

PROGRAMMING THE SMARTPHONE

2003

INTRODUCING THE SMARTPHONE	3
A SMARTPHONE APPLICATION.....	5
HelloSP.rc.....	6
HelloSP.h	6
HelloSP.cpp.....	7
THE SMARTPHONE'S MENUBAR CONTROL.....	10
CREATING A SMARTPHONE MENUBAR CONTROL.....	11
WORKING WITH THE BUTTONS AND MENUS.....	12
THE BACK BUTTON AND OTHER INTERESTING BUTTONS	14
MESSAGE BOXES	16
DIALOG BOXES	17
SCROLLING DIALOGS.....	17
SMARTPHONE CONTROLS.....	18
TEXT CONTROLS	18
EXPANDABLE EDIT CONTROLS	20
SPINNER CONTROLS	21
FILE OPERATION IN THE SMARTPHONE.....	22
COMMUNICATION.....	23
PHONE API.....	23
Dialing the Phone	24
Viewing the Call Log	24
THE CONNECTION MANAGER	26
Connecting.....	26
Setting and Querying Status	28
SMS MESSAGING.....	28
Accessing the SMS System	29
Sending a Message	29
Receiving a Message	32
Configuring the SMS System.....	33
THE SMSTALK EXAMPLE.....	35
SMSTalk.rc	35
SMSTalk.h	38
SMSTalk.cpp.....	39
SMARTPHONE SECURITY.....	49

Programming the Smartphone

The following chapters are from *Programming Microsoft Windows CE, Third Edition*, by Douglas Boling, published by Microsoft Press. © 2003 by Douglas McConnaughey Boling. Reprinted here by permission of the author.

Programming the Smartphone

Given the inevitability of Moore's Law, Windows CE powered devices become both smaller and more powerful each year. One result is convergence, the merging of two or more separate smart devices into one. The Smartphone is one result of the trend toward convergence.

The Smartphone is a Windows CE-based cellular phone. Like the Pocket PC, all Smartphones regardless of manufacturer share the same configuration of Windows CE. Also, Smartphones come bundled with a set of applications such as an address book, calendar, and e-mail program. Microsoft produces a standard Smartphone build that individual manufacturers, and cellular providers can then enhance with branding and additional applications.

Smartphones are one of the more unique implementations of Windows CE devices. First, they are quite small, with screen resolutions of 176 by 220. They don't have touch screens and have a limited set of hardware buttons. Because Smartphones need to be on to receive calls, but also require long battery life, the CPUs used by these devices are slower than is standard in Pocket PCs. Finally, because cellular phones are used by people while they are doing other tasks such as working, walking, or even driving, the user interface of the application has to be much simpler than is normal on a PC or a Pocket PC. All of these things radically change the requirements of a Smartphone application.

Another challenge of developing software for the Smartphone is security. Unlike the Pocket PC, the cellular provider that sells the phone might limit the device's ability to load and run programs. The Smartphone implements the Windows CE module-based security scheme discussed in Chapter 10. Because of this restriction, a locked cell phone may either restrict unsigned applications or not run them at all. Depending on the provider, applications may need to be signed with an encryption key supplied by the provider or a trusted third party. Unfortunately, some cellular providers see this restriction as a possible revenue stream to extract money either from the user or the application developer or both.

Fortunately, this chapter isn't completely in vain even if developers are limited in writing Smartphone applications because the sections on communication features such as TAPI, SMS, and the SIM manager apply not only to the Smartphone but also to the Pocket PC Phone edition. Pocket PCs don't have the restrictions on third-party applications, so applications will run on those devices.

Introducing the Smartphone

The Smartphone user interface is an integration of the hardware implementation of the device and the implementation of the application being run on the device. More so than applications on other devices, Smartphone applications depend on the use of a small set of hardware buttons dedicated to particular tasks. This in itself forces a certain conformity across Smartphone applications.

Figure 19-1 shows a diagram of a hypothetical Smartphone. Notice the two buttons immediately below the screen. These two buttons provide the major application defined input to the application. Of course, the other buttons on the numeric keypad and the simple joystick provide input, but typically the device defines their function. The other buttons on this hypothetical device are the Home button, for returning the phone to the home screen, and the Back button, used to return to the previous screen.



Figure 19-1

The diagram of a Smartphone device

The Smartphone display has a layout similar to the Pocket PC. The top of the display has the navigation bar which shows the title text of the foreground window, and status icons for the phone, battery, and application notification icons. At the bottom of the screen is the phone's specialized *MenuBar* control, which is much-simplified over the Pocket PC's *MenuBar* control. The application's data is displayed in between.

The home screen of the Smartphone contains a summary of the various installed applications on the system. The joystick can be used to scroll to each of the items on the home screen to select the associated application. The home screen *MenuBar* also provides

a Programs menu that is the Smartphone's version of a Start menu. The default action of the right button is to dial the voicemail service for the phone.

Figure 19-2 shows the Programs menu from the Home screen. Notice that each menu item has a number associated with it. The user can use the keypad to select the menu item associated with the number on the keypad, allowing quick selection of a menu item. Alternatively, the user can use the joystick to highlight the proper menu item and select it by pressing in on the joystick. All menus displayed in the Smartphone have these item numbers displayed by default.



Figure 19-2

The Programs menu on the Home screen

A Smartphone Application

As with the Pocket PC, Smartphone applications have to perform certain tasks so they will operate correctly. Like Pocket PC applications, Smartphone applications must ensure that only one instance of itself must be running any one time. Also like the Pocket PC, the application should create a control with *SHCreateMenuBar*, although the function creates a simplified menu bar. The application might need to override the function of the Back button. For the main window, the Back button is passed on to the shell so that it can restore the previous application's window. Finally, and this can't be overemphasized, the application user interface must be simple. Figure 19-3 shows *HelloSP* running on a Smartphone. The text is centered in the client window, which has been sized to fit the area between the *navigation bar* and the *MenuBar* control

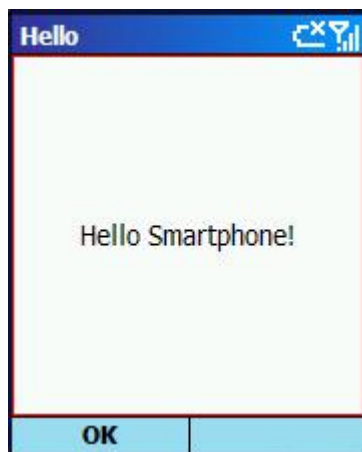


Figure 19-3

The HelloSP example running on a Smartphone

Listing 19-1 shows the first example in this chapter, *HelloSP*.

HelloSP.rc

```
//=====
// Resource file
//
// Written for the book Programming Windows CE
// Copyright (C) 2003 Douglas Boling
//=====
#include "windows.h"           // Windows stuff
#include "aygshell.h"          // Pocket PC stuff
#include "HelloSP.h"           // Program-specific stuff

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

ID_MENU RCDATA
BEGIN
    0,
    1,
    I_IMAGENONE, IDM_EXIT, TBSTATE_ENABLED,
    TBSTYLE_BUTTON | TBSTYLE_AUTOSIZE, IDS_OK, 0, NOMENU,
END

STRINGTABLE DISCARDABLE
BEGIN
    IDS_OK      "OK"
END
```

HelloSP.h

```
//=====
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 2003 Douglas Boling
//=====
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))

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

//-----
// Generic defines used by application
#define ID_MENU      10
#define IDM_EXIT     100
#define IDS_OK       1001

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

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

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

// WM_COMMAND message handlers
LPARAM DoMainCommandExit (HWND, WORD, HWND, WORD);
```

HelloSP.cpp

```
//=====
// HelloSP - A simple application for the Smartphone
//
// Written for the book Programming Windows CE
// Copyright (C) 2003 Douglas Boling
//=====
#include <windows.h>           // For all that Windows stuff
#include <commctrl.h>          // Command bar includes
#include <aygshell.h>           // Pocket PC includes
#include "hellosp.h"           // Program-specific stuff
//-----
// Global data
//
const TCHAR szAppName[] = TEXT("HelloSP");
HINSTANCE hInst;               // Program instance handle

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_PAINT, DoPaintMain,
    WM_COMMAND, DoCommandMain,
    WM_DESTROY, DoDestroyMain,
};
// Command Message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDM_EXIT, DoMainCommandExit,
};
//=====
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;
    HWND hwndMain;

    // 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 TerminateInstance (hInstance, msg.wParam);
}
//-----
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow) {
    WNDCLASS wc;
    HWND hwnd;

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

    // Allow only one instance of the application.
```

Programming the Smartphone

```
hWnd = FindWindow (szAppName, NULL);
if (hWnd) {
    SetForegroundWindow ((HWND)(((DWORD)hWnd) | 0x01));
    return 0;
}
// Register application main window class.
wc.style = CS_VREDRAW | CS_HREDRAW;          // 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 = LoadCursor (NULL, IDC_ARROW);  // Default cursor
wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
wc.lpszMenuName = NULL;                     // Menu name
wc.lpszClassName = szAppName;               // Window class name

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

// Create main window.
hWnd = CreateWindow (szAppName,              // Window class
                    TEXT("Hello"),           // 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
if (!IsWindow (hWnd)) return 0;              // Fail if not created.

// 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 main window
//
//-----
// 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) {
    SHMENUBARINFO mbi;
```

```

// Create a MenuBar.
memset(&mbi, 0, sizeof(SHMENUBARINFO)); // Zero structure
mbi.cbSize = sizeof(SHMENUBARINFO);    // Size field
mbi.hwndParent = hWnd;                  // Parent window
mbi.nToolBarId = ID_MENU;               // ID of toolbar resource
mbi.hInstRes = hInst;                   // Inst handle of app

// Create bar and check for errors.
if (!SHCreateMenuBar(&mbi)) {
    MessageBox (hWnd, TEXT("Couldn't create menu bar"),
        szAppName, MB_OK);
    DestroyWindow (hWnd);
}
// Size the window to fit above the MenuBar
RECT rect, rectDesk;
int cx, cy;
GetWindowRect (mbi.hwndMB, &rect);
GetDesktopRect (GetDesktopWindow (), &rectDesk);
cx = rectDesk.right-rectDesk.left;
cy = (rectDesk.bottom - rectDesk.top) - (rect.bottom - rect.top);
SetWindowPos (hWnd, NULL, 0, 0, cx, cy, SWP_NOMOVE | SWP_NOZORDER);

SHSetNavBarText (hWnd, TEXT("Hello"));
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;
}
//-----
// DoPaintMain - Process WM_PAINT message for window.
//
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
    LPARAM lParam) {
    PAINTSTRUCT ps;
    HPEN hPen, hOld;
    RECT rect;
    HDC hdc;

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

    // Draw a red rectangle around the window.
    hPen = CreatePen (PS_SOLID, 1, RGB (255, 0, 0));
    hOld = (HPEN)SelectObject (hdc, hPen);
    Rectangle (hdc, rect.left, rect.top, rect.right, rect.bottom);
    SelectObject (hdc, hOld);
    DeleteObject (hPen);

    // Draw the standard hello text centered in the window.
    DrawText (hdc, TEXT ("Hello Smartphone!"), -1, &rect,

```

```
DT_CENTER | DT_VCENTER | DT_SINGLELINE);

    EndPaint (hWnd, &ps);
    return 0;
}
//-----
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wParam, 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;
}
```

Listing 19-1

The structure of the program is similar to all the examples in the book. The point of the example is so show that a Smartphone application is a Windows application with the same entry point, the same message loop, and the same general message handlers.

The unique parts of the application start with the Pocket PC–like check for whether another instance of the application is running. There’s some difference of opinion on what should happen when the user starts a second instance of an application. Some style guides suggest that an application ought to close the previous instance and show the new instance. This behavior would provide the user with a clean copy of the application each time they select it. The problem with this concept is that few, if any, of the bundled applications that come with the Smartphone act this way. Instead, the bundled applications place the previous running instance in the foreground, restoring the state of the application as the user left it.

The example code also queries the height of the *MenuBar* control and adjusts the size of the main window to fit above it. Unlike Pocket PC applications, there’s no need to dynamically respond to the SIP popping up over the window because there’s no SIP on the Smartphone.

Another major difference in *HelloSP* is the *MenuBar* control. It’s created with a call to *ShCreateMenuBar*, but its function and action are quite different—different enough to handle it as a separate topic. Let’s dive into the Smartphone’s *MenuBar* control and see how it ticks.

The Smartphone’s MenuBar Control

The Smartphone *MenuBar* is a simplified version of the *MenuBar* control used by the Pocket PC. Because the Smartphone lacks a touch screen, the user interacts with the Smartphone *MenuBar* using two buttons at the base of the screen. The two buttons are aligned with the two possible buttons on the control. The buttons can either be implemented to display a menu or to perform an action directly.

As I mentioned earlier, when a menu is displayed, the first 10 items on the menu are automatically prefixed with a number from 1 through 9 and then 0 corresponding to the 10 digits on the phone keyboard. When the menu is displayed the user can easily select an item by pressing a key on the phone. With the automatic addition of the menu item

numbers, there's no reason to specify underlined navigation characters in the menu items.

While it's possible to put more than 10 items on a menu, the small size of the phone necessitates that the menu scroll, which isn't a very friendly interface design. Another less than friendly interface design is cascaded menus. The Smartphone *MenuBar* control does support cascaded menus, but the extra level of action required by the user causes more work than a cascaded menu provides benefits.

Creating a Smartphone MenuBar Control

Just as on the Pocket PC, the *MenuBar* creation function *SHCreateMenuBar* is used to create a *MenuBar* on the Smartphone. The format of the parameters is the same as in the Pocket PC, as is the format of the *SHMENUBARINFO* structure used to define the structure of the control. The combination toolbar and menu resource used by the Smartphone *MenuBar* is also the same. The difference is that, on the Smartphone, the *MenuBar* is much less flexible. The toolbar resource must define one or two buttons. The *MenuBar* buttons can either be designed to provide a direct action or display a menu when the associated button is pressed.

For review, the prototype of *SHCreateMenuBar* and *SHMENUBARINFO* are shown here:

```
BOOL SHCreateMenuBar (SHMENUBARINFO *pmb);
```

```
typedef struct tagSHMENUBARINFO {
    DWORD cbSize;
    HWND hwndParent;
    DWORD dwFlags;
    UINT nToolBarId;
    HINSTANCE hInstRes;
    int nBmpId;
    int cBmpImages;
    HWND hwndMB;
    COLORREF clrBk;
} SHMENUBARINFO;
```

In the Smartphone *MenuBar* control, the *cbSize*, *hwndParent*, *hInstRes*, and *hwndMB* fields are used, as in the standard menu bar, to confirm the size of the structure, indicate the control parent, provide the instance handle of the module containing the resource, and return the handle of the created control. However, some fields are limited in comparison to their use in the standard menu bar. The *nBmpId* and *cBmpImages* fields must be 0 or the call to create the control will fail. The *clrBk* field is ignored even if the corresponding *SHCMBF_COLORBK* flag is set in the *dwFlags* field. The other defined flags in *dwFlags* also don't make sense in the limited *MenuBar* control.

The *nToolBarId* field must identify a resource structured identically to the menu bar resource described in Chapter 5. That format is shown here.

```
<Menu ID>, <Number of buttons (1 or 2)>,
I_IMAGENONE, <Cmd1ID>, <Btn1State>, <Btn1Style>, <String1ID>, 0, <Menu1Index>
I_IMAGENONE, <Cmd2ID>, <Btn2State>, <Btn2Style>, <String2ID>, 0, <Menu2Index>
```

The first field of the resource is the resource ID of any menu resource being used by the control. The second field can be either 1 or 2 depending on if the control will have one or two buttons on the bar. The remainder of the resource describes one or both buttons on the control. In the previous example, the second and third lines describe the two buttons.

The first field should be set to *I_IMAGENONE* to indicate that the button is text, not a bitmap. Bitmaps are not supported on the Smartphone *MenuBar*. The second field contains the ID value that will be sent to the owner window if the button is not a menu and is pressed. The third field describes the initial state of the button using toolbar state flags. The fourth field describes the style of the button using toolbar style flags. The String ID field must refer to a valid string resource ID that contains the text for that button. The next

to last field should be set to 0, and the last field is the submenu index into the menu identified by the first field in the resource. This field can be *NOMENU* if the button doesn't display a menu when pressed.

An example *SoftKeyBar* control is shown in Figure 19-4.



Figure 19-4

A SoftKeyBar control where the menu button has been pressed

Figure 19-4 shows a Smartphone *MenuBar* with a Done button on the left and a Menu button on the right. This placement of direct action assigned to the left button and an optional menu assigned to the right button is the recommended arrangement suggested by the Smartphone user interface guide. In the figure, the Menu button has been pressed displaying a short menu on the screen. The two-part resource that describes the *MenuBar* control in Figure 19-4 is shown here:

```
ID_BTNBARRES RCDATA MOVEABLE PURE
BEGIN
    ID_MENU, 2,
    I_IMAGENONE, IDM_EXIT, TBSTATE_ENABLED, TBSTYLE_AUTOSIZE | TBSTYLE_BUTTON,
        IDS_DONE, 0, NOMENU,
    I_IMAGENONE, IDM_POP, TBSTATE_ENABLED, TBSTYLE_AUTOSIZE | TBSTYLE_DROPDOWN,
        IDS_MENU, 0, 0,
END

ID_MENU MENU DISCARDABLE
BEGIN
    POPUP "&Dummy"
    BEGIN
        MENUITEM "Week View",          IDM_VIEWWEEK
        MENUITEM "Month View",         IDM_VIEWMONTH
        MENUITEM "Year View",          IDM_VIEWYEAR
        MENUITEM SEPARATOR
        MENUITEM "Options",             IDM_OPTIONS
    END
END
```

The *ID_BTNBARRES* resource describes the two buttons. The first button is a direct action button and the second is a button that displays a menu. One difference between the two buttons, aside from the string resource references, is the different style flag, *TBSTYLE_BUTTON* vs. *TBSTYLE_DROPDOWN*. Another difference is the *NOMENU* index in the first button description, where the second button uses a menu index of 0. The 0 index indicates the first, and in this case only submenu of the *ID_MENU* menu resource.

Working with the Buttons and Menus

Like on the Pocket PC, the buttons and menus hosted by the *MenuBar* control can be modified after the control has been created. The key to modifying *MenuBar* is the use of

the *TB_GETBUTTONINFO* and *TB_SETBUTTONINFO* messages. These toolbar control messages work on the *MenuBar* control because it's derived from the toolbar.

To query the current settings for a button send a *TB_GETBUTTONINFO* message to the *MenuBar*. The *wParam* parameter should have the command ID identifying the button, and *lParam* should point to a *TBBUTTONINFO* structure. The *TBBUTTONINFO* structure is defined as:

```
typedef struct {
    UINT cbSize;
    DWORD dwMask;
    int idCommand;
    int iImage;
    BYTE fsState;
    BYTE fsStyle;
    WORD cx;
    DWORD lParam;
    LPTSTR pszText;
    int cchText;
} TBBUTTONINFO, *LPTBBUTTONINFO;
```

The *cbSize* and *dwMask* fields of *TBBUTTONINFO* should be initialized before the message is sent. The *cbSize* field should be filled with the size of the structure. The *dwMask* field should be set with flags indicating what data is being queried from the control. For the *MenuBar* control, the only flags allowed are *TBIF_TEXT*, *TBIF_STATE*, *TBIF_LPARAM*, and *TBIF_COMMAND*, which specify the text, state, user-defined parameter, and command ID, respectively.

Setting the parameters of a button is just as simple with the use of a *TB_SETBUTTONINFO* message. Here again, *wParam* should contain the ID of the button, and *lParam* should point to a *TBBUTTONINFO* structure. The following code disables a button by first querying the state of the button and then clearing the *TBSTATE_ENABLED* flag in the *fsState* field.

```
int DisableButton (HWND hwndMainWnd, DWORD dwID) {

    HWND hwndMB = SHFindMenuBar (hwndMainWnd);
    TBBUTTONINFO tbi;
    if (!hwndMB)
        return -1;

    memset (&tbi, 0, sizeof (tbi));
    tbi.cbSize = sizeof (tbi);
    tbi.dwMask = TBIF_STATE;
    if (!SendMessage (hwndMB, TB_GETBUTTONINFO, dwID, (LPARAM)&tbi))
        return -2;

    tbi.fsState &= ~TBSTATE_ENABLED;
    SendMessage (hwndMB, TB_SETBUTTONINFO, dwID, (LPARAM)&tbi);

    return 0;
}
```

The menus displayed by the Smartphone *MenuBar* can also be modified by the application. The same messages used to get the menu handles for the *MenuBar* control can be used in the Smartphone version of the control. Sending a *SHCMBM_GETMENU* message to the *MenuBar* control returns the menu attached to the control. The *wParam* and *lParam* parameters are ignored in this message. Sending *SHCMBM_GETSUBMENU* to the control returns the submenu attached to a specific button. In this case, *lParam* should contain the ID of the button, whereas *wParam* is ignored. The following code disables the Options menu item by first getting the handle of the submenu attached to the menu button and then using *EnableMenu* to disable the menu item.

```
HWND hwndMB = SHFindMenuBar (hwndMainWnd);
```

```
if (hwndMB) {  
    HMENU hMenu;  
    hMenu = (HMENU)SendMessage (hwndMB, SHCMBM_GETSUBMENU, 0, IDM_POP);  
    EnableMenuItem (hMenu, IDM_OPTIONS, MF_BYCOMMAND | MF_GRAYED);  
}
```

The other menu functions can be used to manipulate the menus on the *MenuBar* as easily as *EnableMenu* was used in the previous example.

The Back Button and Other Interesting Buttons

One of the more important components of the Smartphone interface is the Back button. This button allows the user to return to the previous screen from the current screen at any time. For the most part, the Back button works as designed without any assistance from the foreground application. There are, however, times when the Back button needs to act differently, and that's when the application has to do a bit of work.

The rules for the operation of the Back button are as follows:

- If the current window is not a dialog box and does not have an edit box, the Back button activates the window that was displayed before the current window was activated. The current window isn't destroyed, it's simply covered by the previous window, which now becomes the active window.
- If the current window is a message box or a modal dialog box without an edit control, the Back button dismisses the message box or dialog box and returns the cancel return code. For message boxes, this value is either *IDNO*, *IDCANCEL*, or *IDOK* for message boxes with only an OK button. For dialog boxes, a *WM_COMMAND* message is sent to the dialog window with the command ID *IDCANCEL*.
- If the window currently displayed contains an edit control, the Back button erases the last character in the control and moves the entry cursor one character to the left.

In the case of the first two rules, the system will provide the default action for the Back button. For the final rule, concerning edit boxes, the application must override the default action and forward the key to the appropriate child control in the window. Fortunately, the Smartphone shell that does most of the work through a couple of helper functions.

If a window contains an edit box, it must override the default action of the Back key in order to pass the key to the child control. To do this, the window sends a *SHCMBM_OVERRIDEKEY* message to the *MenuBar* control for the window. The *wParam* parameter defines the key to override. The following keys are supported for override:

Key	Meaning
<i>VK_TBACK</i>	Back button
<i>VK_TSOFT1</i>	Left <i>SoftKeyBar</i> button
<i>VK_TSOFT2</i>	Right <i>SoftKeyBar</i> button
<i>VK_TVOLUMEUP</i>	Up volume button
<i>VK_TVOLUMEDOWN</i>	Down volume button
<i>VK_TRECORD</i>	Record button

The *lParam* parameter designates the keys to override and the action to take. The lower word of *lParam* contains a mask of flags that designates which flags are valid. The action flags are in the upper word. The first flag that can be set indicates whether a key is to be overridden and the second indicates whether a *WM_HOTKEY* message is sent to the window when the key is pressed. For example, to override the action of the Back button and to be notified with a *WM_HOTKEY* message, a window would send the following message to the *MenuBar*:

```
SendMessage (SHFindMenuBar (hwnd), SHCMBM_OVERRIDEKEY, VK_TBACK,  
            MAKELPARAM (SHMBOF_NODEFAULT | SHMBOF_NOTIFY,
```

```
SHMBOF_NODEFAULT | SHMBOF_NOTIFY));
```

This line sends a *SHCMBM_OVERRIDEKEY* message to the *MenuBar* control owned by *hWnd*. The *wParam* parameter, *VK_TBACK*, indicates the key being overridden. The *MAKELPARAM* macro forms a *DWORD* from two 16-bit words. The first parameter of *MAKELPARAM*, which will become the low word of *lParam*, is the mask field. Here, the *SHMBOF_NODEFAULT* flag is used to indicate that the override state of the Back key is to be set or cleared. The *SHMBOF_NOTIFY* flag indicates that the notification flag will also be set or cleared. The upper word of *lParam* is created from the second parameter of the *MAKELPARAM* macro. Here, the flag *SHMBOF_NODEFAULT* indicates that the Back key will be overridden. The second flag, *SHMBOF_NOTIFY*, tells the *MenuBar* to notify its parent when the Back key is pressed.

To override the Back key but not be notified when it's pressed, the following line could be used:

```
SendMessage (SHFindMenuBar (hWnd), SHCMBM_OVERRIDEKEY, VK_TBACK,
             MAKELPARAM (SHMBOF_NODEFAULT | SHMBOF_NOTIFY,
                         SHMBOF_NODEFAULT));
```

To restore the Back key to its original function, the following line could be used:

```
SendMessage (SHFindMenuBar (hWnd), SHCMBM_OVERRIDEKEY, VK_TBACK,
             MAKELPARAM (SHMBOF_NODEFAULT | SHMBOF_NOTIFY,
                         0));
```

In the previous line, the first parameter of *MAKELPARAM* (the mask bits) indicates that the message is going to set the state of both the default action and the notification state of the button. The second *MAKELPARAM* parameter is 0, which indicates that the default action is to be restored and that notification is to be sent to the *MenuBar* owner.

When *WM_HOTKEY* is received by the window, the *wParam* value contains an ID value for the key that was pressed. The IDs reported by the *MenuBar* are:

Key	Value
<i>VK_TSOFT1</i>	0
<i>VK_TSOFT2</i>	1
<i>VK_TBACK</i>	2
<i>VK_TVOLUMEUP</i>	3
<i>VK_TVOLUMEDOWN</i>	4
<i>VK_TRECORD</i>	5

The *lParam* value also indicates the key but in a different way. The high word of *lParam* will contain the virtual key code of the key that was pressed. The lower word of *lParam* contains the modifier flags for the key. The only flag that is of interest on the Smartphone is *MOD_KEYUP*, which is set if the key is being released. Other modifier flags are documented such as the Shift, Alt, and Control keys, but those flags are seldom used because the Smartphone doesn't have these keys.

For windows that contain edit controls, the notification of a Back key press needs to be forwarded to the proper child control. This forwarding is done by using a *SHCMBM_OVERRIDEKEY* message to redirect the Back key. Once the key is overridden, a press of the key sends a *WM_HOTKEY* message to the owner of the *MenuBar*. To fully support passing the backspace to the edit control, each *WM_HOTKEY* message received as a result of a Back key press needs to be forwarded to the control with the function *SHSendBackToFocusWindow*, defined as:

```
void SHSendBackToFocusWindow (UINT uMsg, WPARAM wp, LPARAM lp);
```

The three parameters are the message being handled, as well as the *wParam* and *lParam* values for the message. The following code fragment shows the use of this function.

```
case WM_HOTKEY:
    if (HIWORD (lParam) == VK_TBACK)
        SHSendBackToFocusWindow (wMsg, wParam, lParam);
}
```

The override state of a key changed by a *SHCMBM_OVERRIDEKEY* message is removed when the corresponding *MenuBar* control is destroyed so there's no need to manually restore the default state before the window is destroyed.

Message Boxes

MessageBox, the standard Windows CE function for displaying message boxes, works on the Smartphone with some limitations. Because of the limit of two buttons on the *SoftKeyBar*, message boxes are limited to one or two buttons. The supported flags are *MB_OK*, *MB_OKCANCEL*, *MB_RETRYCANCEL*, and *MB_YESNO*. Using the flags *MB_ABORTRETRYIGNORE* or *MB_YESNOCANCEL*, which would normally result in message boxes with three buttons, will cause *MessageBox* to fail. Figure 19-5 shows a typical message box.



Figure 19-5

A message box on the Smartphone

The message box in Figure 19-5 is created with the following line:

```
MessageBox (hwnd, TEXT("Do you want to delete the file?"), TEXT("Box Title"),
            MB_OKCANCEL | MB_ICONEXCLAMATION);
```

The interesting thing about the message box is the bold text, *Alert*, that is not part of the text passed to the *MessageBox* function. The Smartphone places this text on the message box depending on the icon specified in the *MessageBox* call. The text displayed depends on the language settings of the phone and optionally the customization done by the service provider, but for a US English phone the text is:

MessageBox flag	Text	Icon
<i>MB_ICONEXCLAMATION</i>	Alert	Exclamation mark within a circle
<i>MB_ICONHAND</i>	Error	Exclamation mark within a circle
<i>MB_ICONQUESTION</i>	Confirm	Question mark within a circle
<i>MB_ICONASTERISK</i>	Info	Lowercase I within a circle

The message text is displayed in smaller text below the large, bold prompt text. The title text is displayed on the caption bar across the top of the screen.

If the message box has OK as one of the options, the user can press the action button on the phone to return *IDOK*. For other button settings such as Yes/No or Retry/Cancel, the action button has no effect and the user has to press one of the two buttons associated with

the *MenuBar*. If the user presses the Back button, the message box is dismissed and the function returns the negative response, either No for a Yes/No choice, Cancel for Yes/Cancel, or OK if the message box only has an OK button.

Dialog Boxes

Message boxes will only get you so far on a Smartphone. Dialog boxes, or dialogs, can be created on the Smartphone as long as a few rules are observed. As on the Pocket PC, dialog boxes on the Smartphone must be sized to fit the full screen. This is accomplished with a call to *SHInitDialog* during the handling of the *WM_INITDIALOG* message. Smartphone dialogs must also create a *SoftKeyBar* control to provide a place for key input and to handle the Back button if the dialog contains one or more edit boxes.

The following code shows the handling of a *WM_INITDIALOG* message for a typical Smartphone dialog box that contains at least one edit control.

```
// Specify that the dialog box should stretch full screen
SHINITDLGINFO shidi;

memset (&shidi, 0, sizeof(shidi));
shidi.dwMask = SHIDIM_FLAGS;
shidi.dwFlags = SHIDIF_SIZEDLGFULLSCREEN/*SHIDIF_FULLSCREENNOMENUBAR*/;
shidi.hDlg = hWnd;
if(!SHInitDialog(&shidi))
    return FALSE;

// set up MenuBar menu
SHMENUBARINFO mbi;

memset (&mbi, 0, sizeof(SHMENUBARINFO));
mbi.cbSize = sizeof(SHMENUBARINFO);
mbi.hwndParent = hWnd;
mbi.nToolBarId = ID_MENUDLG;
mbi.hInstRes = hInst;

// If we could not initialize the dialog box, return an error
if (!SHCreateMenuBar(&mbi)) {
    DestroyWindow (hWnd);
    return FALSE;
}
// Override back key since we have an edit control
SendMessage(mbi.hwndMB, SHCMBM_OVERRIDEKEY, VK_TBACK,
    MAKELPARAM (SHMBOF_NODEFAULT | SHMBOF_NOTIFY,
        SHMBOF_NODEFAULT | SHMBOF_NOTIFY));
// set the title bar
SHSetNavBarText (hWnd, TEXT("Dialog Title Text"));
```

Navigating controls in a Smartphone dialog is different from a standard Windows CE system because the Smartphone doesn't have a Tab key to transfer focus between controls. Instead, controls within a Smartphone dialog box are organized vertically, and the Up and Down cursor keys are used to switch focus to the next control in the tab order. (The control order is still referred to as *tab order* even though the Smartphone lacks a Tab key.) Each individual control has been modified to use the left and right cursor keys to provide navigation and selection within the control.

Scrolling Dialogs

Implementing a dialog box that scrolls is typically not an easy task. On the Smartphone, with its tiny screen, it's sometimes a necessity. Fortunately, the shell makes implementing scrolling dialogs fairly painless. All that is required is that the dialog box template has the *WS_VSCROLL* style flag set. When the user changes focus from control to control, if the next control is below the visible part of the screen, the shell scrolls the dialog automatically. If the focus is at the bottom most control and the user presses the Down

cursor key, the Smartphone switches the focus back to the first control and scrolls the dialog back to the top.

If the application repositions any of the controls on a scrolling dialog, it needs to tell the dialog manager about the new positions. This is done with the Smartphone unique message, `DM_RESETSCROLL`. If the controls are repositioned in the `WM_INITDIALOG` message or in any message before `WM_INITDIALOG`, sending `DM_RESETSCROLL` isn't required.

Smartphone Controls

The limited display and lack of touch screen or mouse on the Smartphone have driven Microsoft to modify the behavior of some of the standard controls. The goal is to make controls that are navigable from simple cursor commands as well as use very little space.

The standard Windows CE controls still exist on the Smartphone. It's possible to create a standard list box in a Smartphone window, but once you navigate into a list box using the cursor keys the control won't relinquish focus to other controls in the window because it uses the Up and Down cursor keys for internal navigation between the items in the list box. Standard multi-line edit controls have the same problem. Other controls such as combo boxes don't make much sense in the limited screen space of a Smartphone. Because of this, Smartphone windows or dialog boxes use a limited and somewhat different set of controls. One of these enhanced controls is the oft-overlooked edit control.

Text Controls

Inputting text is a bit of a challenge on a Smartphone. The keyboard is limited to a number pad and a handful of cursor keys and is not conducive to text input. The Smartphone relies on the developer providing a context for the contents of the edit field, such as whether the control will contain numbers, words, or names. Depending on the context, the edit box can interpret taps on the numeric keypad in different ways.

Most OEMs support two different text input modes on their Smartphones. The first is referred to as *multi-tap* and is familiar to anyone who has entered text on a cellular phone during the last 10 years. In multi-tap mode, multiple presses of each number key result in the system scrolling through the letters assigned to that number. The letters assigned are derived from the standard letter assignment that was set by American Telephone and Telegraph eons ago. Because that original assignment didn't include the letters Q and Z, a great trivia question by the way, the layout has been slightly modified and is now an international standard. The layout is shown in Figure 19-6.

1 Punctuation	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ
* Shift	0	# Space

Figure 19-6

The standard letter assignments on a telephone keypad

When the user wants to enter a letter, the number assigned to the letter is tapped multiple times until that letter appears. For example to enter a K, the user would tap the 5 key twice. Each number scrolls through its assigned letters followed by its number. So tapping the 8 key multiple times would scroll through the characters T, U, V, and 8. Pausing between taps for a short time locks in the selection and moves the cursor to the right. The 1 key is used to enter standard punctuation characters such as . , ? ! - ' @ : 1. Tapping the * key makes the next letter upper case. The # key inserts spaces, and the 0 key enters 0.

There are special cases for some keys. Pressing and holding the # key displays a screen of symbols. The user can then use the cursor keys to select the symbol needed. Pressing and holding the # key switches between multi-tap mode, T9 mode, and numeric mode.

T9 mode is a text-entry scheme that makes amazingly accurate guesses at what the user is trying to enter. The user taps the number key assigned to each letter and then types in the number for the next letter of the word, and so on. The T9 system makes a guess at the word from the possible combination of the letters possible from the first number and the letters from the second number. As more numbers are entered, the system predicts the word being entered with an almost uncanny accuracy. T9 works great with standard words but does have trouble with names.

A third entry mode is numeric. As you might guess, this mode simply takes the assigned number for each key entered and places it in the edit box.

Edit controls on the Smartphone have an enhanced interface over the standard run-of-the-mill Windows edit control. The first enhancement is the ability for the edit control to be switched between the various text input modes.

The *EM_SETINPUTMODE* message sets the input mode of the control. The *wParam* value isn't used, but *lParam* can contain one of the following values: *EIM_SPELL* for multi-tap mode, *EIM_AMBIG* for ambiguous or T9 mode, *EIM_NUMBERS* for numeric mode, and *EIM_TEXT* for the default entry mode set by the user.

In addition to the text mode, modifier flags can also be passed to set the shift state or the caps lock state. To change the shift state, the *IMMF_SHIFT* flag is combined with the text mode. To set the shift state, combine *IMMF_SHIFT* with *IMMF_SETCLR_SHIFT*. To clear the shift state, use *IMMF_SHIFT* alone. The caps lock state can be set in a similar manner. Passing the *IMMF_CAPSLOCK* flag indicates that the caps lock state is being modified. Combining *IMMF_CAPSLOCK* with *IMMF_SETCLR_CAPSLOCK* sets the caps lock state, whereas passing *IMMF_CAPSLOCK* alone clears the caps lock state.

The current mode can be queried with the *EM_GETINPUTMODE* message. To query the default input mode of the control, pass *FALSE* in *lParam*. Passing *TRUE* in *lParam* returns the default input mode for the control if it doesn't have focus or the actual input mode if it does have focus. The actual and default can be different because the user can change the input mode by pressing and holding the # key.

The *EM_SETSYMBOLS* message sets the symbols that are displayed when the user presses the 1 key. For this message, *lParam* points to a string that contains the symbols to be made available to the user.

The *EM_SETEXTENDEDSTYLE* message sets the extended edit style bits for the control. The *wParam* parameter contains a mask defining which of the extended style bits are to be changed. The state of each flag is set in *lParam*. So, to clear a particular extended style flag, the flag would be passed in *wParam*, but the corresponding bit in *lParam* would be 0 to indicate that the new state of the flag is to be 0.

The only extended style bit currently defined is *ES_EX_CLEARONBACKPRESSHOLD*. This flag causes the control to interpret a press and hold of the Back key as a command to clear the contents of the edit control. To prevent this action, this style bit can be cleared by the application. This flag is set by default on single line edit controls and cleared by default on multiline edit controls.

Expandable Edit Controls

Edit controls on the Smartphone can be made expandable with an up-down buddy control. In a dialog resource, the combination might look like the following:

```
EDITTEXT          IDD_TEXT, 5, 60, 75, 12, WS_TABSTOP | ES_MULTILINE
CONTROL "",       IDD_UDEDIT, UPDOWN_CLASS, UDS_AUTOBUDDY |
                  UDS_ALIGNRIGHT | UDS_EXPANDABLE | UDS_NOSCROLL,
                  0, 0, 0, 0
```

The first line defines the edit control, the up-down control is declared immediately afterward. The *CONTROL* resource tag is used for this control because the resource compiler doesn't use a specific keyword such as *EDIT* to declare an up-down control. The up-down control is assigned an ID value to identify the source of *WM_NOTIFY* messages it sends to its parent. The control doesn't need a position because its position will be defined by the location of the edit control. The up-down control's style flags configure it for this specific use. The *UDS_AUTOBUDDY* flag tells the control to associate itself with the closest control in the z-order. Because the edit box was declared immediately before the up-down control, the edit box will be closest in the z-order. The *UDS_ALIGNRIGHT* flag tells the control to attach to the right side of the edit control.

UDS_NOSCROLL tells the up-down control not to display Up and Down arrows like a scroll bar. The Up and Down arrows can be used for incrementing and decrementing numbers in an associated edit control. Here, we simply want the Right arrow to indicate to the user that the edit control is expandable.

Finally, the *UDS_EXPANDABLE* flag indicates that the edit control is expandable. When the edit control has the focus and the user hits the action key, the edit control will be expanded to fill the entire screen, showing as much of the contents of the edit control as possible. The expanded mode is much like a multi-line edit control in that the action key moves the cursor to a new line. A *MenuBar* is automatically created with OK and Cancel buttons to accept or discard any changes made in expanded mode. Figure 19-7 shows an expandable edit control in both normal and expanded modes.

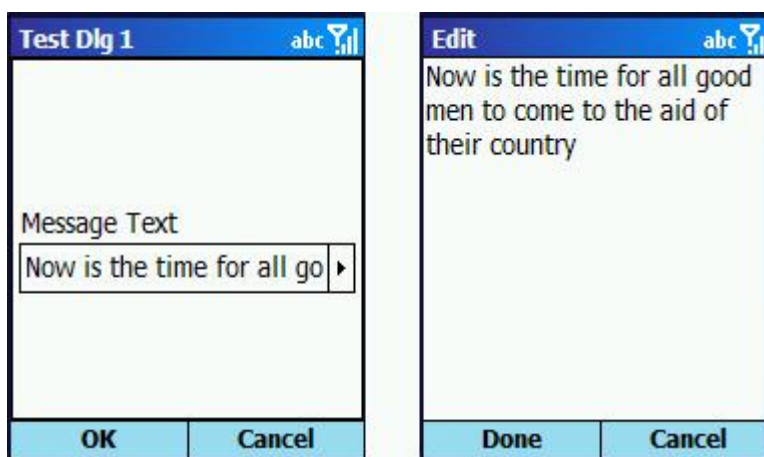


Figure 19-7

An expandable edit control in normal and expanded modes

An expandable edit control can be created manually as well as through dialog resource

definitions. The following code creates an expandable edit control manually.

```
hWndEdit = CreateWindow (TEXT("edit"), NULL, WS_VISIBLE |
                        WS_TABSTOP | ES_AUTOHSCROLL | ES_AUTOVSCROLL |
                        ES_MULTILINE, 9, 15, 100, 75, hWnd,
                        (HMENU)IDD_TEXT, hInst, 0L);

hWndUpDown = CreateWindow (UPDOWN_CLASS, NULL, WS_VISIBLE | UDS_AUTOBUDDY |
                           UDS_ALIGNRIGHT | UDS_EXPANDABLE | UDS_NOSCROLL,
                           0, 0, 0, 0, hWnd, (HMENU)IDD_UDTEXT, hInst, 0L);

SendMessage(hWndUpDown, UDM_SETBUDDY, (WPARAM)hWndEdit, 0);
```

The only difference between this code and the dialog resource is the sending of a `UDM_SETBUDDY` message to the up-down control to associate it with the edit control.

Spinner Controls

The spinner is a modified list box control buddied with an up-down control. The result is a single line control that can be *spun* to display each item in the list box. Unlike the standard list box, the spinner control uses the Left and Right cursor keys to *spin* between items in the box. As with other Smartphone controls, the Up and Down cursor keys transfer focus to the next control in the tab order.

Creating a spinner consists of creating both the list box and up-down controls. In a dialog resource, the combination would look like the following:

```
LISTBOX      IDD_LISTCITIES, 5, 60, 75, 12, WS_TABSTOP
CONTROL "",  IDD_CITIESUD, UPDOWN_CLASS,
              UDS_AUTOBUDDY | UDS_HORZ | UDS_ALIGNRIGHT | UDS_ARROWKEYS |
              UDS_SETBUDDYINT | UDS_WRAP | UDS_EXPANDABLE, 0, 0, 0, 0
```

The first line defines the list box. The only unusual thing about the declaration is that the list box is only 12 dialog units high, just tall enough for one item. The up-down control is declared immediately after the list box. The `CONTROL` resource tag is used for this control because the resource compiler doesn't use a specific keyword such as `LISTBOX` to declare an up-down control. The large number of style flags configures the control for this specific use. Many of the same style flags are used when creating expandable edit controls, so I'll only cover the new ones. The `UDS_HORZ` flag tells the control to create Left and Right arrows instead of Up and Down arrows attached to the list box.

The `UDS_SETBUDDYINT` flag tells the up-down control to manipulate the text of its buddy, in this case the list box, and have it scroll among the different items in the list. The `UDS_WRAP` flag tells the control to wrap the list so that if the list box is at the last item in the list and the user presses the right button, the list box will show the first item in the list.

The up-down control can have the additional style flag of `UDS_NOSCROLL`. This flag prevents the user from spinning the data with the Left and Right cursor keys. This style isn't much use unless it's combined with the `UDS_EXPANDABLE` flag so the control can be expanded, allowing the user to change the selection. When the user expands the spinner, it sends a `WM_NOTIFY` message to the parent window with the `UDN_EXPANDING` command. Figure 19-8 shows a spinner control in both normal and expanded modes.

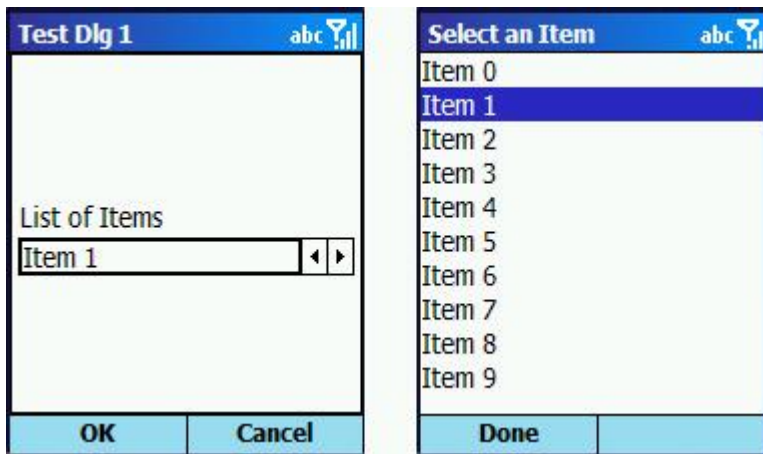


Figure 19-8

A spinner control in normal and expanded modes

A spinner can also be created manually with two calls to *CreateWindow*, as shown here:

```
HWND hwndList = CreateWindow (TEXT("listbox"), NULL, WS_VISIBLE |
                             WS_BORDER | WS_TABSTOP, 5, 5, 75, 20, hwnd,
                             (HMENU)IDD_LISTCITIES, hInst, 0L);

HWND hwndUpDown = CreateWindow (UPDOWN_CLASS, NULL, WS_VISIBLE | UDS_HORZ |
                                UDS_ALIGNRIGHT | UDS_ARROWKEYS |
                                UDS_SETBUDDYINT | UDS_WRAP | UDS_EXPANDABLE,
                                0, 0, 0, 0, hwnd, 0, hInst, 0L);

SendMessage (hwndUpDown, UDM_SETBUDDY, (WPARAM)hwndList, 0);
```

Here, like in the expandable edit control, the only difference between the two methods is the extra message sent to the up-down control to tell it the ID of its buddy list box and a few manually added style flags needed when creating the list box.

File Operation in the Smartphone

Unlike the Pocket PC, data in the object store of the Smartphone is volatile. When the phone is turned off, the data goes away. Fortunately, there is a way to persistently save data in the file system.

The Smartphone implements an external file system using flash memory internal to the phone. This file system is just like a file system that would appear if a Compact Flash or SD card was inserted into the system, but in this case the file system is not removable. The room in the internal persistent store is limited,, so it's not a great area for storing huge databases or MP3 files. Still, it does provide a place to store application state data or other information. While the method for finding external file systems was discussed in Chapter 8, the Smartphone extends a standard shell call that will return the proper subdirectory in which an application can store its data.

The function *SHGetSpecialFolderPath* was covered in Chapter 16. The Smartphone adds an additional constant, *CSIDL_APPDATA*. Using this CSIDL value will return the name of the application data folder that is persistent. An application can then create a subdirectory under the persistent folder where it can store its data. Because the persistent folder is used by all applications, you should be careful to uniquely name your application's directory. The following code demonstrates finding the application data folder and creating a directory.

```
int CreateAppFolder (HWND hwnd, TCHAR *pszAppFolder, int nMax) {
    const TCHAR szMyAppFolderName[] = TEXT ("ProgWinCESpSample");
    TCHAR szPath[MAX_PATH];
```

```
// It doesn't help to have a path longer than MAX_PATH
if (nMax > MAX_PATH)
    nMax = MAX_PATH;

BOOL f = SHGetSpecialFolderPath (hWnd, szPath, CSIDL_APPDATA, FALSE);

// See if everything will fit in output string
int nLen = lstrlen (szPath);
if (nLen + 2 + (int)lstrlen (szMyAppFolderName) > nMax)
    return -2;

// Copy app folder name to parameter
lstrcpy (pszAppFolder, szPath);

// Append directory separator character as needed
if (szPath[nLen] != TEXT ('\\'))
    lstrcat (pszAppFolder, TEXT("\\"));

// Append my folder name
lstrcat (pszAppFolder, szMyAppFolderName);

// See if directory exists.
if (GetFileAttributes (pszAppFolder) == 0xffffffff) {
    // See why call failed
    if (GetLastError () == ERROR_PATH_NOT_FOUND) {
        // Wasn't there, create the directory
        if (!CreateDirectory (pszAppFolder, NULL))
            return -3;
    } else
        return -4; // Dir created but inaccessible
} else
    return 1;      // Indicate directory already exists
return 0;         // Indicate directory created
}
```

Data stored in the registry is persistent. The Smartphone takes steps to save and restore the registry when the system shuts down and starts back up.

Communication

The Smartphone would be nothing but a small PDA with a limited interface if it weren't for its communication features. An intelligent, mobile device, in a pocket and always connected to the Internet, is a powerful platform. The communication features discussed in this section are also implemented on the Pocket PC Phone edition, which is quite helpful for the learning curve of developers, who can apply their knowledge both on the Pocket PC and the Smartphone.

Communication services cover a number of areas, from dialing voice calls to sending messages to other phones to connecting to the Internet through either a wired or wireless connection. For the most part, these services are accessed through easy-to-use shell APIs that wrap the traditional but more complex Windows APIs such as TAPI, the Telephone API.

The folks that design communication functions design the most flexible and therefore most complex functions possible. The functions I describe in this section have what at times seems like dozens of parameters, each with a dozen options. Complete books can be, and have been, written about many of the following topics, so I can't cover completely each of the topics in a single part of a chapter. Still, this discussion will provide a good introduction to these functions and should give you a good start on adding communication features in your applications.

Phone API

The phone API provides a basic set of functions to make calls and query the call log. The phone API is convenient because it avoids the need for most applications to dive directly into the TAPI and make calls with that rather involved interface. TAPI is supported on the Smartphone and Pocket PC Phone edition, but working with it isn't required for all but the most manipulative of applications.

Dialing the Phone

The first communication feature likely to be tried by a programmer new to the Smartphone or Pocket PC Phone edition is to dial the phone. The system provides a simple and effective function for this, *PhoneMakeCall*, prototyped as:

```
LONG PhoneMakeCall (PHONEMAKECALLINFO *ppmci);
```

The *PHONEMAKECALLINFO* structure is defined as:

```
typedef struct tagPHONEMAKECALLINFO{
    DWORD cbSize;
    DWORD dwFlags;
    PCWSTR pszDestAddress;
    PCWSTR pszAppName;
    PCWSTR pszCalledParty;
    PCWSTR pszComment;
} PHONEMAKECALLINFO, * PHONEMAKECALLINFO;
```

The first field is the standard size field that needs to be filled with the size of the structure. The *dwFlags* field can contain one of two flags: *PMCF_PROMPTBEFORECALLING*, which tells the system to prompt the user before initiating the call, or *PMCF_DEFAULT*, which tells the system to make the call without asking the user whether the call should be made. Even with the *PMCF_DEFAULT* flag, the system will display a notification that the call is being made. The *pszDestAddress* field should point to a string containing the phone number to call. The string can contain the standard phone number separator characters such as dashes, spaces, and parentheses. The *pszCalledParty* field should point to a string that identifies the called party. This string is displayed, along with the number, on the call-in-progress notification that is displayed when the call is made. The *pszAppName* and *pszComment* fields should be set to 0.

Viewing the Call Log

The system maintains a log of all calls made to and from the phone. The call log is a simple database that keeps information such as the time of the call, its duration, and the number of the other phone, as well as details such as whether the call was incoming, the phone is roaming, and so on. The call log can be accessed with a few simple functions. To open the call log, call the aptly named *PhoneOpenCallLog*, prototyped as:

```
HRESULT PhoneOpenCallLog (HANDLE * ph);
```

The function returns *S_OK* if successful and *ERROR_FAIL* otherwise. An extended error code can be retrieved with *GetLastError*. Some Smartphone systems don't allow the call log to be opened. There might or might not be a system reason for getting this error, but many phones don't allow this function to succeed. There isn't an issue with Pocket PC phone edition systems. Also, opening the call log isn't necessary for calling *PhoneMakeCall*. If *PhoneOpenCallLog* is successful, a handle will be placed in the value pointed to by the parameter *ph*, and the seek pointer of the call log will be set to point to the first entry in the log.

Once opened, entries in the call log can be queried with the function *PhoneGetCallLogEntry*, prototyped as:

```
HRESULT PhoneGetCallLogEntry (HANDLE h, PCALLLOGENTRY pentry);
```

The handle is the one received from the call to *PhoneOpenCallLog*. The *dbSize* field of the

CALLLOGENTRY structure needs to be initialized with the size of the structure before the function is called.

If the function returns without error, *pentry* points to a structure that contains data about the call, and the seek pointer of the call log is moved to the next entry. Repeated calls to *PhoneGetCallLogEntry* will enumerate the entire call log. When no more entries are in the log, the function will fail with the extended error code 259 indicating no more entries are available.

The *CALLLOGENTRY* structure is defined as:

```
typedef struct {
    DWORD cbSize;
    FILETIME ftStartTime;
    FILETIME ftEndTime;
    IOM iom;
    BOOL fOutgoing:1;
    BOOL fConnected:1;
    BOOL fEnded:1;
    BOOL fRoam:1;
    CALLERIDTYPE cidt;
    PTSTR pszNumber;
    PTSTR pszName;
    PTSTR pszNameType;
    PTSTR pszNote;
} CALLLOGENTRY, * PCALLLOGENTRY;
```

The *ftStartTime* and *ftEndTime* fields are *FILETIME* structures that provide the start and end times of the call. The *iom* field contains an enumeration indicating if the call was incoming, was outgoing, or was missed. The next four fields are Booleans detailing the conditions of the call: *fOutgoing* is *TRUE* if the call was made from the device; *fConnected* is set if the call actually made a connection; *fEnded* is *TRUE* if the call was terminated by the callers and *FALSE* if the call was dropped; and *fRoam* is set if the call was made from outside the phone's home area. The *cidt* field is an enumeration indicating if the caller ID for the call was available, blocked, or unavailable.

The *pszNumber* field points to a string indicating the number of the phone number of the calling phone or the phone being called. The *pszName* field identifies the name associated with the number. The *pszNameType* field points to a string that indicates which number—home, work, or mobile—was associated with the contact. The string is in the form of a character, typically either *h*, *w*, or *m* for home, work, or mobile, respectively. The *pszNote* field is a string that is supposed to point to a string containing the name of a notes file for the call. This field isn't always filled in by the system.

The seek pointer of the call log can be moved with

```
HRESULT PhoneSeekCallLog (HANDLE h, CALLLOGSEEK seek, DWORD iRecord,
                          LPDWORD piRecord);
```

The handle value is the handle returned by *PhoneOpenCallLog*. The seek value can be set to either *CALLLOGSEEK_BEGINNING* or *CALLLOGSEEK_END*, depending of whether the passed offset is based from the beginning or end of the log. The parameter *iRecord* is the zero-based offset from the beginning or end of the log. The *piRecord* parameter points to a *DWORD* that receives the index, from the beginning of the log, of the resulting record. Once the seek pointer is moved to a specific record, the record can then be read with *PhoneGetCallLogEntry*.

The phone log should be closed with a call to *PhoneCloseCallLog*, prototyped as:

```
HRESULT PhoneCloseCallLog (HANDLE h);
```

The single parameter is the handle returned from *PhoneOpenCallLog*.

The Connection Manager

The connection manager is a centralized location to request a connection to external data networks regardless of the connection method. The connection manager presents both a single point of connection configuration to the user and a single place where applications can go to programmatically connect to the network. The connection manager also frees the application from having to know what connections are the best to use given the different costs and speeds of the various connections. For example, the connection manager knows it's better to access the Internet via ActiveSync if possible instead of connecting via a cellular connection.

Connecting

An application can connect to a network in three ways using the connection manager. An application can request a connection synchronously, request a connection asynchronously, or schedule a time for a connection to be made. Typically, an application will call *ConnMgrEstablishConnection* to request a connection be made asynchronously and then be notified when the connection is made. The prototype is:

```
HRESULT ConnMgrEstablishConnection (CONNMGR_CONNECTIONINFO * pConnInfo,  
                                   HANDLE * phConnection);
```

The function returns an *HRESULT* and, if successful, a handle to the connection in the variable pointed to by *phConnection*. The other parameter of the call is a pointer to a *CONNMGR_CONNECTIONINFO* structure, defined as:

```
typedef struct _CONNMGR_CONNECTIONINFO {  
    DWORD cbSize;  
    DWORD dwParams;  
    DWORD dwFlags;  
    DWORD dwPriority;  
    BOOL bExclusive;  
    BOOL bDisabled;  
    GUID guidDestNet;  
    HWND hwnd;  
    UINT uMsg;  
    LPARAM lParam;  
    ULONG ulMaxCost;  
    ULONG ulMinRcvBw;  
    ULONG ulMaxConnLatency;  
} CONNMGR_CONNECTIONINFO;
```

The traditional size field should be set to the size of the structure. The *dwParams* field indicates which optional fields in the structure are filled with valid data. I'll mention the flags in this field as I discuss the optional fields. The *dwFlags* field indicates the proxies supported by the application. The supported flags include proxy flags for HTTP, WAP, SOCKS4 and SOCKS5. If no flags are specified, only a direct Internet connection is attempted.

The *dwPriority* field indicates how important the connection is to the application. The priority ranges from *CONNMGR_PRIORITY_VOICE* for a voice connection, the highest priority, to *CONNMGR_PRIORITY_LOWBKGND*, which indicates a connection will only be made if another connection is currently active that satisfies the request. The *bExclusive* field should be set to *TRUE* if the connection should not be shared among the other applications in the system. If *bDisabled* is *TRUE*, *ConnMgrEstablishConnection* will check the connection to see whether it can be made, but the connection won't actually be made. The *guidDestNet* field indicates the network to connect to. The GUID of the various network connections can be determined using *ConnMgrEnumDestinations* discussed later. This field will be used only if the *dwParams* field contains the *CONNMGR_PARAM_DESTNETID* flag.

The next three fields, *hWnd*, *wMsg*, and *lParam*, are used to provide feedback to the application about the connection. The connection manager sends message values of *wMsg* indicating progress of the connection to the window indicated by *hWnd*. The *lParam* value is passed in the progress message in the *lParam* value. The *wParam* value of the message provides a connection status. The status values are defined in the *ConnMgr.h* include file. Additional connection status can be queried by calling *ConnMgrConnectionStatus*. This function, along with the various connection states, is discussed later.

The final three parameters, *ulMaxCost*, *ulMinRcvBw*, and *ulMaxConnLatency*, define additional conditions that the connection manager should use when choosing what path to use when making a connection. These fields are only used if the corresponding flag is set in the *dwParams* parameter.

To connect synchronously, an application can call *ConnMgrEstablishConnectionSync*, prototyped as:

```
HRESULT ConnMgrEstablishConnectionSync (CONNMGR_CONNECTIONINFO *pConnInfo,
                                         HANDLE *phConnection,
                                         DWORD dwTimeout, DWORD *pdwStatus);
```

The parameters are similar to *ConnMgrEstablishConnection* with the addition of *dwTimeout*, which defines the time in milliseconds that the function should wait for a connection to be made, and *pdwStatus*, which points to a *DWORD* that is filled in with the resulting status of the connection.

The connection manager can also be requested to make a connection at a scheduled time using the function *ConnMgrRegisterScheduledConnection*. If the device is off when the scheduled time arrives, it will automatically turn on and attempt the connection. *ConnMgrRegisterScheduledConnection* is defined as:

```
HRESULT ConnMgrRegisterScheduledConnection (SCHEDULEDCONNECTIONINFO
*pSCI);
```

The single parameter is a pointer to a *SCHEDULEDCONNECTIONINFO* structure, defined as:

```
typedef struct _SCHEDULEDCONNECTIONINFO {
    GUID guidDest;
    UINT64 uiStartTime;
    UINT64 uiEndTime;
    UINT64 uiPeriod;
    TCHAR szAppName[MAX_PATH];
    TCHAR szCmdLine[MAX_PATH];
    TCHAR szToken[32];
    BOOL bPiggyback;
} SCHEDULEDCONNECTIONINFO;
```

The *guidDest* field defines the network to connect to. The *szToken* field should be set to a string that uniquely identifies this scheduled connection. This string will be used if the connection needs to be canceled.

The *uiStartTime* and *uiEndTime* fields define the time that the requests should start and stop, and *uiPeriod* defines how often the requests should be repeated. These times are defined in the structure as 64-bit numbers that are 100 nanosecond intervals since January 1, 1601. Conveniently this format is the same as the *FILETIME* structure. The *uiPeriod* value can be set to 0 to indicate that the system will never automatically attempt the connection.

When the connection is made, the application name pointed to by *szAppName* will be launched with the command line specified in *szCmdLine*. If the *bPiggyback* field is set to *TRUE*, the application will be launched whenever a connection is made to the network matching the *guidDest* field. Setting this field to *TRUE* and setting *uiPeriod* to 0 allows an

application to monitor any connection to a specific network without scheduling a connection of its own.

The scheduled connection can be canceled with a call to *ConnMgrUnregisterScheduledConnection*, defined as:

```
HRESULT ConnMgrUnregisterScheduledConnection (LPCWSTR pwszToken);
```

The single parameter is the 32-character token string passed in the *szToken* field of *SCHEDULEDCONNECTIONINFO* when the connection was scheduled.

Setting and Querying Status

Querying the status of a connection can be accomplished with a call to

```
HRESULT ConnMgrConnectionStatus (HANDLE hConnection, DWORD *pdwStatus);
```

The parameters are the handle to a current connection handle and a pointer to a *DWORD* that will receive the current status. The status flags are listed here:

Status	Value	Description
Unknown	00	Status unknown.
Connected	0x10	The connection is up.
Disconnected	0x20	The connection is disconnected.
Connection failed	0x21	The connection attempt failed.
Connection canceled	0x22	The user aborted the connection.
Connection disabled	0x23	The connection is ready to connect but is disabled.
No path to destination	0x24	No network path could be found to destination.
Waiting for path	0x25	The connection is waiting for a path to the destination.
Waiting for phone	0x26	A voice call is in progress.
Waiting connection	0x40	Attempting to connect.
Waiting for resource	0x41	Resource is in use by another connection.
Waiting for network	0x42	No path could be found to destination.
Waiting disconnection	0x80	The connection is being brought down.
Waiting connection abort	0x81	The connection attempt is being aborted.

The various connections can be enumerated using the function *ConnMgrEnumDestinations*, prototyped as:

```
HRESULT ConnMgrEnumDestinations (int nIndex,  
                                CONNMGR_DESTINATION_INFO *pDestInfo);
```

To use this function, an application calls *ConnMgrEnumDestinations* repeatedly, the first time with the *nIndex* set to 0 and then incrementing *nIndex* for each call. If the function is successful, data about the connection will be provided in the *CONNMGR_DESTINATION_INFO* structure pointed to by *pDestInfo*. The structure is defined as:

```
typedef struct _CONNMGR_DESTINATION_INFO {  
    GUID guid;  
    TCHAR szDescription[CONNMGR_MAX_DESC];  
} CONNMGR_DESTINATION_INFO;
```

The structure has two fields, the GUID of the specific network and a string describing the network. The GUID of the network can be used to specify the network when connecting.

SMS Messaging

The Short Message Service, or SMS, is a popular way, as the name implies, to exchange short text messages between cellular phones. By default, the InBox application on both the Smartphone and the Pocket PC Phone edition reads SMS messages. On the Smartphone, the Pocket InBox is also responsible for composing and sending SMS messages. On the Pocket PC, SMS messages can be composed and sent from a menu on the phone application. In addition to these applications, both systems expose a set of functions that allows third-party applications to send and receive SMS messages.

The process of sending and receiving SMS messages involves getting an SMS handle for sending and another for receiving. The message is composed, and the address is defined as the phone number of the receiving phone. Instead of sending the message directly to the phone, however, the message is sent to the SMS Service Center, which forwards the message on to the destination phone.

Receiving a message involves blocking on an event that is signaled when a message has been received. The message can then be read with a call to the SMS system. Because of the blocking nature of the reading process, this task is usually accomplished with a secondary thread.

The SMS system doesn't provide any way of saving the read messages. Instead, the application that is responsible for receiving the messages is responsible for saving the messages in a database if the user wants them saved. On the Pocket PC and Smartphone, Pocket InBox saves the messages in the e-mail database.

Accessing the SMS System

The task of sending or receiving messages starts with accessing the SMS subsystem using *SmsOpen*, prototyped as:

```
HRESULT SmsOpen (const LPCTSTR ptsMessageProtocol,  
                 const DWORD dwMessageModes, SMS_HANDLE* const psmshHandle,  
                 HANDLE* const phMessageAvailableEvent);
```

The first parameter is a string that describes the type of message that the application is interested in sending or receiving. For sending and receiving basic text messages, the *SMS_MSGTYPE_TEXT* protocol should be used. The include file *Sms.h* defines a number of other protocols that can be used for broadcast, status, and a couple of control protocols. The *dwMessageModes* parameter should be set for either *SMS_MODE_RECEIVE* or *SMS_MODE_SEND* depending on if the open is to send or receive messages. The *psmshHandle* parameter points to an *SMS_HANDLE* value that will receive the SMS handle if the function is successful.

The final parameter, *phMessageAvailableEvent*, points to an event handle but is used only when opening to receive messages. When asking for a handle to send SMS messages, the parameter should be *NULL*. When asking for a handle to receive messages, this parameter must point to a standard Windows CE event handle that was previously created by the application. This event handle should be an auto-reset event, not initially signaled. The event will be signaled when a message has been received by the system. The application should not set the event or close the event handle. The event will be closed when the application calls *SmsClose*.

The return value of *SmsOpen* will be *ERROR_SUCCESS* if the call was successful. Otherwise, an error code will be returned.

Sending a Message

Sending a message is accomplished by calling the rather involved *SmsSendMessage* function, prototyped as:

```
HRESULT SmsSendMessage (const SMS_HANDLE smshHandle,
```

```
const SMS_ADDRESS * const pmsaSMSCAddress,  
const SMS_ADDRESS * const pmsaDestinationAddress,  
const SYSTEMTIME * const pstValidityPeriod,  
const BYTE * const pbData, const DWORD dwDataSize,  
const BYTE * const pbProviderSpecificData,  
const DWORD dwProviderSpecificDataSize,  
const SMS_DATA_ENCODING smsdeDataEncoding,  
const DWORD dwOptions,  
SMS_MESSAGE_ID * psmsmidMessageID);
```

The first parameter is an SMS handle that was opened for sending a message. The *pmsaSMSCAddress* parameter points to a *SMS_ADDRESS* structure that contains the phone number of the SMS service center. In most cases, this parameter can be *NULL* to indicate that the system should use the default SMS center address. The *pmsaDestinationAddress* parameter points to another *SMS_ADDRESS* structure that contains the destination address of the message. I'll discuss the format of the *SMS_ADDRESS* structure in a moment.

The fourth parameter of *SmsSendMessage*, *pstValidityPeriod*, sets the length of time the message can remain undelivered at the server before it's deleted. Contrary to the parameter's *SYSTEMTIME* type, the format of this field is not a *SYSTEMTIME* but a data type defined by the SMS specification. This parameter can be *NULL*.

The next two fields are the pointer to the message text and the length in bytes of the text. The maximum length of a single message is 140 bytes or 160 7-bit characters. The format of the data is defined in the *smsdeDataEncoding* parameter discussed later.

The *pbProviderSpecificData* parameter points to optional provider-specific data. The provider-specific data is a structure specialized to the message format being used when sending the data. The format of the *TEXT_PROVIDER_SPECIFIC_DATA* structure, used when sending standard text messages, is described later. The *dwProviderSpecificDataSize* parameter should be set to the size of the structure pointed to by *pbProviderSpecificData*. See the discussion of the *TEXT_PROVIDER_SPECIFIC_DATA* structure for special handling of this parameter.

The *smsdeDataEncoding* parameter describes how the message data is encoded. For most messages, the parameter should be set to *SMSDE_OPTIMAL* to tell the SMS code to define the optimal encoding. The other values are *SMSDE_GSM* to use the 7-bit GSM encoding and *SMSDE_UCS2* to specify a Unicode UCS2 encoding. The *dwOptions* parameter specifies how the service center will handle retries. If the parameter is set to *SMS_OPTION_DELIVERY_NONE*, the service center will retry sending the message. If *SMS_OPTION_DELIVERY_NO_RETRY* is specified, the message won't retry the delivery.

The final parameter points to a *DWORD* that will receive a message ID value. The message ID can be used to get status of the message using the function *SmsGetMessageStatus*. This parameter can be *NULL* if the message ID isn't needed.

The return value of *SmsSendMessage* is a standard *HRESULT* value, with *S_OK* (0) indicating success. The function can take a short but noticeable amount of time to complete so it's best to display a wait cursor or use some other method to prevent the user from thinking the system is momentarily locked up.

The structures used by *SmsSendMessage* included the *SMS_ADDRESS* and *TEXT_PROVIDER_SPECIFIC_DATA* structures. The *SMS_ADDRESS* structure is defined as:

```
typedef struct sms_address_tag {  
    SMS_ADDRESS_TYPE smsatAddressType;  
    TCHAR ptsAddress[SMS_MAX_ADDRESS_LENGTH];  
} SMS_ADDRESS, *LPSMS_ADDRESS;
```

The first field of the structure is the address type. For most uses, this field can be set to *SMSAT_INTERNATIONAL*. The second field is the address, for *SMSAT_INTERNATIONAL*, the address is in the form of a phone number complete with country code and area code as in +12225551212.

When sending standard text messages, the provider specific data structure used is *TEXT_PROVIDER_SPECIFIC_DATA*, shown here. Other structures can be used when sending other types of message, but for brevity I'll only describe this structure.

```
typedef struct text_provider_specific_data_tag {
    DWORD dwMessageOptions;
    PROVIDER_SPECIFIC_MESSAGE_CLASS psMessageClass;
    PROVIDER_SPECIFIC_REPLACE_OPTION psReplaceOption;
    DWORD dwHeaderDataSize;
    BYTE pbHeaderData[SMS_DATAGRAM_SIZE];
    BOOL fMessageContainsEMSHeaders;
    DWORD dwProtocolID;
} TEXT_PROVIDER_SPECIFIC_DATA;
```

This structure definition is somewhat misleading because only the first three fields are used when sending a standard text message. The additional fields are only used for message concatenation. The first field is the message options field that can optionally request that various message bits be set, such as reply path, discard, or status. The message class value ranges from *PS_MESSAGE_CLASS0* through *PS_MESSAGE_CLASS3* and *PS_MESSAGE_CLASSUNSPECIFIED*. The message class indicates how the service center handles the message. For text messages, *PS_MESSAGE_CLASS0* is used. The *psReplaceOption* field contains *PSRO_NONE* for standard messages. The field can be set to *PSRO_REPLACE_TYPE_n*, where *n* is a value from 1 through 7. If a replace type field is set, the message will replace a message at the destination with the same parameters and the same replace type value.

A function of this complexity deserves an example. The following code calls *SmsOpen*, fills in the proper structures, and then sends the message. The SMS handle is then closed with *SmsClose*.

```
SMS_HANDLE smshHandle;
SMS_ADDRESS smsaDestination;
TEXT_PROVIDER_SPECIFIC_DATA tpsd;
SMS_MESSAGE_ID smsmidMessageID = 0;

// try to open an SMS Handle
HRESULT hr = SmsOpen(SMS_MSGTYPE_TEXT, SMS_MODE_SEND, &smshHandle, NULL);
if (hr != ERROR_SUCCESS) {
    printf ("SmsOpen fail %x %d", hr, GetLastError());
    return 0;
}
// Create the destination address
memset (&smsaDestination, 0, sizeof (smsaDestination));
smsaDestination.smsatAddressType = SMSAT_INTERNATIONAL;
lstrcpy(smsaDestination.ptsAddress, TEXT("+18005551212"));

// Set up provider specific data
tpsd.dwMessageOptions = PS_MESSAGE_OPTION_NONE;
tpsd.psMessageClass = PS_MESSAGE_CLASS0;
tpsd.psReplaceOption = PSRO_NONE;

char szMessage[] = "Watson! Come here, I need you!";

// Send the message, indicating success or failure
hr = SmsSendMessage (smshHandle, NULL, &smsaDestination, NULL,
    (PBYTE) szMessage, strlen(szMessage)+1,
    (PBYTE) &tpsd, 12, SMSDE_OPTIMAL,
    SMS_OPTION_DELIVERY_NONE, &smsmidMessageID);
if (hr == ERROR_SUCCESS)
    printf ("Message sent");
```

```
SmsClose (smshHandle);
```

SmsOpen is called with *SMS_MODE_SEND* to open a handle for sending. Notice that the final parameter is *NULL* because there isn't a need for the read event handle. The destination address is then filled in with a 10-digit phone number. The provider specific data is constructed with no options and message class 0. The message is then constructed, and the call to *SmsSendMessage* is made. Notice that the size of the provider specific data is not set to *sizeof (TEXT_PROVIDER_SPECIFIC_DATA)* because this is a simple, standalone message and the last few fields of *TEXT_PROVIDER_SPECIFIC_DATA* aren't used. After the message is sent, the handle is closed with *SmsClose*, which has the handle as the single parameter.

Receiving a Message

Receiving a message is accomplished with the function *SmsReadMessage*. When opening a SMS handle for reading, an event handle must be passed as the last parameter. The SMS system will use this handle to signal the application when a message has been received.

When the event is signaled, the application can get an idea of the size of the incoming message by calling *SmsGetMessageSize*, prototyped as:

```
HRESULT SmsGetMessageSize (const SMS_HANDLE smshHandle,  
                           DWORD * const pdwDataSize);
```

The two parameters are the SMS handle that was opened previously and a pointer to a *DWORD* that will receive the message size. The size received is not necessarily the exact size of the message. Instead, it's an upper bound that can be used to allocate the buffer that receives the message.

With a buffer allocated, the message can be read using *SmsReadMessage*, prototyped as:

```
HRESULT SmsReadMessage (const SMS_HANDLE smshHandle,  
                        SMS_ADDRESS * const psmsaSMSCAddress,  
                        SMS_ADDRESS * const psmsaSourceAddress,  
                        SYSTEMTIME * const pstReceiveTime,  
                        BYTE * const pbBuffer, DWORD dwBufferSize,  
                        BYTE * const pbProviderSpecificBuffer,  
                        DWORD dwProviderSpecificDataBuffer,  
                        DWORD* pdwBytesRead);
```

The first parameter is a SMS handle that was opened in receive mode. The second parameter is an optional *SMS_ADDRESS* structure that can receive the number of the SMS service center that sent the message. If the message center address is of no interest, this parameter can be set to *NULL*. The third parameter points to an *SMS_ADDRESS* structure that will be filled in with the address of the message received. The *pstReceiveTime* parameter points to a *SYSTEMTIME* structure that will receive the UTC-based time of the message. This parameter can be *NULL* if the time isn't required. The next two parameters, *pbBuffer* and *dwBufferSize*, are the pointer to the buffer to receive the data and the size of the buffer. The *pbProviderSpecificBuffer* parameter points to a buffer that will receive the provider-specific data that accompanies the message, and *dwProviderSpecificDataBuffer* contains the size of the buffer. The final parameter points to a *DWORD* that will receive the size of the message received.

SmsReadMessage will fail if there is no message to be read, so the application must block on the event used when *SmsOpen* was called and only read the message when the event is signaled. The blocking nature of the process means that *SmsReadMessage*, or at least the wait on the event object, should be done in a secondary thread. The following code is a separate thread that creates an event, opens an SMS handle, blocks on the event, and if signaled reads the message.

```
DWORD ThreadRead (PVOID pArg) {  
    SMS_ADDRESS smsaDestination;
```

```

TEXT_PROVIDER_SPECIFIC_DATA tpsd;
SMS_HANDLE smshHandle;

HANDLE hRead = CreateEvent (NULL, FALSE, FALSE, NULL);
// Open an SMS Handle
HRESULT hr = SmsOpen (SMS_MSGTYPE_TEXT, SMS_MODE_RECEIVE,
                     &smshHandle, &hRead);
if (hr != ERROR_SUCCESS) {
    printf ("SmsOpen fail %x %d\r\n", hr, GetLastError());
    return 0;
}
// Wait for message to come in.
int rc = WaitForSingleObject (hRead, 30000);
if (rc != WAIT_OBJECT_0) {
    printf ("WaitForSingleObject %d\r\n", rc);
    SmsClose (smshHandle);
    return 0;
}
memset (&smsaDestination, 0, sizeof (smsaDestination));
DWORD dwSize, dwRead = 0;

hr = SmsGetMessageSize (smshHandle, &dwSize);
if (hr != ERROR_SUCCESS) {
    dwSize = 1024;
    return 0;
}
char *pMessage = (char *)malloc (dwSize+1);
memset (&tpsd, 0, sizeof (tpsd));
hr = SmsReadMessage (smshHandle, NULL, &smsaDestination, NULL,
                    (PBYTE)pMessage, dwSize,
                    (PBYTE)&tpsd, sizeof(TEXT_PROVIDER_SPECIFIC_DATA),
                    &dwRead);
if (hr == ERROR_SUCCESS) {
    printf ("Dst Address >%s<\r\n", smsaDestination.ptsAddress);
    printf ("Msg: >%s<", pMessage);
} else
    printf ("Failed %x LastErr:%d\r\n", hr, GetLastError());
free (pMessage);
SmsClose (smshHandle);
printf ("ThreadExit");
return 0;
}

```

This code could be better written to check the length of the received data and to insure that the message is zero terminated.

Configuring the SMS System

There are a number of functions in the SMS API that are provided for querying the state and managing the SMS system. The SMS phone number for the device can be queried with a call to *SmsGetPhoneNumber*, defined as:

```
HRESULT SmsGetPhoneNumber (SMS_ADDRESS* const psmsaAddress);
```

The only parameter is an *SMS_ADDRESS* structure that is filled in with the phone number of the device.

The status of a sent message can be queried with *SmsQueryMessageStatus*, prototyped as:

```

HRESULT SmsGetMessageStatus (const SMS_HANDLE smshHandle,
                             SMS_MESSAGE_ID smsmidMessageID,
                             SMS_STATUS_INFORMATION * const psmssiStatusInformation,
                             const DWORD dwTimeout);

```

The first two parameters are the SMS handle and the message ID that was returned by *SmsSendMessage*. The *dwTimeout* value is the time, in milliseconds, that the function should wait for status information from the SMS service center. If the function returns successfully, the *SMS_STATUS_INFORMATION* structure is filled with status information about the message. The structure is defined as:

```
typedef struct sms_status_information_tag {
    SMS_MESSAGE_ID smsmidMessageID;
    DWORD dwMessageStatus0;
    DWORD dwMessageStatus1;
    SMS_ADDRESS smsaRecipientAddress;
    SYSTEMTIME stServiceCenterTimeStamp;
    SYSTEMTIME stDischargeTime;
} SMS_STATUS_INFORMATION, *LPSMS_STATUS_INFORMATION;
```

The first field is the ID of the message. The next two fields contain status flags that define the state of the message. There are two fields because there are more than 32 status flags defined. The *SMS_ADDRESS* field is filled with the destination address of the message. The *stServiceCenterTimeStamp* field contains the time the message was received by the service center. The *stDischargeTime* field is a time that depends on the status flags returned in the two *dwMessageStatus* fields.

The SMS service center number can be queried and set with the functions *SmsGetSMSC* and *SmsSetSMSC*, prototyped as:

```
HRESULT SmsGetSMSC (SMS_ADDRESS* const pmsaSMSCAddress);
```

and

```
HRESULT SmsSetSMSC (const SMS_ADDRESS * const pmsaSMSCAddress);
```

Both functions take a single parameter, a pointer to an *SMS_ADDRESS* structure. Typically, the telephony provider preconfigures this service center number in the phone.

The current time can be estimated with a call to *SmsGetTime*, prototyped as:

```
HRESULT SmsGetTime (SYSTEMTIME * const ptsCurrentTime,
    DWORD * const pdwErrorMargin);
```

The time returned is based on the time received by the SMS service center the last time the system received a timestamp. The time is a UTC number so it needs to be corrected for the local time zone. The *pdwErrorMargin* parameter should point to a *DWORD* that receives an estimated error margin, in seconds, for the time. If an error margin can't be determined, the error margin will be set to 0xFFFFFFFF.

An application can ask to be started when a message is received by calling the function *SmsSetMessageNotification*, prototyped as:

```
HRESULT SmsSetMessageNotification (const SMSREGISTRATIONDATA * pmsrd);
```

The single parameter is a pointer to a *SMSREGISTRATIONDATA* structure, defined as:

```
typedef struct smsregistrationdata_tag {
    DWORD cbSize;
    TCHAR tszAppName[SMS_MAX_APPNAME_LENGTH];
    TCHAR tszParams[SMS_MAX_PARAMS_LENGTH];
    TCHAR tszProtocolName[SMS_MAX_PROTOCOLNAME_LENGTH];
} SMSREGISTRATIONDATA, *LPSMSREGISTRATIONDATA;
```

The *cbSize* field should be set with the size of the structure before calling the function. The *tszAppName* and *tszParams* fields specify the application name and command line for the application when it's launched. The *tszProtocolName* field should be set to the message protocol for the messages the application wants to receive. For example, if the application wants to receive standard text messages, the field should be set to *SMS_MSGTYPE_TEXT*.

When the application no longer wants to be notified when messages are received, it can call *SmsClearMessageNotification*, prototyped as:

```
HRESULT SmsClearMessageNotification (const LPCTSTR tszProtocolName);
```

The single parameter is the message protocol that was specified when *SmsSetMessageNotification* was called.

The SMSTalk Example

The following example uses a number of the techniques discussed in this chapter to create an application that sends and receives SMS messages. *SMSTalk* is a dialog-based, multithreaded application that monitors the SMS read queue as well as provides a method for the user to compose and send SMS messages.

The example is designed to run both on the Smartphone and the Pocket PC phone edition. The example is designed for binary compatibility, as opposed to source code compatibility. Notice that SMSTalk checks for the Smartphone and makes the necessary changes to the user interface at run time. Figure 19-9 shows the *SMSTalk* main dialog on both a Smartphone and a Pocket PC phone edition device.

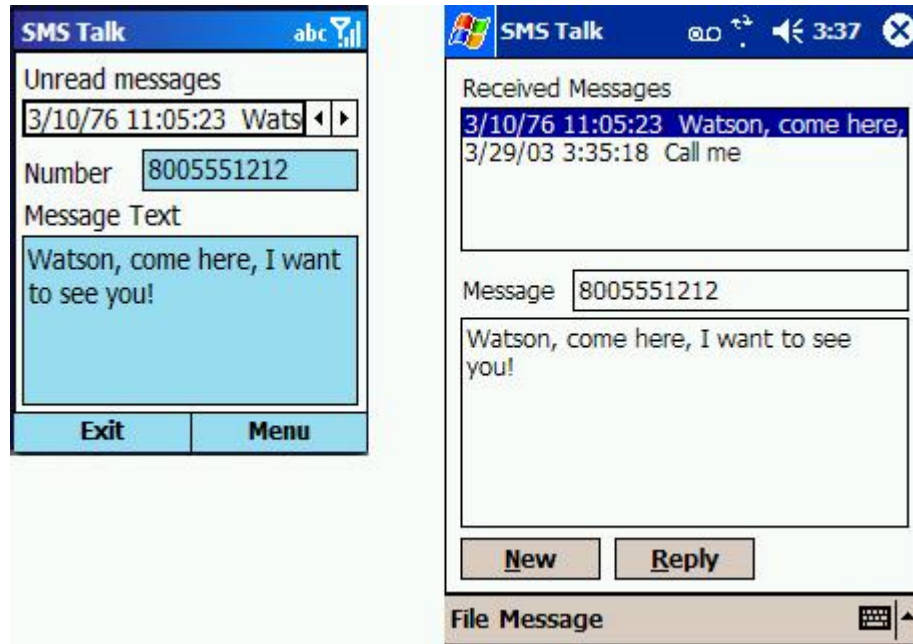


Figure 19-9

The SMSTalk application running on both a Smartphone and a Pocket PC

The dialogs look different because the application uses different dialog box templates depending on the device the application is running on. The source code, shown in Listing 19-2, has a rather long resource file because all the dialogs must be described twice, once for each device. The resource file also contains different menu bar templates for the two devices.

SMSTalk.rc

```
//=====
// Resource file
//
// Written for the book Programming Windows CE
// Copyright (C) 2003 Douglas Boling
//=====
#include "windows.h"
#include "aygshell.h"
#include "SMSTalk.h"                                // Program-specific stuff

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

//-----
// Main window dialog template for Pocket PC
//
```

Programming the Smartphone

```
SMSTalk_PPC DIALOG discardable 25, 5, 120, 98
STYLE WS_OVERLAPPED | WS_VISIBLE | WS_SYSMENU | DS_MODALFRAME
CAPTION "SMS Talk"
FONT 8, "System"
BEGIN
    LTEXT "Received Messages", -1, 4, 4, 128, 10,
    LISTBOX IDD_MSGLIST, 4, 14, 128, 48, WS_TABSTOP

    LTEXT "Message", -1, 4, 62, 32, 10,
    EDITTEXT IDD_MSGADDR, 36, 60, 96, 12, WS_TABSTOP |
        ES_NUMBER
    EDITTEXT IDD_MSGTEXT, 4, 74, 128, 60, WS_TABSTOP |
        ES_MULTILINE | ES_WANTRETURN | ES_AUTOVSCROLL
    PUSHBUTTON "&New", ID_CMDNEW, 4, 137, 40, 12, WS_TABSTOP
    PUSHBUTTON "&Reply", ID_CMDREPLY, 48, 137, 40, 12, WS_TABSTOP
END
//-----
// Main window dialog template for Smartphone
//
SMSTalk_SP DIALOG discardable 25, 5, 120, 98
STYLE WS_OVERLAPPED | WS_VISIBLE | WS_CAPTION | WS_SYSMENU |
    DS_CENTER | DS_MODALFRAME
CAPTION "SMS Talk"
BEGIN
    LTEXT "Unread messages", -1, 2, 2, 96, 8,
    LISTBOX IDD_MSGLIST, 2, 11, 96, 12, WS_TABSTOP
    CONTROL "", IDD_MSGLISTUD, UPDOWN_CLASS,
        UDS_AUTOBUDDY | UDS_HORZ | UDS_ALIGNRIGHT | UDS_ARROWKEYS |
        UDS_SETBUDDYINT | UDS_WRAP | UDS_EXPANDABLE,
        0, 0, 0, 0
    LTEXT "Number", -1, 2, 24, 34, 8,
    EDITTEXT IDD_MSGADDR, 36, 23, 62, 10, ES_READONLY

    LTEXT "Message Text", -1, 2, 34, 96, 8,
    EDITTEXT IDD_MSGTEXT, 2, 43, 96, 40, WS_TABSTOP |
        ES_MULTILINE | ES_READONLY
    CONTROL "", IDD_MSGTEXTUD, UPDOWN_CLASS, UDS_AUTOBUDDY |
        UDS_HORZ | UDS_ARROWKEYS | UDS_SETBUDDYINT |
        UDS_WRAP | UDS_EXPANDABLE | UDS_NOSCROLL,
        0, 0, 0, 0
END
//-----
// Compose window dialog template for Pocket PC
//
WriteMsgDlg_PPC DIALOG discardable 25, 5, 120, 98
STYLE WS_OVERLAPPED | WS_VISIBLE | WS_SYSMENU | DS_MODALFRAME
CAPTION "Compose Message"
BEGIN
    LTEXT "Number", -1, 4, 6, 32, 10,
    EDITTEXT IDD_MSGADDR, 36, 4, 96, 12, WS_TABSTOP |
        ES_NUMBER
    LTEXT "Message Text", -1, 4, 20, 128, 10,
    EDITTEXT IDD_MSGTEXT, 4, 30, 128, 54, WS_TABSTOP |
        ES_MULTILINE | ES_WANTRETURN | ES_AUTOVSCROLL
    PUSHBUTTON "&Send", ID_CMDSSEND, 4, 90, 40, 12, WS_TABSTOP
    PUSHBUTTON "&Cancel", IDCANCEL, 48, 90, 40, 12, WS_TABSTOP
END
//-----
// Compose window dialog template for Smartphone
//
WriteMsgDlg_SP DIALOG discardable 25, 5, 120, 98
STYLE WS_OVERLAPPED | WS_VISIBLE | WS_CAPTION | WS_SYSMENU |
    DS_CENTER | DS_MODALFRAME
CAPTION "SMS Talk"
BEGIN
    LTEXT "Number", -1, 2, 2, 96, 8,
    EDITTEXT IDD_MSGADDR, 2, 11, 96, 10, WS_TABSTOP

    LTEXT "Message Text", -1, 2, 24, 96, 8,
    EDITTEXT IDD_MSGTEXT, 2, 33, 96, 50, WS_TABSTOP |
```

```

                                ES_MULTILINE
CONTROL "",          IDD_MSGTEXTUD, UPDOWN_CLASS, UDS_AUTOBUDDY |
                                UDS_HORZ | UDS_ARROWKEYS | UDS_SETBUDDYINT |
                                UDS_WRAP | UDS_EXPANDABLE | UDS_NOSCROLL,
                                0, 0, 0, 0

END
//-----
// String resource table
//
STRINGTABLE DISCARDABLE
BEGIN
    IDS_EXIT      "Exit"
    IDS_MENU      "Menu"
    IDS_MSG       "Message"
    IDS_FILE      "File"
    IDS_OK        "OK"
    IDS_CANCEL    "Cancel"
    IDS_SEND      "Send"
END
//-----
// SoftKeyBar resource on main window for Smartphone
//
ID_MENU_SP RCDATA MOVEABLE PURE
BEGIN
    ID_MENU_SP, 2,
    I_IMAGENONE, IDOK, TBSTATE_ENABLED, TBSTYLE_BUTTON |
                                TBSTYLE_AUTOSIZE, IDS_EXIT, 0, NOMENU,
    I_IMAGENONE, IDPOP, TBSTATE_ENABLED, TBSTYLE_DROPDOWN |
                                TBSTYLE_AUTOSIZE, IDS_MENU, 0, 0,
END

ID_MENU_SP MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "Reply",          ID_CMDREPLY
        MENUITEM "New Message"     ID_CMDNEW
        MENUITEM SEPARATOR
        MENUITEM "Delete",         ID_CMDDEL
    END
END
//-----
// SoftKeyBar resource on Compose dialog for Smartphone
//
ID_DLGMENU_SP RCDATA MOVEABLE PURE
BEGIN
    ID_MENU_SP, 2,
    I_IMAGENONE, ID_CMDSND, TBSTATE_ENABLED, TBSTYLE_BUTTON |
                                TBSTYLE_AUTOSIZE, IDS_SEND, 0, NOMENU,
    I_IMAGENONE, IDCANCEL, TBSTATE_ENABLED, TBSTYLE_BUTTON |
                                TBSTYLE_AUTOSIZE, IDS_CANCEL, 0, NOMENU,
END
//-----
// Menu bar resource main window for Pocket PC
//
ID_MENU_PPC RCDATA MOVEABLE PURE
BEGIN
    ID_MENU_PPC, 2,
    I_IMAGENONE, IDFILE, TBSTATE_ENABLED, TBSTYLE_DROPDOWN |
                                TBSTYLE_AUTOSIZE, IDS_FILE, 0, 0,
    I_IMAGENONE, IDPOP, TBSTATE_ENABLED, TBSTYLE_DROPDOWN |
                                TBSTYLE_AUTOSIZE, IDS_MSG, 0, 1
END

ID_MENU_PPC MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&About",          IDM_ABOUT
        MENUITEM "E&xit",           IDOK
    END
END

```

```

END
POPUP "&Help"
BEGIN
    MENUITEM "&Delete",                ID_CMDDEL
    MENUITEM SEPARATOR
    MENUITEM "&Reply",                ID_CMDREPLY
    MENUITEM "New Message"            ID_CMDNEW
END
END

```

SMSTalk.h

```

//=====
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 2003 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.
    BOOL (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                // Structure associates
    UINT Code;                    // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD); // function.
};

//-----
// Program defines used by application
//
typedef struct {
    SMS_ADDRESS smsAddr;
    SYSTEMTIME stMsg;
    int nSize;
    WCHAR wcMessage[160];
} MYMSG_STRUCT, *PMYMSG_STRUCT;

#define MAX_MSGS 250
typedef struct {
    int nMsgCnt;
    MYMSG_STRUCT pMsgs[MAX_MSGS];
} MYMSG_DBASE, *PMYMSG_DBASE;

//-----
// Generic defines used by application

#define MYMSG_TELLNOTIFY    (WM_USER + 100)

#define ID_ICON            1

#define ID_MENU_SP         100
#define ID_MENU_PPC        101
#define ID_DLGMENU_SP      102

#define IDD_MSGLIST        110                // Control IDs
#define IDD_MSGLISTUD      111
#define IDD_MSGTEXT        112
#define IDD_MSGTEXTUD      113
#define IDD_MSGADDR        114

#define IDM_EXIT            200
#define ID_CMSEND          201
#define ID_CMDNEW          202

```

```
#define ID_CMDREPLY      203
#define ID_CMDDEL       204
#define ID_CMDREAD      205
#define IDM_ABOUT       206
#define IDFILE          207
#define IDPOP           208

#define IDS_EXIT        401
#define IDS_MENU        402
#define IDS_MSG         403
#define IDS_FILE        404
#define IDS_OK          405
#define IDS_CANCEL      406
#define IDS_SEND        407

//-----
// Function prototypes
//
void ErrorBox (HWND hWnd, LPCTSTR lpszFormat, ...);
BOOL OnSmartPhone(void);
int RefreshMessageList (HWND hWnd, int nSel);
int SetButtons (HWND hWnd);

// Window procedures
BOOL CALLBACK MainDlgProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK WriteDlgProc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK AboutDlgProc(HWND, UINT, WPARAM, LPARAM);

// Message handlers
BOOL DoCreateDialogMain (HWND, UINT, WPARAM, LPARAM);
BOOL DoInitDialogMain (HWND, UINT, WPARAM, LPARAM);
BOOL DoCommandMain (HWND, UINT, WPARAM, LPARAM);
BOOL DoHotKeyMain (HWND, UINT, WPARAM, LPARAM);
BOOL DoTellNotifyMain (HWND, UINT, WPARAM, LPARAM);

// Command functions
LPARAM DoMainCommandExit (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandDelMessage (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandReplyMessage (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandNewMessage (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandMsgList (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandAbout (HWND, WORD, HWND, WORD);

// Thread prototype
DWORD WINAPI MonitorThread (PVOID pArg);
```

SMSTalk.cpp

```
//=====
// SMSTalk - Demonstrates SMS messaging system
//
// Written for the book Programming Windows CE
// Copyright (C) 2003 Douglas Boling
//=====
#include <windows.h>           // For all that Windows stuff
#include <aygshell.h>         // Extended shell defines
#include <tpcshell.h>
#include <sms.h>              // SMS functions
#include "SMSTalk.h"          // Program-specific stuff

#define MY_MSGWAITING_STRING TEXT("SMSMsgReadEvent")
#define EMPTY_MSG_LIST      TEXT("<No new messages>")
#define MAXMESSAGELEN       4096
//-----
// Global data
//
const TCHAR szAppName[] = TEXT ("SMSTalk");
const TCHAR szOtherApp[] = TEXT("Another application already \
has the SMS system open.\n\nPlease close the (email?) application");
HINSTANCE hInst;              // Program instance handle
```

```

HWND g_hMain = 0;
HANDLE g_hReadEvent = 0;
HANDLE g_hQuitEvent = 0;
BOOL g_fContinue = TRUE;
BOOL g_fOnSPhone = FALSE;
PMYMSG_DBASE g_pMsgDB = 0;

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateDialogMain,
    WM_INITDIALOG, DoInitDialogMain,
    WM_COMMAND, DoCommandMain,
    WM_HOTKEY, DoHotKeyMain,
    MYMSG_TELLNOTIFY, DoTellNotifyMain,
};
// Command Message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDD_MSGLIST, DoMainCommandMsgList,
    IDOK, DoMainCommandExit,
    IDCANCEL, DoMainCommandExit,
    ID_CMDREPLY, DoMainCommandReplyMessage,
    ID_CMDNEW, DoMainCommandNewMessage,
    ID_CMDDEL, DoMainCommandDelMessage,
    IDM_ABOUT, DoMainCommandAbout,
};
//=====
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    INT i;
    HWND hWnd;
    HANDLE hThread;
    TCHAR szDlgTemplate[32];
    SMS_HANDLE smshHandle;

    hInst = hInstance;

    // Look to see if another instance of the app is running.
    hWnd = FindWindow (NULL, szAppName);
    // See if we were launched with a command line
    if (*lpCmdLine) {
        // Check to see if app started due to notification.
        if (lstrcmp (lpCmdLine, MY_MSGWAITING_STRING) == 0) {
            if (hWnd) {
                SendMessage (hWnd, MYMSG_TELLNOTIFY, 0, 0);
            }
        }
    }
    // Set first instance to the foreground and exit
    if (hWnd) {
        SetForegroundWindow (((HWND)(((DWORD)hWnd) | 0x01)));
        return 0;
    }
    // See if we're running on a smartphone.
    g_fOnSPhone = OnSmartPhone ();

    // Allocate message array
    g_pMsgDB = (PMYMSG_DBASE)LocalAlloc (LPTR, sizeof (MYMSG_DBASE));
    if (g_pMsgDB == 0) {
        ErrorBox (NULL, TEXT("Out of memory"));
        return -1;
    }
    g_pMsgDB->nMsgCnt = 0;

    // Create secondary thread for timer event notification.
    // then try to open an SMS Handle for reading messages.
    g_hQuitEvent = CreateEvent (NULL, FALSE, FALSE, NULL);
    g_hReadEvent = CreateEvent (NULL, FALSE, FALSE, NULL);
    HRESULT hr = SmsOpen (SMS_MSGTYPE_TEXT, SMS_MODE_RECEIVE,

```

```

        &smshHandle, &g_hReadEvent);
if (hr == SMS_E_RECEIVEHANDLEALREADYOPEN) {
    MessageBox (hwnd, (LPCTSTR)szOtherApp);
    return 0;
} else if (hr != ERROR_SUCCESS) {
    MessageBox (hwnd, TEXT("SmsOpen fail %x %d"), hr, GetLastError());
    return 0;
}
hThread = CreateThread (NULL, 0, MonitorThread, (PVOID)smshHandle,
                        0, (DWORD *)&i);
if (hThread == 0)
    return -1;

// Display dialog box as main window. Use different template if
// running on the smartphone
if (g_fOnSPhone)
    _tcscpy (szDlgTemplate, TEXT("SMSTalk_SP"));
else
    _tcscpy (szDlgTemplate, TEXT("SMSTalk_PPC"));

DialogBoxParam (hInstance, szDlgTemplate, NULL, MainDlgProc,
                (LPARAM)lpCmdLine);
// Signal notification thread to terminate
g_fContinue = FALSE;
SetEvent (g_hQuitEvent);
WaitForSingleObject (hThread, 1000);
CloseHandle (hThread);
CloseHandle (g_hQuitEvent); // Don't close ReadEvent, SMS does that
if (g_pMsgDB) LocalFree (g_pMsgDB);
return 0;
}
//=====
// Message handling procedures for main window
//-----
// MainDlgProc - Callback function for application window
//
BOOL CALLBACK MainDlgProc (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 FALSE;
}
//-----
// DoCreateDialogMain - Process WM_CREATE message for window.
//
BOOL DoCreateDialogMain (HWND hwnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {

    if (!g_fOnSPhone) {
        // set up Menu bar for Pocket PC
        SHMENUBARINFO mbi;

        memset (&mbi, 0, sizeof(SHMENUBARINFO));
        mbi.cbSize = sizeof(SHMENUBARINFO);
        mbi.hwndParent = hwnd;
        mbi.nToolBarId = ID_MENU_PPC; // IDM_MENU;
        mbi.hInstRes = hInst;

        // If we could not initialize the dialog box, return an error
        if (!SHCreateMenuBar(&mbi)) {
            MessageBox (hwnd, TEXT("Menubar failed"));
            DestroyWindow (hwnd);
        }
    }
}

```

```

        return FALSE;
    }
}
return TRUE;
}
//-----
// DoInitDialogMain - Process WM_INITDIALOG message for window.
//
BOOL DoInitDialogMain (HWND hWnd, UINT wParam, WPARAM wParam,
                      LPARAM lParam) {
    // Save the window handle
    g_hMain = hWnd;

    // Specify that the dialog box should stretch full screen
    SHINITDLGINFO shidi;
    memset (&shidi, 0, sizeof(shidi));
    shidi.dwMask = SHIDIM_FLAGS;
    shidi.dwFlags = SHIDIF_SIZEIDLG ;//SHIDIF_SIZEIDLGFULLSCREEN
    shidi.hDlg = hWnd;
    if(!SHInitDialog(&shidi))
        return FALSE;

    // Create menubar
    SHMENUBARINFO mbi;
    memset (&mbi, 0, sizeof(SHMENUBARINFO));
    mbi.cbSize = sizeof(SHMENUBARINFO);
    mbi.hwndParent = hWnd;
    if (g_fOnSPhone)
        mbi.nToolBarId = ID_MENU_SP;
    else
        mbi.nToolBarId = ID_MENU_PPC;
    mbi.hInstRes = hInst;

    // If we could not initialize the dialog box, return an error
    if (!SHCreateMenuBar(&mbi)) {
        MessageBox (hWnd, TEXT("Menubar failed"));
        DestroyWindow (hWnd);
        return FALSE;
    }

    // This is only needed on the smartphone
    if (g_fOnSPhone) {
        // Override back key since we have an edit control
        SendMessage (SHFindMenuBar (hWnd), SHCMBM_OVERRIDEKEY, VK_TBACK,
                     MAKELPARAM (SHMBOF_NODEFAULT | SHMBOF_NOTIFY,
                                   SHMBOF_NODEFAULT | SHMBOF_NOTIFY));
    }
    // set the title bar
    SHSetNavBarText (hWnd, TEXT("SMS Talk"));

    SendDlgItemMessage (hWnd, IDD_MSGLIST, LB_ADDSTRING, 0,
                       (LPARAM)EMPTY_MSG_LIST);
    SetButtons (hWnd);
    return TRUE;
}
//-----
// DoCommandMain - Process WM_COMMAND message for window.
//
BOOL DoCommandMain (HWND hWnd, UINT wParam, 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) {
            (*MainCommandItems[i].Fxn)(hWnd, idItem, hWndCtl,
                                        wNotifyCode);
            return TRUE;
        }
    }
    return FALSE;
}

//-----
// DoHotKeyMain - Process WM_HOTKEY message for window.
//
BOOL DoHotKeyMain (HWND hWnd, UINT wParam, WPARAM wParam, LPARAM lParam) {

    SHSendBackToFocusWindow (wParam, wParam, lParam);
    return 0;
}

//-----
// DoTellNotifyMain - Process MYMSG_TELLNOTIFY message for window.
//
BOOL DoTellNotifyMain (HWND hWnd, UINT wParam, WPARAM wParam,
                      LPARAM lParam) {
    RefreshMessageList (hWnd, lParam);
    SetButtons (hWnd);
    return 0;
}

//=====
// Command handler routines
//-----
// DoMainCommandExit - Process Program Exit command.
//
LPARAM DoMainCommandExit (HWND hWnd, WORD idItem, HWND hWndCtl,
                          WORD wNotifyCode) {
    EndDialog (hWnd, 0);
    return 0;
}

//-----
// DoMainCommandDelMessage - Process Read message button.
//
LPARAM DoMainCommandDelMessage (HWND hWnd, WORD idItem, HWND hWndCtl,
                               WORD wNotifyCode) {
    int i, nSel;
    nSel = SendDlgItemMessage (hWnd, IDD_MSGLIST, LB_GETCURSEL, 0, 0);
    if (nSel != LB_ERR) {
        for (i = nSel; i < g_pMsgDB->nMsgCnt-1; i++)
            g_pMsgDB->pMsgs[i] = g_pMsgDB->pMsgs[i+1];
        g_pMsgDB->nMsgCnt--;
        RefreshMessageList (hWnd, -1);
    }
    SetButtons (hWnd);
    return 0;
}

//-----
// DoMainCommandReplyMessage - Process Reply message button.
//
LPARAM DoMainCommandReplyMessage (HWND hWnd, WORD idItem, HWND hWndCtl,
                                  WORD wNotifyCode) {
    int nSel;
    LPCTSTR lpTemplate;
    LPARAM lp = 0;
    nSel = SendDlgItemMessage (hWnd, IDD_MSGLIST, LB_GETCURSEL, 0, 0);
    if (nSel != LB_ERR)
        lp = (LPARAM)&g_pMsgDB->pMsgs[nSel].smsAddr;

    // Display reply dialog box.
    if (g_fOnSPhone)
        lpTemplate = TEXT("WriteMsgDlg_SP");
    else
        lpTemplate = TEXT("WriteMsgDlg_PPC");

    DialogBoxParam (hInst, lpTemplate, NULL, WriteDlgProc, lp);
}

```

```

    return 0;
}
//-----
// DoMainCommandNewMessage - Process New message button.
//
LPARAM DoMainCommandNewMessage (HWND hWnd, WORD idItem, HWND hwndCtl,
                                WORD wNotifyCode) {
    LPCTSTR lpTemplate;
    // Display reply dialog box.
    if (g_fOnSPhone)
        lpTemplate = TEXT("WriteMsgDlg_SP");
    else
        lpTemplate = TEXT("WriteMsgDlg_PPC");

    DialogBoxParam (hInst, lpTemplate, NULL, WriteDlgProc, 0);
    return 0;
}
//-----
// DoMainCommandAbout - Process About menu item.
//
LPARAM DoMainCommandAbout (HWND hWnd, WORD idItem, HWND hwndCtl,
                           WORD wNotifyCode) {
    TCHAR szAbout[] = TEXT("SMS Talk\nCopyright 2003\nDouglas Boling");

    // Display about information in a message box.
    MessageBox (hWnd, szAbout, TEXT("About"), MB_OK | MB_ICONASTERISK);
    return 0;
}
//-----
// DoMainCommandMsgList - Process message list listbox.
//
LPARAM DoMainCommandMsgList (HWND hWnd, WORD idItem, HWND hwndCtl,
                             WORD wNotifyCode) {
    if (wNotifyCode == LBN_SELCHANGE)
        SetButtons (hWnd);
    return 0;
}
//-----
// RefreshMessageList - Fill in message listbox from message array
//
int RefreshMessageList (HWND hWnd, int nSel) {
    TCHAR szStr[256];
    int i;

    SendDlgItemMessage (hWnd, IDD_MSGLIST, LB_RESETCONTENT, 0, 0);
    for (i = 0; i < g_pMsgDB->nMsgCnt; i++) {
        wprintf (szStr, TEXT("%d/%02d/%02d %d:%02d:%02d %s"),
                g_pMsgDB->pMsgs[i].stMsg.wMonth,
                g_pMsgDB->pMsgs[i].stMsg.wDay,
                g_pMsgDB->pMsgs[i].stMsg.wYear%100,
                g_pMsgDB->pMsgs[i].stMsg.wHour,
                g_pMsgDB->pMsgs[i].stMsg.wMinute,
                g_pMsgDB->pMsgs[i].stMsg.wSecond,
                g_pMsgDB->pMsgs[i].wcMessage);
        SendDlgItemMessage (hWnd, IDD_MSGLIST, LB_ADDSTRING, 0,
                            (LPARAM)szStr);
    }
    if (g_pMsgDB->nMsgCnt == 0)
        SendDlgItemMessage (hWnd, IDD_MSGLIST, LB_ADDSTRING, 0,
                            (LPARAM)EMPTY_MSG_LIST);
    else {
        if (nSel != -1)
            SendDlgItemMessage (hWnd, IDD_MSGLIST, LB_SETCURSEL, 0, nSel);
        else
            SendDlgItemMessage (hWnd, IDD_MSGLIST, LB_SETCURSEL, 0, i-1);
    }
    return 0;
}
//-----
// SetButtons - Utility function to compute enabled state of btns

```

```
//
int SetButtons (HWND hWnd) {
    int nSel;
    BOOL fReply = FALSE;
    TCHAR szText[128];
    LPTSTR pMsg = TEXT(""), pNum = TEXT("");

    nSel = SendDlgItemMessage (hWnd, IDD_MSGLIST, LB_GETCURRESEL, 0, 0);
    if (nSel != LB_ERR) {
        SendDlgItemMessage (hWnd, IDD_MSGLIST, LB_GETTEXT, nSel,
            (LPARAM)szText);
        if (_tcscmp (szText, EMPTY_MSG_LIST)) {
            fReply = TRUE;
            pNum = g_pMsgDB->pMsgs[nSel].smsAddr.ptsAddress;
            pMsg = g_pMsgDB->pMsgs[nSel].wcMessage;
        }
    }
    EnableWindow (GetDlgItem (hWnd, ID_CMDREPLY), fReply);

    // Set the text in the number and message fields
    SetWindowText (GetDlgItem (hWnd, IDD_MSGADDR), pNum);
    SetWindowText (GetDlgItem (hWnd, IDD_MSGTEXT), pMsg);

    // Disable the menu bar button if necessary
    TBBUTTONINFO tbi;
    HWND hwndMB = SHFindMenuBar (hWnd);
    memset (&tbi, 0, sizeof (tbi));
    tbi.cbSize = sizeof (tbi);
    tbi.dwMask = TBIF_STATE;
    if(SendMessage (hwndMB, TB_GETBUTTONINFO, IDPOP, (LPARAM)&tbi)) {
        if (fReply)
            tbi.fsState |= TBSTATE_ENABLED;
        else
            tbi.fsState &= ~TBSTATE_ENABLED;
        SendMessage (hwndMB, TB_SETBUTTONINFO, IDPOP, (LPARAM)&tbi);
    }
    return nSel;
}
//-----
// SendSmsMessage - Send an SMS message
//
HRESULT SendSmsMessage (HWND hWnd, SMS_ADDRESS smsDest, LPTSTR pMsg) {
    HRESULT hr;
    SMS_HANDLE smshHandle;
    TEXT_PROVIDER_SPECIFIC_DATA tpsd;
    SMS_MESSAGE_ID smsmidMessageID = 0;

    // try to open an SMS Handle
    hr = SmsOpen(SMS_MSGTYPE_TEXT, SMS_MODE_SEND, &smshHandle, NULL);
    if (hr != ERROR_SUCCESS)
        return hr;

    // Set up provider specific data
    tpsd.dwMessageOptions = PS_MESSAGE_OPTION_NONE;
    tpsd.psMessageClass = PS_MESSAGE_CLASS0;
    tpsd.psReplaceOption = PSRO_NONE;
    tpsd.dwHeaderDataSize = 0;

    // Send the message, indicating success or failure
    hr = SmsSendMessage (smshHandle, NULL, &smsDest, NULL, (PBYTE)pMsg,
        lstrlen(pMsg) * sizeof (TCHAR),
        (PBYTE) &tpsd, 12, SMSDE_OPTIMAL,
        SMS_OPTION_DELIVERY_NONE, &smsmidMessageID);
    SmsClose (smshHandle);
    return hr;
}
//=====
// WriteMsg Dialog procedure
//
BOOL CALLBACK WroteDlgProc (HWND hWnd, UINT wParam, WPARAM wParam,
```

```

        LPARAM lParam) {
SHINITDLGINFO shidi;
static SMS_ADDRESS smsDest;
TCHAR szMsg[SMS_DATAGRAM_SIZE+2];
HRESULT hr;

SHInputDialog (hWnd, wMsg, wParam);
switch (wMsg) {
case WM_INITDIALOG:
    // Specify that the dialog box should stretch full screen
    memset (&shidi, 0, sizeof(shidi));
    shidi.dwMask = SHIDIM_FLAGS;
    shidi.dwFlags = SHIDIF_SIZEDLGFULLSCREEN;
    if (!g_fOnSPhone) shidi.dwFlags |= SHIDIF_DONEBUTTON;
    shidi.hDlg = hWnd;
    if(!SHInitDialog(&shidi))
        return FALSE;

    // This is only needed on the smartphone
    if (g_fOnSPhone) {
        // Create MenuBar
        SHMENUBARINFO mbi;
        memset (&mbi, 0, sizeof(SHMENUBARINFO));
        mbi.cbSize = sizeof(SHMENUBARINFO);
        mbi.hwndParent = hWnd;
        mbi.nToolBarId = ID_DLGMENU_SP;
        mbi.hInstRes = hInst;

        // If we could not initialize the dialog box, return an error
        if (!SHCreateMenuBar(&mbi)) {
            MessageBox (hWnd, TEXT("Menubar failed"));
            DestroyWindow (hWnd);
            return FALSE;
        }

        // Override back key since we have an edit control
        SendMessage (SHFindMenuBar (hWnd), SHCMBM_OVERRIDEKEY, VK_TBACK,
            MAKELPARAM (SHMBOF_NODEFAULT | SHMBOF_NOTIFY,
                SHMBOF_NODEFAULT | SHMBOF_NOTIFY));
        // Set input mode of number field to numbers
        SendDlgItemMessage (hWnd, IDD_MSGADDR, EM_SETINPUTMODE, 0,
            EIM_NUMBERS);
    }

    SendDlgItemMessage (hWnd, IDD_MSGTEXT, EM_LIMITTEXT,
        SMS_DATAGRAM_SIZE, 0);
    // If there is a reply address passed, place it in control
    if (lParam) {
        // Copy dest address
        smsDest = *(SMS_ADDRESS *)lParam;
        SetDlgItemText (hWnd, IDD_MSGADDR, smsDest.ptsAddress);
        SetFocus (GetDlgItem (hWnd, IDD_MSGTEXT));
        return FALSE;
    } else {
        smsDest.smsatAddressType = SMSAT_INTERNATIONAL;
        memset (smsDest.ptsAddress, 0,
            sizeof (smsDest.ptsAddress));
    }
    return TRUE;

case WM_HOTKEY:
    SHSendBackToFocusWindow (wMsg, wParam, lParam);
    return TRUE;

case WM_COMMAND:
    switch (LOWORD (wParam)) {
        case ID_CMDSEND:
            GetDlgItemText (hWnd, IDD_MSGADDR, smsDest.ptsAddress,
                dim (smsDest.ptsAddress));
            GetDlgItemText (hWnd, IDD_MSGTEXT, szMsg, dim (szMsg));

```

```

        hr = SendSmsMessage (hWnd, smsDest, szMsg);
        if (hr == 0)
            MessageBox (hWnd, TEXT("Message Sent"), szAppName,
                        MB_OK | MB_ICONASTERISK);
        else
            ErrorBox (hWnd, TEXT ("Send message fail %x %d"),
                    hr, GetLastError());
    case IDOK:
    case IDCANCEL:
        EndDialog (hWnd, 0);
        return TRUE;
    }
    break;
}
return FALSE;
}
//-----
// ErrorBox - Displays a message box with a formatted string
//
void ErrorBox (HWND hWnd, LPCTSTR lpszFormat, ...) {
    int nBuf;
    TCHAR szBuffer[512];

    va_list args;
    va_start(args, lpszFormat);

    nBuf = _vstprintf(szBuffer, lpszFormat, args);
    MessageBox (hWnd, szBuffer, TEXT("Error Msg"), MB_OK);
    va_end(args);
}
//-----
// OnSmartPhone - Determines if we're running on a smartphone
//
BOOL OnSmartPhone (void) {
    TCHAR szPlat[128];
    int rc;
    rc = SystemParametersInfo(SPI_GETPLATFORMTYPE, dim(szPlat), szPlat, 0);
    if (rc) {
        if (lstrcmpi (szPlat, TEXT("Smartphone"))) == 0)
            return TRUE;
    }
    return FALSE;
}
//=====
// MonitorThread - Monitors event for timer notification
//
DWORD WINAPI MonitorThread (PVOID pArg) {
    TEXT_PROVIDER_SPECIFIC_DATA tpsd;
    SMS_HANDLE smshHandle = (SMS_HANDLE)pArg;
    PMYMSG_STRUCT pNextMsg;
    BYTE bBuffer[MAXMESSAGELEN];
    PBYTE pIn;
    SYSTEMTIME st;
    HANDLE hWait[2];
    HRESULT hr;
    int rc;
    DWORD dwInSize, dwSize, dwRead = 0;

    hWait[0] = g_hReadEvent;          // Need two events since it isn't
    hWait[1] = g_hQuitEvent;         // allowed for us to signal SMS event.

    while (g_fContinue) {
        rc = WaitForMultipleObjects (2, hWait, FALSE, INFINITE);
        if (!g_fContinue || (rc != WAIT_OBJECT_0))
            break;
        // Point to the next free entry in the array
        pNextMsg = &g_pMsgDB->pMsgs[g_pMsgDB->nMsgCnt];

        // Get the message size
        hr = SmsGetMessageSize (smshHandle, &dwSize);

```

```

if (hr != ERROR_SUCCESS) continue;

// Check for message larger than std buffer
if (dwSize > sizeof (pNextMsg->wcMessage)) {
    if (dwSize > MAXMESSAGELEN)
        continue;
    pIn = pBuffer;
    dwInSize = MAXMESSAGELEN;
} else {
    pIn = (PBYTE)pNextMsg->wcMessage;
    dwInSize = sizeof (pNextMsg->wcMessage);
}
// Set up provider specific data
tpsd.dwMessageOptions = PS_MESSAGE_OPTION_NONE;
tpsd.psMessageClass = PS_MESSAGE_CLASS0;
tpsd.psReplaceOption = PSRO_NONE;
tpsd.dwHeaderDataSize = 0;
// Read the message
hr = SmsReadMessage (smshHandle, NULL, &pNextMsg->smsAddr, &st,
                    (PBYTE)pIn, dwInSize, (PBYTE)&tpsd,
                    sizeof(TEXT_PROVIDER_SPECIFIC_DATA),
                    &dwRead);
if (hr == ERROR_SUCCESS) {
    // Convert GMT message time to local time
    FILETIME ft, ftLocal;
    SystemTimeToFileTime (&st, &ft);
    FileTimeToLocalFileTime (&ft, &ftLocal);
    FileTimeToSystemTime (&ftLocal, &pNextMsg->stMsg);

    // If using alt buffer, copy to std buff
    if ((DWORD)pIn == (DWORD)pNextMsg->wcMessage) {
        pNextMsg->nSize = (int) dwRead;
    } else {
        memset (pNextMsg->wcMessage, 0,
                sizeof(pNextMsg->wcMessage));
        memcpy (pNextMsg->wcMessage, pIn,
                sizeof(pNextMsg->wcMessage)-2);
        pNextMsg->nSize = sizeof(pNextMsg->wcMessage);
    }
    // Increment message count
    if (g_pMsgDB->nMsgCnt < MAX_MSGS-1) {
        if (g_hMain)
            PostMessage (g_hMain, MYMSG_TELLNOTIFY, 1,
                        g_pMsgDB->nMsgCnt);
        g_pMsgDB->nMsgCnt++;
    }
} else {
    MessageBox (g_hMain, TEXT("Error %x (%d) reading msg"),
                hr, GetLastError());
    break;
}
}
SmsClose (smshHandle);
return 0;
}

```

Listing 19-2

The example illustrates a number of techniques used in a Smartphone application. The dialog templates use both expandable edit fields and spinner controls. The Back button is overridden in both the main dialog and the Compose dialog because both contain edit controls. The edit control that holds the destination number in the Compose dialog has its input mode overridden to numeric mode because the address to be entered is more than likely a phone number.

The SMS code in the application uses a separate thread, appropriately named *MonitorThread*, to monitor incoming SMS messages and uses a send routine, *SendSmsMessage*, that sends messages. Before the monitor thread is launched, the

SmsOpen is called to request a read access handle. This call will fail if another application currently has an open SMS handle with read access. If the open fails, *SMSTalk* displays a message box notifying the user of the problem and terminates. On the Pocket PC, this failure is quite likely because the e-mail program Pocket Inbox stays running in the background. Pocket Inbox can be closed by selecting the Inbox from the Start menu and then entering Ctrl+Q from the soft keyboard.

Sending an SMS message is accomplished by selecting the Reply or the New menu item. First a dialog is displayed so that the message can be composed. If the user selects to send the message, the program calls *SendSmsMessage*. This routine opens an SMS handle for write access, fills in the appropriate structures, and sends the message in the simple text format.

Incoming messages are saved in an array in memory. The messages can then be viewed by highlighting a message in the list box on the Pocket PC, or in the spinner on the Smartphone. *SMSTalk* does not save the messages because the goal is to demonstrate the features of the SMS system and the Smartphone with the least amount of clutter. Because of this arrangement, any messages received by *SMSTalk* will be lost when the program terminates. Saving the messages in a file would be a great enhancement and is left as a task for the reader.

Smartphone Security

The Smartphone has one significant difference from the Pocket PC in that it implements Windows CE's module-level security scheme. This means that unless a specific device has been "unlocked", all applications and DLLs have to be certified to run on the device. As I mentioned in Chapter 10, the system can prevent modules, EXEs and DLLs from being run at all. Even when allowed to run, there are two levels of run-time privilege that can be granted, *trusted* and *run*. Trusted modules have full run of the operating system. They can call any function and modify any registry key. Run-level modules, sometimes called *untrusted modules*, can only call a limited set of the API and are prevented from modifying certain critical sections of the registry. Not that the limited set of functions is all that limited—a run-level module can still call 95 percent of the functions—it's just that the module can't call the set of functions that might compromise the integrity of the system. In addition to the restricted functions listed in Listing 10-1 in Chapter 10, the Smartphone restricts an additional set of communication-related functions. A list of these addition functions is shown in Figure 19-10.

ExtAPI

lineRegister
lineSetCallBarringPassword
lineSetCallBarringState
lineUnregister
lineSetPreferredOperator
lineSetEquipmentState
lineGetGeneralInfo
lineManageCalls
lineSetGprsClass
lineGetNumberCalls
lineSetHSCSDState
lineGetUSSD
lineSendUSSD
lineSetSendCallerIDState
lineSetCallWaitingState

Short Message Service

SmsSetMessageNotification
SmsClearMessageNotification
SmsReceiveAllMessagesFromSIM

SIM Manager

simUnlockPhone
simSetLockingStatus
simGetSmsStorageStatus
simChangeLockingPassword
simReadMessage
simWriteMessage
simDeleteMessage
simReadRecord
simWriteRecord
simGetRecordInfo

CPM

CPMRegister
CPMShutdown
CPMStatus
CPMRegisterTest

Connection Manager

ConnMgrProviderMessage

Radio Interface Layer

SmsSetSMSC

All Radio Interface Layer functions*

* The system can be configured to allow the RIL functions to be called by untrusted applications

Figure 19-10

The list of restricted communication functions in the Smartphone

The level of security implemented on a particular Smartphone is set by the telecommunication service that sells the phone. The service defines the protection level after considering security on its network, its phones, and the profit potential of restricting all software to be sold through its own service. Market pressures will drive this issue's evolution.

These last few chapters have covered the Pocket PC and Smartphone in detail. Now it's time for some fun and games. Both the Smartphone and Pocket PC support a unique API called the Game API, to assist game writers in implementing cool games on these devices. GAPI is simple, fairly straightforward, and kind of neat.