA
Edit                   PSN_SETA
Edit        id:d2      WM_COMM
Edit        id:d2      WM_COMM
Edit        id:d2      WM_COMM
Edit        id:d2      WM_COMMAND: EN_KILLFOCUS
Edit        id:d3      WM_COMMAND: EN_SETFOCUS
Edit        id:d3      WM_COMMAND: EN_KILLFOCUS
Edit        id:d4      WM_COMMAND: EN_SETFOCUS
Edit        id:d4      WM_COMMAND: EN_KILLFOCUS
Edit                   PSN_KILLACTIVE
Button                 PSN_SETACTIVE
Button      id:ce      WM_DRAWITEM  Action:1  State:0
```

Figure 4-9. *The DlgDemo window.*

The basic dialog box is a simple "about box" launched by selecting the Help About menu. The property sheet is launched by selecting the File Property Sheet menu. The property sheet dialog contains five pages corresponding to the different windows in the CtlView example. The common dialog boxes are launched from the File Open, File Save, File Color, and File Print menu items. These last two menu items are disabled when the program is run on a Palm-size PC since those common dialog boxes aren't supported on that platform. The DlgDemo source code is shown in Figure 4-10.

DlgDemo.rc

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

#include "windows.h"                    //
#include "DlgDemo.h"                    // Program-specific stuff

//----------------------------------------------------------------------
// Icons and bitmaps
//
ID_ICON      ICON   "DlgDemo.ico"    // Program icon
IDI_BTNICON  ICON   "btnicon.ico"    // Bitmap used in owner-draw button
statbmp      BITMAP "statbmp.bmp"    // Bitmap used in static window

//----------------------------------------------------------------------
// Menu
//
ID_MENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "Open...",                      IDM_OPEN
        MENUITEM "Save...",                      IDM_SAVE
        MENUITEM SEPARATOR
        MENUITEM "Color...",                     IDM_COLOR
        MENUITEM "Print...",                     IDM_PRINT
        MENUITEM SEPARATOR
        MENUITEM "Property Sheet",               IDM_SHOWPROPSHEET
        MENUITEM "Modeless Dialog",              IDM_SHOWMODELESS
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                        IDM_EXIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...",                    IDM_ABOUT
    END
END

//----------------------------------------------------------------------
// Property page templates
//
```

Figure 4-10. *The DlgDemo program.* *(continued)*

Figure 4-10. *continued*

```
ID_BTNPAGE DIALOG discardable 0, 0, 125, 90
CAPTION "Buttons"
BEGIN
    PUSHBUTTON "Button 1",            IDC_PUSHBTN,   5,  5,  80,  12,
                                          WS_TABSTOP | BS_NOTIFY
    CHECKBOX "Check Box",             IDC_CHKBOX,  5,  20,  80,  12,
                                          WS_TABSTOP | BS_NOTIFY
    AUTOCHECKBOX "Auto check box"     IDC_ACHKBOX,   5,  35,  80,  12,
                                          WS_TABSTOP
    AUTO3STATE "Auto 3-state box",    IDC_A3STBOX,   5,  50,  80,  12,
                                          WS_TABSTOP
    AUTORADIOBUTTON "Auto radio button 1",
                                      IDC_RADIO1,    5,  65,  80,  12,
                                          WS_TABSTOP | WS_GROUP
    AUTORADIOBUTTON "Auto radio button 2",
                                      IDC_RADIO2,    5,  75,  80,  12
    PUSHBUTTON "",                    IDC_OWNRDRAW, 95,   5,  30,  30,
                                          BS_OWNERDRAW
END

ID_EDITPAGE DIALOG discardable 0, 0,  80,  80
CAPTION "Edit"
BEGIN
    EDITTEXT                       IDC_SINGLELINE,   5,   5,  70,  12,
                                          WS_TABSTOP
    EDITTEXT                       IDC_MULTILINE,    5,  20,  70,  40,
                                          WS_TABSTOP | ES_MULTILINE
    EDITTEXT                       IDC_PASSBOX,      5,  65,  70,  12,
                                          WS_TABSTOP | ES_PASSWORD
END

ID_LISTPAGE DIALOG discardable 0, 0,  125,  80
CAPTION "List"
BEGIN
    COMBOBOX                 IDC_COMBOBOX,      5,   5,  70,  60,
                                          WS_TABSTOP | CBS_DROPDOWN
    LISTBOX                  IDC_SNGLELIST,   5,  20,  50,  60,
                                          WS_TABSTOP
    LISTBOX                  IDC_MULTILIST,  60,  20,  50,  60,
                                          WS_TABSTOP | LBS_EXTENDEDSEL
END

ID_STATPAGE DIALOG discardable 0, 0, 130,  80
CAPTION "Static"
BEGIN
    LTEXT "Left text",             IDC_LEFTTEXT,    5,   5,  70,  20
    RTEXT "Right text",            IDC_RIGHTTEXT,   5,  30,  70,  20
    CTEXT "Center text",            IDC_CENTERTEXT,  5,  55,  70,  20,
                                            WS_BORDER
    ICON IDI_BTNICON                IDC_ICONCTL,    95,   5,  32,  32
    CONTROL "statbmp",              IDC_BITMAPCTL,  "static", SS_BITMAP,
                                                    95,  40,  32,  32
END

ID_SCROLLPAGE DIALOG discardable 0, 0,  60,  80
CAPTION "Scroll"
BEGIN
    SCROLLBAR                       IDC_LRSCROLL,  5,   5,  70,  12,
                                            WS_TABSTOP
    SCROLLBAR                       IDC_UDSCROLL, 80,   5,  12,  70,
                                            WS_TABSTOP | SBS_VERT
END
//----------------------------------------------------------------------
// Clear list modeless dialog box template.
//
Clearbox DIALOG discardable 60, 10,  70, 30
STYLE  WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU | DS_MODALFRAME
CAPTION "Clear"
BEGIN
    DEFPUSHBUTTON "Clear Listbox"
                      IDD_CLEAR,   5,   5,  60,   20
END
//----------------------------------------------------------------------
// About box dialog box template
//
aboutbox DIALOG discardable 10, 10, 132, 40
STYLE  WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU | DS_CENTER |
       DS_MODALFRAME
CAPTION "About"
BEGIN
    ICON    ID_ICON                  -1,   5,   5,   0,   0

    LTEXT "DlgDemo - Written for the book Programming Windows \
           CE Copyright 1998 Douglas Boling"
                                     -1,  28,   5, 100,  30
END
```

DlgDemo.h

```
//======================================================================
// Header file
//
// Written for the book Programming Windows CE
```

(continued)

Figure 4-10. *continued*

```
// 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_RPTLIST  2                         // ID for report list box

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

// Menu item IDs
#define  IDM_OPEN           100
#define  IDM_SAVE           101
#define  IDM_COLOR          102
#define  IDM_PRINT          103
#define  IDM_SHOWPROPSHEET  104
#define  IDM_SHOWMODELESS   105
#define  IDM_EXIT           106

#define  IDM_ABOUT          110

#define  IDI_BTNICON        120

// Identifiers for the property page resources
#define  ID_BTNPAGE         50
#define  ID_EDITPAGE        51
#define  ID_LISTPAGE        52
#define  ID_STATPAGE        53
#define  ID_SCROLLPAGE      54
```

```
// Button window defines
#define  IDC_PUSHBTN        200
#define  IDC_CHKBOX         201
#define  IDC_ACHKBOX        202
#define  IDC_A3STBOX        203
#define  IDC_RADIO1         204
#define  IDC_RADIO2         205
#define  IDC_OWNRDRAW       206

// Edit window defines
#define  IDC_SINGLELINE     210
#define  IDC_MULTILINE      211
#define  IDC_PASSBOX        212

// List box window defines
#define  IDC_COMBOBOX       220
#define  IDC_SNGLELIST      221
#define  IDC_MULTILIST      222

// Static control window defines
#define  IDC_LEFTTEXT       230
#define  IDC_RIGHTTEXT      231
#define  IDC_CENTERTEXT     232
#define  IDC_ICONCTL        233
#define  IDC_BITMAPCTL      234

// Scroll bar window defines
#define  IDC_LRSCROLL       240
#define  IDC_UDSCROLL       241

// Control IDs for modeless dialog box
#define  IDD_CLEAR          500

// User-defined message to add a line to the window
#define MYMSG_ADDLINE   (WM_USER + 10)

//----------------------------------------------------------------------
// Program-specific structures
//
typedef struct {
    TCHAR *pszLabel;
    DWORD wNotification;
} NOTELABELS, *PNOTELABELS;
```

(continued)

Figure 4-10. *continued*

```
//----------------------------------------------------------------------
// Function prototypes
//
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 DoCommandMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoAddLineMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);

// Command functions
LPARAM DoMainCommandOpen (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandSave (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandColor (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandPrint (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandShowProp (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandModeless (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandExit (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandAbout (HWND, WORD, HWND, WORD);

// Dialog box procedures
BOOL CALLBACK BtnDlgProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK EditDlgProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK ListDlgProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK StaticDlgProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK ScrollDlgProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK ModelessDlgProc (HWND, UINT, WPARAM, LPARAM);
```

DlgDemo.c

```
//======================================================================
// DlgDemo - 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 <commdlg.h>                   // Common dialog box includes
#include <prsht.h>                     // Property sheet includes

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

//----------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("DlgDemo");
HINSTANCE hInst;                       // Program instance handle
HWND g_hwndMlDlg = 0;                  // Handle to modeless dialog box

HINSTANCE hLib = 0;                    // Handle to CommDlg lib
FARPROC lpfnChooseColor = 0;           // Ptr to color common dialog fn
FARPROC lpfnPrintDlg = 0;              // Ptr to print common dialog fn

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

// Command message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDM_OPEN, DoMainCommandOpen,
    IDM_SAVE, DoMainCommandSave,
    IDM_SHOWPROPSHEET, DoMainCommandShowProp,
    IDM_SHOWMODELESS, DoMainCommandModeless,
    IDM_COLOR, DoMainCommandColor,
    IDM_PRINT, DoMainCommandPrint,
    IDM_EXIT, DoMainCommandExit,
    IDM_ABOUT, DoMainCommandAbout,
};
//
// Labels for WM_NOTIFY notifications
//
NOTELABELS nlPropPage[] = {{TEXT ("PSN_SETACTIVE  "), (PSN_FIRST-0)},
                           {TEXT ("PSN_KILLACTIVE "), (PSN_FIRST-1)},
                           {TEXT ("PSN_APPLY      "), (PSN_FIRST-2)},
                           {TEXT ("PSN_RESET      "), (PSN_FIRST-3)},
                           {TEXT ("PSN_HASHELP    "), (PSN_FIRST-4)},
                           {TEXT ("PSN_HELP       "), (PSN_FIRST-5)},
                           {TEXT ("PSN_WIZBACK    "), (PSN_FIRST-6)},
                           {TEXT ("PSN_WIZNEXT    "), (PSN_FIRST-7)},
```

(continued)

Figure 4-10. *continued*

```
                                    {TEXT ("PSN_WIZFINISH  "), (PSN_FIRST-8)},
                                    {TEXT ("PSN_QUERYCANCEL"), (PSN_FIRST-9)},
};
int nPropPageSize = dim(nlPropPage);

// Labels for the property pages
TCHAR *szPages[] = {TEXT ("Button"),
                    TEXT ("Edit  "),
                    TEXT ("List  "),
                    TEXT ("Static"),
                    TEXT ("Scroll"),
};
//======================================================================
// Program entry point
//
HWND hwndMain;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    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)) {
        // If modeless dialog box is created, let it have
        // the first crack at the message.
        if ((g_hwndMlDlg == 0) ||
            (!IsDialogMessage (g_hwndMlDlg, &msg))) {
            TranslateMessage (&msg);
            DispatchMessage (&msg);
        }
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//----------------------------------------------------------------------
```

```
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;

    // 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;

    // Get the Color and print dialog function pointers.
    hLib = LoadLibrary (TEXT ("COMMDLG.DLL"));
    if (hLib) {
        lpfnChooseColor = GetProcAddress (hLib, TEXT ("ChooseColor"));
        lpfnPrintDlg = GetProcAddress (hLib, TEXT ("PrintDlg"));
    }
    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 ("Dialog 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
```

(continued)

Figure 4-10. *continued*

```
                          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) {
    if (hLib)
        FreeLibrary (hLib);
    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 i, nHeight;
    LPCREATESTRUCT lpcs;
    HMENU hMenu;

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

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);
    // Add the menu.
    CommandBar_InsertMenubar (hwndCB, hInst, ID_MENU, 0);
    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);

    // See color and print functions not found, disable menus.
    hMenu = CommandBar_GetMenu (hwndCB, 0);
    if (!lpfnChooseColor)
        EnableMenuItem (hMenu, IDM_COLOR, MF_BYCOMMAND | MF_GRAYED);
    if (!lpfnPrintDlg)
        EnableMenuItem (hMenu, IDM_PRINT, MF_BYCOMMAND | MF_GRAYED);

    nHeight = CommandBar_Height (hwndCB);
    //
    // Create report window.  Size it so that it fits under
    // the command bar and fills the remaining client area.
    //
    hwndChild = CreateWindowEx (0, TEXT ("listbox"),
                         TEXT (""), WS_VISIBLE | WS_CHILD | WS_VSCROLL |
                         LBS_USETABSTOPS | LBS_NOINTEGRALHEIGHT, 0,
                         nHeight, lpcs->cx, lpcs->cy - nHeight,
                         hWnd, (HMENU)IDC_RPTLIST,
                         lpcs->hInstance, NULL);

    // Destroy frame if window not created.
    if (!IsWindow (hwndChild)) {
        DestroyWindow (hWnd);
        return 0;
    }
    // Initialize tab stops for display list box.
    i = 40;
    SendMessage (hwndChild, LB_SETTABSTOPS, 1, (LPARAM)&i);
    return 0;
}
//----------------------------------------------------------------------
// DoCommandMain - Process WM_COMMAND message for window.
```

(continued)

Figure 4-10. *continued*

```
//
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;
}
//----------------------------------------------------------------------
// DoAddLineMain - Process MYMSG_ADDLINE message for window.
//
LRESULT DoAddLineMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    TCHAR szOut[128];
    INT i;

    // If nothing in wParam, just fill in spaces.
    if (wParam == -1) {
        // Print message only.
        lstrcpy (szOut, (LPTSTR)lParam);
    } else {
        // If no ID val, ignore that field.
        if (LOWORD (wParam) == 0xffff)
            // Print prop page and message.
            wsprintf (szOut, TEXT ("%s \t        \t %s"),
                      szPages[HIWORD (wParam) - ID_BTNPAGE],
                      (LPTSTR)lParam);
        else
            // Print property page, control ID, and message.
            wsprintf (szOut, TEXT ("%s \t id:%x \t %s"),
                      szPages[HIWORD (wParam) - ID_BTNPAGE],
                      LOWORD (wParam), (LPTSTR)lParam);
    }
    i = SendDlgItemMessage (hWnd, IDC_RPTLIST, LB_ADDSTRING, 0,
                            (LPARAM)(LPCTSTR)szOut);

    if (i != LB_ERR)
        SendDlgItemMessage (hWnd, IDC_RPTLIST, LB_SETTOPINDEX, i,
                            (LPARAM)(LPCTSTR)szOut);
    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) {
    OPENFILENAME of;
    TCHAR szFileName [MAX_PATH] = {0};
    const LPTSTR pszOpenFilter = TEXT ("All Documents (*.*)\0*.*\0\0");
    TCHAR szOut[128];
    INT rc;

    // Initialize filename.
    szFileName[0] = '\0';

    // Initialize File Open structure.
    memset (&of, 0, sizeof (of));

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

    rc = GetOpenFileName (&of);
```

(continued)

Figure 4-10. *continued*

```
    wsprintf (szOut,
              TEXT ("GetOpenFileName returned: %x, filename: %s"),
              rc, szFileName);
    SendMessage (hWnd, MYMSG_ADDLINE, -1, (LPARAM)szOut);
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandSave - Process File Save command.
//
LPARAM DoMainCommandSave (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {
    OPENFILENAME of;
    TCHAR szFileName [MAX_PATH] = {0};
    const LPTSTR pszOpenFilter = TEXT ("All Documents (*.*)\0*.*\0\0");
    TCHAR szOut[128];
    INT rc;

    // Initialize filename.
    szFileName[0] = '\0';

    // Initialize File Open structure.
    memset (&of, 0, sizeof (of));

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

    rc = GetSaveFileName (&of);

    wsprintf (szOut,
              TEXT ("GetSaveFileName returned: %x, filename: %s"),
              rc, szFileName);
    SendMessage (hWnd, MYMSG_ADDLINE, -1, (LPARAM)szOut);
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandColor - Process File Color command.
//
LPARAM DoMainCommandColor (HWND hWnd, WORD idItem, HWND hwndCtl,
                           WORD wNotifyCode) {
    CHOOSECOLOR cc;
    static COLORREF cr[16];
    TCHAR szOut[128];
    INT rc;
    // Initialize color structure.
    memset (&cc, 0, sizeof (cc));
    memset (&cr, 0, sizeof (cr));

    cc.lStructSize = sizeof (cc);
    cc.hwndOwner = hWnd;
    cc.hInstance = hInst;
    cc.rgbResult = RGB (0, 0, 0);
    cc.lpCustColors = cr;
    cc.Flags = CC_ANYCOLOR;

    rc = (lpfnChooseColor) (&cc);

    wsprintf (szOut, TEXT ("Choose Color returned: %x, color: %x"),
              rc, cc.rgbResult);
    SendMessage (hWnd, MYMSG_ADDLINE, -1, (LPARAM)szOut);
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandPrint - Process File Print command.
//
LPARAM DoMainCommandPrint (HWND hWnd, WORD idItem, HWND hwndCtl,
                           WORD wNotifyCode) {
    PRINTDLG pd;
    INT rc;

    // Initialize print structure.
    memset (&pd, 0, sizeof (pd));

    pd.cbStruct = sizeof (pd);
    pd.hwndOwner = hWnd;
    pd.dwFlags = PD_SELECTALLPAGES;

    rc = (lpfnPrintDlg) (&pd);

    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandShowProp - Process show property sheet command.
//
LPARAM DoMainCommandShowProp(HWND hWnd, WORD idItem, HWND hwndCtl,
                             WORD wNotifyCode) {

    PROPSHEETPAGE psp[5];
    PROPSHEETHEADER psh;
    INT i;
```

(continued)

Figure 4-10. *continued*

```
    // Zero all the property page structures.
    memset (&psp, 0, sizeof (psp));
    // Fill in default values in property page structures.
    for (i = 0; i < dim(psp); i++) {
        psp[i].dwSize = sizeof (PROPSHEETPAGE);
        psp[i].dwFlags = PSP_DEFAULT;
        psp[i].hInstance = hInst;
        psp[i].lParam = (LPARAM)hWnd;
    }
    // Set the dialog box templates for each page.
    psp[0].pszTemplate = MAKEINTRESOURCE (ID_BTNPAGE);
    psp[1].pszTemplate = MAKEINTRESOURCE (ID_EDITPAGE);
    psp[2].pszTemplate = MAKEINTRESOURCE (ID_LISTPAGE);
    psp[3].pszTemplate = MAKEINTRESOURCE (ID_STATPAGE);
    psp[4].pszTemplate = MAKEINTRESOURCE (ID_SCROLLPAGE);

    // Set the dialog box procedures for each page.
    psp[0].pfnDlgProc = BtnDlgProc;
    psp[1].pfnDlgProc = EditDlgProc;
    psp[2].pfnDlgProc = ListDlgProc;
    psp[3].pfnDlgProc = StaticDlgProc;
    psp[4].pfnDlgProc = ScrollDlgProc;

    // Initialize property sheet structure.
    psh.dwSize = sizeof (PROPSHEETHEADER);
    psh.dwFlags = PSH_PROPSHEETPAGE;
    psh.hwndParent = hWnd;
    psh.hInstance = hInst;
    psh.pszCaption = TEXT ("Property Sheet Demo");
    psh.nPages = dim(psp);
    psh.nStartPage = 0;
    psh.ppsp = psp;
    psh.pfnCallback = 0;

    // Create and display property sheet.
    i = PropertySheet (&psh);
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandModelessDlg - Process the File Modeless menu command.
//
LPARAM DoMainCommandModeless(HWND hWnd, WORD idItem, HWND hwndCtl,
                             WORD wNotifyCode) {
```

Page 00265

```
    // Only create dialog box if not already created.
    if (g_hwndMlDlg == 0)
        // Use CreateDialog to create modeless dialog box.
        g_hwndMlDlg = CreateDialog (hInst, TEXT ("Clearbox"), hWnd,
                                    ModelessDlgProc);
    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;
}
//----------------------------------------------------------------------
// 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;
}
//======================================================================
// Modeless ClearList dialog box procedure.
//
BOOL CALLBACK ModelessDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                               LPARAM lParam) {

    switch (wMsg) {
        case WM_COMMAND:
            switch (LOWORD (wParam)) {
                case IDD_CLEAR:
                    // Send message to list box to clear it.
                    SendDlgItemMessage (GetWindow (hWnd, GW_OWNER),
                                        IDC_RPTLIST,
                                        LB_RESETCONTENT, 0, 0);
                    return TRUE;

                case IDOK:
                case IDCANCEL:
                    // Modeless dialog boxes can't use EndDialog.
                    DestroyWindow (hWnd);
```

(continued)

Figure 4-10. *continued*

```
                    // Set hwnd value to zero to indicate that
                    // the dialog box is destroyed.
                    g_hwndMlDlg = 0;
                    return TRUE;

            }
            break;
    }
    return FALSE;
}
//======================================================================
// About dialog box 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;
}
```

BtnDlg.c

```
//======================================================================
// BtnDlg - Button dialog box window code
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                    // For all that Windows stuff
#include <prsht.h>                      // Property sheet includes
#include "DlgDemo.h"                    // Program-specific stuff

extern HINSTANCE hInst;

LRESULT DrawButton (HWND hWnd, LPDRAWITEMSTRUCT pdi);
//----------------------------------------------------------------------
// Global data
```

```
//
// Identification strings for various WM_COMMAND notifications
NOTELABELS nlBtn[] = {{TEXT ("BN_CLICKED "),       0},
                      {TEXT ("BN_PAINT   "),       1},
                      {TEXT ("BN_HILITE  "),       2},
                      {TEXT ("BN_UNHILITE"),       3},
                      {TEXT ("BN_DISABLE "),       4},
                      {TEXT ("BN_DOUBLECLICKED"),  5},
                      {TEXT ("BN_SETFOCUS "),      6},
                      {TEXT ("BN_KILLFOCUS"),      7}
};
extern NOTELABELS nlPropPage[];
extern int nPropPageSize;

// Handle for icon used in owner-draw icon
HICON hIcon = 0;
//======================================================================
// BtnDlgProc - Button page dialog box procedure
//
BOOL CALLBACK BtnDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                          LPARAM lParam) {
    TCHAR szOut[128];
    HWND hwndMain;
    INT i;

    switch (wMsg) {

        case WM_INITDIALOG:
            // The generic parameter contains the
            // top-level window handle.
            hwndMain = (HWND)((LPPROPSHEETPAGE)lParam)->lParam;
            // Save the window handle in the window structure.
            SetWindowLong (hWnd, DWL_USER, (LONG)hwndMain);

            // Load icon for owner-draw window.
            hIcon = LoadIcon (hInst, MAKEINTRESOURCE (IDI_BTNICON));

            // We need to set the initial state of the radio buttons.
            CheckRadioButton (hWnd, IDC_RADIO1, IDC_RADIO2, IDC_RADIO1);
            return TRUE;
        //
        // Reflect WM_COMMAND messages to main window.
        //
        case WM_COMMAND:
            // Since the check box is not an auto check box, the button
            // has to be set manually.
```

(continued)

Figure 4-10. *continued*

```
            if ((LOWORD (wParam) == IDC_CHKBOX) &&
                (HIWORD (wParam) == BN_CLICKED)) {
                // Get the current state, complement, and set.
                i = SendDlgItemMessage (hWnd, IDC_CHKBOX, BM_GETCHECK,
                                        0, 0);
                if (i)
                    SendDlgItemMessage (hWnd, IDC_CHKBOX, BM_SETCHECK,
                                        0, 0);
                else
                    SendDlgItemMessage (hWnd, IDC_CHKBOX, BM_SETCHECK,
                                        1, 0);
            }

            // Get the handle of the main window from the user word.
            hwndMain = (HWND) GetWindowLong (hWnd, DWL_USER);

            // Look up button notification.
            lstrcpy (szOut, TEXT ("WM_COMMAND: "));
            for (i = 0; i < dim(nlBtn); i++) {
                if (HIWORD (wParam) == nlBtn[i].wNotification) {
                    lstrcat (szOut, nlBtn[i].pszLabel);
                    break;
                }
            }
            if (i == dim(nlBtn))
                wsprintf (szOut, TEXT ("WM_COMMAND notification: %x"),
                          HIWORD (wParam));

            SendMessage (hwndMain, MYMSG_ADDLINE,
                         MAKEWPARAM (LOWORD (wParam),ID_BTNPAGE),
                         (LPARAM)szOut);
            return TRUE;

        //
        // Reflect notify message.
        //
        case WM_NOTIFY:
            // Get the handle of the main window from the user word.
            hwndMain = (HWND) GetWindowLong (hWnd, DWL_USER);

            // Look up notify message.
            for (i = 0; i < nPropPageSize; i++) {
```

Page 00269

```
                if (((NMHDR *)lParam)->code ==
                                    nlPropPage[i].wNotification) {
                    lstrcpy (szOut, nlPropPage[i].pszLabel);
                    break;
                }
            }
            if (i == nPropPageSize)
                wsprintf (szOut, TEXT ("Notify code:%d"),
                          ((NMHDR *)lParam)->code);

            SendMessage (hwndMain, MYMSG_ADDLINE,
                         MAKEWPARAM (-1,ID_BTNPAGE), (LPARAM)szOut);

            return FALSE;  // Return false to force default processing.

        case WM_DRAWITEM:
            DrawButton (hWnd, (LPDRAWITEMSTRUCT)lParam);
            return TRUE;
    }
    return FALSE;
}

//----------------------------------------------------------------------
// DrawButton - Draws an owner-draw button.
//
LRESULT DrawButton (HWND hWnd, LPDRAWITEMSTRUCT pdi) {

    HPEN hPenShadow, hPenLight, hPenDkShadow, hOldPen;
    POINT ptOut[3], ptIn[3];
    HBRUSH hBr, hOldBr;
    TCHAR szOut[128];
    HWND hwndMain;
    LOGPEN lpen;

    // Get the handle of the main window from the user word.
    hwndMain = (HWND) GetWindowLong (hWnd, DWL_USER);

    // Reflect the messages to the report window.
    wsprintf (szOut, TEXT ("WM_DRAWITEM  Action:%x  State:%x"),
              pdi->itemAction, pdi->itemState);

    SendMessage (hwndMain, MYMSG_ADDLINE,
                 MAKEWPARAM (pdi->CtlID, ID_BTNPAGE),
                 (LPARAM)szOut);
```

(continued)

Figure 4-10. *continued*

```
    // Create pens for drawing.
    lpen.lopnStyle = PS_SOLID;
    lpen.lopnWidth.x = 3;
    lpen.lopnWidth.y = 3;
    lpen.lopnColor = GetSysColor (COLOR_3DSHADOW);
    hPenShadow = CreatePenIndirect (&lpen);

    lpen.lopnWidth.x = 1;
    lpen.lopnWidth.y = 1;
    lpen.lopnColor = GetSysColor (COLOR_3DLIGHT);
    hPenLight = CreatePenIndirect (&lpen);

    lpen.lopnColor = GetSysColor (COLOR_3DDKSHADOW);
    hPenDkShadow = CreatePenIndirect (&lpen);

    // Create a brush for the face of the button.
    hBr = CreateSolidBrush (GetSysColor (COLOR_3DFACE));

    // Draw a rectangle with a thick outside border to start the
    // frame drawing.
    hOldPen = SelectObject (pdi->hDC, hPenShadow);
    hOldBr = SelectObject (pdi->hDC, hBr);
    Rectangle (pdi->hDC, pdi->rcItem.left, pdi->rcItem.top,
               pdi->rcItem.right, pdi->rcItem.bottom);

    // Draw the upper left inside line.
    ptIn[0].x = pdi->rcItem.left + 1;
    ptIn[0].y = pdi->rcItem.bottom - 3;
    ptIn[1].x = pdi->rcItem.left + 1;
    ptIn[1].y = pdi->rcItem.top + 1;
    ptIn[2].x = pdi->rcItem.right - 3;
    ptIn[2].y = pdi->rcItem.top+1;

    // Select a pen to draw shadow or light side of button.
    if (pdi->itemState & ODS_SELECTED) {
        SelectObject (pdi->hDC, hPenDkShadow);
    } else {
        SelectObject (pdi->hDC, hPenLight);
    }
    Polyline (pdi->hDC, ptIn, 3);

    // If selected, also draw a bright line inside the lower
    // right corner.
    if (pdi->itemState & ODS_SELECTED) {
        SelectObject (pdi->hDC, hPenLight);
        ptIn[1].x = pdi->rcItem.right- 3;
        ptIn[1].y = pdi->rcItem.bottom - 3;
        Polyline (pdi->hDC, ptIn, 3);
    }
    // Now draw the black outside line on either the upper left or lower
    // right corner.
    ptOut[0].x = pdi->rcItem.left;
    ptOut[0].y = pdi->rcItem.bottom-1;
    ptOut[2].x = pdi->rcItem.right-1;
    ptOut[2].y = pdi->rcItem.top;

    SelectObject (pdi->hDC, hPenDkShadow);
    if (pdi->itemState & ODS_SELECTED) {
        ptOut[1].x = pdi->rcItem.left;
        ptOut[1].y = pdi->rcItem.top;
    } else {
        ptOut[1].x = pdi->rcItem.right-1;
        ptOut[1].y = pdi->rcItem.bottom-1;
    }
    Polyline (pdi->hDC, ptOut, 3);

    // Draw the icon.
    if (hIcon) {
        ptIn[0].x = (pdi->rcItem.right - pdi->rcItem.left)/2 -
                    GetSystemMetrics (SM_CXICON)/2 - 2;
        ptIn[0].y = (pdi->rcItem.bottom - pdi->rcItem.top)/2 -
                    GetSystemMetrics (SM_CYICON)/2 - 2;
        // If pressed, shift image down one pel to simulate the press.
        if (pdi->itemState & ODS_SELECTED) {
            ptOut[1].x += 2;
            ptOut[1].y += 2;
        }
        DrawIcon (pdi->hDC, ptIn[0].x, ptIn[0].y, hIcon);
    }

    // If button has the focus, draw the dotted rect inside the button.
    if (pdi->itemState & ODS_FOCUS) {
        pdi->rcItem.left += 3;
        pdi->rcItem.top += 3;
        pdi->rcItem.right -= 4;
        pdi->rcItem.bottom -= 4;
        DrawFocusRect (pdi->hDC, &pdi->rcItem);
    }

    // Clean up. First select the original brush and pen into the DC.
    SelectObject (pdi->hDC, hOldBr);
    SelectObject (pdi->hDC, hOldPen);
```

(continued)

Figure 4-10. *continued*

```
    // Now delete the brushes and pens created.
    DeleteObject (hBr);
    DeleteObject (hPenShadow);
    DeleteObject (hPenDkShadow);
    DeleteObject (hPenLight);
    return 0;
}
```

EditDlg.c

```
//======================================================================
// EditDlg - Edit dialog box window code
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                 // For all that Windows stuff
#include <prsht.h>                   // Property sheet includes
#include "DlgDemo.h"                 // Program-specific stuff

extern HINSTANCE hInst;
//----------------------------------------------------------------------
// Global data
//
// Identification strings for various WM_COMMAND notifications
NOTELABELS nlEdit[] = {{TEXT ("EN_SETFOCUS "), 0x0100},
                       {TEXT ("EN_KILLFOCUS"), 0x0200},
                       {TEXT ("EN_CHANGE   "), 0x0300},
                       {TEXT ("EN_UPDATE   "), 0x0400},
                       {TEXT ("EN_ERRSPACE "), 0x0500},
                       {TEXT ("EN_MAXTEXT  "), 0x0501},
                       {TEXT ("EN_HSCROLL  "), 0x0601},
                       {TEXT ("EN_VSCROLL  "), 0x0602},
};
extern NOTELABELS nlPropPage[];
extern int nPropPageSize;
//======================================================================
// EditDlgProc - Button page dialog box procedure
//
BOOL CALLBACK EditDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                           LPARAM lParam) {
    TCHAR szOut[128];
    HWND hwndMain;
    INT i;
```

```
switch (wMsg) {

    case WM_INITDIALOG:
        // The generic parameter contains the
        // top-level window handle.
        hwndMain = (HWND)((LPPROPSHEETPAGE)lParam)->lParam;
        // Save the window handle in the window structure.
        SetWindowLong (hWnd, DWL_USER, (LONG)hwndMain);
        return TRUE;
    //
    // Reflect WM_COMMAND messages to main window.
    //
    case WM_COMMAND:
        // Get the handle of the main window from the user word.
        hwndMain = (HWND) GetWindowLong (hWnd, DWL_USER);

        // Look up button notification.
        lstrcpy (szOut, TEXT ("WM_COMMAND: "));
        for (i = 0; i < dim(nlEdit); i++) {
            if (HIWORD (wParam) == nlEdit[i].wNotification) {
                lstrcat (szOut, nlEdit[i].pszLabel);
                break;
            }
        }
        if (i == dim(nlEdit))
            wsprintf (szOut, TEXT ("WM_COMMAND notification: %x"),
                      HIWORD (wParam));

        SendMessage (hwndMain, MYMSG_ADDLINE,
                     MAKEWPARAM (LOWORD (wParam),ID_EDITPAGE),
                     (LPARAM)szOut);
        return TRUE;

    //
    // Reflect notify message.
    //
    case WM_NOTIFY:
        // Get the handle of the main window from the user word.
        hwndMain = (HWND) GetWindowLong (hWnd, DWL_USER);

        // Look up notify message.
        for (i = 0; i < nPropPageSize; i++) {
            if (((NMHDR *)lParam)->code ==
                                  nlPropPage[i].wNotification) {
                lstrcpy (szOut, nlPropPage[i].pszLabel);
                break;
```

(continued)

Figure 4-10. *continued*

```
                }
            }
            if (i == nPropPageSize)
                wsprintf (szOut, TEXT ("Notify code:%d"),
                          ((NMHDR *)lParam)->code);

            SendMessage (hwndMain, MYMSG_ADDLINE,
                         MAKEWPARAM (-1,ID_EDITPAGE), (LPARAM)szOut);

            return FALSE;  // Return false to force default processing.
    }
    return FALSE;
}
```

ListDlg.c

```
//======================================================================
// ListDlg - List box dialog window code
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                  // For all that Windows stuff
#include <prsht.h>                    // Property sheet includes
#include "DlgDemo.h"                  // Program-specific stuff

extern HINSTANCE hInst;
//----------------------------------------------------------------------
// Global data
//
NOTELABELS nlList[] = {{TEXT ("LBN_ERRSPACE "), (-2)},
                       {TEXT ("LBN_SELCHANGE"), 1},
                       {TEXT ("LBN_DBLCLK   "), 2},
                       {TEXT ("LBN_SELCANCEL"), 3},
                       {TEXT ("LBN_SETFOCUS "), 4},
                       {TEXT ("LBN_KILLFOCUS"), 5},
};

NOTELABELS nlCombo[] = {{TEXT ("CBN_ERRSPACE    "), (-1)},
                        {TEXT ("CBN_SELCHANGE   "), 1},
                        {TEXT ("CBN_DBLCLK      "), 2},
                        {TEXT ("CBN_SETFOCUS    "), 3},
                        {TEXT ("CBN_KILLFOCUS   "), 4},
                        {TEXT ("CBN_EDITCHANGE  "), 5},
                        {TEXT ("CBN_EDITUPDATE  "), 6},
                    {TEXT ("CBN_DROPDOWN    "), 7},
                    {TEXT ("CBN_CLOSEUP     "), 8},
                    {TEXT ("CBN_SELENDOK    "), 9},
                    {TEXT ("CBN_SELENDCANCEL"), 10},
};

extern NOTELABELS nlPropPage[];
extern int nPropPageSize;
//======================================================================
// ListDlgProc - Button page dialog box procedure
//
BOOL CALLBACK ListDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                           LPARAM lParam) {
    TCHAR szOut[128];
    HWND hwndMain;
    INT i;

    switch (wMsg) {

        case WM_INITDIALOG:
            // The generic parameter contains the
            // top-level window handle.
            hwndMain = (HWND)((LPPROPSHEETPAGE)lParam)->lParam;
            // Save the window handle in the window structure.
            SetWindowLong (hWnd, DWL_USER, (LONG)hwndMain);

            // Fill the list and combo boxes.
            for (i = 0; i < 20; i++) {
                wsprintf (szOut, TEXT ("Item %d"), i);
                SendDlgItemMessage (hWnd, IDC_SNGLELIST, LB_ADDSTRING,
                                    0, (LPARAM)szOut);

                SendDlgItemMessage (hWnd, IDC_MULTILIST, LB_ADDSTRING,
                                    0, (LPARAM)szOut);

                SendDlgItemMessage (hWnd, IDC_COMBOBOX, CB_ADDSTRING,
                                    0, (LPARAM)szOut);
            }
            // Provide default selection for the combo box.
            SendDlgItemMessage (hWnd, IDC_COMBOBOX, CB_SETCURSEL, 0, 0);
            return TRUE;
        //
        // Reflect WM_COMMAND messages to main window.
        //
```

(continued)

Figure 4-10. *continued*

```
        case WM_COMMAND:
            // Get the handle of the main window from the user word.
            hwndMain = (HWND) GetWindowLong (hWnd, DWL_USER);

            // Report the WM_COMMAND messages.
            lstrcpy (szOut, TEXT ("WM_COMMAND: "));
            if (LOWORD (wParam) == IDC_COMBOBOX) {
                for (i = 0; i < dim(nlCombo); i++) {
                    if (HIWORD (wParam) == nlCombo[i].wNotification) {
                        lstrcat (szOut, nlCombo[i].pszLabel);
                        break;
                    }
                }
                if (i == dim(nlCombo))
                    wsprintf (szOut,
                              TEXT ("WM_COMMAND notification: %x"),
                              HIWORD (wParam));
            } else {
                for (i = 0; i < dim(nlList); i++) {
                    if (HIWORD (wParam) == nlList[i].wNotification) {
                        lstrcat (szOut, nlList[i].pszLabel);
                        break;
                    }
                }
                if (i == dim(nlList))
                    wsprintf (szOut,
                              TEXT ("WM_COMMAND notification: %x"),
                              HIWORD (wParam));
            }
            SendMessage (hwndMain, MYMSG_ADDLINE,
                         MAKEWPARAM (LOWORD (wParam),ID_LISTPAGE),
                        (LPARAM)szOut);
            return TRUE;

        //
        // Reflect notify message.
        //
        case WM_NOTIFY:
            // Get the handle of the main window from the user word.
            hwndMain = (HWND) GetWindowLong (hWnd, DWL_USER);

            // Look up notify message.
            for (i = 0; i < nPropPageSize; i++) {
                if (((NMHDR *)lParam)->code ==
                                      nlPropPage[i].wNotification) {
```

Page 00277

```
                    lstrcpy (szOut, nlPropPage[i].pszLabel);
                    break;
                }
            }
            if (i == nPropPageSize)
                wsprintf (szOut, TEXT ("Notify code:%d"),
                          ((NMHDR *)lParam)->code);

            SendMessage (hwndMain, MYMSG_ADDLINE,
                         MAKEWPARAM (-1,ID_LISTPAGE),
                         (LPARAM)szOut);
            return FALSE;  // Return false to force default processing.
    }
    return FALSE;
}
```

StaticDlg.c

```
//======================================================================
// StaticDlg - Static control dialog box window code
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                  // For all that Windows stuff
#include <prsht.h>                    // Property sheet includes
#include "DlgDemo.h"                  // Program-specific stuff

extern HINSTANCE hInst;
//----------------------------------------------------------------------
// Global data
//
// Identification strings for various WM_COMMAND notifications
NOTELABELS nlStatic[] = {{TEXT ("STN_CLICKED"), 0},
                         {TEXT ("STN_ENABLE "), 2},
                         {TEXT ("STN_DISABLE"), 3},
};
extern NOTELABELS nlPropPage[];
extern int nPropPageSize;
//======================================================================
// StaticDlgProc - Button page dialog box procedure
//
BOOL CALLBACK StaticDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                             LPARAM lParam) {
    TCHAR szOut[128];
```

(continued)

Figure 4-10. *continued*

```
    HWND hwndMain;
    INT i;

    switch (wMsg) {

        case WM_INITDIALOG:
            // The generic parameter contains the
            // top-level window handle.
            hwndMain = (HWND)((LPPROPSHEETPAGE)lParam)->lParam;
            // Save the window handle in the window structure.
            SetWindowLong (hWnd, DWL_USER, (LONG)hwndMain);
            return TRUE;
        //
        // Reflect WM_COMMAND messages to main window.
        //
        case WM_COMMAND:
            // Get the handle of the main window from the user word.
            hwndMain = (HWND) GetWindowLong (hWnd, DWL_USER);

            // Look up button notification.
            lstrcpy (szOut, TEXT ("WM_COMMAND: "));
            for (i = 0; i < dim(nlStatic); i++) {
                if (HIWORD (wParam) == nlStatic[i].wNotification) {
                    lstrcat (szOut, nlStatic[i].pszLabel);
                    break;
                }
            }
            if (i == dim(nlStatic))
                wsprintf (szOut, TEXT ("WM_COMMAND notification: %x"),
                          HIWORD (wParam));

            SendMessage (hwndMain, MYMSG_ADDLINE,
                         MAKEWPARAM (LOWORD (wParam),ID_STATPAGE),
                         (LPARAM)szOut);
            return TRUE;

        //
        // Reflect notify message.
        //
        case WM_NOTIFY:
            // Get the handle of the main window from the user word.
            hwndMain = (HWND) GetWindowLong (hWnd, DWL_USER);

            // Look up notify message.
            for (i = 0; i < nPropPageSize; i++) {
                if (((NMHDR *)lParam)->code ==
                    nlPropPage[i].wNotification) {
                        lstrcpy (szOut, nlPropPage[i].pszLabel);
                    break;
                }
            }
            if (i == nPropPageSize)
                wsprintf (szOut, TEXT ("Notify code:%d"),
                          ((NMHDR *)lParam)->code);

            SendMessage (hwndMain, MYMSG_ADDLINE,
                         MAKEWPARAM (-1,ID_STATPAGE), (LPARAM)szOut);

            return FALSE;  // Return false to force default processing.
    }
    return FALSE;
}
```

ScrollDlg.c

```
//======================================================================
// ScrollDlg - Scroll bar dialog box window code
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                 // For all that Windows stuff
#include <prsht.h>                   // Property sheet includes
#include "DlgDemo.h"                 // Program-specific stuff

extern HINSTANCE hInst;
//----------------------------------------------------------------------
// Global data
//
// Identification strings for various WM_xSCROLL notifications
NOTELABELS nlVScroll[] = {{TEXT ("SB_LINEUP        "), 0},
                          {TEXT ("SB_LINEDOWN      "), 1},
                          {TEXT ("SB_PAGEUP        "), 2},
                          {TEXT ("SB_PAGEDOWN      "), 3},
                          {TEXT ("SB_THUMBPOSITION"), 4},
                          {TEXT ("SB_THUMBTRACK    "), 5},
                          {TEXT ("SB_TOP           "), 6},
                          {TEXT ("SB_BOTTOM        "), 7},
                          {TEXT ("SB_ENDSCROLL     "), 8},
};
```

(continued)

Figure 4-10. *continued*

```
NOTELABELS nlHScroll[] = {{TEXT ("SB_LINELEFT     "), 0},
                          {TEXT ("SB_LINERIGHT    "), 1},
                          {TEXT ("SB_PAGELEFT     "), 2},
                          {TEXT ("SB_PAGERIGHT    "), 3},
                          {TEXT ("SB_THUMBPOSITION"), 4},
                          {TEXT ("SB_THUMBTRACK   "), 5},
                          {TEXT ("SB_LEFT         "), 6},
                          {TEXT ("SB_RIGHT        "), 7},
                          {TEXT ("SB_ENDSCROLL    "), 8},
};
extern NOTELABELS nlPropPage[];
extern int nPropPageSize;
//======================================================================
// EditDlgProc - Button page dialog box procedure
//
BOOL CALLBACK ScrollDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                             LPARAM lParam) {
    TCHAR szOut[128];
    SCROLLINFO si;
    HWND hwndMain;
    INT i, sPos;

    switch (wMsg) {

        case WM_INITDIALOG:
            // The generic parameter contains
            // the top-level window handle.
            hwndMain = (HWND)((LPPROPSHEETPAGE)lParam)->lParam;
            // Save the window handle in the window structure.
            SetWindowLong (hWnd, DWL_USER, (LONG)hwndMain);
            return TRUE;
        //
        // Reflect WM_COMMAND messages to main window.
        //
        case WM_VSCROLL:
        case WM_HSCROLL:
            // Get the handle of the main window from the user word.
            hwndMain = (HWND) GetWindowLong (hWnd, DWL_USER);

            // Update the report window.
            // Determine whether from horizontal or vertical scroll bar.
            if (GetDlgItem (hWnd, 101) == (HWND)lParam) {
                for (i = 0; i < dim(nlVScroll); i++) {
                    if (LOWORD (wParam) == nlVScroll[i].wNotification) {
                        lstrcpy (szOut, nlVScroll[i].pszLabel);
                        break;
                }
            }
            if (i == dim(nlVScroll))
                wsprintf (szOut, TEXT ("notification: %x"),
                          HIWORD (wParam));
        } else {
            for (i = 0; i < dim(nlHScroll); i++) {
                if (LOWORD (wParam) == nlHScroll[i].wNotification) {
                    lstrcpy (szOut, nlHScroll[i].pszLabel);
                    break;
                }
            }
            if (i == dim(nlHScroll))
                wsprintf (szOut, TEXT ("notification: %x"),
                          HIWORD (wParam));
        }
        SendMessage (hwndMain, MYMSG_ADDLINE,
                     MAKEWPARAM (-1, ID_SCROLLPAGE), (LPARAM)szOut);

        // Get scroll bar position.
        si.cbSize = sizeof (si);
        si.fMask = SIF_POS;
        GetScrollInfo ((HWND)lParam, SB_CTL, &si);
        sPos = si.nPos;

        // Act on the scroll code.
        switch (LOWORD (wParam)) {
        case SB_LINEUP:      // Also SB_LINELEFT
            sPos -= 2;
            break;

        case SB_LINEDOWN:    // Also SB_LINERIGHT
            sPos += 2;
            break;

        case SB_PAGEUP:      // Also SB_PAGELEFT
            sPos -= 10;
            break;

        case SB_PAGEDOWN:    // Also SB_PAGERIGHT
            sPos += 10;
            break;

        case SB_THUMBPOSITION:
            sPos = HIWORD (wParam);
            break;
        }
```

(continued)

Figure 4-10. *continued*

```
            // Check range.
            if (sPos < 0)
                sPos = 0;
            if (sPos > 100)
                sPos = 100;

            // Update scrollbar position.
            si.cbSize = sizeof (si);
            si.nPos = sPos;
            si.fMask = SIF_POS;
            SetScrollInfo ((HWND)lParam, SB_CTL, &si, TRUE);

            return TRUE;

        //
        // Reflect notify message.
        //
        case WM_NOTIFY:
            // Get the handle of the main window from the user word.
            hwndMain = (HWND) GetWindowLong (hWnd, DWL_USER);

            // Look up notify message.
            for (i = 0; i < nPropPageSize; i++) {
                if (((NMHDR *)lParam)->code ==
                        nlPropPage[i].wNotification) {
                            lstrcpy (szOut, nlPropPage[i].pszLabel);
                            break;
                }
            }
            if (i == nPropPageSize)
                wsprintf (szOut, TEXT ("Notify code:%d"),
                          ((NMHDR *)lParam)->code);

            SendMessage (hwndMain, MYMSG_ADDLINE,
                         MAKEWPARAM (-1, ID_SCROLLPAGE), (LPARAM)szOut);

            return FALSE;  // Return false to force default processing.
    }
    return FALSE;
}
```

The dialog box procedures for each of the property pages report all WM_COMMAND and WM_NOTIFY messages back to the main window where they're displayed in a list box contained in the main window. The property page dialog box procedures mirror the child window procedures of the CtlView example, the differences being that the page procedures don't have to create their controls, and they field the WM_INITDIALOG message to initialize the controls. The page procedures also use the technique of storing information in their window structures—in this case, the window handle of the main window of the example. This is necessary because the parent window of the pages is the property sheet, not the main window. The window handle is conveniently accessible during the WM_INITDIALOG message because it's loaded into the user-definable parameter in the PROPSHEETPAGE structure by the main window when the property sheet is created. Each page procedure copies the parameter from the PROPSHEETPAGE structure into the DWL_USER field of the window structure available to all dialog box procedures. When other messages are handled, the handle is then queried using *GetWindowLong*. The page procedures also field the WM_NOTIFY message so that they, too, can be reflected back to the main window.

As with CtlView, the best way to learn from DlgDemo is to run the program and watch the different WM_COMMAND and WM_NOTIFY messages that are sent by the controls and the property sheet. Opening the property sheet and switching between the pages results in a flood of WM_NOTIFY messages informing the individual pages of what's happening. It's also interesting to note that when the OK button is pressed on the property sheet, the PSN_APPLY messages are sent only to property pages that have been displayed.

The menu handlers that display the Print and Color common dialogs work with a bit of a twist. Since the Palm-size PC doesn't support these dialogs, DlgDemo can't call the functions directly. That would result in these two functions being implicitly linked at run time. Since the Palm-size PC doesn't have these common dialogs and therefore these functions, Windows CE wouldn't be able to resolve the implicit links to all the functions in the program and therefore the program wouldn't be able to load. So, instead of calling the functions directly, you explicitly link these functions in *InitApp* by loading the common dialog DLL using *LoadLibrary* and getting pointers to the functions using *GetProcAddress*. If DlgDemo is running on a Palm-size PC, the GetProcAddress function fails and returns 0 for the function pointer. In *OnCreateMain*, a check is made to see whether these function pointers are 0, and if so, the Print and Color menu items are disabled. In the menu handler functions *DoMainCommandColor* and *DoMainCommandPrint*, the function pointers returned by *GetProcAddress* are used to call the functions. This extra effort isn't necessary if you know your program will run only on a system that supports a specific set of functions, but every once in a while, this technique comes in handy.

CONCLUSION

This chapter has covered a huge amount of ground, from basic child windows to controls and on to dialog boxes and property sheets. My goal wasn't to teach everything there is to know about these topics. Instead, I've tried to introduce these program elements, provide a few examples, and point out the subtle differences between the way they're handled by Windows CE and the desktop versions of Windows.

This chapter also marks the end of the introductory section, "Windows Programming Basics." In these first four chapters, I've talked about fundamental Windows programming while also using a basic Windows CE application to introduce the concepts of the system message queue, windows, and messages. I've given you an overview of how to paint text and graphics in a window and how to query the user for input. Finally, I talked about the windows hierarchy, controls, and dialog boxes. For the remainder of the book, I move from description of the elements common to both Windows CE and the desktop versions of Windows to the unique nature of Windows CE programming. I begin this process in Chapter 5 by talking about another set of controls, the *common controls,* this time with an emphasis on controls unique to Windows CE.

Part II

WINDOWS CE BASICS

Chapter 5

Common Controls and Windows CE

As Microsoft Windows matured as an operating system, it became apparent that the basic controls provided by Windows were insufficient for the sophisticated user interfaces that users demanded. Microsoft developed a series of additional controls, called *common controls,* for their internal applications and later made the dynamic link library (DLL) containing the controls available to application developers. Starting with Microsoft Windows 95 and Microsoft Windows NT 3.5, the common control library was bundled with the operating system. (Although this hasn't stopped Microsoft from making interim releases of the DLL as the common control library was enhanced.) With each release of the common control DLL, new controls and new features are added to old controls. As a group, the common controls are less mature than the standard Windows controls and therefore show greater differences between implementations across the various versions of Windows. These differences aren't just between Microsoft Windows CE and other versions of Windows, but also between Windows NT, Windows 95, and Microsoft Windows 98. The functionality of the common controls in Windows CE tracks most closely with the common controls delivered with Windows 98, although not all of the Windows 98 features are supported.

It isn't the goal of this chapter to cover in depth all the common controls. That would take an entire book. Instead, I'll cover the controls and features of controls the Windows CE programmer will most often need when writing Windows CE applications. I'll start with the command bar and then look at the month calendar and time and date picker controls. Finally, I'll finish up with the list view control. By the end of the chapter, you might not know every common control inside and out, but you will be able to see how the common controls work in general. And you'll have the background to look at the documentation and understand the common controls not covered.

PROGRAMMING COMMON CONTROLS

Since the common controls are separate from the core operating system, the DLL that contains them must be initialized before any of the common controls can be used. Under all versions of Windows, including Windows CE, you can call the function

```
void InitCommonControls (void);
```

to load the library and register all the common control classes.

Another function added recently to the common control library and supported by Windows CE is this one:

```
BOOL InitCommonControlsEx (LPINITCOMMONCONTROLSEX lpInitCtrls);
```

This function allows an application to load and initialize only selected common controls. This function is handy under Windows CE because loading only the necessary controls can reduce the memory impact. The only parameter to this function is a two-field structure that contains a size field and a field that contains a set of flags indicating which common controls should be registered. Figure 5-1 shows the available flags and their associated controls.

Flag	*Control Classes Initialized*
ICC_BAR_CLASSES	Toolbar
	Status bar
	Trackbar
	Command bar
ICC_COOL_CLASSES	Rebar
ICC_DATE_CLASSES	Date and time picker Month calendar control
ICC_LISTVIEW_CLASSES	List view
	Header control
ICC_PROGRESS_CLASS	Progress bar control
ICC_TAB_CLASSES	Tab control
ICC_TREEVIEW_CLASSES	Tree view control
ICC_UPDOWN_CLASS	Up-down control

Figure 5-1. *Flags for selected common controls.*

Once the common control DLL has been initialized, these controls can be treated as any other control. But since the common controls aren't formally part of the Windows core functionality, an additional include file, commctrl.h, must be included.

The programming interface for the common controls is similar to standard Windows controls. Each of the controls has a set of custom style flags that configure the look and behavior of the control. Messages specific to each control are sent to configure, manipulate, and cause the control to perform actions. One major difference between the standard windows controls and common controls is that notifications of events or requests for service are sent via WM_NOTIFY messages instead of WM_COMMAND messages as in the standard controls. This technique allows the notifications to contain much more information than would be allowed using WM_COMMAND message notifications.

One additional difference when programming common controls is that most of the control-specific messages that can be sent to the common controls have predefined macros that make sending the message look as if your application is calling a function. So, instead of using an LVM_INSERTITEM message to a list view control to insert an item, as in

```
nIndex = (int) SendMessage (hwndLV, LVM_INSERTITEM, 0, (LPARAM)&lvi);
```

an application could just as easily have used the line:

```
nIndex = ListView_InsertItem (hwndLV, &lvi);
```

There's no functional difference between the two lines; the advantage of these macros is clarity. The macros themselves are defined in commctrl.h along with the other definitions required for programming the common controls. One problem with the macros is that the compiler doesn't perform the type checking on the parameters that would normally occur if the macro were an actual function. This is also true of the *SendMessage* technique, in which the parameters must be typed as WPARAM and LPARAM types, but at least with messages the lack of type checking is obvious. All in all though, the macro route provides better readability. One exception to this system of macros are the calls made to the command bar control and the command bands control. Those controls actually have a number of true functions in addition to a large set of macro-wrapped messages. As a rule, I'll talk about messages as messages, not as their macro equivalents. That should help differentiate what is a message or macro and what is a true function.

THE COMMON CONTROLS

Windows CE's special niche—small personal productivity devices—has driven the requirements for the common controls in Windows CE. The frequent need for time and date references for schedule and task management applications has led to inclusion of the date and time picker control and the month calendar control. The small screens of personal productivity devices inspired the space-saving *command bar*. Mating the command bar with the *rebar control* that was created for Internet Explorer 3.0 has produced the *command bands control*. The command bands control provides even more room for menus, buttons, and other controls across the top of a Windows CE application. You've seen glimpses of the command bar control in Chapter 1 and again in Chapters 3 and 4. It's time you were formally introduced.

The Command Bar

Briefly, a *command bar* control combines a menu and a toolbar. This combination is valuable because, as I've pointed out before, the combination of a menu and toolbar on one line saves screen real estate on space-constrained Windows CE displays. To the programmer, the command bar looks like a toolbar with a number of helper functions that make programming the command bar a breeze. In addition to the command bar functions, you can also use most toolbar messages when you're working with command bars.

The command bands control was added to Windows CE in version 2.0. A command bands control is a rebar control that, by default, contains a command bar in each band of the control. The rebar control is a fairly new common control; it's a container of controls that the user can drag around the application window. It was previously known as a *Cool Bar* when it first appeared in the common control DLL delivered with Internet Explorer 3.0. Given that command bands are nothing more than command bars in a rebar control, knowing how to program a command bar is most of the battle when learning how to program the command bands control.

Creating a command bar

You build a command bar in a number of steps, each defined by a particular function. The command bar is created, the menu is added, buttons are added, other controls are added, tool tips are added, and finally, the Close and Help buttons are appended to the right side of the command bar.

You begin the process of creating a command bar with a call to

```
HWND CommandBar_Create (HINSTANCE hInst, HWND hwndParent,
                        int idCmdBar);
```

The function requires the program's instance handle, the handle of the parent window, and an ID value for the control. If successful, the function returns the handle to the newly created command bar control. But a bare command bar isn't much use to the application. It takes a menu and a few buttons jazz it up.

Command bar menus

You can add a menu to a command bar by calling one of two functions. The first function is this:

```
BOOL CommandBar_InsertMenubar (HWND hwndCB, HINSTANCE hInst,
                              WORD idMenu, int iButton);
```

The first two parameters of this function are the handle of the command bar and the instance handle of the application. The *idMenu* parameter is the resource ID of the menu to be loaded into the command bar. The last parameter is the index of the button to the immediate left of the menu. Because the Windows CE guidelines specify that the menu should be at the left end of the command bar, this parameter should be set to 0, which indicates that all the buttons are to the right of the menu.

A shortcoming of the *CommandBar_InsertMenubar* function is that it requires the menu to be loaded from a resource. You can't configure the menu on the fly. Of course, it would be possible to load a dummy menu and manipulate the contents of the menu with the various menu functions, but here's an easier method.

The function

```
BOOL CommandBar_InsertMenubarEx (HWND hwndCB, HINSTANCE hInst,
                                LPTSTR pszMenu, int iButton);
```

was added in Windows CE 2.0. The difference between *CommandBar_InsertMenubarEx* and *CommandBar_InsertMenubar* is the change in the third parameter, *pszMenu*. This parameter can be either the name of a menu resource or the handle to a menu previously created by the program. If the *pszMenu* parameter is a menu handle, the *hInst* parameter must be NULL.

Once a menu has been loaded into a command bar, the handle to the menu can be retrieved at any time using

```
HMENU CommandBar_GetMenu (HWND hwndCB, int iButton);
```

The second parameter, *iButton*, is the index of the button to the immediate left of the menu. This mechanism provides the ability to identify more than one menu on the command bar. However, given the Windows CE design guidelines, you should see only one menu on the bar. With the menu handle, you can manipulate the structure of the menu using the many menu functions available.

If an application modifies the menu on the command bar, the application must call

```
BOOL CommandBar_DrawMenuBar (HWND hwndCB, int iButton);
```

which forces the menu on the command bar to be redrawn. Here again, the parameters are the handle to the command bar and the index of the button to the left of the menu. Under Windows CE, you must use *CommandBar_DrawMenuBar* instead of *DrawMenuBar*, which is the standard function used to redraw the menu under other versions of Windows.

Command bar buttons

Adding buttons to a command bar is a two-step process, and is similar to adding buttons to a toolbar. First the bitmap images for the buttons must be added to the command bar. Second the buttons are added, with each of the buttons referencing one of the images in the bitmap list that was previously added.

The command bar maintains its own list of bitmaps for the buttons in an internal image list. Bitmaps can be added to this image list one at a time or as a group of images contained in a long and narrow bitmap. For example, for a bitmap to contain four 16-by-15-bit images, the dimensions of the bitmap added to the command bar would be 64 by 15 bits. Figure 5-2 shows this bitmap image layout.

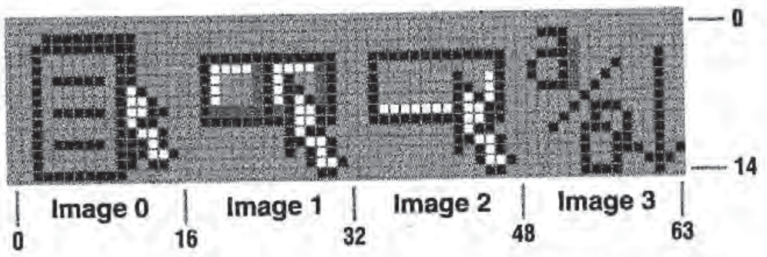


Figure 5-2. *Layout of a bitmap that contains four 16-by-15-bit images.*

Loading a image bitmap is accomplished using

```
int CommandBar_AddBitmap (HWND hwndCB, HINSTANCE hInst, int idBitmap,
                          int iNumImages, int iReserved, int iReserved);
```

This first two parameters are, as is usual with a command bar function, the handle to the command bar and the instance handle of the executable. The third parameter, *idBitmap*, is the resource ID of the bitmap image. The fourth parameter, *iNumImages*, should contain the number of images in the bitmap being loaded. Multiple bitmap images can be loaded into the same command bar by calling *CommandBar_AddBitmap* as many times as is needed.

Two predefined bitmaps provide a number of images that are commonly used in command bars and toolbars. You load these images by setting the *hInst* parameter in *CommandBar_AddBitmap* to HINST_COMMCTRL and setting the *idBitmap* parameter to either IDB_STD_SMALL_COLOR or IDB_VIEW_SMALL_COLOR. The images contained in these bitmaps are shown in Figure 5-3. The buttons on the top line contain the bitmaps from the standard bitmap while the second-line buttons contain the bitmaps from the standard view bitmap.

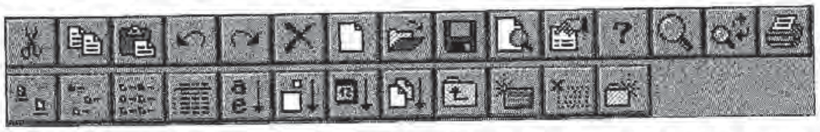


Figure 5-3. *Images in the two standard bitmaps provided by the common control DLL.*

The index values to these images are defined in commctrl.h, so you don't need to know the exact order in the bitmaps. The constants are

Constants to access the standard bitmap

STD_CUT	Edit/Cut button image
STD_COPY	Edit/Copy button image
STD_PASTE	Edit/Paste button image
STD_UNDO	Edit/Undo button image
STD_REDOW	Edit/Redo button image
STD_DELETE	Edit/Delete button image
STD_FILENEW	File/New button image
STD_FILEOPEN	File/Open button image
STD_FILESAVE	File/Save button image
STD_PRINTPRE	Print preview button image
STD_PROPERTIES	Properties button image
STD_HELP	Help button (Use Commandbar_Addadornments function to add a help button to the command bar.)
STD_FIND	Find button image
STD_REPLACE	Replace button image
STD_PRINT	Print button image

Constants to access the standard view bitmap

VIEW_LARGEICONS	View/Large Icons button image
VIEW_SMALLICONS	View/Small Icons button image
VIEW_LIST	View/List button image
VIEW_DETAILS	View/Details button image
VIEW_SORTNAME	Sort by name button image
VIEW_SORTSIZE	Sort by size button image
VIEW_SORTDATE	Sort by date button image
VIEW_SORTTYPE	Sort by type button image
VIEW_PARENTFOLDER	Go to Parent folder button image
VIEW_NETCONNECT	Connect network drive button image
VIEW_NETDISCONNECT	Disconnect network drive button image
VIEW_NEWFOLDER	Create new folder button image

Referencing images

The images loaded into the command bar are referenced by their index into the list of images. For example, if the bitmap loaded contained five images, and the image to be referenced was the fourth image into the bitmap, the zero-based index value would be 3.

If more than one set of bitmap images was added to the command bar using multiple calls to *CommandBar_AddBitmap*, the images' subsequent lists are referenced according to the previous count of images plus the index into that list. For example, if two calls were made to *CommandBar_AddBitmap* to add two sets of images, with the first call adding five images and the second adding four images, the third image of the second set would be referenced with the total number of images added in the first bitmap (5) plus the index into the second bitmap (2) resulting in an index value of 5 + 2 = 7.

Once the bitmaps have been loaded, the buttons can be added using one of two functions. The first function is this one:

```
BOOL CommandBar_AddButtons (HWND hwndCB, UINT uNumButtons,
                            LPTBBUTTON lpButtons);
```

CommandBar_AddButtons adds a series of buttons to the command bar at one time. The function is passed a count of buttons and a pointer to an array of TBBUTTON structures. Each element of the array describes one button. The TBBUTTON structure is defined as the following:

```
typedef struct {
    int iBitmap;
    int idCommand;
    BYTE fsState;
    BYTE fsStyle;
    DWORD dwData;
    int iString;
} TBBUTTON;
```

The *iBitmap* field specifies the bitmap image to be used by the button. This is, as I just explained, the zero-based index into the list of images. The second parameter is the command ID of the button. This ID value is sent via a WM_COMMAND message to the parent when a user clicks the button.

The *fsState* field specifies the initial state of the button. The allowable values in this field are the following:

- *TBSTATE_ENABLED* The button is enabled. If this flag isn't specified, the button is disabled and is grayed.
- *TBSTATE_HIDDEN* The button isn't visible on the command bar.
- *TBSTATE_PRESSED* This button is displayed in a depressed state.
- *TBSTATE_CHECKED* The button is initially checked. This state can be used only if the button has the TBSTYLE_CHECKED style.
- *TBSTATE_INDETERMINATE* The button is grayed.

One last flag is specified in the documentation, TBSTATE_WRAP, but it doesn't have a valid use in a command bar. This flag is used by toolbars when a toolbar wraps across more than one line.

The *fsStyle* field specifies the initial style of the button, which defines how the button acts. The button can be defined as a standard push button, a check button, a drop-down button, or a check button that resembles a radio button but allows only one button in a group to be checked. The possible flags for the *fsStyle* field are the following:

- *TBSTYLE_BUTTON* The button looks like a standard push button.
- *TBSTYLE_CHECK* The button is a check button that toggles between checked and unchecked states each time the user clicks the button.
- *TBSTYLE_GROUP* Defines the start of a group of buttons.
- *TBSTYLE_CHECKGROUP* The button is a member of a group of check buttons that act like a radio buttons in that only one button in the group is checked at any one time.
- *TBSTYLE_DROPDOWN* The button is a drop-down list button.
- *TBSTYLE_AUTOSIZE* The button's size is defined by the button text.
- *TBSTYLE_SEP* Defines a separator (instead of a button) that inserts a small space between buttons.

The *dwData* field of the TBBUTTON structure is an application-defined value. This value can be set and queried by the application using the TB_SETBUTTONINFO and TB_ GETBUTTONINFO messages. The *iString* field defines the index into the command bar string array that contains the text for the button. The *iString* field can also be filled with a pointer to a string that contains the text for the button.

The other function that adds buttons to a command bar is this one:

```
BOOL CommandBar_InsertButton (HWND hwndCB, int iButton,
                              LPTBBUTTON lpButton);
```

This function inserts one button into the command bar to the left of the button referenced by the *iButton* parameter. The parameters in this function mimic the parameters in *CommandBar_AddButtons* with the exception that the *lpButton* parameter points to a single TBBUTTON structure. The *iButton* parameter specifies the position on the command bar of the new button.

Working with command bar buttons

When a user presses a command bar button other than a drop-down button, the command bar sends a WM_COMMAND message to the parent window of the command bar. So, handling button clicks on the command bar is just like handling menu commands. In fact, since many of the buttons on the command bar have menu command equivalents, it's customary to use the same command IDs for the buttons and the like functioning menus, thus removing the need for any special processing for the command bar buttons.

The command bar maintains the checked and unchecked state of check and checkgroup buttons. After the buttons have been added to the command bar, their states can be queried or set using two messages, TB_ISBUTTONCHECKED and TB_CHECKBUTTON. (The TB_ prefix in these messages indicates the close relationship between the command bar and the toolbar controls.) The TB_ISBUTTONCHECKED message is sent with the ID of the button to be queried passed in the *wParam* parameter this way:

```
fChecked = SendMessage (hwndCB, TB_ISBUTTONCHECKED, wID, 0);
```

where *hwndCB* is the handle to the command bar containing the button. If the return value from the TB_ISBUTTONCHECKED message is nonzero, the button is checked. To place a button in the checked state, send a TB_CHECKBUTTON message to the command bar, as in

```
SendMessage (hwndCB, TB_CHECKBUTTON, wID, TRUE);
```

To uncheck a checked button, replace the TRUE value in *lParam* with FALSE.

A new look for disabled buttons

Windows CE allows you to easily modify the way a command bar or toolbar button looks when the button is disabled. Command bars and toolbars maintain two image lists: the standard image list that I described previously and a disabled image list used to store bitmaps that you can employ for disabled buttons.

To use this new feature, you need to create and load a second image list for disabled buttons. The easiest way to do this is to create the image list for the normal states of the buttons using the techniques I described when I talked about *CommandBar_AddBitmap*. (Image lists in toolbars are loaded with the message TB_LOADIMAGES.) Once that image list complete, simply copy the original image list and modify the bitmaps of the images to create disabled counterparts to the original images. Then load the new image list back into the command bar or toolbar. A short code fragment that accomplishes this chore is shown below.

```
HBITMAP hBmp, hMask;
HIMAGELIST hilDisabled, hilEnabled;

// Load the bitmap and mask to be used in the disabled image list.
hBmp = LoadBitmap (hInst, TEXT ("DisCross"));
hMask = LoadBitmap (hInst, TEXT ("DisMask"));
```

```
// Get the std image list and copy it.
hilEnabled = (HIMAGELIST)SendMessage (hwndCB, TB_GETIMAGELIST, 0, 0);
hilDisabled = ImageList_Duplicate (hilEnabled);

// Replace one bitmap in the disabled list.
ImageList_Replace (hilDisabled, VIEW_LIST, hBmp, hMask);

// Set the disabled image list.
SendMessage (hwndCB, TB_SETDISABLEDIMAGELIST, 0, (LPARAM) hilDisabled);
```

The code fragment first loads a bitmap and a mask bitmap that will replace one of the images in the disabled image list. You retrieve the current image list by sending a TB_GETIMAGELIST message to the command bar, and then you duplicate it using *ImageList_Duplicate*. One image in the image list is then replaced by the bitmap that was loaded earlier.

This example replaces only one image, but in a real-world example many images might be replaced. If all the images were replaced, it might be easier to build the disabled image list from scratch instead of copying the standard image list and replacing a few bitmaps in it. Once the new image list is created, you load it into the command bar by sending a TB_SETDISABLEDIMAGELIST message. The code that I just showed you works just as well for toolbars under Windows CE as it does for command bars.

Drop-down buttons

The drop-down list button is a more complex animal than the standard button on a command bar. The button looks to the user like a button that, when pressed, displays a list of items for the user to select from. To the programmer, a drop-down button is actually a combination of a button and a menu that is displayed when the user clicks on the button. Unfortunately, the command bar does little to support a drop-down button except to modify the button appearance to indicate that the button is a drop-down button and to send a special notification when the button is clicked by the user. It's up to the application to display the menu.

The notification of the user clicking a drop-down button is sent to the parent window of the command bar by a WM_NOTIFY message with a notification value of TBN_DROPDOWN. When the parent window receives the TBN_DROPDOWN notification, it must create a pop-up menu immediately below the drop-down button identified in the notification. The menu is filled by the parent window with whatever selections are appropriate for the button. When one of the menu items is selected, the menu will send a WM_COMMAND message indicating the menu item picked and the menu will be dismissed. The easiest way to understand how to handle a drop-down button notification is to look at the following procedure that handles a TBN_DROPDOWN notification.

```
LRESULT DoNotifyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    LPNMHDR pNotifyHeader;
    LPNMTOOLBAR pNotifyToolBar;
    RECT rect;
    TPMPARAMS tpm;
    HMENU hMenu;

    // Get pointer to notify message header.
    pNotifyHeader = (LPNMHDR)lParam;

    if (pNotifyHeader->code == TBN_DROPDOWN) {

        // Get pointer to toolbar notify structure.
        pNotifyToolBar = (LPNMTOOLBAR)lParam;

        // Get the rectangle of the drop-down button.
        SendMessage (pNotifyHeader->hwndFrom, TB_GETRECT,
                     pNotifyToolBar->iItem, (LPARAM)&rect);

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

        // Prevent the menu from covering the button.
        tpm.cbSize = sizeof (tpm);
        CopyRect (&tpm.rcExclude, &rect);

        // Load the menu resource to display under the button.
        hMenu = GetSubMenu (LoadMenu (hInst, TEXT ("popmenu")),0);

        // Display the menu.  This function returns after the
        // user makes a selection or dismisses the menu.
        TrackPopupMenuEx (hMenu, TPM_LEFTALIGN | TPM_VERTICAL,
                          rect.left, rect.bottom, hWnd, &tpm);
    }
    return 0;
}
```

After the code determines that the message is a TBN_DROPDOWN notification, the first task of the notification handler code is to get the rectangle of the drop-down button. The rectangle is queried so that the drop-down menu can be positioned immediately below the button. To do this, the routine sends a TB_GETRECT message to the command bar with the ID of the drop-down button passed in *wParam* and a pointer to a rectangle structure in *lParam*.

Since the rectangle returned is in the coordinate base of the parent window, and pop-up menus are positioned in screen coordinates, the coordinates must be converted from one basis to the other. You accomplish this using the function

```
MapWindowPoints (HWND hwndFrom, HWND hwndTo,
                 LPPOINT lppoints, UINT cPoints);
```

The first parameter is the handle of the window in which the coordinates are originally based. The second parameter is the handle of the window to which you want to map the coordinates. The third parameter is a pointer to an array of points to be translated; the last parameter is the number of points in the array. In the routine I just showed you, the window handles are the command bar handle and the desktop window handle, respectively.

Once the rectangle has been translated into desktop coordinates, the pop-up, or context, menu can be created. You do this by first loading the menu from the resource, then displaying the menu with a call to *TrackPopupMenuEx*. That function is prototyped as

```
BOOL TrackPopupMenuEx (HMENU hmenu, UINT fuFlags, int x, int y,
                       HWND hwnd, LPTPMPARAMS lptpm);
```

The *hMenu* parameter is the handle of the menu to be displayed. The *hwnd* parameter identifies the window to receive the WM_COMMAND message if a menu item is selected. The TPMPARAMS structure contains a rectangle that won't be covered up by the menu when it is displayed. For our purposes, this rectangle is set to the dimensions of the drop-down button so that the button won't be covered by the pop-up menu. The *fuFlags* field can contain a number of values that define the placement of the menu. For drop-down buttons, the only flag needed is TPM_VERTICAL. If TMP_VERTICAL is set, the menu leaves uncovered as much of the horizontal area of the exclude rectangle as possible. The *TrackPopupMenuEx* function doesn't return until an item on the menu has been selected or the menu has been dismissed by the user tapping on another part of the screen.

Combo boxes on the command bar

Combo boxes on a command bar are much easier to implement than drop-down buttons. You add a combo box by calling

```
HWND CommandBar_InsertComboBox (HWND hwndCB, HINSTANCE hInst,
                                int iWidth, UINT dwStyle,
                                WORD idComboBox,
                                int iButton);
```

This function inserts a combo box on the command bar to the left of the button indicated by the *iButton* parameter. The width of the combo box is specified, in pixels, by the *iWidth* parameter. The *dwStyle* parameter specifies the style of the combo box.

The allowable style flags are any valid Windows CE combo box style and window styles. The function automatically adds the WS_CHILD and WS_VISIBLE flags when creating the combo box. The *idComboBox* parameter is the ID for the combo box that will be used when WM_COMMAND messages are sent notifying the parent window of a combo box event. Experienced Windows programmers will be happy to know that *CommandBar_InsertComboBox* takes care of all the "parenting" problems that occur when a control is added to a standard Windows toolbar. That one function call is all that is needed to create a properly functioning combo box on the command bar.

Once a combo box is created, you program it on the command bar the same way you would a stand-alone combo box. Since the combo box is a child of the command bar, you must query the window handle of the combo box by passing the handle of the command bar to *GetDlgItem* with the ID value of the combo box, as in the following code:

```
hwndCombobox = GetDlgItem (GetDlgItem (hWnd, IDC_CMDBAR),
                           IDC_COMBO));
```

However, the WM_COMMAND messages from the combo box are sent directly to the parent of the command bar, so handling combo box events is identical to handling them from a combo box created as a child of the application's top-level window.

Command bar tool tips

Tool tips are small windows that display descriptive text that labels a command bar button when the stylus is held down over the control. Tool tips under Windows CE are implemented in a completely different way from how they're implemented under Windows 98 and Windows NT.

You add tool tips to a command bar by using this function:

```
BOOL CommandBar_AddToolTips (HWND hwndCB, UINT uNumToolTips,
                             LPTSTR lpToolTips);
```

The *lpToolTips* parameter must point to an array of pointers to strings. The *uNumToolTips* parameter should be set to the number of elements in the string pointer array. The *CommandBar_AddToolTips* function doesn't copy the strings into its own storage. Instead, the location of the string array is saved. This means that the block of memory containing the string array must not be released until the command bar is destroyed.

Each string in the array becomes the tool tip text for a control or separator on the command bar excluding the menu. The first string in the array becomes the tool tip for the first control or separator, the second string is assigned to the second control or separator, and so on. So, even though combo boxes and separators don't display tool tips, they must have entries in the string array so that all the text lines up with the proper buttons.

Other command bar functions

A number of other functions assist in command bar management. The *CommandBar_Height* function returns the height of the command bar and is used in all the example programs that use the command bar. Likewise, the *CommandBar_AddAdornments* function is also used whenever a command bar is used. This function, prototyped as

```
BOOL CommandBar_AddAdornments (HWND hwndCB, DWORD dwFlags,
                               DWORD dwReserved);
```

places a Close button and, if you want, a Help button and an OK button on the extreme right of the command bar. You pass a CMDBAR_HELP flag to the *dwFlags* parameter to add a Help button, and you pass a CMDBAR_OK flag to add an OK button.

The Help button is treated differently from other buttons on the command bar. When the Help button is pressed, the command bar sends a WM_HELP message to the owner of the command bar instead of the standard WM_COMMAND message. The OK button's action is more traditional. When it is pressed, a WM_COMMAND message is sent with a control ID of IDOK. *CommandBar_AddAdornments* must be called after all other conrols of the command bar have been added.

A command bar can be hidden by calling

```
BOOL CommandBar_Show (HWND hwndCB, BOOL fShow);
```

The *fShow* parameter is set to TRUE to show the command bar and FALSE to hide a command bar. The visibility of a command bar can be queried with this:

```
BOOL CommandBar_IsVisible (HWND hwndCB);
```

Finally, a command bar can be destroyed using this:

```
void CommandBar_Destroy (HWND hwndCB);
```

Although a command bar is automatically destroyed when its parent window is destroyed, sometimes it's more convenient to destroy a command bar manually. This is often done if a new command bar is needed for a different mode of the application. Of course, you can create multiple command bars, hiding all but one and switching between them by showing only one at a time, but this isn't good programming practice under Windows CE because all those hidden command bars take up valuable RAM that could be used elsewhere. The proper method is to destroy and create command bars on the fly. You can create a command bar fast enough so that a user shouldn't notice any delay in the application when a new command bar is created.

Design guidelines for command bars

Because command bars are a major element of Windows CE applications, it's not surprising that Microsoft has a rather strong set of rules for their use. Many of these rules are similar to the design guidelines for other versions of Windows, such as the recommendations for the ordering of main menu items and the use of tool tips. Most of these guidelines are already second nature for Windows programmers.

The menu should be the left-most item on the command bar. The order of the main menu items should be from left to right: File, Edit, View, Insert, Format, Tools, and Window. Of course, most applications have all of those menu items but the order of the items used should follow the suggested order. For buttons, the order is from left to right; New, Open, Save, and Print for file actions; and Bold, Italic, and Underline for font style.

The CmdBar Example Program

The CmdBar example demonstrates the basics of command bar operation. On startup, the example creates a bar with only a menu and a close button. Selecting the different items from the view menu creates various command bars showing the capabilities of the command bar control. The source code for CmdBar is shown in Figure 5-4.

CmdBar.rc

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

//----------------------------------------------------------------------
// Icons and bitmaps
//
ID_ICON       ICON   "cmdbar.ico"       // Program icon
DisCross      BITMAP "cross.bmp"        // Disabled button image
DisMask       BITMAP "mask.bmp"         // Disabled button image mask
SortDropBtn   BITMAP "sortdrop.bmp"     // Sort drop-down button image

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

Figure 5-4. *The CmdBar program.*

```
    POPUP "&View"
    BEGIN
        MENUITEM "&Standard",                       IDM_STDBAR
        MENUITEM "&View",                           IDM_VIEWBAR
        MENUITEM "&Combination",                    IDM_COMBOBAR
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...",                       IDM_ABOUT
    END
END

popmenu MENU DISCARDABLE
BEGIN
    POPUP "&Sort"
    BEGIN
        MENUITEM "&Name",                           IDC_SNAME
        MENUITEM "&Type",                           IDC_STYPE
        MENUITEM "&Size",                           IDC_SSIZE
        MENUITEM "&Date",                           IDC_SDATE
    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 "CmdBar - Written for the book Programming Windows \
           CE Copyright 1998 Douglas Boling"
                                         -1,  40,   5, 110,  30
END
```

CmdBar.h

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

(continued)

Figure 5-4. *continued*

```
#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 band ID
#define  ID_ICON             10                 // Icon resource ID
#define  ID_MENU             11                 // Main menu resource ID
#define  IDC_COMBO           12                 // Combo box on cmd bar ID

// Menu item IDs
#define  IDM_EXIT            101                // File menu
#define  IDM_STDBAR          111                // View menu
#define  IDM_VIEWBAR         112
#define  IDM_COMBOBAR        113
#define  IDM_ABOUT           120                // Help menu

// Command bar button IDs
#define  IDC_NEW             201
#define  IDC_OPEN            202
#define  IDC_SAVE            203
#define  IDC_CUT             204
#define  IDC_COPY            205
#define  IDC_PASTE           206
#define  IDC_PROP            207

#define  IDC_LICON           301
#define  IDC_SICON           302
#define  IDC_LIST            303
#define  IDC_RPT             304
#define  IDC_SNAME           305
#define  IDC_STYPE           306
#define  IDC_SSIZE           307
#define  IDC_SDATE           308
```

```
#define  IDC_DPSORT           350

#define  STD_BMPS             (STD_PRINT+1)      // Number of bmps in
                                                 // std imglist
#define  VIEW_BMPS            (VIEW_NEWFOLDER+1) // Number of bmps in
                                                 // view imglist

//----------------------------------------------------------------------
// Function prototypes
//
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 DoCommandMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoNotifyMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);

// Command functions
LPARAM DoMainCommandExit (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandVStd (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandVView (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandVCombo (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandAbout (HWND, WORD, HWND, WORD);

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

CmdBar.c

```
//======================================================================
// CmdBar - Command bar 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 "CmdBar.h"                  // Program-specific stuff
```

(continued)

Figure 5-4. *continued*

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

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

// Command Message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDM_EXIT, DoMainCommandExit,
    IDM_STDBAR, DoMainCommandVStd,
    IDM_VIEWBAR, DoMainCommandVView,
    IDM_COMBOBAR, DoMainCommandVCombo,
    IDM_ABOUT, DoMainCommandAbout,
};

// Standard file bar button structure
const TBBUTTON tbCBStdBtns[] = {
//  BitmapIndex       Command    State        Style        UserData String
    {0,               0,         0,           TBSTYLE_SEP,       0,   0},
    {STD_FILENEW,     IDC_NEW,   TBSTATE_ENABLED,
                                              TBSTYLE_BUTTON,    0,   0},
    {STD_FILEOPEN,    IDC_OPEN,  TBSTATE_ENABLED,
                                              TBSTYLE_BUTTON,    0,   0},
    {STD_FILESAVE,    IDC_SAVE,  TBSTATE_ENABLED,
                                              TBSTYLE_BUTTON,    0,   0},
    {0,                 0,       0,           TBSTYLE_SEP,       0,   0},
    {STD_CUT,         IDC_CUT,   TBSTATE_ENABLED,
                                              TBSTYLE_BUTTON,    0,   0},
    {STD_COPY,        IDC_COPY,  TBSTATE_ENABLED,
                                              TBSTYLE_BUTTON,    0,   0},
    {STD_PASTE,       IDC_PASTE, TBSTATE_ENABLED,
                                              TBSTYLE_BUTTON,    0,   0},
    {0,                 0,       0,           TBSTYLE_SEP,       0,   0},
    {STD_PROPERTIES,  IDC_PROP,  TBSTATE_ENABLED,
        TBSTYLE_BUTTON,    0,   0}
};
```

```
// Standard view bar button structure
const TBBUTTON tbCBViewBtns[] = {
//  BitmapIndex        Command     State         Style         UserData String
    {0,                0,          0,            TBSTYLE_SEP,         0,  0},
    {VIEW_LARGEICONS,  IDC_LICON,  TBSTATE_ENABLED | TBSTATE_CHECKED,
                                                 TBSTYLE_CHECKGROUP,  0,  0},
    {VIEW_SMALLICONS,  IDC_SICON,  TBSTATE_ENABLED,
                                                 TBSTYLE_CHECKGROUP,  0,  0},
    {VIEW_LIST,        IDC_LIST,   0,            TBSTYLE_CHECKGROUP,  0,  0},
    {VIEW_DETAILS,     IDC_RPT,    TBSTATE_ENABLED,
                                                 TBSTYLE_CHECKGROUP,  0,  0},
    {0,                0,          TBSTATE_ENABLED,
                                                 TBSTYLE_SEP,         0,  0},
    {VIEW_SORTNAME,    IDC_SNAME,  TBSTATE_ENABLED | TBSTATE_CHECKED,
                                                 TBSTYLE_CHECKGROUP,  0,  0},
    {VIEW_SORTTYPE,    IDC_STYPE,  TBSTATE_ENABLED,
                                                 TBSTYLE_CHECKGROUP,  0,  0},
    {VIEW_SORTSIZE,    IDC_SSIZE,  TBSTATE_ENABLED,
                                                 TBSTYLE_CHECKGROUP,  0,  0},
    {VIEW_SORTDATE,    IDC_SDATE,  TBSTATE_ENABLED,
                                                 TBSTYLE_CHECKGROUP,  0,  0},
    {0,                0,          0,            TBSTYLE_SEP,         0,  0},
};
// Tooltip string list for view bar
const TCHAR *pViewTips[] = {TEXT (""),
                            TEXT ("Large"),
                            TEXT ("Small"),
                            TEXT ("List"),
                            TEXT ("Details"),
                            TEXT (""),
                            TEXT ("Sort by Name"),
                            TEXT ("Sort by Type"),
                            TEXT ("Sort by Size"),
                            TEXT ("Sort by Date"),
};

// Combination standard and view bar button structure
const TBBUTTON tbCBCmboBtns[] = {
//  BitmapIndex        Command     State         Style         UserData String
    {0,                0,          0,            TBSTYLE_SEP,         0,  0},
    {STD_FILENEW,      IDC_NEW,    TBSTATE_ENABLED,
                                                 TBSTYLE_BUTTON,      0,  0},
    {STD_FILEOPEN,     IDC_OPEN,   TBSTATE_ENABLED,
                                                 TBSTYLE_BUTTON,      0,  0},
    {STD_PROPERTIES,   IDC_PROP,   TBSTATE_ENABLED,
                                                 TBSTYLE_BUTTON,      0,  0},
```

(continued)

Figure 5-4. *continued*

```
    {0,                 0,          0,            TBSTYLE_SEP,          0,  0},
    {STD_CUT,           IDC_CUT,    TBSTATE_ENABLED,
                                                  TBSTYLE_BUTTON,       0,  0},
    {STD_COPY,          IDC_COPY,   TBSTATE_ENABLED,
                                                  TBSTYLE_BUTTON,       0,  0},
    {STD_PASTE,         IDC_PASTE,  TBSTATE_ENABLED,
                                                  TBSTYLE_BUTTON,       0,  0},
    {0,                 0,          0,            TBSTYLE_SEP,          0,  0},
    {STD_BMPS + VIEW_LARGEICONS,
                        IDC_LICON,  TBSTATE_ENABLED | TBSTATE_CHECKED,
                                                  TBSTYLE_CHECKGROUP, 0,  0},
    {STD_BMPS + VIEW_SMALLICONS,
                        IDC_SICON,  TBSTATE_ENABLED,
                                                  TBSTYLE_CHECKGROUP, 0,  0},
    {STD_BMPS + VIEW_LIST,
                        IDC_LIST,   TBSTATE_ENABLED,
                                                  TBSTYLE_CHECKGROUP, 0,  0},
    {STD_BMPS + VIEW_DETAILS,
                        IDC_RPT,    TBSTATE_ENABLED,
                                                  TBSTYLE_CHECKGROUP, 0,  0},
    {0,                 0,          0,            TBSTYLE_SEP,          0,  0},
    {STD_BMPS + VIEW_BMPS,
                        IDC_DPSORT,TBSTATE_ENABLED,
                                                  TBSTYLE_DROPDOWN,     0,  0}
};

//======================================================================
// 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;
    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,             // Window class
                         TEXT ("CmdBar Demo"), // Window title
                         WS_VISIBLE,           // Style flags
```

(continued)

Figure 5-4. *continued*

```
                          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) {
    HWND hwndCB;

    // 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);
    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 pNotifyHeader;
    LPNMTOOLBAR pNotifyToolBar;
```

(continued)

Figure 5-4. *continued*

```
    RECT rect;
    TPMPARAMS tpm;
    HMENU hMenu;

    // Get pointer to notify message header.
    pNotifyHeader = (LPNMHDR)lParam;

    if (pNotifyHeader->code == TBN_DROPDOWN) {

        // Get pointer to toolbar notify structure.
        pNotifyToolBar = (LPNMTOOLBAR)lParam;

        if (pNotifyToolBar->iItem == IDC_DPSORT) {

            // Get the rectangle of the drop-down button.
            SendMessage (pNotifyHeader->hwndFrom, TB_GETRECT,
                         pNotifyToolBar->iItem, (LPARAM)&rect);

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

            // Prevent the menu from covering the button.
            tpm.cbSize = sizeof (tpm);
            CopyRect (&tpm.rcExclude, &rect);

            hMenu = GetSubMenu (LoadMenu (hInst, TEXT ("popmenu")),0);
            TrackPopupMenuEx (hMenu, TPM_LEFTALIGN | TPM_VERTICAL,
                              rect.left, rect.bottom, hWnd, &tpm);
        }
    }
    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;
}
//----------------------------------------------------------------------
// DoMainCommandViewStd - Displays a standard edit-centric cmd bar
//
LPARAM DoMainCommandVStd (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {
    HWND hwndCB;

    // If a command bar exists, kill it.
    if (hwndCB = GetDlgItem (hWnd, IDC_CMDBAR))
        CommandBar_Destroy (hwndCB);

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

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

    // Insert buttons.
    CommandBar_AddBitmap (hwndCB, HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
                          STD_BMPS, 0, 0);

    CommandBar_AddButtons (hwndCB, dim(tbCBStdBtns), tbCBStdBtns);

    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandVView - Displays a standard edit-centric cmd bar
//
LPARAM DoMainCommandVView (HWND hWnd, WORD idItem, HWND hwndCtl,
                           WORD wNotifyCode) {
    INT i;
    HWND hwndCB;
    TCHAR szTmp[64];
    HBITMAP hBmp, hMask;
    HIMAGELIST hilDisabled, hilEnabled;

    // If a command bar exists, kill it.
    if (hwndCB = GetDlgItem (hWnd, IDC_CMDBAR))
        CommandBar_Destroy (hwndCB);
```

(continued)

Figure 5-4. *continued*

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

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

    // Insert buttons, first add a bitmap and then the buttons.
    CommandBar_AddBitmap (hwndCB, HINST_COMMCTRL, IDB_VIEW_SMALL_COLOR,
                          VIEW_BMPS, 0, 0);

    // Load bitmaps for disabled image.
    hBmp = LoadBitmap (hInst, TEXT ("DisCross"));
    hMask = LoadBitmap (hInst, TEXT ("DisMask"));

    // Get the current image list and copy.
    hilEnabled = (HIMAGELIST)SendMessage (hwndCB, TB_GETIMAGELIST, 0, 0);
    hilDisabled = ImageList_Duplicate (hilEnabled);

    // Replace a button image with the disabled image.
    ImageList_Replace (hilDisabled, VIEW_LIST, hBmp, hMask);

    // Set disabled image list.
    SendMessage (hwndCB,  TB_SETDISABLEDIMAGELIST, 0,
                 (LPARAM)hilDisabled);

    // Add buttons to the command bar.
    CommandBar_AddButtons (hwndCB, dim(tbCBViewBtns), tbCBViewBtns);

    // Add tooltips to the command bar.
    CommandBar_AddToolTips (hwndCB, dim(pViewTips), pViewTips);

    // Add a combo box between the view icons and the sort icons.
    CommandBar_InsertComboBox (hwndCB, hInst, 75,
                               CBS_DROPDOWNLIST | WS_VSCROLL,
                               IDC_COMBO, 6);
    // Fill in combo box.
    for (i = 0; i < 10; i++) {
        wsprintf (szTmp, TEXT ("Item %d"), i);
        SendDlgItemMessage (hwndCB, IDC_COMBO, CB_INSERTSTRING, -1,
                            (LPARAM)szTmp);
    }
    SendDlgItemMessage (hwndCB, IDC_COMBO, CB_SETCURSEL, 0, 0);

    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    return 0;
}
```

```
//----------------------------------------------------------------------
// DoMainCommandVCombo - Displays a combination of file and edit buttons
//
LPARAM DoMainCommandVCombo (HWND hWnd, WORD idItem, HWND hwndCtl,
                            WORD wNotifyCode) {
    HWND hwndCB;

    // If a command bar exists, kill it.
    if (hwndCB = GetDlgItem (hWnd, IDC_CMDBAR))
        CommandBar_Destroy (hwndCB);

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

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

    // Add two bitmap lists plus custom bmp for drop-down button.
    CommandBar_AddBitmap (hwndCB, HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
                          STD_BMPS, 0, 0);
    CommandBar_AddBitmap (hwndCB, HINST_COMMCTRL, IDB_VIEW_SMALL_COLOR,
                          VIEW_BMPS, 0, 0);
    CommandBar_AddBitmap (hwndCB, NULL,
                          (int)LoadBitmap (hInst, TEXT ("SortDropBtn")),
                          1, 0, 0);

    CommandBar_AddButtons (hwndCB, dim(tbCBCmboBtns), tbCBCmboBtns);

    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    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
//
```

(continued)

Figure 5-4. *continued*

```
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;
}
```

Each of the three command bars created in CmdBar demonstrate different capabilities of the command bar control. The first command bar, created in the routine *DoMainCommandVStd* creates a vanilla command bar with a menu and a set of buttons. The button structure for this command bar is defined in the array *tbCBStdBtns,* which is defined near the top of CmdBar.C.

The second command bar, created in the routine *DoMainCommandVView*, contains two groups of checkgroup buttons separated by a combo box. This command bar also demonstrates the use of a separate image for a disabled button. The list view button, the third button on the bar, is disabled. The image for that button in the image list for disabled buttons is replaced with a bitmap that looks like an X.

The *DoMainCommandVCombo* routine creates the third command bar. It uses both the standard and view bitmap images as well as a custom bitmap for a drop-down button. This command bar demonstrates the technique of referencing the images in an image list that contains multiple bitmaps. The drop-down button is serviced by the *OnNotifiyMain* routine where a pop-up menu is loaded and displayed when a TBN_DROPDOWN notification is received.

Command Bands

Command bands appeared in Windows CE 2.0 and are a valuable feature, especially in their capacity to contain separate bands that can be dragged around by a user. Each individual band can have a "gripper" that can be used to drag the band to a new position. A band can be in a minimized state, showing only its gripper and, if you want, an icon; in a maximized state, covering up the other bands on the line; or restored, sharing space with the other bands on the same line. You can even move bands to a new row, creating a multiple-row command band.

The standard use of a command bands control is to break up the elements of a command bar—menu, buttons, and other controls—into separate bands. This allows users to rearrange these elements as they see fit. Users can also expose or overlap separate bands as needed in order to provide a larger total area for menus, buttons, and other controls.

Creating a command bands control

Creating a command bands control is straightforward, if a bit more involved than creating a command bar control. You create the control by calling

```
HWND CommandBands_Create (HINSTANCE hinst, HWND hwndParent, UINT wID,
                          DWORD dwStyles, HIMAGELIST himl);
```

The *dwStyles* parameter accepts a number of flags that define the look and operation of the command bands control. These styles match the rebar styles; the command bands control is, after all, closely related to the rebar control.

- *RBS_AUTOSIZE* Bands are automatically reformatted if the size or position of the control is changed.
- *RBS_BANDBORDERS* Each band is drawn with lines to separate adjacent bands.
- *RBS_FIXEDORDER* Bands can be moved but always remain in the same order.
- *RBS_SMARTLABELS* When minimized, a band is displayed with its icon. When restored or maximized, the band's label text is displayed.
- *RBS_VARHEIGHT* Each row in the control is vertically sized to the minimum required by the bands on that row. Without this flag, the height of every row is defined by the height of the tallest band in the control.
- *CCS_VERT* Creates a vertical command bands control.
- *RBS_VERTICALGRIPPER* Displays a gripper appropriate for a vertical command bar. This flag is ignored unless CCS_VERT is set.

Of these styles, the RBS_SMARTLABLES and RBS_VARHEIGHT are the two most frequently used flags. The RBS_SMARTLABLES flag lets you choose an attractive appearance for the command bands control without requiring any effort from the application. The RBS_VARHEIGHT flag is important if you use controls in a band other than the default command bar. The CCS_VERT style creates a vertical command bands control, but because Windows CE doesn't support vertical menus, any band with a menu won't be displayed correctly in a vertical band. As you'll see, however, you can hide a particular band when the control is orientated vertically.

IMAGE LISTS FOR COMMAND BANDS CONTROLS

I touched on image lists earlier. Command bars and toolbars use image lists internally to manage the images used on buttons. Image lists can be managed in a stand-alone image list control. This control is basically a helper control that assists applications in managing a series of like-size images. The image list control in Windows CE is identical to the image list control under Windows NT and Windows 98, with the exception that the Windows CE version can't contain cursors for systems built without mouse/cursor support. For the purposes of the command bands control, the image list just needs to be created and a set of bitmaps added that will represent the individual bands when they're minimized. An example of the minimal code required for this is shown here:

```
himl = ImageList_Create (16, 16, ILC_COLOR, 2, 0);
hBmp = LoadBitmap (hInst, TEXT ("CmdBarBmps"));
ImageList_Add (himl, hBmp, NULL);
DeleteObject (hBmp);
```

The *ImageList_Create* function takes the dimensions of the images to be loaded, the format of the images (ILC_COLOR is the default), the number of images initially in the list, and the number to be added. The two images are then added by loading a double-wide bitmap that contains two images and calling *ImageList_Add*. After the bitmap has been loaded into the image list, it should be deleted.

Adding bands

You can add bands to your application by passing an array of REBARBANDINFO structures that describe each band to the control. The function is

```
BOOL CommandBands_AddBands (HWND hwndCmdBands, HINSTANCE hinst,
                            UINT cBands, LPREBARBANDINFO prbbi);
```

Before you call this function, you must fill out a REBARBANDINFO structure for each of the bands to be added to the control. The structure is defined as

```
typedef struct tagREBARBANDINFO{
    UINT cbSize;
    UINT fMask;
    UINT fStyle;
    COLORREF clrFore;
    COLORREF clrBack;
    LPTSTR lpText;
    UINT cch;
    int iImage;
    HWND hwndChild;
    UINT cxMinChild;
    UINT cyMinChild;
    UINT cx;
    HBITMAP hbmBack;
    UINT wID;
    UINT cyChild;
    UINT cyMaxChild;
    UINT cyIntegral;
    UINT cxIdeal;
    LPARAM lParam;
} REBARBANDINFO;
```

Fortunately, although this structure looks imposing, many of the fields can be ignored because there are default actions for uninitialized fields. As usual with a Windows structure, the *cbSize* field must be filled with the size of the structure as a fail-safe measure when the structure is passed to Windows. The *fMask* field is filled with a number of flags that indicate which of the remaining fields in the structure are filled with valid information. I'll describe the flags as I cover each of the fields.

The *fStyle* field must be filled with the style flags for the band if the RBBIM_STYLE flag is set in the *fMask* field. The allowable flags are the following:

- *RBBS_BREAK* The band will start on a new line.
- *RBBS_FIXEDSIZE* The band can't be sized. When this flag is specified, the gripper for the band isn't displayed.
- *RBBS_HIDDEN* The band won't be visible when the command band is created.
- *RBBS_GRIPPERALWAYS* The band will have a sizing grip, even if it's the only band in the command band.
- *RBBS_NOGRIPPER* The band won't have a sizing grip. The band therefore can't be moved by the user.
- *RBBS_NOVERT* The band won't be displayed if the command bands control is displayed vertically due to the CCS_VERT style.
- *RBBS_CHILDEDGE* The band will be drawn with an edge at the top and bottom of the band.
- *RBBS_FIXEDBMP* The background bitmap of the band doesn't move when the band is resized.

For the most part, these flags are self-explanatory. Although command bands are usually displayed across the top of a window, they can be created as vertical bands and displayed down the left side of a window. In that case, the RBBS_NOVERT style