Figure 7-5. *continued*

```
// ValidateTime - Trival error checking of time field
//
BOOL ValidateTime (TCHAR *pStr) {
    BOOL fSep = FALSE;
    TCHAR *pPtr;

    pPtr = pStr;
    // See if field contains only numbers and up to one colon.
    while (*pPtr) {
        if (*pPtr == TEXT (':')) {
            if (fSep)
                return FALSE;
            fSep = TRUE;
        } else if ((*pPtr < TEXT ('0')) || (*pPtr > TEXT ('9')))
            return FALSE;
        pPtr++;
    }
    // Reject empty field.
    if (pPtr > pStr)
        return TRUE;
    return FALSE;
}
//======================================================================
// EditTrack dialog procedure
//
BOOL CALLBACK EditTrackDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                                LPARAM lParam) {
    static LPTRACKINFO lpti;

    switch (wMsg) {
        case WM_INITDIALOG:
            lpti = (LPTRACKINFO)lParam;
            SendDlgItemMessage (hWnd, IDD_TRACK, EM_SETLIMITTEXT,
                                sizeof (lpti->szTrack), 0);
            SendDlgItemMessage (hWnd, IDD_TIME, EM_SETLIMITTEXT,
                                sizeof (lpti->szTime), 0);
            // See if new album or edit of old one.
            if (lstrlen (lpti->szTrack) == 0) {
                SetWindowText (hWnd, TEXT ("New Track"));
            } else {
                SetDlgItemText (hWnd, IDD_TRACK, lpti->szTrack);
                SetDlgItemText (hWnd, IDD_TIME, lpti->szTime);
            }
            return TRUE;
        case WM_COMMAND:
            switch (LOWORD (wParam)) {
                case IDOK:
                    Edit_GetText (GetDlgItem (hWnd, IDD_TRACK),
                               lpti->szTrack, sizeof (lpti->szTrack));
                    Edit_GetText (GetDlgItem (hWnd, IDD_TIME),
                                lpti->szTime, sizeof (lpti->szTime));
                    if (ValidateTime (lpti->szTime))
                        EndDialog (hWnd, 1);
                    else
                        MessageBox (hWnd, TEXT ("Track time must \
be entered in mm:ss format"),
                                      TEXT ("Error"), MB_OK);
                    return TRUE;
                case IDCANCEL:
                    EndDialog (hWnd, 0);
                    return TRUE;
            }
        break;
    }
    return FALSE;
}
//======================================================================
// EditAlbum dialog procedure
//
BOOL CALLBACK EditAlbumDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                                LPARAM lParam) {
    static PCEPROPVAL *ppRecord;
    static int nTracks;
    PCEPROPVAL pRecord, pRecPtr;
    TCHAR *pPtr, szTmp[128];
    HWND hwndTList, hwndCombo;
    TRACKINFO ti;
    BOOL fEnable;
    INT i, nLen, rc;

    switch (wMsg) {
        case WM_INITDIALOG:
            ppRecord = (PCEPROPVAL *)lParam;
            pRecord = *ppRecord;

            hwndCombo = GetDlgItem (hWnd, IDD_CATEGORY);
            hwndTList = GetDlgItem (hWnd, IDD_TRACKS);

            Edit_LimitText (GetDlgItem (hWnd, IDD_NAME), MAX_NAMELEN);
```

(continued)

Figure 7-5. *continued*

```
        Edit_LimitText (GetDlgItem (hWnd, IDD_ARTIST),
                        MAX_ARTISTLEN);
        // Set tabstops on track list box.
        i = 110;
        ListBox_SetTabStops (hwndTList, 1, &i);
        // Initialize category combo box.
        for (i = 0; i < dim(pszCategories); i++)
            ComboBox_AddString (hwndCombo, pszCategories[i]);
        ComboBox_SetCurSel (hwndCombo, 3);
        nTracks = 0;

        // See if new album or edit of old one.
        if (pRecord == 0) {
            SetWindowText (hWnd, TEXT ("New Album"));
        } else {
            // Copy the data from the record to album structure.
            for (i = 0; i < NUM_DB_PROPS; i++) {
                switch (pRecord->propid) {
                case PID_NAME:
                    SetDlgItemText (hWnd, IDD_NAME,
                                    pRecord->val.lpwstr);
                    break;
                case PID_ARTIST:
                    SetDlgItemText (hWnd, IDD_ARTIST,
                                    pRecord->val.lpwstr);
                    break;
                case PID_CATEGORY:
                    ComboBox_SetCurSel (hwndCombo,
                                        pRecord->val.iVal);
                    break;
                case PID_TRACKS:
                    pPtr = (TCHAR *)pRecord->val.blob.lpb;
                    for (i = 0; *pPtr; i++){
                      ListBox_InsertString (hwndTList,i,pPtr);
                        pPtr += lstrlen (pPtr) + 1;
                        nTracks++;
                    }
                    break;
                }
                pRecord++;
            }
        }
        // Select first track or disable buttons if no tracks.
        if (nTracks)
            ListBox_SetCurSel (GetDlgItem (hWnd, IDD_TRACKS), 3);
            else {
                EnableWindow (GetDlgItem (hWnd, IDD_DELTRACK),
                              FALSE);
                EnableWindow (GetDlgItem (hWnd, IDD_EDITTRACK),
                              FALSE);
            }
            return TRUE;

        case WM_COMMAND:
            hwndTList = GetDlgItem (hWnd, IDD_TRACKS);
            hwndCombo = GetDlgItem (hWnd, IDD_CATEGORY);
            pRecord = *ppRecord;
            switch (LOWORD (wParam)) {
                case IDD_TRACKS:
                    switch (HIWORD (wParam)) {
                    case LBN_DBLCLK:
                        PostMessage (hWnd, WM_COMMAND,
                                     MAKELONG(IDD_EDITTRACK, 0), 0);
                        break;
                    case LBN_SELCHANGE:
                        i = ListBox_GetCurSel (hwndTList);
                        if (i == LB_ERR)
                            fEnable = FALSE;
                        else
                            fEnable = TRUE;
                        EnableWindow (GetDlgItem (hWnd,
                                      IDD_DELTRACK), fEnable);
                        EnableWindow (GetDlgItem (hWnd,
                                      IDD_EDITTRACK), fEnable);
                        break;
                    }
                    return TRUE;

                case IDD_NEWTRACK:
                    memset (&ti, 0, sizeof (ti));
                    rc = DialogBoxParam (hInst,
                        TEXT ("EditTrackDlg"), hWnd,
                        EditTrackDlgProc, (LPARAM)&ti);
                    if (rc) {
                        wsprintf (szTmp, TEXT ("%s\t%s"),
                                  ti.szTrack, ti.szTime);
                        i = ListBox_GetCurSel (hwndTList);
                        if (i != LB_ERR)
                            i++;
                        i = ListBox_InsertString (hwndTList, i,
                                                  szTmp);
```

(continued)

Figure 7-5. *continued*

```
                ListBox_SetCurSel (hwndTList, i);
            }
            return TRUE;

        case IDD_EDITTRACK:
            i = ListBox_GetCurSel (hwndTList);
            if (i != LB_ERR) {
                ListBox_GetText (hwndTList, i, szTmp);
                pPtr = szTmp;
                while ((*pPtr != TEXT ('\t')) &&
                       (*pPtr != TEXT ('\0')))
                    pPtr++;
                if (*pPtr == TEXT ('\t'))
                    *pPtr++ = TEXT ('\0');

                lstrcpy (ti.szTime, pPtr);
                lstrcpy (ti.szTrack, szTmp);
                rc = DialogBoxParam (hInst,
                                     TEXT ("EditTrackDlg"),
                                     hWnd, EditTrackDlgProc,
                                     (LPARAM)&ti);
                if (rc) {
                    wsprintf (szTmp, TEXT ("%s\t%s"),
                              ti.szTrack, ti.szTime);
                    i = ListBox_GetCurSel (hwndTList);
                    ListBox_DeleteString (hwndTList, i);
                    ListBox_InsertString (hwndTList, i,
                                          szTmp);
                    ListBox_SetCurSel (hwndTList, i);
                }
            }
            return TRUE;

        case IDD_DELTRACK:
            // Grab the current selection and remove
            // it from list box.
            i = ListBox_GetCurSel (hwndTList);
            if (i != LB_ERR) {
                rc = MessageBox (hWnd,
                                 TEXT ("Delete this item?"),
                                 TEXT ("Track"), MB_YESNO);
                if (rc == IDYES) {
                    i=ListBox_DeleteString (hwndTList,i);
                    if (i > 0)
                        i--;
                    ListBox_SetCurSel (hwndTList, i);
                }
            }
            return TRUE;

        case IDOK:
            // Be lazy and assume worst case size values.
            nLen = sizeof (CEPROPVAL) * NUM_DB_PROPS +
                   MAX_NAMELEN + MAX_ARTISTLEN +
                   MAX_TRACKNAMELEN;
            // See if prev record, alloc if not.
            if (pRecord) {
                // Resize record if necessary.
                if (nLen > (int)LocalSize (pRecord))
                    pRecPtr =
                       (PCEPROPVAL)LocalReAlloc (pRecord,
                       nLen, LMEM_MOVEABLE);
                else
                    pRecPtr = pRecord;
            } else
                pRecPtr = LocalAlloc (LMEM_FIXED, nLen);
            if (!pRecPtr)
                return 0;
            // Copy the data from the controls to a
            // marshaled data block with the structure
            // at the front and the data in the back.
            pRecord = pRecPtr;
            nTracks = ListBox_GetCount (hwndTList);
            pPtr = (TCHAR *)((LPBYTE)pRecPtr +
                     (sizeof (CEPROPVAL) * NUM_DB_PROPS));
            // Zero structure to start over.
            memset (pRecPtr, 0, LocalSize (pRecPtr));

            pRecPtr->propid = PID_NAME;
            pRecPtr->val.lpwstr = pPtr;
            GetDlgItemText (hWnd, IDD_NAME, pPtr,
                            MAX_NAMELEN);
            pPtr += lstrlen (pPtr) + 1;
            pRecPtr++;

            pRecPtr->propid = PID_ARTIST;
            pRecPtr->val.lpwstr = pPtr;
            GetDlgItemText (hWnd, IDD_ARTIST, pPtr,
                            MAX_ARTISTLEN);
            pPtr += lstrlen (pPtr) + 1;
            pRecPtr++;
```

(continued)

Figure 7-5. *continued*

```
                    pRecPtr->propid = PID_RELDATE;
                    pRecPtr->val.iVal = 0;
                    pRecPtr++;

                    pRecPtr->propid = PID_CATEGORY;
                    pRecPtr->val.iVal =
                                ComboBox_GetCurSel (hwndCombo);
                    pRecPtr++;

                    pRecPtr->propid = PID_NUMTRACKS;
                    pRecPtr->val.iVal = nTracks;
                    pRecPtr++;

                    pRecPtr->propid = PID_TRACKS;
                    pRecPtr->val.blob.lpb = (LPBYTE)pPtr;

                    // Get the track titles from the list box.
                    rc = MAX_TRACKNAMELEN;
                    for (i = 0; i < nTracks; i++) {
                        // Make sure we have the room in the buff.
                        rc -= ListBox_GetTextLen(hwndTList, i);
                        if (rc)
                            ListBox_GetText (hwndTList, i, pPtr);
                        else {
                            nTracks = i;
                            break;
                        }
                        pPtr += lstrlen (pPtr) + 1;
                    }
                    *pPtr++ = TEXT ('\0');
                    pRecPtr->val.blob.dwCount =
                            (LPBYTE)pPtr - pRecPtr->val.blob.lpb;
                    *ppRecord = pRecord;
                    EndDialog (hWnd, 1);
                    return TRUE;

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

```
//======================================================================
// About dialog procedure
//
BOOL CALLBACK AboutDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                            LPARAM lParam) {
    switch (wMsg) {
        case WM_COMMAND:
            switch (LOWORD (wParam)) {
                case IDOK:
                case IDCANCEL:
                    EndDialog (hWnd, 0);
                    return TRUE;
            }
        break;
    }
    return FALSE;
}
```

The program uses a virtual list view control to display the records in the database. As I explained in Chapter 5, virtual list views don't store any data internally. Instead, the control makes calls back to the owning window using notification messages to query the information for each item in the list view control. The WM_NOTIFY handler *OnNotifyMain* calls *GetItemData* to query the database in response to the list view control sending LVN_GETDISPINFO notifications. The *GetItemInfo* function first seeks the record to read then reads all the properties of a database record with one call to *CeReadRecordProps*. Since the list view control typically uses the LVN_GETDISPINFO notification multiple times for one item, *GetItemInfo* saves the data from the last record read. If the next read is of the same record, the program uses the cached data instead of rereading the database.

As I've explained before, you can change the way you sort by simply closing the database and reopening it in one of the other sort modes. The list view control is then invalidated, causing it to again request the data for each record being displayed. With a new sort order defined, the seek that happens with each database record read automatically sorts the data by the sort order defined when the database was opened.

AlbumDB doesn't use the new *Ex* database functions provided by Windows CE 2.1 based systems. This allows the program to run under earlier versions of the operating system. To modify the example to use separate database volumes, only minor changes would be necessary. First a global variable *g_guidDB* of type CEOID would be added. In the *DoCreateMain* routine, code such as the following, which mounts the volume, would be added.

```
if (!CeMountDBVol (&g_guidDB, TEXT ("\\Albums.cdb"), OPEN_ALWAYS)) {
    wsprintf (szErr, TEXT ("Database mount failed. rc %d"),
              GetLastError());
    MessageBox (NULL, szErr, szAppName, MB_OK);
}
```

The following code would be added to the *OnDestroyMain* routine to unmount the volume:

```
if (!CHECK_INVALIDGUID (&g_guidDB))
    CeUnmountDBVol (&g_guidDB);
```

Finally, the *OpenCreateDB* routine would be replaced by this version:

```
HANDLE OpenCreateDB (HWND hWnd, int *pnRecords) {
    INT i, rc;
    CEOIDINFO oidinfo;
    CEDBASEINFO dbi;
    TCHAR szErr[128];
    CENOTIFYREQUEST cenr;

    g_oidDB = 0;
    cenr.dwSize = sizeof (cenr);
    cenr.hWnd = hWnd;
    cenr.dwFlags = 0;                    // Use old style notifications.
    cenr.hHeap = 0;
    cenr.dwParam = 0;

    g_hDB = CeOpenDatabaseEx (&g_guidDB, &g_oidDB, TEXT ("\\Albums"),
                              g_nLastSort, 0, &cenr);
    if (g_hDB == INVALID_HANDLE_VALUE) {
        rc = GetLastError();
        if (rc == ERROR_FILE_NOT_FOUND) {
            i = 0;
            dbi.rgSortSpecs[i].propid = PID_NAME;
            dbi.rgSortSpecs[i++].dwFlags = 0;

            dbi.rgSortSpecs[i].propid = PID_ARTIST;
            dbi.rgSortSpecs[i++].dwFlags = 0;

            dbi.rgSortSpecs[i].propid = PID_CATEGORY;
            dbi.rgSortSpecs[i++].dwFlags = 0;

            dbi.dwFlags = CEDB_VALIDCREATE;
            lstrcpy (dbi.szDbaseName, TEXT ("\\Albums"));
            dbi.dwDbaseType = 0;
            dbi.wNumSortOrder = 3;

            g_oidDB = CeCreateDatabaseEx (&g_guidDB, &dbi);
            if (g_oidDB == 0) {
                wsprintf (szErr,
                          TEXT ("Database create failed. rc %d"),
                          GetLastError());
                MessageBox (hWnd, szErr, szAppName, MB_OK);
                return 0;
            }
            g_hDB = CeOpenDatabaseEx (&g_guidDB, &g_oidDB, NULL,
                                      g_nLastSort, 0,  &cenr);
        }
    } else if (g_hDB == 0){
        wsprintf (szErr,
                  TEXT ("Database open failed. rc %X  ext err:%d"),
                  g_hDB, GetLastError());
        MessageBox (hWnd, szErr, szAppName, MB_OK);
    }
    CeOidGetInfoEx (&g_guidDB, g_oidDB, &oidinfo);
    *pnRecords = oidinfo.infDatabase.wNumRecords;
    return g_hDB;
}
```

THE REGISTRY

The registry is a system database used to store configuration information in applications and in Windows itself. The registry as defined by Windows CE is similar but not identical in function and format to the registries under Windows 98 and Windows NT. In other words, for an application, most of the same registry access functions exist, but the layout of the Windows CE registry doesn't exactly follow either Windows 98 or Windows NT.

As in all versions of Windows, the registry is made up of keys and values. Keys can contain keys or values or both. Values contain data in one of a number of predefined formats. Since keys can contain keys, the registry is distinctly hierarchical. The highest level keys, the root keys, are specified by their predefined numeric constants. Keys below the root keys and values are identified by their text name. Multiple levels of keys can be specified in one text string by separating the keys with a backslash (\).

To query or modify a value, the key containing the value must first be opened, the value queried and or written, then the key closed. Keys and values can also be enumerated so that an application can determine what a specific key contains. Data in the registry can be stored in a number of different predefined data types. Among the available data types are strings, 32-bit numbers, and free form binary data.

Registry Organization

The Windows CE registry supports three of the high-level, root keys seen on other Windows platforms, HKEY_LOCAL_MACHINE, HKEY_CURRENT_USER, and HKEY_CLASSES_ROOT. As with other Windows platforms, Windows CE uses the HKEY_LOCAL_MACHINE key to store hardware and driver configuration data, the HKEY_CURRENT_USER to store user-specific configuration data, and the HKEY_CLASSES_ROOT key to store file type matching and OLE configuration data.

As a practical matter, the registry is used by applications and drivers to store state information that needs to be saved across invocations. Applications typically store their current state when they are requested to close and then restore this state when they are launched again. The traditional location for storing data in the registry by an application is obtained by means of the following structure:

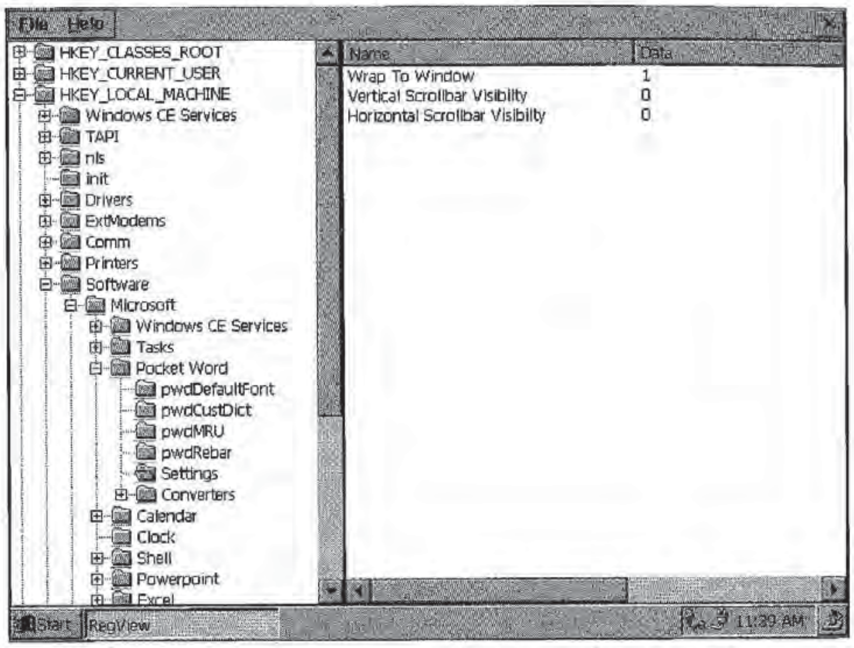
{ROOT_KEY}\Software\{Company Name}\{Company Product}

In this template, the ROOT_KEY is either HKEY_LOCAL_MACHINE for machine-specific data such as what optional components of an application may be installed on the machine or HKEY_CURRENT_USER for user-specific information, such as the list of the user's last-opened files. Under the Software key, the company's name that wrote the application is used followed by the name of the specific application. For example, Microsoft saves the configuration information for Pocket Word under the key

HKEY_LOCAL_MACHINE\Software\Microsoft\Pocket Word

While this hierarchy is great for segregating registry values from different applications from one another, it's best not to create too deep a set of keys. Because of the way the registry is designed, it takes less memory to store a value than it does a key. Because of this, you should design you registry storage so that it uses fewer keys and more values. To optimize even further, it's more efficient to store more information in one value than to have the same information stored across a number of values.

The window in Figure 7-6 shows the hierarchy of keys used to store data for Pocket Word. The left pane shows the hierarchy of keys down to the Settings key under the Pocket Word key. In the Settings key, three values are stored: Wrap To Window, Vertical Scrollbar Visibility, and Horizontal Scrollbar Visibility. In this case, these values are DWORDs, but they could have been strings or other data types.



Name	Data
Wrap To Window	1
Vertical Scrollbar Visibilty	0
Horizontal Scrollbar Visibilty	0

Figure 7-6. *You can see the hierarchy of the registry by looking at the values stored by Pocket Word.*

The Registry API

Now let's turn toward the Windows CE registry API. In general, the registry API provides all the functions necessary to read and write data in the registry as well as enumerate the keys and data store within. Windows CE doesn't support the security features of the registry that are supported under Windows NT.

Opening and creating keys

A registry key is opened with a call to this function:

```
LONG RegOpenKeyEx (HKEY hKey, LPCWSTR lpszSubKey, DWORD ulOptions,
                   REGSAM samDesired, PHKEY phkResult);
```

The first parameter is the key that contains the second parameter, the subkey. This first key must be either one of the root key constants or a previously opened key. The subkey to open is specified as a text string that contains the key to open. This subkey string can contain multiple levels of subkeys as long as each subkey is separated by a backslash. For example, to open the subkey HKEY_LOCAL_MACHINE\Software\Microsoft\Pocket Word, an application could either call *RegOpenKeyEx* with HKEY_LOCAL_MACHINE as the key and Software\Microsoft\Pocket Word as the subkey or it could open the Software\Microsoft key and then make a call with that opened handle to *RegOpenKeyEx* specifying the subkey Pocket Word. Key and value names aren't case specific.

Windows CE ignores the *ulOptions* and *samDesired* parameters. To remain compatible with future versions of the operating system that might use security features, these parameters should be set to 0 for *ulOptions* and NULL for *samDesired*. The *phkResult* parameter should point to a variable that will receive the handle to the opened key. The function, if successful, returns a value of ERROR_SUCCESS and an error code if it fails.

Another method for opening a key is

```
LONG RegCreateKeyEx (HKEY hKey, LPCWSTR lpszSubKey, DWORD Reserved,
                     LPWSTR lpszClass, DWORD dwOptions,
                     REGSAM samDesired,
                     LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                     PHKEY phkResult, LPDWORD lpdwDisposition);
```

The difference between *RegCreateKeyEx* and *RegOpenKeyEx*, aside from the extra parameters, is that *RegCreateKeyEx* creates the key if it didn't exist before the call. The first two parameters, the key handle and the subkey name, are the same as in *RegOpenKeyEx*. The *Reserved* parameter should be set to 0. The *lpClass* parameter points to a string that contains the class name of the key if it's to be created. This parameter can be set to NULL if no class name needs to be specified. The *dwOptions* and *samDesired* and *lpSecurityAttributes* parameters should be set to 0, NULL, and NULL respectively. The *phkResult* parameter points to the variable that receives the handle to the opened or newly created key. The *lpdwDisposition* parameter points to a variable that's set to indicate whether the key was opened or created by the call.

Reading registry values

You can query registry values by first opening the key containing the values of interest and calling this function:

```
LONG RegQueryValueEx (HKEY hKey, LPCWSTR lpszValueName,
                      LPDWORD lpReserved, LPDWORD lpType,
                      LPBYTE lpData, LPDWORD lpcbData);
```

The *hKey* parameter is the handle of the key opened by *RegCreateKeyEx* or *RegOpenKeyEx*. The *lpszValueName* is the name of the value that's being queried. The *lpType* parameter is a pointer to a variable that receives the variable type. This variable is filled with The *lpData* parameter points to the buffer to receive the data, while the *lpcbData* parameter points to a variable that receives the size of the data. If *RegQueryValueEx* is called with the *lpData* parameter equal to NULL, Windows returns the size of the data but doesn't return the data itself. This allows applications to first query the size and type of the data before actually receiving it.

Writing registry values

You set a registry value by calling

```
LONG RegSetValueEx (HKEY hKey, LPCWSTR lpszValueName, DWORD Reserved,
                    DWORD dwType, const BYTE *lpData, DWORD cbData);
```

The parameters here are fairly obvious: the handle to the open key followed by the name of the value to set. The function also requires that you pass the type of data, the data itself, and the size of the data. The data type parameter is simply a labeling aid for the application that eventually reads the data. Data in the registry is stored in a binary format and returned in that same format. Specifying a different type has no effect on how the data is stored in the registry or how it's returned to the application. However, given the availability of third-party registry editors, you should make every effort to specify the appropriate data type in the registry.

The data types can be one of the following:

- *REG_SZ* A zero-terminated Unicode string
- *REG_EXPAND_SZ* A zero-terminated Unicode string with embedded environment variables
- *REG_MULTI_SZ* A series of zero-terminated Unicode strings terminated by two zero characters
- *REG_DWORD* A 4-byte binary value
- *REG_BINARY* Free-form binary data
- *REG_DWORD_BIG_ENDIAN* A DWORD value stored in big-endian format
- *REG_DWORD_LITTLE_ENDIAN* Equivalent to REG_DWORD
- *REG_LINK*
- *REG_NONE*
- *REG_RESOURCE_LIST*

Deleting keys and values

You delete a registry key by calling

```
LONG RegDeleteKey (HKEY hKey, LPCWSTR lpszSubKey);
```

The parameters are the handle to the open key and the name of the subkey you plan to delete. For the deletion to be successful, the key must not be currently open. You can delete a value by calling

```
LONG RegDeleteValue (HKEY hKey, LPCWSTR lpszValueName);
```

A wealth of information can be gleaned about a key by calling this function:

```
LONG RegQueryInfoKey (HKEY hKey, LPWSTR lpszClass, LPDWORD lpcchClass,
                      LPDWORD lpReserved, LPDWORD lpcSubKeys,
                      LPDWORD lpcchMaxSubKeyLen,
                      LPDWORD lpcchMaxClassLen,
                      LPDWORD lpcValues, LPDWORD lpcchMaxValueNameLen,
                      LPDWORD lpcbMaxValueData,
                      LPDWORD lpcbSecurityDescriptor,
                      PFILETIME lpftLastWriteTime);
```

The only input parameter to this function is the handle to a key. The function returns the class of the key, if any, as well as the maximum lengths of the subkeys and values under the key. The last two parameters, the security attributes and the last write time, are unsupported under Windows CE and should be set to NULL.

Closing keys

You close a registry key by calling

```
LONG RegCloseKey (HKEY hKey);
```

When a registry key is closed, Windows CE flushes any unwritten key data to the registry before returning from the call.

Enumerating registry keys

In some instances, you'll find it helpful to be able to query a key to see what subkeys and values it contains. You accomplish this with two different functions: one to query the subkeys, another to query the values. The first function

```
LONG RegEnumKeyEx (HKEY hKey, DWORD dwIndex, LPWSTR lpszName,
                   LPDWORD lpcchName, LPDWORD lpReserved,
                   LPWSTR lpszClass,
                   LPDWORD lpcchClass, PFILETIME lpftLastWriteTime);
```

enumerates the subkeys of a registry key through repeated calls. The parameters to pass the function are the handle of the opened key and an index value. To enumerate the first subkey, the *dwIndex* parameter should be 0. For each subsequent call to *RegEnumKeyEx*, *dwIndex* should be incremented to get the next subkey. When there are no more subkeys to be enumerated, *RegEnumKeyEx* returns ERROR_NO_MORE_ITEMS.

For each call to *RegEnumKeyEx*, the function returns the name of the subkey, and its classname. The last write time parameter isn't supported under Windows CE.

Values within a key can be enumerated with a call to this function:

```
LONG RegEnumValue (HKEY hKey, DWORD dwIndex, LPWSTR lpszValueName,
                   LPDWORD lpcchValueName, LPDWORD lpReserved,
                   LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData);
```

Like *RegEnumKey*, this function is called repeatedly, passing index values to enumerate the different values stored under the key. When the function returns ERROR_NO_MORE_ITEMS, there are no more values under the key. *RegEnumValue* returns the name of the values, the data stored in the value, as well as its data type and the size of the data.

The RegView Example Program

The following program is a registry viewer application. It allows a user to navigate the trees in the registry and examine the contents of the data stored. Unlike RegEdit, which is provided by Windows NT and Windows 98, RegView doesn't let you edit the registry. However, such an extension wouldn't be difficult to make. Figure 7-7 contains the code for the RegView program.

RegView.rc

```
//======================================================================
// Resource file
//
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include "windows.h"
#include "regview.h"                    // Program-specific stuff

//----------------------------------------------------------------------
// Icons and bitmaps
//
ID_ICON ICON   "regview.ico"            // Program icon
ID_BMPS BITMAP "TVBmps.bmp"

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

Figure 7-7. *The RegView program.* *(continued)*

Figure 7-7. *continued*

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

RegView.h

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

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

```
struct decodeNotify {                               // Structure associates
    UINT Code;                                      // control IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, LPNMHDR);      // notify handler.
};

//----------------------------------------------------------------------
// Generic defines used by application
#define  ID_ICON              1                     // App icon resource ID
#define  ID_BMPS              2                     // Bitmap resource ID

#define  IDC_CMDBAR           10                    // Command band ID
#define  ID_MENU              11                    // Main menu resource ID
#define  ID_TREEV             12                    // Tree view control ID
#define  ID_LISTV             13                    // List view control ID

// Menu item IDs
#define  IDM_EXIT             101                   // File menu
#define  IDM_ABOUT            150                   // Help menu

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

INT EnumChildren (HWND, HTREEITEM, HKEY, LPTSTR);
DWORD CountChildren (HKEY, LPTSTR, LPTSTR);
INT EnumValues (HWND, HKEY, LPTSTR);
INT DisplayValue (HWND, INT, LPTSTR, PBYTE, DWORD, DWORD);
INT GetTree (HWND, HTREEITEM, HKEY *, TCHAR *, INT);
HTREEITEM InsertTV (HWND, HTREEITEM, TCHAR *, LPARAM, DWORD);
INT InsertLV (HWND, INT, LPTSTR, LPTSTR);
HWND CreateLV (HWND, RECT *);
HWND CreateTV (HWND, RECT *);

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

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

(continued)

Figure 7-7. *continued*

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

// Notify functions
LPARAM DoMainNotifyListV (HWND, WORD, HWND, LPNMHDR);
LPARAM DoMainNotifyTreeV (HWND, WORD, HWND, LPNMHDR);

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

RegView.c

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

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

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

INT nDivPct = 40;                    // Divider setting between windows

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_SIZE, DoSizeMain,
    WM_COMMAND, DoCommandMain,
    WM_NOTIFY, DoNotifyMain,
    WM_DESTROY, DoDestroyMain,
};
// Command message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDM_EXIT, DoMainCommandExit,
    IDM_ABOUT, DoMainCommandAbout,
};
// Notification message dispatch for MainWindowProc
const struct decodeNotify MainNotifyItems[] = {
    ID_LISTV, DoMainNotifyListV,
    ID_TREEV, DoMainNotifyTreeV,
};
//======================================================================
//
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    HWND hwndMain;
    MSG msg;
    int rc = 0;

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

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

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

    // Register application main window class.
    wc.style = 0;                             // Window style
    wc.lpfnWndProc = MainWndProc;             // Callback function
    wc.cbClsExtra = 0;                        // Extra class data
    wc.cbWndExtra = 0;                        // Extra window data
    wc.hInstance = hInstance;                 // Owner handle
```

(continued)

Figure 7-7. *continued*

```
    wc.hIcon = NULL,                              // Application icon
    wc.hCursor = NULL;                            // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                      // Menu name
    wc.lpszClassName = szAppName;                 // Window class name

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

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

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

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

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
```

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

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

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

(continued)

Figure 7-7. *continued*

```
    // Create the tree view control in the left pane.
    SetRect (&rect, 0, nHeight, lpcs->cx/3, lpcs->cy - nHeight);
    hwndChild = CreateTV (hWnd, &rect);

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

    // Create the list view control in right pane.
    SetRect (&rect, lpcs->cx/3, nHeight, (lpcs->cx*2)/3,
             lpcs->cy - nHeight);
    hwndChild = CreateLV (hWnd, &rect);

    // Destroy frame if window not created.
    if (!IsWindow (hwndChild)) {
        DestroyWindow (hWnd);
        return 0;
    }
    // Insert the base keys.
    InsertTV (hWnd, NULL, TEXT ("HKEY_CLASSES_ROOT"),
                        (LPARAM)HKEY_CLASSES_ROOT, 1);
    InsertTV (hWnd, NULL, TEXT ("HKEY_CURRENT_USER"),
              (LPARAM)HKEY_CURRENT_USER, 1);
    InsertTV (hWnd, NULL, TEXT ("HKEY_LOCAL_MACHINE"),
              (LPARAM)HKEY_LOCAL_MACHINE, 1);
    InsertTV (hWnd, NULL, TEXT ("HKEY_USERS"),
              (LPARAM)HKEY_USERS, 1);
    return 0;
}
//----------------------------------------------------------------------
// DoSizeMain - Process WM_SIZE message for window.
//
LRESULT DoSizeMain (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam){
    HWND hwndLV, hwndTV;
    RECT rect;
    INT nDivPos;

    hwndTV = GetDlgItem (hWnd, ID_TREEV);
    hwndLV = GetDlgItem (hWnd, ID_LISTV);

    // Adjust the size of the client rect to take into account
    // the command bar height.
    GetClientRect (hWnd, &rect);
    rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));
    nDivPos = ((rect.right - rect.left) * nDivPct)/100;

    SetWindowPos (hwndTV, NULL, rect.left, rect.top,
                  nDivPos, rect.bottom - rect.top,
                  SWP_NOZORDER);

    SetWindowPos (hwndLV, NULL, nDivPos, rect.top,
                  (rect.right - rect.left) - nDivPos,
                  rect.bottom - rect.top, SWP_NOZORDER);
    return 0;
}
//----------------------------------------------------------------------
// DoCommandMain - Process WM_COMMAND message for window.
//
LRESULT DoCommandMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    WORD idItem, wNotifyCode;
    HWND hwndCtl;
    INT  i;

    // Parse the parameters.
    idItem = (WORD) LOWORD (wParam);
    wNotifyCode = (WORD) HIWORD (wParam);
    hwndCtl = (HWND) lParam;

    // Call routine to handle control message.
    for (i = 0; i < dim(MainCommandItems); i++) {
        if (idItem == MainCommandItems[i].Code)
            return (*MainCommandItems[i].Fxn)(hWnd, idItem, hwndCtl,
                                              wNotifyCode);
    }
    return 0;
}
//----------------------------------------------------------------------
// DoNotifyMain - Process WM_NOTIFY message for window.
//
LRESULT DoNotifyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    UINT    idItem;
    HWND    hCtl;
    LPNMHDR pHdr;
    INT     i;

    // Parse the parameters.
    idItem = wParam;
    pHdr = (LPNMHDR) lParam;
    hCtl = pHdr->hwndFrom;
```

(continued)

Figure 7-7. *continued*

```
    // Call routine to handle control message.
    for (i = 0; i < dim(MainNotifyItems); i++) {
        if (idItem == MainNotifyItems[i].Code)
            return (*MainNotifyItems[i].Fxn)(hWnd, idItem, hCtl, pHdr);
    }
    return 0;
}
//----------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
//======================================================================
// Command handler routines
//----------------------------------------------------------------------
// DoMainCommandExit - Process Program Exit command.
//
LPARAM DoMainCommandExit (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {

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

    // Use DialogBox to create modal dialog box.
    DialogBox (hInst, TEXT ("aboutbox"), hWnd, AboutDlgProc);
    return 0;
}
//======================================================================
// Notify handler routines
//----------------------------------------------------------------------
// DoMainNotifyListV - Process notify message for list view.
//
LPARAM DoMainNotifyListV (HWND hWnd, WORD idItem, HWND hwndCtl,
                          LPNMHDR pnmh) {
    return 0;
}
```

```
//----------------------------------------------------------------------
// DoMainNotifyTreeV - Process notify message for list view.
//
LPARAM DoMainNotifyTreeV (HWND hWnd, WORD idItem, HWND hwndCtl,
                          LPNMHDR pnmh) {

    LPNM_TREEVIEW pNotifyTV;
    TCHAR szKey[256];
    HKEY hRoot;
    HTREEITEM hChild, hNext;
    INT i;

    pNotifyTV = (LPNM_TREEVIEW) pnmh;

    switch (pnmh->code) {
        case TVN_ITEMEXPANDED:
            if (pNotifyTV->action == TVE_COLLAPSE) {
                // Delete the children so that on next open, they will
                // be reenumerated.
                hChild = TreeView_GetChild (hwndCtl,
                                     pNotifyTV->itemNew.hItem);
                while (hChild) {
                    hNext = TreeView_GetNextItem (hwndCtl, hChild,
                                                  TVGN_NEXT);
                    TreeView_DeleteItem (hwndCtl, hChild);
                    hChild = hNext;
                }
            }
            break;

        case TVN_SELCHANGED:
            GetTree (hWnd, pNotifyTV->itemNew.hItem, &hRoot,
                     szKey, dim(szKey));
            EnumValues (hWnd, hRoot, szKey);
            break;

        case TVN_ITEMEXPANDING:
            if (pNotifyTV->action == TVE_EXPAND) {
                GetTree (hWnd, pNotifyTV->itemNew.hItem, &hRoot,
                         szKey, dim(szKey));
                i = EnumChildren (hWnd, pNotifyTV->itemNew.hItem,
                                  hRoot, szKey);
            }
            break;
    }
    return 0;
}
```

(continued)

Figure 7-7. *continued*

```
//----------------------------------------------------------------------
// CreateLV - Create list view control.
//
HWND CreateLV (HWND hWnd, RECT *prect) {
    HWND hwndLV;
    LVCOLUMN lvc;

    //
    // Create report window.  Size it so that it fits under
    // the command bar and fills the remaining client area.
    //
    hwndLV = CreateWindowEx (0, WC_LISTVIEW, TEXT (""),
                         WS_VISIBLE | WS_CHILD | WS_VSCROLL |
                         WS_BORDER | LVS_REPORT,
                         prect->left, prect->top,
                         prect->right - prect->left,
                         prect->bottom - prect->top,
                         hWnd, (HMENU)ID_LISTV,
                         hInst, NULL);
    // Add columns.
    if (hwndLV) {
        lvc.mask = LVCF_TEXT | LVCF_WIDTH | LVCF_FMT | LVCF_SUBITEM |
                   LVCF_ORDER;
        lvc.fmt = LVCFMT_LEFT;
        lvc.cx = 120;
        lvc.pszText = TEXT ("Name");
        lvc.iOrder = 0;
        lvc.iSubItem = 0;
        SendMessage (hwndLV, LVM_INSERTCOLUMN, 0, (LPARAM)&lvc);

        lvc.mask |= LVCF_SUBITEM;
        lvc.pszText = TEXT ("Data");
        lvc.cx = 250;
        lvc.iOrder = 1;
        lvc.iSubItem = 1;
        SendMessage (hwndLV, LVM_INSERTCOLUMN, 1, (LPARAM)&lvc);
    }
    return hwndLV;
}
//----------------------------------------------------------------------
// InitTreeView - Initialize tree view control.
//
HWND CreateTV (HWND hWnd, RECT *prect) {
    HBITMAP hBmp;
    HIMAGELIST himl;
    HWND hwndTV;
    //
    // Create tree view.  Size it so that it fits under
    // the command bar and fills the left part of the client area.
    //
    hwndTV = CreateWindowEx (0, WC_TREEVIEW,
                         TEXT (""), WS_VISIBLE | WS_CHILD | WS_VSCROLL |
                         WS_BORDER | TVS_HASLINES | TVS_HASBUTTONS |
                         TVS_LINESATROOT, prect->left, prect->top,
                         prect->right, prect->bottom,
                         hWnd, (HMENU)ID_TREEV, hInst, NULL);

    // Destroy frame if window not created.
    if (!IsWindow (hwndTV))
        return 0;

    // Create image list control for tree view icons.
    himl = ImageList_Create (16, 16, ILC_COLOR, 2, 0);
    // Load first two images from one bitmap.
    hBmp = LoadBitmap (hInst, MAKEINTRESOURCE (ID_BMPS));
    ImageList_Add (himl, hBmp, NULL);
    DeleteObject (hBmp);

    TreeView_SetImageList(hwndTV, himl, TVSIL_NORMAL);
    return hwndTV;
}
//----------------------------------------------------------------------
// InsertLV - Add an item to the list view control.
//
INT InsertLV (HWND hWnd, INT nItem, LPTSTR pszName, LPTSTR pszData) {

    HWND hwndLV = GetDlgItem (hWnd, ID_LISTV);
    LVITEM lvi;
    INT rc;

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

    lvi.mask = LVIF_TEXT;
    lvi.iItem = nItem;
    lvi.iSubItem = 1;
    lvi.pszText = pszData;
```

(continued)

Figure 7-7. *continued*

```
    rc = SendMessage (hwndLV, LVM_SETITEM, 0, (LPARAM)&lvi);
    return 0;
}
//----------------------------------------------------------------------
// InsertTV - Insert item into tree view control.
//
HTREEITEM InsertTV (HWND hWnd, HTREEITEM hParent, TCHAR *pszName,
                    LPARAM lParam, DWORD nChildren) {
    TV_INSERTSTRUCT tvis;

    HWND hwndTV = GetDlgItem (hWnd, ID_TREEV);
    // Initialize the insertstruct.
    memset (&tvis, 0, sizeof (tvis));
    tvis.hParent = hParent;
    tvis.hInsertAfter = TVI_LAST;
    tvis.item.mask = TVIF_TEXT | TVIF_PARAM | TVIF_CHILDREN |
                     TVIF_IMAGE;
    tvis.item.pszText = pszName;
    tvis.item.cchTextMax = lstrlen (pszName);
    tvis.item.iImage = 1;
    tvis.item.iSelectedImage = 1;
    tvis.item.lParam = lParam;
    if (nChildren)
        tvis.item.cChildren = 1;
    else
        tvis.item.cChildren = 0;

    return TreeView_InsertItem (hwndTV, &tvis);
}
//----------------------------------------------------------------------
// GetTree - Compute the full path of the tree view item.
//
INT GetTree (HWND hWnd, HTREEITEM hItem, HKEY *pRoot, TCHAR *pszKey,
             INT nMax) {
    TV_ITEM tvi;
    TCHAR szName[256];
    HTREEITEM hParent;
    HWND hwndTV = GetDlgItem (hWnd, ID_TREEV);

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

    hParent = TreeView_GetParent (hwndTV, hItem);
    if (hParent) {
        // Get the parent of the parent of the...
        GetTree (hWnd, hParent, pRoot, pszKey, nMax);
        // Get the name of the item.
        tvi.mask = TVIF_TEXT;
        tvi.hItem = hItem;
        tvi.pszText = szName;
        tvi.cchTextMax = dim(szName);
        TreeView_GetItem (hwndTV, &tvi);

        lstrcat (pszKey, TEXT ("\\"));
        lstrcat (pszKey, szName);
    } else {
        *pszKey = TEXT ('\0');
        szName[0] = TEXT ('\0');
        // Get the name of the item.
        tvi.mask = TVIF_TEXT | TVIF_PARAM;
        tvi.hItem = hItem;
        tvi.pszText = szName;
        tvi.cchTextMax = dim(szName);
        if (TreeView_GetItem (hwndTV, &tvi))
            *pRoot = (HTREEITEM)tvi.lParam;
        else {
            INT rc = GetLastError();
        }
    }
    return 0;
}
//----------------------------------------------------------------------
// DisplayValue - Display the data depending on the type.
//
INT DisplayValue (HWND hWnd, INT nCnt, LPTSTR pszName, PBYTE pbData,
                  DWORD dwDSize, DWORD dwType) {
    TCHAR szData[512];
    INT i, len;

    switch (dwType) {
    case REG_MULTI_SZ:
    case REG_EXPAND_SZ:
    case REG_SZ:
        lstrcpy (szData, (LPTSTR)pbData);
        break;

    case REG_DWORD:
        wsprintf (szData, TEXT ("%X"), *(int *)pbData);
        break;
```

(continued)

Figure 7-7. *continued*

```
        case REG_BINARY:
            szData[0] = TEXT ('\0');
            for (i = 0; i < (int)dwDSize; i++) {
                len = lstrlen (szData);
                wsprintf (&szData[len], TEXT ("%02X "), pbData[i]);
                if (len > dim(szData) - 6)
                    break;
            }
            break;
        default:
            wsprintf (szData, TEXT ("Unknown type: %x"), dwType);
        }
        InsertLV (hWnd, nCnt, pszName, szData);
        return 0;
}
//----------------------------------------------------------------------
// EnumValues - Enumerate each of the values of a key.
//
INT EnumValues (HWND hWnd, HKEY hRoot, LPTSTR pszKey) {
    INT nCnt = 0, rc;
    DWORD dwNSize, dwDSize, dwType;
    TCHAR szName[MAX_PATH];
    BYTE bData[1024];
    HKEY hKey;

    if (lstrlen (pszKey)) {
        if (RegOpenKeyEx (hRoot, pszKey, 0, 0, &hKey) != ERROR_SUCCESS)
            return 0;
    } else
        hKey = hRoot;

    // Clean out list view.
    ListView_DeleteAllItems (GetDlgItem (hWnd, ID_LISTV));

    // Enumerate the values in the list view control.
    nCnt = 0;
    dwNSize = dim(szName);
    dwDSize = dim(bData);
    rc = RegEnumValue (hKey, nCnt, szName, &dwNSize,
                       NULL, &dwType, bData, &dwDSize);

    while (rc == ERROR_SUCCESS) {
        // Display the value in the list view control.
        DisplayValue (hWnd, nCnt, szName, bData, dwDSize, dwType);
        dwNSize = dim(szName);
        dwDSize = dim(bData);
        nCnt++;
        rc = RegEnumValue (hKey, nCnt, szName, &dwNSize,
                           NULL, &dwType, bData, &dwDSize);
    }
    if (hKey != hRoot)
        RegCloseKey (hKey);
    return 1;
}
//----------------------------------------------------------------------
// CountChildren - Count the number of children of a key.
//
DWORD CountChildren (HKEY hRoot, LPTSTR pszKeyPath, LPTSTR pszKey) {
    TCHAR *pEnd;
    DWORD dwCnt;
    HKEY hKey;

    pEnd = pszKeyPath + lstrlen (pszKeyPath);
    lstrcpy (pEnd, TEXT ("\\"));
    lstrcat (pEnd, pszKey);

    if (RegOpenKeyEx(hRoot, pszKeyPath, 0, 0, &hKey) == ERROR_SUCCESS){
        RegQueryInfoKey (hKey, NULL, NULL, 0, &dwCnt, NULL, NULL, NULL,
                         NULL, NULL, NULL, NULL);
        RegCloseKey (hKey);
    }
    *pEnd = TEXT ('\0');
    return dwCnt;
}
//----------------------------------------------------------------------
// EnumChildren - Enumerate the child keys of a key.
//
INT EnumChildren (HWND hWnd, HTREEITEM hParent, HKEY hRoot,
                  LPTSTR pszKey) {
    INT i = 0, rc;
    DWORD dwNSize;
    DWORD dwCSize;
    TCHAR szName[MAX_PATH];
    TCHAR szClass[256];
    FILETIME ft;
    DWORD nChild;
    HKEY hKey;
    TVITEM tvi;
```

(continued)

Figure 7-7. *continued*

```
    // All keys but root need to be opened.
    if (lstrlen (pszKey)) {
        if (RegOpenKeyEx (hRoot, pszKey, 0, 0, &hKey) != ERROR_SUCCESS) {
            rc = GetLastError();
            return 0;
        }
    } else
        hKey = hRoot;

    dwNSize = dim(szName);
    dwCSize = dim(szClass);
    rc = RegEnumKeyEx (hKey, i, szName, &dwNSize, NULL,
                       szClass, &dwCSize, &ft);
    while (rc == ERROR_SUCCESS) {

        nChild = CountChildren (hRoot, pszKey, szName);
        // Add key to tree view.
        InsertTV (hWnd, hParent, szName, 0, nChild);
        dwNSize = dim(szName);
        rc = RegEnumKeyEx (hKey, ++i, szName, &dwNSize,
                           NULL, NULL, 0, &ft);
    }
    // If this wasn't the a root key, close it.
    if (hKey != hRoot)
        RegCloseKey (hKey);

    // If no children, remove expand button.
    if (i == 0) {
        tvi.hItem = hParent;
        tvi.mask = TVIF_CHILDREN;
        tvi.cChildren = 0;
        TreeView_SetItem (GetDlgItem (hWnd, ID_TREEV), &tvi);
    }
    return i;
}
//======================================================================
// About Dialog procedure
//
BOOL CALLBACK AboutDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                            LPARAM lParam) {
    switch (wMsg) {
        case WM_COMMAND:
            switch (LOWORD (wParam)) {
                case IDOK:
                case IDCANCEL:
                        EndDialog (hWnd, 0);
                        return TRUE;
                }
            break;
        }
        return FALSE;
    }
```

The workhorses of this program are the enumeration functions that query what keys and values are under each key. As a key is opened in the tree view control, the control sends a WM_NOTIFY message. In response, RegView enumerates the items below that key and fills the tree view with the child keys and the list view control with the values.

CONCLUSION

We have covered a huge amount of ground in this chapter. The file system, while radically different under the covers, presents a standard Win32 interface to the programmer and a familiar directory structure to the user. The database API is unique to Windows CE and provides a valuable function for the information-centric devices that Windows CE supports. The registry structure and interface are quite familiar to Windows programmers and should present no surprises.

The last two chapters have covered memory and the file system. Now it's time to look at the third part of the kernel triumvirate, processes and threads. As with the other parts of Windows CE, the API will be familiar if perhaps a bit smaller. However, the underlying architecture of Windows CE does make itself known.

Chapter 8

Processes and Threads

Like Windows NT, Windows CE is a fully multitasking and multithreaded operating system. What does that mean? In this chapter I'll present a few definitions and then some explanations to answer that question.

A *process* is a single instance of an application. If two copies of Microsoft Pocket Word are running, two unique processes are running. Every process has its own, protected, 32-MB address space as described in Chapter 6. Windows CE enforces a limit of 32 separate processes that can run at any time.

Each process has at least one *thread*. A thread executes code within a process. A process can have multiple threads running "at the same time." I put the phrase *at the same time* in quotes because, in fact, only one thread executes at any instant in time. The operating system simulates the concurrent execution of threads by rapidly switching between the threads, alternatively stopping one thread and switching to another.

PROCESSES

Windows CE treats processes differently than does Windows 98 or Windows NT. First and foremost, Windows CE has the aforementioned system limit of 32 processes being run at any one time. When the system starts, at least four processes are created: NK.EXE, which provides the kernel services; FILESYS.EXE, which provides file system services; GWES.EXE, which provides the GUI support; and DEVICE.EXE, which loads and maintains the device drivers for the system. On most systems, other processes are also started, such as the shell, EXPLORER.EXE, and, if the system is connected to a PC, REPLLOG.EXE and RAPISRV.EXE, which service the link between the PC and the Windows CE system. This leaves room for about 24 processes that the user or other applications that are running can start. While this sounds like a harsh limit, most systems don't need that many processes. A typical H/PC that's being used heavily might have 15 processes running at any one time.

Windows CE diverges from its desktop counterparts in other ways. Compared with processes under Windows 98 or Windows NT, Windows CE processes contain much less state information. Since Windows CE supports neither drives nor the concept of a current directory, the individual processes don't need to store that information. Windows CE also doesn't maintain a set of environment variables, so processes don't need to keep an environment block. Windows CE doesn't support handle inheritance, so there's no need to tell a process to enable handle inheritance. Because of all this, the parameter-heavy *CreateProcess* function is passed mainly NULLs and zeros, with just a few parameters actually used by Windows CE.

Many of the process and thread-related functions are simply not supported by Windows CE because the system doesn't support certain features supported by Windows 98 or Windows NT. Since Windows CE doesn't support an environment, all the Win32 functions dealing with the environment don't exist in Windows CE. While Windows CE supports threads, it doesn't support fibers, a lightweight version of a thread supported by Windows NT. So, the fiber API doesn't exist under Windows CE. Some functions aren't supported because there's an easy way to work around the lack of the function. For example, *GetCommandLine* doesn't exist in Windows CE, so an application needs to save a pointer to the command line passed to WinMain if it needs to access it later. Finally, *ExitProcess* doesn't exist under Windows CE. But, as you might expect, there's a workaround that allows a process to close.

Enough of what Windows CE doesn't do; let's look at what you can do with Windows CE.

Creating a Process

The function for creating another process is

```
BOOL CreateProcess (LPCTSTR lpApplicationName,
                    LPTSTR lpCommandLine,
                    LPSECURITY_ATTRIBUTES lpProcessAttributes,
                    LPSECURITY_ATTRIBUTES lpThreadAttributes,
                    BOOL bInheritHandles, DWORD dwCreationFlags,
                    LPVOID lpEnvironment,
                    LPCTSTR lpCurrentDirectory,
                    LPSTARTUPINFO lpStartupInfo,
                    LPPROCESS_INFORMATION lpProcessInformation);
```

While the list of parameters looks daunting, most of the parameters must be set to NULL or 0 because Windows CE doesn't support security or current directories, nor does it handle inheritance. This results in a function prototype that looks more like this:

```
BOOL CreateProcess (LPCTSTR lpApplicationName,
                    LPTSTR lpCommandLine,  NULL, NULL, FALSE,
                    DWORD dwCreationFlags, NULL, NULL, NULL,
                    LPPROCESS_INFORMATION lpProcessInformation);
```

The parameters that remain start with a pointer to the name of the application to launch. Windows CE looks for the application in the following directories, in this order:

1. The path, if any, specified in the *lpApplicationName*.
2. For Windows CE 2.1 or later, the path specified in the *SystemPath* value in [HKEY_LOCAL_MACHINE]\Loader. For earlier versions, the root of any external storage devices, such as PC Cards.
3. The windows directory, (\Windows).
4. The root directory in the object store, (\).

This action is different from Windows NT, where *CreateProcess* searches for the executable only if *lpApplicationName* is set to NULL and the executable name is passed through the *lpCcommnadLine* parameter. In the case of Windows CE, the application name must be passed in the *lpApplicaitonName* parameter because Windows CE doesn't support the technique of passing a NULL in *lpApplicationName* with the application name as the first token in the *lpCommandLine* parameter.

The *lpCommandLine* parameter specifies the command line that will be passed to the new process. The only difference between Windows CE and Windows NT in this parameter is that under Windows CE the command line is always passed as a Unicode string. And, as I mentioned previously, you can't pass the name of the executable as the first token in *lpCommandLine*.

The *dwCreationFlags* parameter specifies the initial state of the process after it has been loaded. Windows CE limits the allowable flags to the following:

- *0* Creates a standard process.
- *CREATE_SUSPENDED* Creates the process, then suspends the primary thread.
- *DEBUG_PROCESS* The process being created is treated as a process being debugged by the caller. The calling process receives debug information from the process being launched.
- *DEBUG_ONLY_THIS_PROCESS* When combined with DEBUG_PROCESS, debugs a process but doesn't debug any child processes that are launched by the process being debugged.

- *CREATE_NEW_CONSOLE* Forces a new console to be created. This is supported only in Windows CE 2.1 and later.

The only other parameter of *CreateProcess* used by Windows CE is *lpProcessInformation*. This parameter can be set to NULL, or it can point to a PROCESS_INFORMATION structure that's filled by *CreateProcess* with information about the new process. The PROCESS_INFORMATION structure is defined this way:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION;
```

The first two fields in this structure are filled with the handles of the new process and the handle of the primary thread of the new process. These handles are useful for monitoring the newly created process, but with them comes some responsibility. When the system copies the handles for use in the PROCESS_INFORMATION structure, it increments the use count for the handles. This means that, if you don't have any use for the handles, the calling process must close them. Ideally, they should be closed immediately following a successful call to *CreateProcess*. I'll describe some good uses for these handles later in this chapter in the section, "Synchronization."

The other two fields in the PROCESS_INFORMATION structure are filled with the process ID and primary thread ID of the new process. These ID values aren't handles but simply unique identifiers that can be passed to Windows functions to identify the target of the function. Be careful when using these IDs. If the new process terminates and another new one is created, the system can reuse the old ID values. You must take measures to assure that ID values for other processes are still identifying the process you're interested in before using them. For example, you can, by using synchronization objects, be notified when a process terminates. When the process terminated, you would then know not to use the ID values for that process.

Using the create process is simple, as you can see in the following code fragment:

```
TCHAR szFileName[MAX_PATH];
TCHAR szCmdLine[64];
DWORD dwCreationFlags;
PROCESS_INFORMATION pi;
INT rc;

lstrcpy (szFileName, TEXT ("calc"));
lstrcpy (szCmdLine, TEXT (""));
dwCreationFlags = 0;
```

```
rc = CreateProcess (szFileName, szCmdLine, NULL, NULL, FALSE,
                    dwCreationFlags, NULL, NULL, NULL, &pi);
if (rc) {
    CloseHandle (pi.hThread);
    CloseHandle (pi.hProcess);
}
```

This code launches the standard Calculator applet found on Handheld PCs and Palm-size PCs. Since the file name doesn't specify a path, *CreateProcess* will, using the standard Windows CE search path, find calc.exe in the \Windows directory. Because I didn't pass a command line to *Calc*, I could have simply passed a NULL value in the *lpCmdLine* parameter. But I passed a null string in *szCmdLine* to differentiate the *lpCmdLine* parameter from the many other parameters in *CreateProcess* that aren't used. I used the same technique for *dwCreationFlags*. If the call to *CreateProcess* is successful, it returns a nonzero value. The code above checks for this, and if the call was successful, closes the process and thread handles returned in the PROCESS_INFORMATION structure. Remember that this must be done by all Win32 applications to prevent memory leaks.

Terminating a Process

A process can terminate itself by simply returning from the *WinMain* procedure. For console applications, a simple return from *main* suffices. Windows CE doesn't support the *ExitProcess* function found in Windows 98 and Windows NT. Instead, you can have the primary thread of the process call *ExitThread*. Under Windows CE, if the primary thread terminates, the process is terminated as well, regardless of what other threads are currently active in the process. The exit code of the process will be the exit code provided by *ExitThread*. You can determine the exit code of a process by calling

```
BOOL GetExitCodeProcess (HANDLE hProcess, LPDWORD lpExitCode);
```

The parameters are the handle to the process and a pointer to a DWORD that receives the exit code that was returned by the terminating process. If the process is still running, the return code is the constant STILL_ACTIVE.

You can terminate another process. But while it's possible to do that, you shouldn't be in the business of closing other processes. The user might not be expecting that process to be closed without his or her consent. If you need to terminate a process (or close a process, which is the same thing but much nicer a word), the following methods can be used.

If the process to be closed is one that you created, you can use some sort of interprocess communication to tell the process to terminate itself. This is the most advisable method because you've designed the target process to be closed by another party. Another method of closing a process is to send the main window of the process a WM_CLOSE message. This is especially effective on the Palm-size PC, where applications are designed to respond to WM_CLOSE messages by quietly saving their state and closing. Finally, if all else fails and you absolutely must close another process, you can use *TerminateProcess*.

TerminateProcess is prototyped as

```
BOOL TerminateProcess (HANDLE hProcess, DWORD uExitCode);
```

The two parameters are the handle of the process to terminate and the exit code the terminating process will return.

Other Processes

Of course, to terminate another process, you've got to know the handle to that process. You might want to know the handle for a process for other reasons, as well. For example, you might want to know *when* the process terminates. Windows CE supports two additional functions that come in handy here (both of which are seldom discussed). The first function is *OpenProcess*, which returns the handle of an already running process. *OpenProcess* is prototyped as

```
HANDLE OpenProcess (DWORD dwDesiredAccess, BOOL bInheritHandle,
                    DWORD dwProcessId);
```

Under Windows CE, the first parameter isn't used and should be set to 0. The *bInheritHandle* parameter must be set to FALSE because Windows CE doesn't support handle inheritance. The final parameter is the process ID value of the process you want to open.

The other function useful in this circumstance is

```
DWORD GetWindowThreadProcessId (HWND hWnd, LPDWORD lpdwProcessId);
```

This function takes a handle to a window and returns the process ID for the process that created the window. So, using these two functions, you can trace a window back to the process that created it.

Two other functions allow you to directly read from and write to the memory space of another process. These functions are

```
BOOL ReadProcessMemory (HANDLE hProcess, LPCVOID lpBaseAddress,
                        LPVOID lpBuffer, DWORD nSize,
                        LPDWORD lpNumberOfBytesRead);
```

and

```
BOOL WriteProcessMemory (HANDLE hProcess, LPVOID lpBaseAddress,
                         LPVOID lpBuffer, DWORD nSize,
                         LPDWORD lpNumberOfBytesWritten);
```

The parameters for these functions are fairly self-explanatory. The first parameter is the handle of the remote process. The second parameter is the base address in the other process's address space of the area to be read or written. The third and fourth parameters specify the name and the size of the local buffer in which the data is to be read from or written to. Finally, the last parameter specifies the bytes actually read or written. Both functions require that the entire area being read to or written from must be accessible. Typically, you use these functions for debugging but there's no requirement that this be their only use.

THREADS

A thread is, fundamentally, a unit of execution. That is, it has a stack and a processor context, which is a set of values in the CPU internal registers. When a thread is suspended, the registers are pushed onto the thread's stack, the active stack is changed to the next thread to be run, that thread's CPU state is pulled off its stack, and the new thread starts executing instructions.

Threads under Windows CE are similar to threads under Windows NT or Windows 98. Each process has a primary thread. Using the functions that I describe below, a process can create any number of additional threads within the process. The only limit to the number of threads in a Windows CE process is the memory and process address space available for the thread's stack.

Threads within a process share the address space of the process. Memory allocated by one thread is accessible to all threads in the process. Threads share the same access rights for handles whether they be file handles, memory objects handles, or handles to synchronization objects.

Before Windows CE 2.1, the size of all thread stacks was set at around 58 KB. Starting with Windows CE 2.1, the stack size of all threads created within a process is set by the linker. (The linker switch for setting the stack size in Microsoft Visual C++ is */stack*.) Secondary threads under Windows CE 2.1 are created with the same stack size as the primary thread.

The System Scheduler

Windows CE schedules threads in a preemptive manner. Threads run for a *quantum* or time slice, which is usually 25 milliseconds on H/PCs and Palm-size PCs. (OEMs developing custom hardware can specify a different quantum.) After that time, if the thread hasn't already relinquished its time slice and if the thread isn't a time-critical thread, it's suspended and another thread is scheduled to run. Windows CE chooses which thread to run based on a priority scheme. Threads of a higher priority are scheduled before threads of lower priority.

The rules for how Windows CE allocates time among the threads are quite different from Windows NT and from Windows 98. Unlike Windows NT, Windows CE processes don't have a *priority class*. Under Windows NT, a process is created with a priority class. Threads derive their priority based on the priority class of their parent processes. A process with a higher-priority class has threads that run at a higher priority than threads in a lower-priority class process. Threads within a process can then refine their priority within that process by setting their relative thread priority.

Because Windows CE has no priority classes, all processes are treated as peers. Individual threads can have different priorities, but the process that the thread runs within doesn't influence those priorities. Also, unlike Windows NT, the foreground thread in Windows CE doesn't get a boost in priority.

In Windows CE, a thread can have one of eight priority levels. Those priorities are listed below:

- *THREAD_PRIORITY_TIME_CRITICAL* Indicates 3 points above normal priority. Threads of this priority aren't preempted.
- *THREAD_PRIORITY_HIGHEST* Indicates 2 points above normal priority.
- *THREAD_PRIORITY_ABOVE_NORMAL* Indicates 1 point above normal priority.
- *THREAD_PRIORITY_NORMAL* Indicates normal priority. All threads are created with this priority.
- *THREAD_PRIORITY_BELOW_NORMAL* Indicates 1 point below normal priority.
- *THREAD_PRIORITY_LOWEST* Indicates 2 points below normal priority.
- *THREAD_PRIORITY_ABOVE_IDLE* Indicates 3 points below normal priority.
- *THREAD_PRIORITY_IDLE* Indicates 4 points below normal priority.

All higher-priority threads run before lower-priority threads. This means that before a thread set to run at particular priority can be scheduled, all threads that have a higher priority must be *blocked*. A blocked thread is one that's waiting on some system resource or synchronization object before it can continue. Threads of equal priority are scheduled in a round-robin fashion. Once a thread has voluntarily given up its time slice, is blocked, or has completed its time slice, all other threads of the same priority are allowed to run before the original thread is allowed to continue. If a thread of higher priority is unblocked and a thread of lower priority is currently running, the lower-priority thread is immediately suspended and the higher-priority thread is scheduled. Lower-priority threads can never preempt a higher-priority thread.

There are two exceptions to the rules I just stated. If a thread has a priority of THREAD_PRIORITY_TIME_CRITICAL, it's never preempted, even by another THREAD_PRIORITY_TIME_CRITICAL thread. As you can see, a THREAD_PRIORITY_TIME_CRITICAL thread can and will starve everyone else in the system unless written carefully. This priority is reserved by convention for interrupt service threads in device drivers, which are written so that each thread quickly performs its task and releases its time slice.

The other exception to the scheduling rules happens if a low-priority thread owns a resource that a higher-priority thread is waiting on. In this case, the low-priority thread is temporarily given the higher-priority thread's priority in a scheme known as *priority inversion*, so that it can quickly accomplish its task and free the needed resource.

While it might seem that lower-priority threads never get a chance to run in this scheme, it works out that threads are almost always blocked, waiting on something to free up before they can be scheduled. Threads are always created at THREAD_PRIORITY_NORMAL, so, unless they proactively change their priority level, a thread is usually at an equal priority to most of the other threads in the system. Even at the normal priority level, threads are almost always blocked. For example, an application's primary thread is typically blocked waiting on messages. Other threads should be designed to block on one of the many synchronization objects available to a Windows CE application.

Never Do This!

What's not supported by the arrangement I just described, or by any other thread-based scheme, is code like the following:

```
while (bFlag == FALSE) {
    // Do nothing, and spin
}
// Now do something.
```

This kind of code isn't just bad manners, since it wastes CPU power, it's a death sentence to a battery-powered Windows CE device. To understand why this is important, I need to digress into a quick lesson on Windows CE power management.

Windows CE is designed so that when all threads are blocked, which happens over 90 percent of the time, it calls down to the OEM Abstraction Layer (the equivalent to the BIOS on an MS-DOS machine) to enter a low-power waiting state. Typically, this low-power state means that the CPU is halted; that is, it simply stops executing instructions. Because the CPU isn't executing any instructions, no power-consuming reads and writes of memory are performed by the CPU. At this point, the only power necessary for the system is to maintain the contents of the RAM and light the display. This low-power mode can reduce power consumption by up to 99 percent of what is required when a thread is running in a well-designed system.

Doing a quick back-of-the-envelope calculation, say a Palm-size PC is designed to run for 15 hours on a couple of AAA batteries. Given that the system turns itself off after a few minutes of non-use, this 15 hours translates into a month or two of battery life in the device for the user. (I'm basing this calculation on the assumption that the system indeed spends 90 percent or more of its time in its low-power idle state.) Say a poorly written application thread spins on a variable instead of blocking. While this application is running, the system will never enter its low-power state. So, instead of

900 minutes of battery time (15 hours × 60 minutes/hour), the system spends 100 percent of its time at full power, resulting in a battery life of slightly over 98 minutes, or right at 1.5 hours. So, as you can see, it's good to have the system in its low-power state.

Fortunately, since Windows applications usually spend their time blocked in a call to *GetMessage,* the system power management works by default. However, if you plan on using multiple threads in your application, you must use synchronization objects to block threads while they're waiting. First, let's look at how to create a thread, and then I'll dive into the synchronization tools available to Windows CE programs.

Creating a Thread

You create a thread by calling this function:

```
HANDLE CreateThread (LPSECURITY_ATTRIBUTES lpThreadAttributes,
                     DWORD dwStackSize,
                     LPTHREAD_START_ROUTINE lpStartAddress,
                     LPVOID lpParameter, DWORD dwCreationFlags,
                     LPDWORD lpThreadId);
```

As with *CreateProcess*, Windows CE doesn't support a number of the parameters in *CreateThread*, and so they are set to NULL or 0 as appropriate. For *CreateThread*, the *lpThreadAttributes*, and *dwStackSize* parameters aren't supported. The parameter *lpThreadAttributes* must be set to NULL and *dwStackSize* is ignored by the system and should be set to 0. The third parameter, *lpStartAddress*, must point to the start of the thread routine. The *lpParameter* parameter in *CreateThread* is an application-defined value that's passed to the thread function as its one and only parameter. The *dwCreationFlags* parameter can be set to either 0 or CREATE_SUSPENDED. If CREATE_SUSPENDED is passed, the thread is created in a suspended state and must be resumed with a call to *ResumeThread*. The final parameter is a pointer to a DWORD that receives the newly created thread's ID value.

The thread routine should be prototyped this way:

```
DWORD WINAPI ThreadFunc (LPVOID lpArg);
```

The only parameter is the *lpParameter* value, passed unaltered from the call to *CreateThread*. The parameter can be an integer or a pointer. Make sure, however, that you don't pass a pointer to a stack-based structure that will disappear when the routine that called *CreateThread* returns.

If *CreateThread* is successful, it creates the thread and returns the handle to the newly created thread. As with *CreateProcess*, the handle returned should be closed when you no longer need the handle. Following is a short code fragment that contains a call to start a thread and the thread routine.

```
//----------------------------------------------------------------------
//
//
HANDLE hThread1;
DWORD dwThread1ID = 0;
INT nParameter = 5;

hThread1 = CreateThread (NULL, 0, Thread2, nParameter, 0,
                         &dwThread1ID);
CloseHandle (hThread1);

//----------------------------------------------------------------------
// Second thread routine
//
DWORD WINAPI Thread2 (PVOID pArg) {

    INT nParam = (INT) pArg;

    //
    // Do something here.
    // .
    // .
    // .
    return 0x15;
}
```

In this code, the second thread is started with a call to *CreateThread*. The *nParameter* value is passed to the second thread as the single parameter to the thread routine. The second thread executes until it terminates, in this case simply by returning from the routine.

A thread can also terminate itself by calling this function:

```
VOID ExitThread (DWORD dwExitCode);
```

The only parameter is the exit code that's set for the thread. That thread exit code can be queried by another thread using this function:

```
BOOL GetExitCodeThread (HANDLE hThread, LPDWORD lpExitCode);
```

The function takes the handle to the thread (not the thread ID) and returns the exit code of the thread. If the thread is still running, the exit code is STILL_ACTIVE, a constant defined as 0x0103. The exit code is set by a thread using *ExitThread* or the value returned by the thread procedure. In the preceding code, the thread sets its exit code to 0x15 when it returns.

All threads within a process are terminated when the process terminates. As I said earlier, a process is terminated when its primary thread terminates.

Setting and querying thread priority

Threads are always created at a priority level of THREAD_PRIORITY_NORMAL. The thread priority can be changed either by the thread itself or by another thread calling this function:

```
BOOL SetThreadPriority (HANDLE hThread, int nPriority);
```

The two parameters are the thread handle and the new priority level. The level passed can be one of the constants described previously, ranging from THREAD_PRIORITY_IDLE up to THREAD_PRIORITY_TIME_CRITICAL. You must be extremely careful when you're changing a thread's priority. Remember that threads of a lower priority almost never preempt threads of higher priority. So, a simple bumping up of a thread one notch above normal can harm the responsiveness of the rest of the system unless that thread is carefully written.

To query the priority level of a thread, call this function:

```
int GetThreadPriority (HANDLE hThread);
```

This function returns the priority level of the thread. You shouldn't use the hard-coded priority levels. Instead, use constants, such as THREAD_PRIORITY_NORMAL, defined by the system. This ensures that any change to the priority scheme in future versions of Windows CE doesn't affect your program.

Suspending and resuming a thread

You can suspend a thread at any time by calling this function:

```
DWORD SuspendThread (HANDLE hThread);
```

The only parameter is the handle to the thread to suspend. The value returned is the *suspend count* for the thread. Windows maintains a suspend count for each thread. Any thread with a suspend count greater than 0 is suspended. Since *SuspendThread* increments the suspend count, multiple calls to *SuspendThread* must be matched with an equal number of calls to *ResumeThread* before a thread is actually scheduled to run. *ResumeCount* is prototyped as

```
DWORD ResumeThread (HANDLE hThread);
```

Here again, the parameter is the handle to the thread and the return value is the previous suspend count. So, if *ResumeThread* returns 1, the thread is no longer suspended.

At times, a thread simply wants to kill some time. Since I've already explained why simply spinning in a *while* loop is a very bad thing to do, you need another way to kill time. The best way to do this is to use this function:

```
void Sleep (DWORD dwMilliseconds);
```

Sleep suspends the thread for at least the number of milliseconds specified in the *dwMilliseconds* parameter. Since the quantum, or time slice, on a Windows CE system is usually 25 milliseconds, specifying very small numbers of milliseconds results in sleeps of at least 25 milliseconds. This strategy is entirely valid, and sometimes it's equally valid to pass a 0 to *Sleep*. When a thread passes a 0 to *Sleep*, it gives up its time slice but is rescheduled immediately according to the scheduling rules I described previously.

Thread Local Storage

Thread local storage is a mechanism that allows a routine to maintain separate instances of data for each thread calling the routine. This capability might not seem like much, but it has some very handy uses. Take the following thread routine:

```
INT g_nGlobal;              // System global variable

int ThreadProc (pStartData) {
    INT nValue1;
    INT nValue2;

    while (unblocked) {
        //
        // Do some work.
        //
    }
    // We're done now, terminate the thread by returning.
    return 0;
}
```

For this example, imagine that multiple threads are created to execute the same routine, *ThreadProc*. Each thread has its own copy of *nValue1* and *nValue2* because these are stack-based variables and each thread has its own stack. All threads, though, share the same static variable, *g_nGlobal*.

Now, imagine that the *ThreadProc* routine calls another routine, *WorkerBee*. As in

```
int g_nGlobal;              // System global variable

int ThreadProc (pStartData) {
    int nValue1;
    int nValue2;
    while (unblocked) {
        WorkerBee();        // Let someone else do the work.
    }
    // We're done now, terminate the thread by returning.
    return 0;
}
```

(continued)

```
int WorkerBee (void) {
    int nLocal1;
    static int nLocal2;
    //
    // Do work here.
    //
    return nLocal1;
}
```

Now *WorkerBee* doesn't have access to any persistent memory that's local to a thread. *nLocal1* is persistent only for the life of a single call to *WorkerBee*. *nLocal2* is persistent across calls to *WorkerBee* but is static and therefore shared among all threads calling *WorkerBee*. One solution would be to have *ThreadProc* pass a pointer to a stack-based variable to *WorkerBee*. This strategy works, but only if you have control over the routines calling *WorkerBee*. What if you're writing a DLL and you need to have a routine in the DLL maintain a different state for each thread calling the routine? You can't define static variables in the DLL because they would be shared across the different threads. You can't define local variables because they aren't persistent across calls to your routine. The answer is to use thread local storage.

Thread local storage allows a process to have its own cache of values that are guaranteed to be unique for each thread in a process. This cache of values is small because an array must be created for every thread created in the process, but it's large enough, if used intelligently. To be specific, the system constant, TLS_MINIMUM_AVAILABLE, is defined to be the number of slots in the TLS array that's available for each process. For Windows CE, like Windows NT, this value is defined as 64. So, each process can have 64 4-byte values that are unique for each thread in that process. For the best results, of course, you must manage those 64 slots well.

To reserve one of the TLS slots, a process calls

```
DWORD TlsAlloc (void);
```

TlsAlloc looks through the array to find a free slot in the TLS array, marks it as *in use*, and then returns an index value to the newly assigned slot. If no slots are available, the function returns -1. It's important to understand that the individual threads don't call *TlsAlloc*. Instead, the process or DLL calls it before creating the threads that will use the TLS slot.

Once a slot has been assigned, each thread can access its unique data in the slot by calling this function:

```
BOOL TlsSetValue (DWORD dwTlsIndex, LPVOID lpTlsValue);
```

and

```
LPVOID TlsGetValue (DWORD dwTlsIndex);
```

For both of these functions, the TLS index value returned by *TlsAlloc* specifies the slot that contains the data. Both *TlsGetValue* and *TlsSetValue* type the data as a PVOID, but the value can be used for any purpose. The advantage of thinking of the TLS value as a pointer is that a thread can allocate a block of memory on the heap, and then keep the pointer to that data in the TLS value. This allows each thread to maintain a block of thread-unique data of almost any size.

One other matter is important to thread local storage. When *TlsAlloc* reserves a slot, it zeros the value in that slot for all currently running threads. All new threads are created with their TLS array initialized to 0 as well. This means that a thread can safely assume that the value in its slot will be initialized to 0. This is helpful for determining whether a thread needs to allocate a memory block the first time the routine is called.

When a process no longer needs the TLS slot, it should call this function:

```
BOOL TlsFree (DWORD dwTlsIndex);
```

The function is passed the index value of the slot to be freed. The function returns TRUE if successful. This function frees only the TLS slot. If threads have allocated storage in the heap and stored pointers to those blocks in their TLS slots, that storage isn't freed by this function. Threads are responsible for freeing their own memory blocks.

SYNCHRONIZATION

With multiple threads running around the system, you need to coordinate the activities. Fortunately, Windows CE supports almost the entire extensive set of standard Win32 synchronization objects. The concept of synchronization objects is fairly simple. A thread *waits* on a synchronization object. When the object is signaled, the waiting thread is unblocked and is scheduled (according to the rules governing the thread's priority) to run.

Windows CE doesn't support some of the synchronization primitives supported by Windows NT. These unsupported elements include semaphores, file change notifications, and waitable timers. Support for semaphores is planed for Windows CE in the near future. The lack of waitable timer support can easily be worked around using the more flexible Notification API, unique to Windows CE.

One aspect of Windows CE unique to it is that the different synchronization objects don't share the same namespace. This means that if you have an event named Bob, you can also have a *mutex* named Bob. (I'll talk about mutexes later in this chapter.) This naming convention is different from Windows NT's rule, where all kernel objects (of which synchronization objects are a part) share the same namespace. While having the same names in Windows CE is possible, it's not advisable. Not only does the practice make your code incompatible with Windows NT, there's no telling whether a redesign of the internals of Windows CE might just enforce this restriction in the future.

Events

The first synchronization primitive I'll describe is the *event object*. An event object is a synchronization object that can be in a *signaled* or *nonsignaled* state. Events are useful to a thread to let it be known that, well, an event has occurred. Event objects can either be created to automatically reset from a signaled state to a nonsignaled state or require a manual reset to return the object to its nonsignaled state. Starting with Windows CE 2.0, events can be named and therefore shared across different processes allowing interprocess synchronization.

An event is created by means of this function:

```
HANDLE CreateEvent (LPSECURITY_ATTRIBUTES lpEventAttributes,
                    BOOL bManualReset, BOOL bInitialState,
                    LPTSTR lpName);
```

As with all calls in Windows CE, the security attributes parameter, *lpEventAttributes*, should be set to NULL. The second parameter indicates whether the event being created requires a manual reset or will automatically reset to a nonsignaled state immediately after being signaled. Setting *bManualReset* to TRUE creates an event that must be manually reset. The *bInitialState* parameter specifies whether the event object is initially created in the signaled or nonsignaled state. Finally, the *lpName* parameter points to an optional string that names the event. Events that are named can be shared across processes. If two processes create event objects of the same name, the processes actually share the same object. This allows one process to signal the other process using event objects. If you don't want a named event, the *lpname* parameter can be set to NULL.

To share an event object across processes, each process must individually create the event object. You can't simply create the event in one process and send the handle of that event to another process. To determine whether a call to *CreateEvent* created a new event object or opened an already created object, you can call *GetLastError* immediately following the call to *CreateEvent*. If *GetLastError* returns ERROR_ALREADY_EXISTS, the call opened an existing event.

Once you have an event object, you'll need to be able to signal the event. You accomplish this using either of the following two functions:

```
BOOL SetEvent (HANDLE hEvent);
```

or

```
BOOL PulseEvent (HANDLE hEvent);
```

The difference between these two functions is that *SetEvent* doesn't automatically reset the event object to a nonsignaled state. For autoreset events, *SetEvent* is all you need because the event is automatically reset once a thread unblocks on the event. For manual reset events, you must manually reset the event with this function:

```
BOOL ResetEvent (HANDLE hEvent);
```

These event functions sound like they overlap, so let's review. An event object can be created to reset itself or require a manual reset. If it can reset itself, a call to *SetEvent* signals the event object. The event is then automatically reset to the nonsignaled state when *one* thread is unblocked after waiting on that event. An event that resets itself doesn't need *PulseEvent* or *ResetEvent*. If, however, the event object was created requiring a manual reset, the need for *ResetEvent* is obvious.

PulseEvent signals the event and then resets the event, which allows *all* threads waiting on that event to be unblocked. So, the difference between *PulseEvent* on a manually resetting event and *SetEvent* on an automatic resetting event is that using *SetEvent* on an automatic resetting event frees only one thread to run even if many threads are waiting on that event. *PulseEvent* frees all threads waiting on that event.

You destroy event objects by calling *CloseHandle*. If the event object is named, Windows maintains a use count on the object so one call to *CloseHandle* must be made for every call to *CreateEvent*.

Waiting...

It's all well and good to have event objects; the question is how to use them. Threads wait on events, as well as on the soon to be described mutex, using one of the following functions: *WaitForSingleObject*, *WaitForMultipleObjects*, *MsgWaitForMultipleObjects*, or *MsgWaitForMultipleObjectsEx*. Under Windows CE, the *WaitForMultiple* functions are limited in that they can't wait for all objects of a set of objects to be signaled. These functions support waiting for *one* object in a set of objects being signaled. Whatever the limitations of waiting, I can't emphasize enough that waiting is good. While a thread is blocked with one of these functions, the thread enters an extremely efficient state that takes very little CPU processing power and battery power.

Another point to remember is that the thread responsible for handling a message loop in your application (usually the application's primary thread) shouldn't be blocked by *WaitForSingleObject* or *WaitForMultipleObjects* because the thread can't be retrieving and dispatching messages in the message loop if it's blocked waiting on an object. The function *MsgWaitForMultipleObjects* gives you a way around this problem, but in a multithreaded environment, it's usually easier to let the primary thread handle the message loop and secondary threads handle the shared resources that require blocking on events.

Waiting on a single object

A thread can wait on a synchronization object with the function:

```
DWORD WaitForSingleObject (HANDLE hHandle, DWORD dwMilliseconds);
```

The function takes two parameters: the handle to the object being waited on and a timeout value. If you don't want the wait to time out, you can pass the value INFINITE in the *dwMilliseconds* parameter. The function returns a value that indicates why the function returned. Calling *WaitForSingleObject* blocks the thread until the event is signaled, the synchronization object is abandoned, or the timeout value is reached.

WaitForSingleObject returns one of the following values:

- *WAIT_OBJECT_0* The specified object was signaled.
- *WAIT_TIMEOUT* The timeout interval elapsed, and the object's state remains nonsignaled.
- *WAIT_ABANDONED* The thread that owned a mutex object being waited on ended without freeing the object.
- *WAIT_FAILED* The handle of the synchronization object was invalid.

You must check the return code from *WaitForSingleObject* to determine whether the event was signaled or simply that the time out had expired. (The WAIT_ABANDONED return value will be relevant when I talk about mutexes soon.)

Waiting on processes and threads

I've talked about waiting on events, but you can also wait on handles to processes and threads. These handles are signaled when their processes or threads terminate. This allows a process to monitor another process (or thread) and perform some action when the process terminates. One common use for this feature is for one process to launch another, and then by blocking on the handle to the newly created process, wait until that process terminates.

The rather irritating routine below is a thread that demonstrates this technique by launching an application, blocking until that application closes, and then relaunching the application:

```
DWORD WINAPI KeepRunning (PVOID pArg) {
    PROCESS_INFORMATION pi;
    TCHAR szFileName[MAX_PATH];
    INT rc = 0;

    // Copy the filename.
    Lstrcpy (szFileName, (LPTSTR)pArg);
    while (1) {
        // Launch the application.
        rc = CreateProcess (szFileName, NULL, NULL, NULL, FALSE,
                            0, NULL, NULL, NULL, &pi);
        // If the application didn't start, terminate thread.
        if (!rc)
            return -1;
        // Close the new process's primary thread handle.
        CloseHandle (pi.hThread);
        // Wait for user to close the application.
        rc = WaitForSingleObject (pi.hProcess, INFINITE);

        // Close the old process handle.
        CloseHandle (pi.hProcess);

        // Make sure we returned from the wait correctly.
        if (rc != WAIT_OBJECT_0)
            return -2;
    }
    return 0;  //This should never get executed.
}
```

This code simply launches the application using *CreateProcess* and waits on the process handle returned in the PROCESS_INFORMATION structure. Notice that the thread closes the child process's primary thread handle and, after the wait, the handle to the child process itself.

Waiting on multiple objects

A thread can also wait on a number of events. The wait can end when any one of the events is signaled. The function that enables a thread to wait on multiple objects is this one:

```
DWORD WaitForMultipleObjects (DWORD nCount, CONST HANDLE *lpHandles,
                              BOOL bWaitAll, DWORD dwMilliseconds);
```

The first two parameters are a count of the number of events or mutexes to wait on and a pointer to an array of handles to these events. The *bWaitAll* parameter must be set to FALSE to indicate the function should return if any of the events are signaled. The final parameter is a timeout value, in milliseconds. As with *WaitForSingleObject*, passing INFINITE in the timeout parameter disables the time out. Windows CE doesn't support the use of *WaitForMultipleObjects* to enable waiting for all events in the array to be signaled before returning.

Like *WaitForSingleObject*, *WaitForMultipleObjects* returns a code that indicates why the function returned. If the function returned due to a synchronization object being signaled, the return value will be WAIT_OBJECT_0 plus an index into the handle array that was passed in the *lpHandles* parameter. For example, if the first handle in the array unblocked the thread, the return code would be WAIT_OBJECT_0; if the second handle was the cause, the return code would be WAIT_OBJECT_0 + 1. The other return codes used by *WaitForSingleObject*—WAIT_TIMEOUT, WAIT_ABANDONED, and WAIT_FAILED—are also returned by *WaitForMultipleObjects* for the same reasons.

Waiting while dealing with messages

The Win32 API provides other functions that allow you to wait on a set of objects as well as messages: these are *MsgWaitForMultipleObjects* and *MsgWaitForMultipleObjectsEx*. Under Windows CE, these functions act identically, so I'll describe only *MsgWaitForMultipleObjects*. This function essentially combines the wait function, *MsgWaitForMultipleObjects*, with an additional check into the message queue so that the function returns if any of the selected categories of messages are received during the wait. The prototype for this function is the following:

```
DWORD MsgWaitForMultipleObjectsEx (DWORD nCount, LPHANDLE pHandles,
                                   BOOL fWaitAll, DWORD dwMilliseconds,
                                   DWORD dwWakeMasks);
```

This function has a number of limitations under Windows CE. As with *WaitForMultipleObjects*, *MsgWaitForMultipleObjectsEx* can't wait for all objects to be signaled. Nor are all the *dwWakeMask* flags supported by Windows CE. Windows CE supports the following flags in *dwWakeMask*. Each flag indicates a category of messages that, when received in the message queue of the thread, causes the function to return.

- *QS_ALLINPUT* Any message has been received.
- *QS_INPUT* An input message has been received.
- *QS_KEY* A key up, key down, or syskey up or down message has been received.
- *QS_MOUSE* A mouse move or mouse click message has been received.
- *QS_MOUSEBUTTON* A mouse click message has been received.
- *QS_MOUSEMOVE* A mouse move message has been received.
- *QS_PAINT* A WM_PAINT message has been received.
- *QS_POSTMESSAGE* A posted message, other than those in this list, has been received.
- *QS_SENDMESSAGE* A sent message, other than those in this list, has been received.
- *QS_TIMER* A WM_TIMER message has been received.

The function is used inside the message loop, so that an action or actions can take place in response to the signaling of a synchronization object while your program is still processing messages.

The return value is WAIT_OBJECT_0 up to WAIT_OBJECT_0 + *nCount* - 1 for the objects in the handle array. If a message causes the function to return, the return value is WAIT_OBJECT_0 + *nCount*. An example of how this function might be used follows. In this code, the handle array has only one entry, *hSyncHandle*.

```
fContinue = TRUE;
while (fContinue) {
    rc = MsgWaitForMultipleObjects (1, &hSyncHandle, FALSE,
                                    INFINITE, QS_ALLINPUT);
    if (rc == WAIT_OBJECT_0) {
        //
        // Do work as a result of sync object.
        //
    } else if (rc == WAIT_OBJECT_0 + 1) {
        // It's a message, process it.
        PeekMessage (&msg, hWnd, 0, 0, PM_REMOVE);
        if (msg.message == WM_QUIT)
            fContinue = FALSE;
        else {
            TranslateMessage (&msg);
            DispatchMessage (&msg);
        }
    }
}
```

Mutexes

Earlier I described the event object. That object resides in either a signaled or nonsignaled state. Another synchronization object is the *mutex*. A mutex is a synchronization object that's signaled when it's not owned by a thread and nonsignaled when it *is* owned. Mutexes are extremely useful for coordinating exclusive access to a resource such as a block of memory across multiple threads.

A thread gains ownership by waiting on that mutex with one of the wait functions. When no other threads own the mutex, the thread waiting on the mutex is unblocked, and implicitly gains ownership of the mutex. After the thread has completed the work that requires ownership of the mutex, the thread must explicitly release the mutex with a call to *ReleaseMutex*.

To create a mutex, call this function:

```
HANDLE CreateMutex (LPSECURITY_ATTRIBUTES lpMutexAttributes,
                    BOOL bInitialOwner, LPCTSTR lpName);
```

The *lpMutexAttributes* parameter should be set to NULL. The *bInitialOwner* parameter lets you specify that the calling thread should immediately own the mutex being created. Finally, the *lpName* parameter lets you specify a name for the object so that it can be shared across other processes. When calling *CreateMutex* with a name specified in the *lpName* parameter, Windows CE checks whether a mutex with the same name has already been created. If so, a handle to the previously created mutex is returned. To determine whether the mutex already exists, call *GetLastError*. It returns ERROR_ALREADY_EXISTS if the mutex has been previously created.

Gaining immediate ownership of a mutex using the *bInitialOwner* parameter works only if the mutex is being created. Ownership isn't granted if you're opening a previously created mutex. If you need ownership of a mutex, be sure to call *GetLast-Error* to determine whether the mutex had been previously committed. If so, call *WaitForSingleObject* to gain ownership of the mutex.

You release the mutex with this function:

```
BOOL ReleaseMutex (HANDLE hMutex);
```

The only parameter is the handle to the mutex.

If a thread owns a mutex and calls one of the wait functions to wait on that same mutex, the wait call immediately returns because the thread already owns the mutex. Since mutexes retain an ownership count for the number of times the wait functions are called, a call to *ReleaseMutex* must be made for each nested call to the wait function.

Critical Sections

Using *critical sections* is another method of thread synchronization. Critical sections are good for protecting sections of code from being executed by two different threads at the same time. Critical sections work by having a thread call *EnterCriticalSection* to indicate that it has entered a critical section of code. If another thread calls *EnterCriticalSection* referencing the same critical section object, it's blocked until the first thread makes a call to *LeaveCriticalSection*. Critical sections can protect more than one linear section of code. All that's required is that all sections of code that need to be protected use the same critical section object. The one limitation of critical sections is that they can be used to coordinate threads only within a process.

To use a critical section, you first create a critical section handle with this function:

```
void InitializeCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
```

The only parameter is a pointer to a CRITICAL_SECTION structure that you define somewhere in your application. Be sure not to allocate this structure on the stack of a function that will be deallocated as soon the function returns. You should also not move or copy the critical section structure. Since the other critical section functions require a pointer to this structure, you'll need to allocate it within the scope of all functions using the critical section. While the CRITICAL_SECTION structure is defined in WINBASE.H, an application doesn't need to manipulate any of the fields in that structure. So, for all practical purposes, think of a pointer to a CRITICAL_SECTION structure as a handle, instead of as a pointer to a structure of a known format.

When a thread needs to enter a protected section of code, it should call this function:

```
void EnterCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
```

The function takes as its only parameter a pointer to the critical section structure initialized with *InitializeCriticalSection*. If the critical section is already owned by another thread, this function blocks the new thread and doesn't return until the other thread releases the critical section. If the thread calling *EnterCriticalSection* already owns the critical section, then a use count is incremented and the function returns immediately.

When a thread leaves a critical section, it should call this function:

```
void LeaveCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
```

As with all the critical section functions, the only parameter is the pointer to the critical section structure. Since critical sections track a use count, one call to *LeaveCriticalSection* must be made for each call to *EnterCriticalSection* by the thread that owns the section.

Finally, when you're finished with the critical section, you should call

```
void DeleteCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
```

This cleans up any system resources used to manage the critical section.

Interlocked Variable Access

Here's one more low-level method for synchronizing threads—using the functions for interlocked access to variables. While programmers with multithread experience already know this, I need to warn you that Murphy's Law[1] seems to come into its own when you're using multiple threads in a program. One of the sometimes overlooked issues in a preemptive multitasking system is that a thread can be preempted in the middle of incrementing or checking a variable. For example, a simple code fragment such as

```
if (!i++) {
    // Do something because i was zero.
}
```

can cause a great deal of trouble. To understand why, let's look into how that statement might be compiled. The assembly code for that if statement might look something like this:

```
load     reg1, [addr of i]            ;Read variable
add      reg2, reg1, 1                ;reg2 = reg1 + 1
store    reg2, [addr of i]            ;Save incremented var
bne      reg1, zero, skipblk          ;Branch reg1 != zero
```

There's no reason that the thread executing this section of code couldn't be preempted by another thread after the load instruction and before the store instruction. If this

1. Murphy's Law: Anything that can go wrong will go wrong. Murphy's first corollary: When something goes wrong, it happens at the worst possible moment.

happened, two threads could enter the block of code when that isn't the way the code is supposed to work. Of course, I've already described a number of methods (such as critical sections and the like) that you can use to prevent such incidents from occurring. But for something like this, a critical section is overkill. What you need is something lighter.

Windows CE supports three of the *interlocked* functions from the Win32 API; *InterlockedIncriment*, *InterlockedDecriment*, and *InterlockedExchange*. Each of these allows a thread to increment, decrement, and exchange a variable without your having to worry about the thread being preempted in the middle of the operation. The functions are prototyped here:

```
LONG InterlockedIncrement(LPLONG lpAddend);

LONG InterlockedDecrement(LPLONG lpAddend);

LONG InterlockedExchange(LPLONG Target, LONG Value);
```

For the interlocked increment and decrement, the one parameter is a pointer to the variable to increment or decrement. The returned value is the new value of the variable after it has been incremented or decremented. The *InterlockedExchange* function takes a pointer to the target variable and the new value for the variable. It returns the previous value of the variable. Rewriting the previous code fragment so that it's thread safe produces this code:

```
if (!InterlockedIncrement(&i)) {
    // Do something because i was zero.
}
```

INTERPROCESS COMMUNICATION

There are many cases where two Windows CE processes need to communicate. The walls between processes that protect processes from one another prevent casual exchanging of data. The memory space of one process isn't exposed to another process. Handles to files or other objects can't be passed from one process to another. Windows CE doesn't support the *DuplicateHandle* function available under Windows NT, which allows one process to open a handle used by another process. Nor, as I mentioned before, does Windows CE support handle inheritance. Some of the other more common methods of interprocess communication, such as named pipes, are also not supported under Windows CE. However, you can choose from plenty of ways to enable two or more processes to exchange data.

Finding Other Processes

Before you can communicate with another process, you have to determine whether it's running on the system. Strategies for finding whether another process is running depend mainly on whether you have control of the other process. If the process to be found is a third-party application in which you have no control over the design of the other process, the best method might be to use *FindWindow* to locate the other process's main window. *FindWindow* can search either by window class or by window title. You can also enumerate the top-level windows in the system using *EnumWindows*. You can also use the ToolHelp debugging functions to enumerate the processes running, but this works only when the ToolHelp DLL is loaded on the system and unfortunately, it generally isn't included, by default, on most systems.

If you're writing both processes, however, it's much easier to enumerate them. In this case, the best methods include using the tools you'll later use in one process to communicate with the other process, such as named mutexes, events, or memory-mapped objects. When you create one of these objects, you can determine whether you're the first to create the object or you're simply opening another object by calling *GetLastError* after another call created the object. And the simplest method might be the best; call *FindWindow* and send a WM_USER message to the main window of the other process.

WM_COPYDATA

Once you've found your target process, the talking can begin. If you're staying at the window level, a simple method of communicating is to send a WM_COPYDATA message. WM_COPYDATA is unique in that it's designed to send blocks of data from one process to another. You can't use a standard, user-defined message to pass pointers to data from one process to another because a pointer isn't valid across processes. WM_COPYDATA gets around this problem by having the system translate the pointer to a block of data from one process's address space to another's. The recipient process is required to copy the data immediately into its own memory space, but this message does provide a quick and dirty method of sending blocks of data from one process to another.

Named memory-mapped objects

The problem with WM_COPYDATA is that it can be used only to copy fixed blocks of data at a specific time. Using a named memory-mapped object, two processes can allocate a shared block of memory that's equally accessible to both processes at the same time. You should use named memory-mapped objects so that the system can maintain a proper use count on the object. This procedure prevents one process from freeing the block when it terminates while the other process is still using the block.

Of course, this level of interaction comes with a price. You need some synchronization between the processes when they're reading and writing data in the shared memory block. The use of named mutexes and named events allows processes to coordinate their actions. Using these synchronization objects requires the use of secondary threads so that the message loop can be serviced, but this isn't an exceptional burden.

I described how to create memory-mapped objects in Chapter 6. The example program that shortly follows uses memory-mapped objects and synchronization objects to coordinate access to the shared block of memory.

Communicating with files and databases

A more basic method of interprocess communication is the use of files or a custom database. These methods provide a robust, if slower, communication path. Slow is relative. Files and databases in the Windows CE object store are slow in the sense that the system calls to access these objects must find the data in the object store, uncompress the data, and deliver it to the process. However, since the object store is based in RAM, you see none of the extreme slowness of a mechanical hard disk that you'd see under Windows NT or Windows 98.

The XTalk Example Program

The following example program, XTalk, uses events, mutexes, and a shared memory-mapped block of memory to communicate among different copies of itself. The example demonstrates the rather common problem of one-to-many communication. In this case, the XTalk window has an edit box with a Send button next to it. When a user taps the Send button, the text in the edit box is communicated to every copy of XTalk running on the system. Each copy of XTalk receives the text from the sending copy and places it in a list box also in the XTalk window. Figure 8-1 shows two XTalk programs communicating.

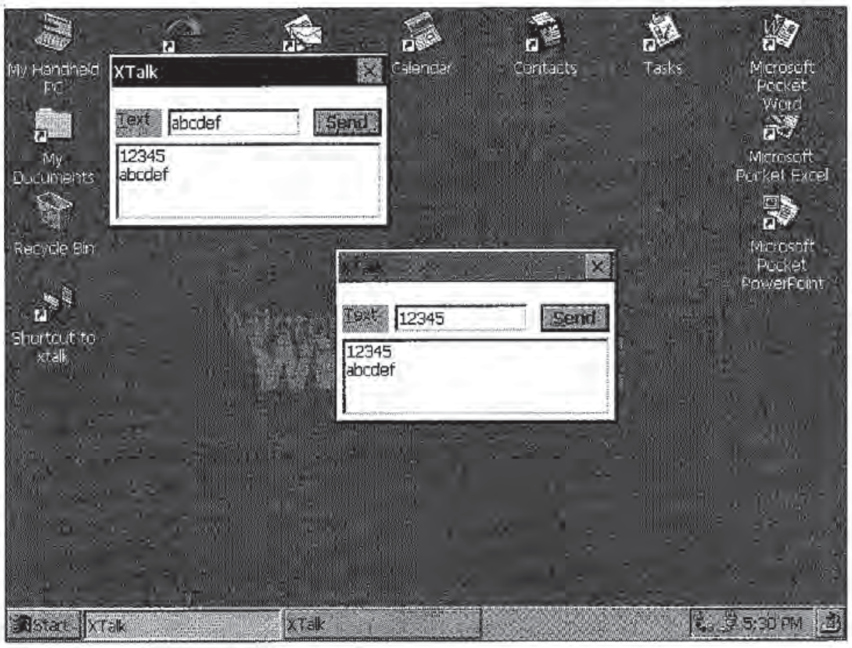


Figure 8-1.
The desktop showing two XTalk windows.

To perform this feat of communication, XTalk uses a named memory-mapped object as a transfer buffer, a mutex to coordinate access to the buffer, and two event objects to indicate the start and end of communication. A third event is used to tell the sender thread to read the text from the edit control and write the contents to the shared memory block. Figure 8-2 shows the source code for XTalk.

XTalk.rc

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

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

//----------------------------------------------------------------------
xtalk DIALOG discardable 10, 10, 120, 60
STYLE  WS_OVERLAPPED | WS_VISIBLE | WS_CAPTION | WS_SYSMENU |
       DS_CENTER | DS_MODALFRAME
CAPTION "XTalk"
CLASS "xtalk"
BEGIN
    LTEXT "&Text"                     -1,   2,  10,  20,  12
    EDITTEXT                IDD_OUTTEXT,  25,  10,  58,  12,
                                           WS_TABSTOP | ES_AUTOHSCROLL
    PUSHBUTTON "&Send",    IDD_SENDTEXT,  88,  10,  30,  12, WS_TABSTOP

    LISTBOX                  IDD_INTEXT,   2,  25, 116,  40,
                                           WS_TABSTOP | WS_VSCROLL
END
```

XTalk.h

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

Figure 8-2. *The source code for XTalk.* *(continued)*

Figure 8-2. *continued*

```
// Copyright (C) 1998 Douglas Boling
//======================================================================
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))
//----------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                             // Structure associates
    UINT Code;                                  // messages
                                                // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                              // Structure associates
    UINT Code;                                  // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);     // function.
};

//----------------------------------------------------------------------
// Generic defines used by application
#define  ID_ICON             1

#define  IDD_INTEXT          10                 // Control IDs
#define  IDD_SENDTEXT        11
#define  IDD_OUTTEXT         12

#define  MMBUFFSIZE          1024               // Size of shared buffer
#define  TEXTSIZE            256

// Interprocess communication structure mapped in shared memory
typedef struct {
    int nAppCnt;
    int nReadCnt;
    TCHAR szText[TEXTSIZE];
} SHAREBUFF;
typedef SHAREBUFF *PSHAREBUFF;

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

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

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

// Command functions
LPARAM DoMainCommandSend (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandExit (HWND, WORD, HWND, WORD);

// Thread functions
int SenderThread (PVOID pArg);
int ReaderThread (PVOID pArg);
```

XTalk.c

```
//======================================================================
// XTalk - A simple application for Windows CE
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                // For all that Windows stuff
#include <commctrl.h>               // Command bar includes
#include "xtalk.h"                  // Program-specific stuff

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

HANDLE g_hMMObj = 0;                // Memory-mapped object
PSHAREBUFF g_pBuff = 0;             // Pointer to mm object
HANDLE g_hmWriteOkay = 0;           // Write mutex
HANDLE g_hSendEvent = 0;            // Local send event
HANDLE g_hReadEvent = 0;            // Shared read data event
HANDLE g_hReadDoneEvent = 0;        // Shared data read event

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_COMMAND, DoCommandMain,
    WM_DESTROY, DoDestroyMain,
};
// Command Message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDOK, DoMainCommandExit,
```

(continued)

Figure 8-2. *continued*

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

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

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

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

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

    if (RegisterClass (&wc) == 0) return 1;
    return 0;
}
//----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow){
    HWND hWnd;
    HANDLE hThread;
    INT rc;
    BOOL fFirstApp = TRUE;

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

    // Create mutex used to share memory-mapped structure.
    g_hmWriteOkay = CreateMutex (NULL, TRUE, TEXT ("XTALKWRT"));
    rc = GetLastError();
    if (rc == ERROR_ALREADY_EXISTS)
        fFirstApp = FALSE;
    else if (rc) return 0;

    // Wait here for ownership to ensure the initialization is done.
    // This is necessary since CreateMutex doesn't wait.
    rc = WaitForSingleObject (g_hmWriteOkay, 2000);
    if (rc != WAIT_OBJECT_0)
        return 0;

    // Create a file-mapping object.
    g_hMMObj = CreateFileMapping ((HANDLE)-1, NULL, PAGE_READWRITE, 0,
                                  MMBUFFSIZE, TEXT ("XTALKBLK"));
    if (g_hMMObj == 0) return 0;

    // Map into memory the file-mapping object.
    g_pBuff = (PSHAREBUFF)MapViewOfFile (g_hMMObj, FILE_MAP_WRITE,
                                         0, 0, 0);
    if (!g_pBuff)
        CloseHandle (g_hMMObj);

    // Initialize structure if first application started.
    if (fFirstApp)
        memset (g_pBuff, 0, sizeof (SHAREBUFF));
```

(continued)

Figure 8-2. *continued*

```
    // Increment app running count. Interlock not needed due to mutex.
    g_pBuff->nAppCnt++;

    // Release the mutex.  We need to release the mutext twice
    // if we owned it when we entered the wait above.
    ReleaseMutex (g_hmWriteOkay);
    if (fFirstApp)
        ReleaseMutex (g_hmWriteOkay);

    // Now create events for read and send notification.
    g_hSendEvent = CreateEvent (NULL, FALSE, FALSE, NULL);
    g_hReadEvent = CreateEvent (NULL, TRUE, FALSE, TEXT ("XTALKREAD"));
    g_hReadDoneEvent = CreateEvent (NULL, FALSE, FALSE,
                                    TEXT ("XTALKDONE"));
    if (!g_hReadEvent || !g_hSendEvent || !g_hReadDoneEvent)
        return 0;

    // Create main window.
    hWnd = CreateDialog (hInst, szAppName, NULL, NULL);
    rc = GetLastError();

    // Create secondary threads for interprocess communication.
    hThread = CreateThread (NULL, 0, SenderThread, hWnd, 0, &rc);
    if (hThread)
        CloseHandle (hThread);
    else {
        DestroyWindow (hWnd);
        return 0;
    }
    hThread = CreateThread (NULL, 0, ReaderThread, hWnd, 0, &rc);
    if (hThread)
        CloseHandle (hThread);
    else {
        DestroyWindow (hWnd);
        return 0;
    }

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

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
```

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

    // Free memory-mapped object.
    if (g_pBuff) {
        // Decrement app running count.
        InterlockedDecrement (&g_pBuff->nAppCnt);
        UnmapViewOfFile (g_pBuff);
    }
    if (g_hMMObj)
        CloseHandle (g_hMMObj);

    // Free mutex.
    if (g_hmWriteOkay)
        CloseHandle (g_hmWriteOkay);

    // Close event handles.
    if (g_hReadEvent)
        CloseHandle (g_hReadEvent);

    if (g_hReadDoneEvent)
        CloseHandle (g_hReadDoneEvent);

    if (g_hSendEvent)
        CloseHandle (g_hSendEvent);
    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);
```

(continued)

Figure 8-2. *continued*

```
}
//----------------------------------------------------------------------
// DoCommandMain - Process WM_COMMAND message for window.
//
LRESULT DoCommandMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    WORD    idItem, wNotifyCode;
    HWND    hwndCtl;
    INT     i;

    // Parse the parameters.
    idItem = (WORD) LOWORD (wParam);
    wNotifyCode = (WORD) HIWORD (wParam);
    hwndCtl = (HWND) lParam;

    // Call routine to handle control message.
    for(i = 0; i < dim(MainCommandItems); i++) {
        if(idItem == MainCommandItems[i].Code)
            return (*MainCommandItems[i].Fxn)(hWnd, idItem, hwndCtl,
                                              wNotifyCode);
    }
    return 0;
}
//----------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
//======================================================================
// Command handler routines
//----------------------------------------------------------------------
// DoMainCommandExit - Process Program Exit command
//
LPARAM DoMainCommandExit (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {

    SendMessage (hWnd, WM_CLOSE, 0, 0);
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandSend - Process Program Send command.
//
LPARAM DoMainCommandSend (HWND hWnd, WORD idItem, HWND hwndCtl,
                            WORD wNotifyCode) {

    SetEvent (g_hSendEvent);
    return 0;
}
//======================================================================
// SenderThread - Performs the interprocess communication
//
int SenderThread (PVOID pArg) {
    HWND hWnd;
    INT nGoCode, rc;
    TCHAR szText[TEXTSIZE];

    hWnd = (HWND)pArg;
    while (1) {
        nGoCode = WaitForSingleObject (g_hSendEvent, INFINITE);
        if (nGoCode == WAIT_OBJECT_0) {
            SendDlgItemMessage (hWnd, IDD_OUTTEXT, WM_GETTEXT,
                                sizeof (szText), (LPARAM)szText);

            rc = WaitForSingleObject (g_hmWriteOkay, 2000);
            if (rc == WAIT_OBJECT_0) {
                lstrcpy (g_pBuff->szText, szText);
                g_pBuff->nReadCnt = g_pBuff->nAppCnt;
                PulseEvent (g_hReadEvent);

                // Wait while reader threads get data.
                while (g_pBuff->nReadCnt)
                    rc = WaitForSingleObject (g_hReadDoneEvent,
                                              INFINITE);
                ReleaseMutex (g_hmWriteOkay);
            }
        } else
            return -1;
    }
    return 0;
}
//======================================================================
// ReaderThread - Performs the interprocess communication
//
int ReaderThread (PVOID pArg) {
    HWND hWnd;
    INT nGoCode, rc, i;
    TCHAR szText[TEXTSIZE];

    hWnd = (HWND)pArg;
```

(continued)

Figure 8-2. *continued*

```
    while (1) {
        nGoCode = WaitForSingleObject (g_hReadEvent, INFINITE);
        if (nGoCode == WAIT_OBJECT_0) {
            i = SendDlgItemMessage (hWnd, IDD_INTEXT, LB_ADDSTRING, 0,
                                    (LPARAM)g_pBuff->szText);
            SendDlgItemMessage (hWnd, IDD_INTEXT, LB_SETTOPINDEX, i, 0);

            InterlockedDecrement (&g_pBuff->nReadCnt);
            SetEvent (g_hReadDoneEvent);
        } else {
            rc = GetLastError();
            wsprintf (szText, TEXT ("rc:%d"), rc);
            MessageBox (hWnd, szText, TEXT ("ReadThread Err"), MB_OK);
        }
    }
    return 0;
}
```

The interesting routines in the XTalk example are the *InitInstance* procedure and the two thread procedures *SenderThread* and *ReaderThread*. The relevant part of *InitInstance* is shown below with the error checking code removed for brevity.

```
// Create mutex used to share memory-mapped structure.
g_hmWriteOkay = CreateMutex (NULL, TRUE, TEXT ("XTALKWRT"));
rc = GetLastError();
if (rc == ERROR_ALREADY_EXISTS)
    fFirstApp = FALSE;

// Wait here for ownership to insure the initialization is done.
// This is necessary since CreateMutex doesn't wait.
rc = WaitForSingleObject (g_hmWriteOkay, 2000);
if (rc != WAIT_OBJECT_0)
    return 0;

// Create a file-mapping object.
g_hMMObj = CreateFileMapping ((HANDLE)-1, NULL, PAGE_READWRITE, 0,
                              MMBUFFSIZE, TEXT ("XTALKBLK"));

// Map into memory the file-mapping object.
g_pBuff = (PSHAREBUFF)MapViewOfFile (g_hMMObj, FILE_MAP_WRITE,
                                     0, 0, 0);

// Initialize structure if first application started.
if (fFirstApp)
    memset (g_pBuff, 0, sizeof (SHAREBUFF));
```

```
// Increment app running count. Interlock not needed due to mutex.
g_pBuff->nAppCnt++;

// Release the mutex.  We need to release the mutex twice
// if we owned it when we entered the wait above.
ReleaseMutex (g_hmWriteOkay);
if (fFirstApp)
    ReleaseMutex (g_hmWriteOkay);

// Now create events for read and send notification.
g_hSendEvent = CreateEvent (NULL, FALSE, FALSE, NULL);
g_hReadEvent = CreateEvent (NULL, TRUE, FALSE, TEXT ("XTALKREAD"));
g_hReadDoneEvent = CreateEvent (NULL, FALSE, FALSE,
                                TEXT ("XTALKDONE"));
```

This code is responsible for creating the necessary synchronization objects as well as creating and initializing the shared memory block. The mutex object is created first with the parameters set to request initial ownership of the mutex object. A call is then made to *GetLastError* to determine whether the mutex object has already been created. If not, the application assumes the first instance of XTalk is running and later will initialize the shared memory block. Once the mutex is created, an additional call is made to *WaitForSingleObject* to wait until the mutex is released. This call is necessary to prevent a late starting instance of XTalk from disturbing communication in progress. Once the mutex is owned, calls are made to *CreateFileMapping* and *MapViewOfFile* to create a named memory-mapped object. Since the object is named, each process that opens the object opens the same object and is returned a pointer to the same block of memory.

Once the shared memory block is created, the first instance of XTalk zeros out the block. This procedure also forces the block of RAM to be committed because memory-mapped objects by default are autocommit blocks. Then *nAppCnt*, which keeps a count of the running instances of XTalk, is incremented. Finally the mutex protecting the shared memory is released. If this is the first instance of XTalk, *ReleaseMutex* must be called twice because it gains ownership of the mutex twice—once when the mutex is created and again when the call to *WaitForSingleObject* is made.

Finally, three event objects are created. *SendEvent* is an unnamed event, local to each instance of XTalk. The primary thread uses this event to signal the sender thread that the user has pressed the Send button and wants the text in the edit box transmitted. The *ReadEvent* is a named event that tells the other instances of XTalk that there's data to be read in the transfer buffer. The *ReadDoneEvent* is a named event signaled by each of the receiving copies of XTalk to indicate that they have read the data.

The two threads, *ReaderThread* and *SenderThread* are created immediately after the main window of XTalk is created. The code for *SenderThread* is shown here:

```
int SenderThread (PVOID pArg) {
    HWND hWnd;
    INT nGoCode, rc;
    TCHAR szText[TEXTSIZE];

    hWnd = (HWND)pArg;
    while (1) {
        nGoCode = WaitForSingleObject (g_hSendEvent, INFINITE);
        if (nGoCode == WAIT_OBJECT_0) {
            SendDlgItemMessage (hWnd, IDD_OUTTEXT, WM_GETTEXT,
                                sizeof (szText), (LPARAM)szText);

            rc = WaitForSingleObject (g_hmWriteOkay, 2000);
            if (rc == WAIT_OBJECT_0) {
                lstrcpy (g_pBuff->szText, szText);
                g_pBuff->nReadCnt = g_pBuff->nAppCnt;
                PulseEvent (g_hReadEvent);

                // Wait while reader threads get data.
                while (g_pBuff->nReadCnt)
                    rc = WaitForSingleObject (g_hReadDoneEvent,
                                              INFINITE);
                ReleaseMutex (g_hmWriteOkay);
            }
        }
    }
    return 0;
}
```

The routine waits on the primary thread of XTalk to signal *SendEvent*. The primary thread of XTalk makes the signal in response to a WM_COMMAND message from the Send button. The thread is then unblocked, reads the text from the edit control, and waits to gain ownership of the *WriteOkay* mutex. This mutex protects two copies of XTalk from writing to the shared block at the same time. When the thread owns the mutex, it writes the string read from the edit control into the shared buffer. It then copies the number of active copies of XTalk into the *nReadCnt* variable in the same shared buffer, and pulses the *ReadEvent* to tell the other copies of XTalk to read the newly written data. A manual resetting event is used so that all threads waiting on the event will be unblocked when the event is signaled.

The thread then waits for the *nReadCnt* variable to return to 0. Each time a reader thread reads the data, the *nReadCnt* variable is decremented and the *ReadDone* event signaled. Note that the thread doesn't spin on this variable but uses an event to tell it when to check the variable again. This would actually be a great place to use

WaitForMultipleObjects and have all reader threads signal when they've read the data, but Windows CE doesn't support the *WaitAll* flag in *WaitForMultipleObjects*.

Finally, when all the reader threads have read the data, the sender thread releases the mutex protecting the shared segment and the thread returns to wait for another send event.

The *ReaderThread* routine is even simpler. Here it is:

```
int ReaderThread (PVOID pArg) {
    HWND hWnd;
    INT nGoCode, rc, i;
    TCHAR szText[TEXTSIZE];

    hWnd = (HWND)pArg;
    while (1) {
        nGoCode = WaitForSingleObject (g_hReadEvent, INFINITE);
        if (nGoCode == WAIT_OBJECT_0) {
            i = SendDlgItemMessage (hWnd, IDD_INTEXT, LB_ADDSTRING, 0,
                                    (LPARAM)g_pBuff->szText);
            SendDlgItemMessage (hWnd, IDD_INTEXT, LB_SETTOPINDEX, i, 0);

            InterlockedDecrement (&g_pBuff->nReadCnt);
            SetEvent (g_hReadDoneEvent);
        }
    }
    return 0;
}
```

The reader thread starts up and immediately blocks on *ReadEvent*. When it's unblocked, it adds the text from the shared buffer into the list box in its window. The list box is then scrolled to show the new line. After this is accomplished, the *nReadCnt* variable is decremented using *InterlockedDecrement* to be thread safe, and the *ReadDone* event is signaled to tell the *SenderThread* to check the read count. After that's accomplished, the routine loops around and waits for another read event to occur.

EXCEPTION HANDLING

Windows CE, along with Visual C++ for Windows CE, supports Microsoft's standard, structured exception handling extensions to the C language, the *__try*, *__except* and *__try*, *__finally* blocks. Note that Visual C++ for Windows CE doesn't support the full C++ exception handling framework with keywords such as *catch* and *throw*.

Windows exception handling is complex and if I were to cover it completely, I could easily write another entire chapter. The following review introduces the concepts to non-Win32 programmers and conveys enough information about the subject for you to get your feet wet. If you want to wade all the way in, the best source for a complete explanation of Win32 exception handling is Jeffrey Richter's *Advanced Windows* third edition (Microsoft Press, 1997).

The *__try, __except* Block

The first construct I'll talk about is the *__try, __except* block which looks like this:

```
__try {

    // Try some code here that might cause an exception.

}
__except (exception filter) {

    // This code is depending on the filter on the except line.

}
```

Essentially, the *try-except* pair allows you the ability to anticipate exceptions and handle them locally instead of having Windows terminate the thread or the process because of an unhandled exception.

The exception filter is essentially a return code that tells Windows how to handle the exception. You can hard code one of the three possible values or call a function that dynamically decides how to respond to the exception.

If the filter returns EXCEPTION_EXECUTE_HANDLER, Windows aborts the execution in the *try* block and jumps to the first statement in the *except* block. This is helpful if you're expecting the exception and you know how to handle it. In the code that follows, the access to memory is protected by a *__try, __except* block.

```
BYTE ReadByteFromMemory (LPBYTE pPtr, BOOL *bDataValid) {
    BYTE ucData = 0;

    *bDataValid = TRUE;
    __try {
        ucData = *pPtr;
    }
    __except (DecideHowToHandleException ()) {
        // The pointer isn't valid, clean up.
        ucData = 0;
        *bDataValid = FALSE;
    }
    return ucData;
}
int DecideHowToHandleException (void) {
    return EXCEPTION_EXECUTE_HANDLER;
}
```

If the memory read line above wasn't protected by a *__try*, *__except* block and an invalid pointer was passed to the routine, the exception generated would have been passed up to the system, causing the thread and perhaps the process to be terminated. If you use the *__try*, *__except* block, the exception is handled locally and the process continues with the error handled locally.

Another possibility is to have the system retry the instruction that caused the exception. You can do this by having the filter return EXCEPTION_CONTINUE_EXECUTION. On the surface, this sounds like a great option—simply fix the problem and retry the operation your program was performing. The problem with this approach is that what will be retried isn't the *line* that caused the exception, but *the machine instruction* that caused the exception. The difference is illustrated by the following code fragment that looks okay but probably won't work:

```
// An example that doesn't work...
int DivideIt (int aVal, int bVal) {
    int cVal;
    __try {
        cVal = aVal / bVal;
    }
    __except (EXCEPTION_CONTINUE_EXECUTION) {
        bVal = 1;
    }
    return cVal;
}
```

The idea in this code is noble; protect the program from a divide-by-zero error by ensuring that if the error occurs, the error is corrected by replacing *bVal* with 1. The problem is that the line

```
cVal = aVal / bVal;
```

is probably compiled to something like the following on a MIPS-compatible CPU:

```
lw    t6,aVal(sp)         ;Load aVal
lw    t7,bVal(sp)         ;Load bVal
div   t6,t7               ;Perform the divide
sw    t6,cVal(sp)         ;Save result into cVal
```

In this case, the third instruction, the *div*, causes the exception. Restarting the code after the exception results in the restart beginning with the *div* instruction. The problem is that the execution needs to start at least one instruction earlier to load the new value from *bVal* into the register. The moral of the story is that attempting to restart code at the point of an exception is risky at best and at worst, unpredictable.

The third option for the exception filter is to not even attempt to solve the problem and to pass the exception up to the next, higher *__try*, *__except* block in code.

This is accomplished by the exception filter returning EXCEPTION_CONTINUE_SEARCH. Since *__try*, *__except* blocks can be nested, it's good practice to handle specific problems in a lower, nested *__try*, *__except* block and more global errors at a higher level.

Determining the problem

With these three options available, it would be nice if Windows let you in on why the exception occurred. Fortunately, Windows provides the function

```
DWORD GetExceptionCode (void);
```

This function returns a code that indicates why the exception occurred in the first place. The codes are defined in WINBASE.H and range from EXCEPTION_ACCESS_VIOLATION to CONTROL_C_EXIT, with a number of codes in between. Another function allows even more information:

```
LPEXCEPTION_POINTERS GetExceptionInformation (void);
```

GetExceptionInformation returns a pointer to a structure that contains pointers to two structures: EXCEPTION_RECORD and CONTEXT. EXCEPTION_RECORD is defined as

```
typedef struct _EXCEPTION_RECORD {
    DWORD ExceptionCode;
    DWORD ExceptionFlags;
    struct _EXCEPTION_RECORD *ExceptionRecord;
    PVOID ExceptionAddress;
    DWORD NumberParameters;
    DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

The fields in this structure go into explicit detail about why an exception occurred. To narrow the problem down even further, you can use the CONTEXT structure. The CONTEXT structure is different for each CPU and essentially defines the exact state of the CPU when the exception occurred.

There are limitations on when these two exception information functions can be called. *GetExecptionCode* can only be called from inside an *except* block or from within the exception filter function. The *GetExceptionInformation* function can be called only from within the exception filter function.

The *__try, __finally* Block

Another tool of the structured exception handling features of the Win32 API is the *__try*, *__finally* block. It looks like this:

```
__try {

    // Do something here.

}
__finally {

    // This code is executed regardless of what happens in the try block.

}
```

The goal of the *__try, __finally* block is to provide a block of code, the *finally* block, that always executes regardless of how the other code in the *try* block attempts to leave the block. If there's no *return, break* or *goto* in the *try* block, the code in the *finally* block executes immediately following the last statement in the *try* block. If the *try* block has a *return* or a *goto* or some other statement that transfers execution out of the *try* block, the compiler insures that the code in the *finally* block will get executed before execution leaves the *try* block. Take, for example, the following code:

```
int ClintSimFunc (int TodaysTask) {

    __try {
        switch (TodaysTask) {
        case THEGOOD:
            //Do the good stuff.
            return 1;
        case THEBAD:
            //Do the bad stuff.
            return 2;
        case THEUGLY:
            //Do the ugly stuff.
            break;
        }
        // Climb the Eiger.
        return 0;
    }
    __finally {
        // Reload the .44.
    }
}
```

In this example, the *try* block can be left three ways: returning after executing the Good case or the Bad case or after executing the Ugly case, which breaks and executes the Eiger code. However the code exits the *try* block, Clint's gun is always reloaded because the *finally* block is always executed.

It works out that having the compiler build the code to protect the *try* block exits tends to create a fair amount of extra code. To help, you can use another statement, *__leave*, which makes it easier for the compiler to recognize what's happening and make a code-efficient path to the *finally* block. Using the *__leave* statement, the code above becomes

```
int ClintSimFunc (int TodaysTask) {
    int nFistfull;

    __try {
        switch (TodaysTask) {
        case THEGOOD:

            //Do the good stuff.
            nFistfull = 1;
            __leave;
        case THEBAD:
            //Do the bad stuff.
            nFistfull = 2;
            __leave;
        case THEUGLY:
            //Do the ugly stuff.
            break;
        }
        // Climb the Eiger.
        nFistFull = 0;
    }
    // The code falls into the __finally block.
    __finally {
        // Reload the .44.
    }
    return nFistfull;
}
```

The *__try*, *__finally* block is helpful for writing clean code because you can use the *__leave* statement to jump out of a sequence of statements that build upon one another and put all the cleanup code in the *finally* block. The *finally* block also has a place in structured exception handling since the *finally* code is executed if an exception in the *try* block causes a premature exit of the block.

In the past three chapters, I've covered the basics of the Windows CE kernel from memory to files to processes and threads. Now it's time to break from this low-level stuff and starting looking outward. The next section covers the different communication aspects of Windows CE. I start at the low level, with explanations of basic serial and I/R communication and TAPI. Chapter 10 covers networking from a Windows CE perspective. Finally, Chapter 11 covers Windows CE to PC communications. That's a fair amount of ground to cover. Let's get started.

Part III

COMMUNICATIONS

Chapter 9

Serial Communications

If there's one area of the Win32 API that Windows CE doesn't skimp, it's in communication. It makes sense. Systems running Windows CE are either mobile, requiring extensive communication functionality, or they're devices generally employed to communicate with remote servers. In this chapter, I introduce the low-level serial and infrared communication APIs. You use the infrared port at this level in almost the same manner as a serial port. The only functional difference is that infrared transmission is *half duplex*, that is, transmission can occur in only one direction at a time.

BASIC DRIVERS

Before I can delve into the serial drivers, we must take a brief look at how Windows CE handles drivers in general. Windows CE separates device drivers into two main groups: native and stream interface. Native drivers, sometimes called *built-in drivers*, are those device drivers that are required for the hardware and were created by the OEM when the Windows CE hardware was designed. Among the devices that have, native drivers are the keyboard, the touch panel, audio, and the PCMCIA controller. These drivers might not support the generic device driver interface I describe below. Instead, they might extend the interface or have a totally custom interface to the operating system. Native drivers frequently require minor changes when a new version of the operation system is released. These drivers are designed using the OEM adaptation kit supplied by Microsoft. A more general adaptation kit, the

Embedded Toolkit (ETK), also enables you to develop built-in drivers. However these drivers are developed, they're tightly bound to the Windows CE operating system and aren't usually replaced after the device has been sold.

On the other hand, stream interface device drivers (which used to be referred to as installable drivers) can be supplied by third-party manufacturers to support hardware added to the system. Since Windows CE systems generally don't have a bus such as an ISA bus or a PCI bus for extra cards, the additional hardware is usually installed via a PCMCIA or a Compact Flash slot. In this case, the device driver would use functions provided by the low-level PCMCIA driver to access the card in the PCMCIA or the Compact Flash slot.

In addition, a device driver might be written to extend the functionality of an existing driver. For example, you might write a driver to provide a compressed or encrypted data stream over a serial link. In this case, an application would access the encryption driver, which would then in turn use the serial driver to access the serial hardware.

Device drivers under Windows CE operate at the same protection level as applications. They differ from applications in that they're DLLs. Most drivers are loaded by the device manager process (DEVICE.EXE) when the system boots. All these drivers, therefore, share the same process address space. Some of the built-in drivers, on the other hand, are loaded by GWE (GWES.EXE). (GWE stands for Graphics Windowing and Event Manager.) These drivers include the display driver (DDI.DLL) as well as the keyboard and touch panel (or mouse) drivers.

Driver Names

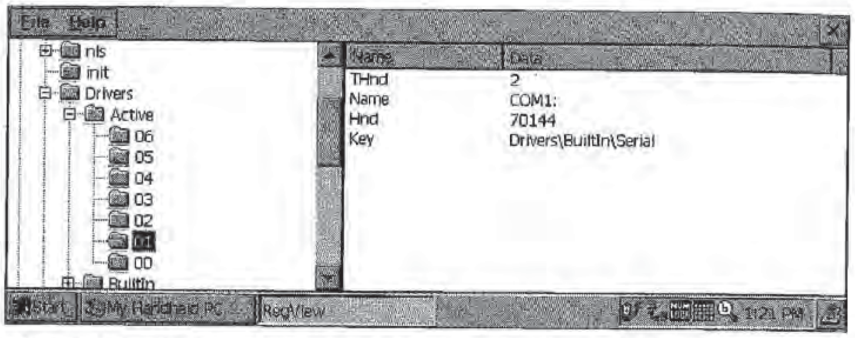
Stream interface device drivers are identified by a three-character name followed by a single digit. This scheme allows for 10 device drivers of one name to be installed on a Windows CE device at any one time. Here are a few examples of some three-character names currently in use:

COM	Serial driver
ACM	Audio compression manager
WAV	Audio wave driver
CON	Console driver

When referencing a stream interface driver, an application uses the three-character name, followed by the single digit, followed by a colon (:). The colon is required under Windows CE for the system to recognize the driver name.

Enumerating the Active Drivers

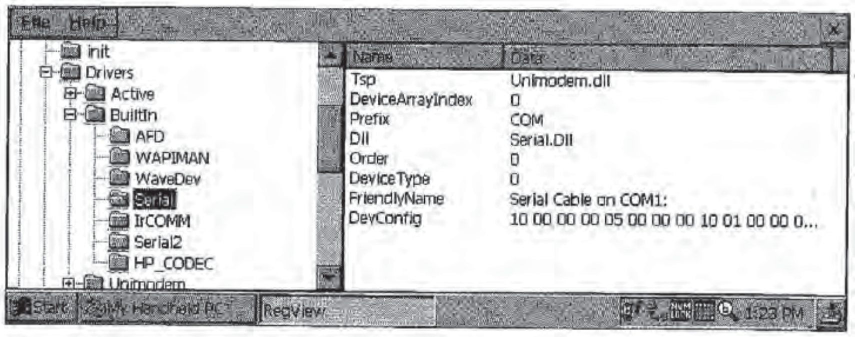
The documented method for determining what drivers are loaded onto a Windows CE system is to look in the registry under the key \Drivers\Active under HKEY_LOCAL_MACHINE. The device manager dynamically updates the subkeys contained here as drivers are loaded and unloaded from the system. Contained in this key is a list of subkeys, one for each active driver. The name of the key is simply a placeholder; it's the values inside the keys that indicate the active drivers. Figure 9-1 shows the registry key for the COM1 serial driver on an HP 620.



Name	Data
THnd	2
Name	COM1:
Hnd	70144
Key	Drivers\BuiltIn\Serial

Figure 9-1. *The registry's active list values for the serial device driver for COM1.*

In Figure 9-1, the *Name* value contains the official five-character name (four characters plus a colon) of the device. The *THnd* and *Hnd* values are handles that are used internally by Windows CE. The interesting entry is the *Key* value. This value points to the registry key where the device driver stores its configuration information. This second key is necessary because the active list is dynamic, changing whenever a device is installed. In the case of the serial driver, its configuration data is generally stored in Drivers\BuiltIn\Serial although you shouldn't hard code this value. Instead, you can look at the *Key* value in the active list to determine the location of a driver's permanent configuration data. The configuration data for the serial driver is shown in Figure 9-2.



Name	Data
Tsp	Unimodem.dll
DeviceArrayIndex	0
Prefix	COM
Dll	Serial.Dll
Order	0
DeviceType	0
FriendlyName	Serial Cable on COM1:
DevConfig	10 00 00 00 05 00 00 00 10 01 00 00 0...

Figure 9-2. *The registry entry for the serial driver.*

You can look in the serial driver registry key for such information as the name of the DLL that actually implements the driver, the three-letter prefix defining the driver name, the order in which the driver wants to be loaded, and something handy for user interfaces, the *friendly name* of the driver. Not all drivers have this friendly name, but when they do, it's a much more user-friendly name than COM2 or NDS1.

Drivers for PCMCIA or Compact Flash Cards have an additional value in their active list key. The *PnpId* value contains the Plug and Play ID string for the card. While this string is more descriptive than the five-character driver name, some PCMCIA and Compact Flash Cards have their *PnpId* strings registered in the system. If so, a registry key for the *PnpId* is located in the *Drivers\PCMCIA* key under HKEY_LOCAL_MACHINE. For example, a PCMCIA Card that had a *PnpId* string *This_is_a_pc_card* would be registered under the key *\Drivers\PCMCIA\This_is_a_pc_card*. That key may contain a *FriendlyName* string for the driver.

Following is a routine (and a small helper routine) that creates a list of active drivers and, if specified, their friendly names. The routine produces a series of Unicode strings, two for each active driver. The first string is the driver name, followed by its friendly name. If a driver doesn't have a friendly name, a zero-length string is inserted in the list. The list ends with a zero-length string for the driver name.

```
//----------------------------------------------------------------------
// AddToList - Helper routine
int AddToList (LPTSTR *pPtr, INT *pnListSize, LPTSTR pszStr) {
    INT nLen = lstrlen (pszStr) + 1;

    if (*pnListSize < nLen)
        return -1;
    lstrcpy (*pPtr, pszStr);
    *pPtr += nLen;
    *pnListSize -= nLen;
    return 0;
}
//----------------------------------------------------------------------
// EnumActiveDrivers - Produces a list of active drivers
//
int EnumActiveDrivers (LPTSTR pszDrvrList, int nListSize) {
    INT i = 0, rc;
    HKEY hKey, hSubKey, hDrvrKey;
    TCHAR szKey[128], szValue[128];
    LPTSTR pPtr = pszDrvrList;
    DWORD dwType, dwSize;

    *pPtr = TEXT ('\0');
    if (RegOpenKeyEx (HKEY_LOCAL_MACHINE, TEXT ("drivers\\active"), 0,
                      0, &hKey) != ERROR_SUCCESS)
        return 0;

    while (1) {
        // Enumerate active driver list.
        dwSize = sizeof (szKey);
        if (RegEnumKeyEx (hKey, i++, szKey, &dwSize, NULL, NULL,
                          NULL, NULL) != ERROR_SUCCESS)
            break;
        // Open active driver key.
        rc = RegOpenKeyEx (hKey, szKey, 0, 0, &hSubKey);
        if (rc != ERROR_SUCCESS)
            continue;

        // Get name of device.
        dwSize = sizeof (szValue);
        rc = RegQueryValueEx (hSubKey, TEXT ("Name"), 0, &dwType,
                              (PBYTE)szValue, &dwSize);
        if (rc != ERROR_SUCCESS)
            szValue[0] = TEXT ('\0');

        if (AddToList (&pPtr, &nListSize, szValue)) {
            rc = -1;
            RegCloseKey (hSubKey);
            break;
        }

        // Get friendly name of device.
        szValue[0] = TEXT ( '\0');
        dwSize = sizeof (szKey);
        rc = RegQueryValueEx (hSubKey, TEXT ("Key"), 0, &dwType,
                              (PBYTE)szKey, &dwSize);
        if (rc == ERROR_SUCCESS) {
            // Get driver friendly name.
            if (RegOpenKeyEx (HKEY_LOCAL_MACHINE, szKey, 0, 0,
                              &hDrvrKey) == ERROR_SUCCESS) {

                dwSize = sizeof (szValue);
                RegQueryValueEx (hDrvrKey, TEXT ("FriendlyName"), 0,
                                 &dwType, (PBYTE)szValue, &dwSize);
                RegCloseKey (hDrvrKey);
            }
        }
        RegCloseKey (hSubKey);
        if (AddToList (&pPtr, &nListSize, szValue)) {
            rc = -1;
            break;
        }
    }
    RegCloseKey (hKey);
    // Add terminating zero.
    if (!rc)
        rc = AddToList (&pPtr, &nListSize, TEXT (""));
    return rc;
}
```

Reading and Writing Device Drivers

Your application accesses device drivers under Windows CE through the file I/O functions, *CreateFile*, *ReadFile*, *WriteFile*, and *CloseHandle*. You open the device using *CreateFile*, with the name of the device being the five-character (three characters plus digit plus colon) name of the driver. Drivers can be opened with all the varied access rights: read only, write only, read/write, or neither read nor write access.

Once a device is open, you can send data to it using *WriteFile* and can read from the device using *ReadFile*. As is the case with file operations, overlapped I/O isn't supported for devices under Windows CE. The driver can be sent control characters using the function (not described in Chapter 7) *DeviceIoControl*. The function is prototyped this way:

```
BOOL DeviceIoControl (HANDLE hDevice, DWORD dwIoControlCode,
                      LPVOID lpInBuffer, DWORD nInBufferSize,
                      LPVOID lpOutBuffer, DWORD nOutBufferSize,
                      LPDWORD lpBytesReturned,
                      LPOVERLAPPED lpOverlapped);
```

The first parameter is the handle to the opened device. The second parameter, *dwIoControlCode*, is the *IoCtl* (pronounced eye-OC-tal) code. This value defines the operation of the call to the driver. The next series of parameters are generic input and output buffers and their sizes. The use of these buffers is dependent on the *IoCtl* code passed in *dwIoControlCode*. The *lpBytesReturned* parameter must point to a DWORD value that will receive the number of bytes returned by the driver in the buffer pointed to by *lpOutBuffer*.

Each driver has its own set of *IoCtl* codes. If you look in the source code for the example serial driver provided in the ETK, you'll see that the following *IoCtl* codes are defined for the COM driver. Note that these codes aren't defined in the Windows CE SDK because an application doesn't need to directly call *DeviceIoControl* using these codes.

IOCTL_SERIAL_SET_BREAK_ON	IOCTL_SERIAL_SET_BREAK_OFF
IOCTL_SERIAL_SET_DTR	IOCTL_SERIAL_CLR_DTR
IOCTL_SERIAL_SET_RTS	IOCTL_SERIAL_CLR_RTS
IOCTL_SERIAL_SET_XOFF	IOCTL_SERIAL_SET_XON
IOCTL_SERIAL_GET_WAIT_MASK	IOCTL_SERIAL_SET_WAIT_MASK
IOCTL_SERIAL_WAIT_ON_MASK	IOCTL_SERIAL_GET_COMMSTATUS
IOCTL_SERIAL_GET_MODEMSTATUS	IOCTL_SERIAL_GET_PROPERTIES
IOCTL_SERIAL_SET_TIMEOUTS	IOCTL_SERIAL_GET_TIMEOUTS
IOCTL_SERIAL_PURGE	IOCTL_SERIAL_SET_QUEUE_SIZE
IOCTL_SERIAL_IMMEDIATE_CHAR	IOCTL_SERIAL_GET_DCB
IOCTL_SERIAL_SET_DCB	IOCTL_SERIAL_ENABLE_IR
IOCTL_SERIAL_DISABLE_IR	

As you can see from the fairly self-descriptive names, the serial driver *IoCtl* functions expose significant function to the calling process. Windows uses these *IoCtl* codes to control some of the specific features of a serial port, such as the handshaking lines and time outs. Each driver has its own set of *IoCtl* codes. I've shown the ones above simply as an example of how the *DeviceIoControl* function is typically used. Under most circumstances, there's no reason for an application to use the *DeviceIoControl* function with the serial driver. Windows provides its own set of functions that then call down to the serial driver using *DeviceIoControl*.

Okay, we've talked enough about generic drivers. It's time to sit down to the meat of the chapter—serial communication. I'll talk first about basic serial connections, and then venture into infrared communication. Windows CE provides excellent support for serial communications, but the API is a subset of the API for Windows NT or Windows 98. Fortunately, the basics are quite similar, and the differences mainly inconsequential.

BASIC SERIAL COMMUNICATION

The interface for a serial device is a combination of generic driver I/O calls and specific communication-related functions. The serial device is treated as a generic, installable, stream device for opening, closing, reading, and writing the serial port. For configuring the port, the Win32 API supports a set of Comm functions. Windows CE supports most of the Comm functions supported by Windows NT and Windows 98.

A word of warning: programming a serial port under Windows CE isn't like programming one under MS-DOS. You can't simply find the base address of the serial port and program the registers directly. While there are ways for a program to gain access to the physical memory space, every Windows CE device has a different physical memory map. Even if you solved the access problem by knowing exactly where the serial hardware resided in the memory map, there's no guarantee the serial hardware is going to be compatible with the 8250 (or, these days, a 16550) serial interface we've all come to know and love in the PC world. In fact, the implementation of the serial port on some Windows CE devices looks nothing like an 8250.

But even if you know where to go in the memory map and the implementation of the serial hardware, you still don't need to "hack down to the hardware." The serial port drivers in Windows CE are efficient, interrupt-driven designs and are written to support its specific serial hardware. If you have any special needs not provided by the base serial driver, you can purchase the Embedded Toolkit and write a serial driver yourself. Aside from that extreme case, there's just no reason not to use the published Win32 serial interface under Windows CE.

Opening and Closing a Serial Port

As with all stream device drivers, a serial port device is opened using *CreateFile*. The name used needs to follow the standards I described previously, with the three letters COM followed by the number of the COM port to open and then a colon. The colon is required under Windows CE and is a departure from the naming convention used for device driver names used in Windows NT and Windows 98. The following line opens COM port 1 for reading and writing:

```
hSer = CreateFile (TEXT ("COM1:"), GENERIC_READ | GENERIC_WRITE,
                   0, NULL, OPEN_EXISTING, 0, NULL);
```

You must pass a 0 in the sharing parameter as well as in the security attributes and the template file parameters of *CreateFile*. Windows CE doesn't support overlapped I/O for devices, so you can't pass the FILE_FLAG_OVERLAPPED flag in the *dwFlagsAndAttributes* parameter. The handle returned is either the handle to the opened serial port or INVALID_HANDLE_VALUE. Remember that, unlike many of the Windows functions, *CreateFile* doesn't return a 0 for a failed open.

You close a serial port by calling *CloseHandle*, as in the following:

```
CloseHandle (hSer);
```

You don't do anything differently when using *CloseHandle* to close a serial device than when you use it to close a file handle.

Reading and Writing to a Serial Port

Just as you use the *CreateFile* function to open a serial port, you use the functions *ReadFile* and *WriteFile* to write to that serial port. Reading data from a serial port is as simple as making this call to *ReadFile*:

```
INT rc;
DWORD cBytes;
BYTE ch;

rc = ReadFile(hSer, &ch, 1, &cBytes, NULL);
```

This call assumes the serial port has been successfully opened with a call to *CreateFile*. If the call is successful, one byte is read into the variable *ch*, and *cBytes* is set to the number of bytes read.

Writing to a serial port is just as simple. The call would look something like the following:

```
INT rc;
DWORD cBytes;
BYTE ch;

ch = TEXT ('a');
rc = WriteFile(hSer, &ch, 1, &cBytes, NULL);
```

This code writes the character *a* to the serial port previously opened. As you may remember from Chapter 7, both *ReadFile* and *WriteFile* return TRUE if successful.

Since overlapped I/O isn't supported under Windows CE, you should be careful not to attempt to read or write a large amount of serial data from your primary thread or from any thread that has created a window. Because those threads are also responsible for handling the message queues for their windows, they can't be blocked waiting on a relatively slow serial read or write. Instead, you should use separate threads for reading and writing the serial port.

You can also transmit a single character using this function:

```
BOOL TransmitCommChar (HANDLE hFile, char cChar);
```

The difference between *TransmitCommChar* and *WriteFile* is that *TransmitCommChar* puts the character to be transmitted at the front of the transmit queue. When you call *WriteFile*, the characters are queued up after any characters that haven't yet been transmitted by the serial driver. *TransmitCommChar* allows you to insert control characters quickly in the stream without having to wait for the queue to empty.

Asynchronous Serial I/O

While Windows CE doesn't support overlapped I/O, there's no reason why you can't use multiple threads to implement the same type of overlapped operation. All that's required is that you launch separate threads to handle the synchronous I/O operations while your primary thread goes about its business. In addition to using separate threads for reading and writing, Windows CE supports the Win32 *WaitCommEvent* function that blocks a thread until one of a group of preselected serial events occurs. I'll demonstrate how to use separate threads for reading and writing a serial port in the CeChat example program later in this chapter.

You can make a thread wait on serial driver events by means of the following three functions:

```
BOOL SetCommMask (HANDLE hFile, DWORD dwEvtMask);
BOOL GetCommMask (HANDLE hFile, LPDWORD lpEvtMask);
```

and

```
BOOL WaitCommEvent (HANDLE hFile, LPDWORD lpEvtMask,
                    LPOVERLAPPED lpOverlapped);
```

To wait on an event, you first set the event mask using *SetCommMask*. The parameters for this function are the handle to the serial device and a combination of the following event flags:

- *EV_BREAK* A break was detected.
- *EV_CTS* The Clear to Send (CTS) signal changed state.

- *EV_DSR* The Data Set Ready (DSR) signal changed state.
- *EV_ERR* An error was detected by the serial driver.
- *EV_RLSD* The Receive Line Signal Detect (RLSD) line changed state.
- *EV_RXCHAR* A character was received.
- *EV_RXFLAG* An event character was received.
- *EV_TXEMPTY* The transmit buffer is empty.

You can set any or all of the flags in this list at the same time using *SetCommMask*. You can query the current event mask using *GetCommMask*.

To wait on the events specified by *SetCommMask*, you call *WaitCommEvent*. The parameters for this call are the handle to the device, a pointer to a DWORD that will receive the reason the call returned, and *lpOverlapped*, which under Windows CE must be set to NULL. The code fragment that follows waits on a character being received or an error. The code assumes that the serial port has already been opened and the handle is contained in *hComPort*.

```
DWORD dwMask;
// Set mask and wait.
SetCommMask (hComPort, EV_RXCHAR | EV_ERR);
if (WaitCommEvent (hComPort, &dwMask, 0) {

    // Use the flags returned in dwMask to determine the reason
    // for returning.
    Switch (dwMask) {
    case EV_RXCHAR:
        //Read character.
        break;
    case EV_ERR:
        // Process error.
        break;
    }
}
```

Configuring the Serial Port

Reading and writing to a serial port is fairly straightforward, but you also must configure the port for the proper baud rate, character size, and so forth. The masochist could configure the serial driver through device I/O control (IOCTL) calls but the *IoCtl* codes necessary for this are exposed only in the Embedded Toolkit, not the Software Development Kit. Besides, here's a simpler method.

You can go a long way in configuring the serial port using two functions, *GetCommState* and *SetCommState*, prototyped here:

```
BOOL SetCommState (HANDLE hFile, LPDCB lpDCB);
BOOL GetCommState (HANDLE hFile, LPDCB lpDCB);
```

Both these functions take two parameters, the handle to the opened serial port and a pointer to a DCB structure. The extensive DCB structure is defined as follows:

```
typedef struct _DCB {
    DWORD DCBlength;
    DWORD BaudRate;
    DWORD fBinary: 1;
    DWORD fParity: 1;
    DWORD fOutxCtsFlow:1;
    DWORD fOutxDsrFlow:1;
    DWORD fDtrControl:2;
    DWORD fDsrSensitivity:1;
    DWORD fTXContinueOnXoff:1;
    DWORD fOutX: 1;
    DWORD fInX: 1;
    DWORD fErrorChar: 1;
    DWORD fNull: 1;
    DWORD fRtsControl:2;
    DWORD fAbortOnError:1;
    DWORD fDummy2:17;
    WORD wReserved;
    WORD XonLim;
    WORD XoffLim;
    BYTE ByteSize;
    BYTE Parity;
    BYTE StopBits;
    char XonChar;
    char XoffChar;
    char ErrorChar;
    char EofChar;
    char EvtChar;
    WORD wReserved1;
} DCB;
```

As you can see from the structure, the *SetCommState* can set a fair number of states. Instead of attempting to fill out the entire structure from scratch, you should use the best method of modifying a serial port, which is to call *GetCommState* to fill in a DCB structure, modify the fields necessary, and then call *SetCommState* to configure the serial port.

The first field in the DCB structure, *DCBlength*, should be set to the size of the structure. The *BaudRate* field should be set to one of the baud rate constants defined in WINBASE.H. The baud rate constants range from CBR_110 for 110 bits per second to CBR_256000 for 256 kilobits per second (Kbps). Just because constants are defined for speeds up to 256 Kbps doesn't mean that all serial ports support that speed. To determine what baud rates a serial port supports, you can call *GetCommProperties*, which I'll describe shortly. Windows CE devices generally support speeds up to 115 Kbps, although some support faster speeds. The *fBinary* field must be set to TRUE because no Win32 operating system currently supports a nonbinary serial transmit mode familiar to MS-DOS programmers. The *fParity* field can be set to TRUE to enable parity checking.

The *fOutxCtsFlow* field should be set to TRUE if the output of the serial port should be controlled by the port CTS line. The *fOutxDsrFlow* field should be set to TRUE if the output of the serial port should be controlled by the DSR line of the serial port. The *fDtrControl* field can be set to one of three values: DTR_CONTROL_DISABLE, which disables the DTR (Data Terminal Ready) line and leaves it disabled; DTR_CONTROL_ENABLE, which enables the DTR line; or DTR_CONTROL_HANDSHAKE, which tells the serial driver to toggle the DTR line in response to how much data is in the receive buffer.

The *fDsrSensitivity* field is set to TRUE, and the serial port ignores any incoming bytes unless the port DSR line is enabled. Setting the *fTXContinueOnXoff* field to TRUE tells the driver to stop transmitting characters if its receive buffer has reached its limit and the driver has transmitted an XOFF character. Setting the *fOutX* field to TRUE specifies that the XON/XOFF control is used to control the serial output. Setting the *fInX* field to TRUE specifies that the XON/XOFF control is used for the input serial stream.

The *fErrorChar* and *ErrorChar* fields are ignored by the default implementation of the Windows CE serial driver although some drivers might support these fields. Likewise, the *fAbortOnError* fields is also ignored. Setting the *fNull* field to TRUE tells the serial driver to discard null bytes received.

The *fRtsControl* field specifies the operation of the RTS (Request to Send) line. The field can be set to one of the following: RTS_CONTROL_DISABLE, indicating that the RTS line is set to the disabled state while the port is open; RTS_CONTROL_ENABLE, indicating that the RTS line is set to the enabled state while the port is open; or RTS_CONTROL_HANDSHAKE, indicating that the RTS line is controlled by the driver. In this mode, if the serial input buffer is less than half full, the RTS line is enabled and disabled otherwise. Finally, RTS_CONTROL_TOGGLE indicates the driver enables the RTS line if there are bytes in the output buffer ready to be transmitted and disables the line otherwise.

The *XonLim* field specifies the minimum number of bytes in the input buffer before an XON character is automatically sent. The *XoffLim* field specifies the maximum number of bytes in the input buffer before the XOFF character is sent. This limit value is computed by taking the size of the input buffer and subtracting the value in *XoffLim*. In the sample Windows CE implementation of the serial driver provided in the ETK, the *XonLim* field is ignored and XON and XOFF characters are sent based on the value in *XoffLim*. However, this behavior might differ in some systems.

The next three fields, *ByteSize*, *Parity*, and *StopBits*, define the format of the serial data word transmitted. The *ByteSize* field specifies the number of bits per byte, usually a value of 7 or 8, but in some older modes the number of bits per byte can be as small as 5. The parity field can be set to the self-explanatory constants EVENPARITY, MARKPARITY, NOPARITY, ODDPARITY, or SPACEPARITY. The *StopBits* field should be set to ONESTOPBIT, ONE5STOPBITS, or TWOSTOPBITS depending on whether you want one, one and a half, or two stop bits per byte.

The next two fields, *XonChar* and *XoffChar*, let you specify the XON and XOFF characters. Likewise, the *EvtChar* field lets you specify the character used to signal an event. If an event character is received, an EV_RXFLAG event is signaled by the driver. This "event" is what triggers the *WaitCommEvent* function to return if the EV_RXFLAG bit is set in the event mask.

Setting the Port Timeout Values

As you can see, *SetCommState* can fine-tune, to almost the smallest detail, the operation of the serial driver. However, one more step is necessary—setting the timeout values for the port. The time out is the length of time Windows CE waits on a read or write operation before *ReadFile* or *WriteFile* automatically returns. The functions that control the serial time outs are the following:

```
BOOL GetCommTimeouts (HANDLE hFile, LPCOMMTIMEOUTS lpCommTimeouts);
```

and

```
BOOL SetCommTimeouts (HANDLE hFile, LPCOMMTIMEOUTS lpCommTimeouts);
```

Both functions take the handle to the open serial device and a pointer to a COMMTIMEOUTS structure, defined as the following:

```
typedef struct _COMMTIMEOUTS {
    DWORD ReadIntervalTimeout;
    DWORD ReadTotalTimeoutMultiplier;
    DWORD ReadTotalTimeoutConstant;
    DWORD WriteTotalTimeoutMultiplier;
    DWORD WriteTotalTimeoutConstant;
} COMMTIMEOUTS;
```

The COMMTIMEOUTS structure provides for a set of timeout parameters that time both the interval between characters and the total time to read and write a block of characters. Time outs are computed in two ways. First *ReadIntervalTimeout* specifies the maximum interval between characters received. If this time is exceeded, the *ReadFile* call returns immediately. The other time out is based on the number of characters you're waiting to receive. The value in *ReadTotalTimeoutMultiplier* is multiplied by the number of characters requested in the call to *ReadFile*, and is added to *ReadTotalTimeoutConstant* to compute a total time out for a call to *ReadFile*.

The write time out can be specified only for the total time spent during the *WriteFile* call. This time out is computed the same way as the total read time out, by specifying a multiplier value, the time in *WriteTotalTimeoutMultiplier*, and a constant value in *WriteTotalTimeoutConstant*. All of the times in this structure are specified in milliseconds.

In addition to the basic time outs that I just described, you can set values in the COMMTIMEOUTS structure to control whether and exactly how time outs are used in calls to *ReadFile* and *WriteFile*. You can configure the time outs in the following ways:

- Time outs for reading and writing as well as an interval time out. Set the fields in the COMMTIMEOUTS structure for the appropriate timeout values.
- Time outs for reading and writing with no interval time out. Set *ReadIntervalTimeout* to 0. Set the other fields for the appropriate timeout values.
- *ReadFile* returns immediately regardless of whether there is data to be read. Set *ReadIntervalTimeout* to MAXDWORD. Set *ReadTotalTimeoutMultiplier* and *ReadTotalTimeoutConstant* to 0.
- *ReadFile* doesn't have a time out. The function doesn't return until the proper number of bytes is returned or an error occurs. Set *ReadIntervalTimeout*, *ReadTotalTimeoutMultiplier*, and *ReadTotalTimeoutConstant* to 0.
- WriteFile doesn't have a time out. Set *WriteTotalTimeoutMultiplier* and *WriteTotalTimeoutConstant* to 0.

The timeout values are important because the worst thing you can do is to spin in a loop waiting on characters from the serial port. While the calls to *ReadFile* and *WriteFile* are waiting on the serial port, the calling threads are efficiently blocked on an event object internal to the driver. This saves precious CPU and battery power during the serial transmit and receive operations. Of course, to block on the *ReadFile* and *WriteFile*, you'll have to create secondary threads because you can't have your primary thread blocked waiting on the serial port.

Another call isn't quite as useful—*SetupComm*, prototyped this way:

```
BOOL SetupComm (HANDLE hFile, DWORD dwInQueue, DWORD dwOutQueue);
```

This function lets you specify the size of the input and output buffers for the driver. However, the sizes passed in *SetupComm* are only recommendations, not requirements to the serial driver. For example, the example implementation of the serial driver in the ETK ignores these recommended buffer sizes.

Querying the Capabilities of the Serial Driver

The configuration functions enable you to configure the serial driver, but with varied implementations of serial ports you need to know just what features a serial port supports before you configure it. The function *GetCommProperties* provides just this service. The function is prototyped this way:

```
BOOL GetCommProperties (HANDLE hFile, LPCOMMPROP lpCommProp);
```

GetCommProperties takes two parameters: the handle to the opened serial driver, and a pointer to a COMMPROP structure defined as

```
typedef struct _COMMPROP {
    WORD wPacketLength;
    WORD wPacketVersion;
    DWORD dwServiceMask;
    DWORD dwReserved1;
    DWORD dwMaxTxQueue;
    DWORD dwMaxRxQueue;
    DWORD dwMaxBaud;
    DWORD dwProvSubType;
    DWORD dwProvCapabilities;
    DWORD dwSettableParams;
    DWORD dwSettableBaud;
    WORD wSettableData;
    WORD wSettableStopParity;
    DWORD dwCurrentTxQueue;
    DWORD dwCurrentRxQueue;
    DWORD dwProvSpec1;
    DWORD dwProvSpec2;
    WCHAR wcProvChar[1];
} COMMPROP;
```

As you can see from the fields of the COMMPROP structure, *GetCommProperties* returns generally enough information to determine the capabilities of the device. Of immediate interest to speed demons is the *dwMaxBaud* field that indicates the maximum baud rate of the serial port. The *dwSettableBaud* field contains bit flags that indicate the allowable baud rates for the port. Both these fields use bit flags that are defined in WINBASE.H. These constants are expressed as BAUD_*xxxx*, as in BAUD_19200, which indicates the port is capable of a speed of 19.2 kbps. Note that these constants are *not* the constants used to set the speed of the serial port in the DCB structure. Those constants are numbers, not bit flags. To set the speed of a COM port in the DCB structure to 19.2 kbps, you would use the constant CBR_19200 in the *BaudRate* field of the DCB structure.

Starting back at the top of the structure are the *wPacketLength* and *wPacketVersion* fields. These fields allow you to request more information from the driver than is supported by the generic call. The *dwServiceMask* field indicates what services the port supports. The only service currently supported is SP_SERIALCOMM, indicating that the port is a serial communication port.

The *dwMaxTxQueue* and *dwMaxRxQueue* fields indicate the maximum size of the output and input buffers internal to the driver. A value of 0 in these fields indicates that you'll encounter no limit in the size of the internal queues. The *dwCurrentTxQueue* and *dwCurrentRxQueue* fields indicate the current size for the queues. These fields are 0 if the queue size can't be determined.

The *dwProvSubType* field contains flags that indicate the type of serial port supported by the driver. Values here include PST_RS232, PST_RS422, and PST_RS423, indicating the physical layer protocol of the port. PST_MODEM indicates a modem device, and PST_FAX tells you the port is a fax device. This field reports what the driver thinks the port is, not what device is attached to the port. For example, if an external modem is attached to a standard, RS-232 serial port, the driver returns the PST_RS232 flag, not the PST_MODEM flag.

The *dwProvCapabilities* field contains flags indicating the handshaking the port supports, such as XON/XOFF, RTS/CTS, and DTR/DSR. This field also shows you whether the port supports setting the characters used for XON/XOFF, parity checking, and so forth. The *dwSettableParams*, *dwSettableData*, and *dwSettableStopParity* fields give you information about how the serial data stream can be configured. Finally, the fields *dwProvSpec1*, *dwProvSpec2*, and *wcProvChar* are used by the driver to return driver-specific data.

Controlling the Serial Port

You can stop and start a serial stream using the following functions:

```
BOOL SetCommBreak (HANDLE hFile);
```

and

```
BOOL ClearCommBreak (HANDLE hFile);
```

The only parameter for both these functions is the handle to the opened COM port. When *SetCommBreak* is called, the COM port stops transmitting characters and places the port in a break state. Communication is resumed with the *ClearCommBreak* function.

You can clear out any characters in either the transmit or receive queues internal to the serial driver using this function:

```
BOOL PurgeComm (HANDLE hFile, DWORD dwFlags);
```

The *dwFlags* parameter can be a combination of the flags PURGE_TXCLEAR and PURGE_RXCLEAR. These flags terminate any pending writes and reads and reset the queues. In the case of PURGE_RXCLEAR, the driver also clears any receive holds due to any flow control states, transmitting an XON character if necessary, and setting RTS and DTR if those flow control methods are enabled. Since Windows CE doesn't support overlapped I/O, the flags PURGE_TXABORT and PURGE_RXABORT, used under Windows NT and Windows 98, are ignored.

The *EscapeCommFunction* provides a more general method of controlling the serial driver. It allows you to set and clear the state of specific signals on the port. On Windows CE devices, it's also used to control serial hardware that's shared between the serial port and the IrDA port. (I'll talk more about infrared data transmission and the Infrared Data Association (IrDA) standard later in this chapter.) The function is prototyped as

```
BOOL EscapeCommFunction (HANDLE hFile, DWORD dwFunc);
```

The function takes two parameters, the handle to the device and a set of flags in *dwFunc*. The flags can be one of the following values:

- *SETDTR* Sets the DTR signal.
- *CLRDTR* Clears the DTR signal.
- *SETRTS* Sets the RTS signal
- *CLRRTS* Clears the RTS) ignal.
- *SETXOFF* Tells the driver to act as if an XOFF character has been received.
- *SETXON* Tells the driver to act as if an XON character has been received.
- *SETBREAK* Suspends serial transmission and sets the port in a break state.
- *CLRBREAK* Resumes serial transmission from a break state.
- *SETIR* Tells the serial port to transmit and receive through the infrared transceiver.
- *CLRIR* Tells the serial port to transmit and receive through the standard serial transceiver.

The SETBREAK and CLRBREAK commands act identically to *SetCommBreak* and *ClearCommBreak* and can be used interchangeably. For example, you can use *EscapeCommFunction* to put the port in a break state and *ClearCommBreak* to restore communication.

Clearing Errors and Querying Status

The function

```
BOOL ClearCommError (HANDLE hFile, LPDWORD lpErrors, LPCOMSTAT lpStat);
```

performs two functions. As you might expect from the name, it clears any error states within the driver so that I/O can continue. The serial device driver is responsible for reporting the errors. The default serial driver returns the following flags in the variable pointed to by *lpErrors*: CE_OVERRUN, CE_RXPARITY, CE_FRAME, and CE_TXFULL. *ClearCommError* also returns the status of the port. The third parameter of *ClearCommError* is a pointer to a COMSTAT structure defined as

```
typedef struct _COMSTAT {
    DWORD fCtsHold : 1;
    DWORD fDsrHold : 1;
    DWORD fRlsdHold : 1;
    DWORD fXoffHold : 1;
    DWORD fXoffSent : 1;
    DWORD fEof : 1;
    DWORD fTxim : 1;
    DWORD fReserved : 25;
    DWORD cbInQue;
    DWORD cbOutQue;
} COMSTAT;
```

The first five fields indicate that serial transmission is waiting for one of the following reasons. It's waiting for a CTS signal, waiting for a DSR signal, waiting for a Receive Line Signal Detect (also known as a Carrier Detect), waiting for an XON character, or it's waiting because an XOFF character was sent by the driver. The *fEor* field indicates that an end-of-file character has been received. The *fTxim* field is TRUE if a character placed in the queue by the *TransmitCommChar* function instead of a call to *WriteFile* is queued for transmission. The final two fields, *cbInQue* and *cbOutQue*, return the number of characters in the input and output queues of the serial driver.

The function

```
BOOL GetCommModemStatus (HANDLE hFile, LPDWORD lpModemStat);
```

returns the status of the modem control signals in the variable pointed to by *lpModemStat*. The flags returned can be any of the following:

- *MS_CTS_ON* Clear to Send (CTS) is active.
- *MS_DSR_ON* Data Set Ready (DSR) is active.
- *MS_RING_ON* Ring Indicate (RI) is active.
- *MS_RLSD_ON* Receive Line Signal Detect (RLSD) is active.

Stay'n Alive

One of the issues with serial communication is preventing the system from powering down while a serial link is active. A Windows CE system determines activity by the number of key presses and screen taps. It doesn't take into account such tasks as a serial port transmitting data. To prevent a Windows CE device from powering off, you can simulate a keystroke using either of the following functions:

```
VOID keybd_event (BYTE bVk, BYTE bScan, DWORD dwFlags,
                  DWORD dwExtraInfo);
```

or

```
UINT SendInput (UINT nInputs, LPINPUT pInputs, int cbSize);
```

These functions can be used to simulate a keystroke that resets the activity timer used by Windows CE to determine when the system should automatically power down. Windows CE supports an additional constant for both these functions—KEYEVENTF_SILENT, which prevents the default keyboard click sound from being played.

THE INFRARED PORT

Windows CE devices almost always have an infrared, IrDA-compatible serial port. In fact, all H/PC and Palm-size PC systems are guaranteed to have one. The IR ports on Windows CE devices are IrDA (Infrared Data Association) compliant. The IrDA standard specifies everything from the physical implementation, such as the frequency of light used, to the handshaking between devices and how remote systems find each other and converse.

The IR port can be used in a variety of ways. At the most basic level, the port can be accessed as a serial port with an IR transmitter and receiver attached. This method is known as *raw IR*. When you're using raw IR, the port isn't IrDA compliant because the IrDA standard requires the proper handshaking for the link. However, raw IR gives you the most control over the IR link. A word of warning: While all Windows CE devices I know currently support raw IR, some might not in the future.

You can also use the IR port in IrComm mode. In this mode, the IR link looks like a serial port. However, under the covers, Windows CE works to hide the differences between a standard serial port and the IR link. This is perhaps the easiest way to link two custom applications because the applications can use the rather simple Comm API while Windows CE uses the IrDA stack to handle the IR link.

The most robust and complex method of using the IR port is to use IrSock. In this mode, the IR link appears to be just another socket. IrSock is an extension to WinSock, the Windows version of the socket interface used by applications communicating with TCP/IP. I'll cover WinSock in Chapter 10, so I'll defer any talk of IrSock until then.

Raw IR

As I mentioned previously, when you use raw IR you're mainly on your own. You essentially have a serial port with an IR transceiver attached to it. Since both the transmitter and receiver use the same ether (the air), collisions occur if you transmit at the same time that you're receiving a stream of data from another device. This doesn't happen when a serial cable connects two serial ports because the cable gives you separate transmit and receive wires that can be used at the same time.

Finding the raw IR port

To use raw IR, you must first find the serial port attached to the IR transceiver. On some Windows CE units, the serial port and the IR port use the same serial hardware. This means you can't use the serial port at the same time you use the IR port. Other Windows CE devices have separate serial hardware for the IR port. Regardless of how a device is configured, Windows CE gives you a separate instance of a COM driver for the IR port that's used for raw IR mode.

There is no official method of determining the COM port used for raw IR. However, the following technique works for current devices. To find the COM port used for raw IR, look in the registry in the \Comm\IrDA key under HKEY_LOCAL_MACHINE. There, you should find the Port key that contains the COM port number for the raw IR device. Below is a short routine that returns the device name of the raw IR port.

```
//----------------------------------------------------------------------
// GetRawIrDeviceName - Returns the device name for the RawIR com port
//
INT GetRawIrDeviceName (LPTSTR pDevName) {
    DWORD dwSize, dwType, dwData;
    HKEY hKey;
    INT rc;

    *pDevName = TEXT ('\0');
    // Open the IrDA key.
    if (RegOpenKeyEx (HKEY_LOCAL_MACHINE, TEXT ("Comm\\IrDA"), 0,
                      0, &hKey) == ERROR_SUCCESS) {

        // Query the device number.
        dwSize = sizeof (dwData);
        if (RegQueryValueEx (hKey, TEXT ("Port"), 0, &dwType,
                             (PBYTE)&dwData, &dwSize) == ERROR_SUCCESS)

            // Check for valid port number. Assume buffer > 5 chars.
            if (dwData < 10)
                wsprintf (pDevName, TEXT ("COM%d:"), dwData);

        RegCloseKey (hKey);
    }
    return lstrlen (pDevName);
}
```

Using raw IR

Once you have the port name, you must perform one more task before you can use the port. If the COM port hardware is being shared by the serial port and the IR port, you must tell the driver to direct the serial stream through the IR transceiver. You do this by first opening the device and calling *EscapeCommFunction*. The command passed to the device is SETIR. When you've finished using the IR port, you should call *EscapeCommFunction* again with the command CLRIR to return the port back to its original serial function.

Once the port is set up, there's one main difference between raw IR and standard serial communication. You have to be careful when using raw IR not to transmit while another device is also transmitting. The two transmissions will collide, corrupting both data streams. With raw IR, you're also responsible for detecting the other device and handling the dropped bytes that will occur as the infrared beam between the two devices is occasionally broken.

IrComm

Using IrComm is much easier than using raw IR. IrComm takes care of remote device detection, collision detection, and data buffering while communication with the other device is temporally interrupted. The disadvantage of IrComm is that it's a point-to-point protocol—only two devices can be connected. In most instances, however, this is sufficient.

Finding the IrComm port

Here again, there's no official method for determining the IrComm port. But you should be able to find the IrComm port by looking in the registry under the Drivers\builtin \IrCOMM key under HKEY_LOCAL_MACHINE. The item to query is the *Index* value, which is the COM device number for the IrComm port. Following is a routine that returns the device name of the IrComm port.

```
//----------------------------------------------------------------------
// GetIrCommDeviceName - Returns the device name for the IrComm port
//
INT GetIrCommDeviceName (LPTSTR pDevName) {
    DWORD dwSize, dwType, dwData;
    HKEY hKey;

    *pDevName = TEXT ('\0');
    // Open the IrDA key.
    if (RegOpenKeyEx (HKEY_LOCAL_MACHINE,
                      TEXT ("Drivers\\BuiltIn\\IrCOMM"), 0,
                      0, &hKey) == ERROR_SUCCESS) {
```

(continued)

```
        // Query the device number.
        dwSize = sizeof (dwData);
        if (RegQueryValueEx (hKey, TEXT ("Index"), 0, &dwType,
                             (PBYTE)&dwData, &dwSize) == ERROR_SUCCESS)

            // Check for valid port number. Assume buffer > 5 chars.
            if (dwData < 10)
                wsprintf (pDevName, TEXT ("COM%d:"), dwData);

        RegCloseKey (hKey);
    }
    return lstrlen (pDevName);
}
```

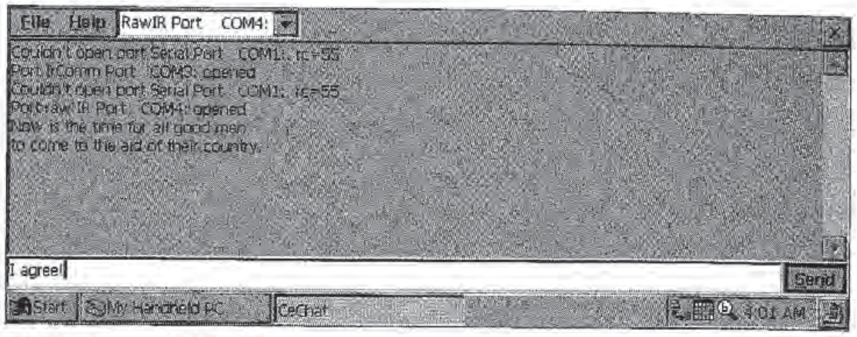
The IrComm port is different in a number of ways from the serial port and the raw IR port. These differences arise from the fact that the IrComm port is a simulated port, not a real device. The IrComm driver uses IrSock to manage the IR link. The driver is then responsible only for reflecting the data stream and a few control characters to simulate the serial connection. If you try to query the communication settings for the IrComm port using *GetCommState*, the DCB returned is all zeros. If you try to set a baud rate or some of the other parameters, and later call *GetCommState* again, the DCB will still be 0. IrSock manages the speed and the handshaking protocol, so IrComm simply ignores your configuration requests.

On the other hand, the IrComm driver happily queues up pending writes waiting on another IrComm device to come within range. After the IrComm driver automatically establishes a link, it transmits the pending bytes to the other device. This assistance is a far cry from raw IR and is what makes using IrComm so easy.

The best way to learn about the characteristics of the two methods of IR communication I've described is to use them. Which brings us to this chapter's example program.

THE CECHAT EXAMPLE PROGRAM

The CeChat program is a simple point-to-point chat program that connects two Windows CE devices using one of the three methods of serial communication covered in this chapter. The CeChat window is shown in Figure 9-3. Most of the window is taken up by the receive text window. Text received from the other device is displayed here. Along the bottom of the screen is the send text window. If you type characters here and either hit the Enter key or tap on the Send button, the text is sent to the other device. The combo box on the command bar selects the serial medium to use: standard serial, raw IR, or IrComm.



```
Couldn't open port Serial Port  COM1:, rc=55
Port IrComm Port  COM3: opened
Couldn't open port Serial Port  COM1:, rc=55
Port raw IR Port  COM4: opened
Now is the time for all good men
to come to the aid of their country.
```

```
I agree!
```

Figure 9-3. *The CeChat window.*

The source code for CeChat is shown in Figure 9-4. CeChat uses three threads to accomplish its work. The primary thread manages the window and the message loop. The two secondary threads handle reading from and writing to the appropriate serial port.

CeChat.rc

```
//======================================================================
// Resource file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include "windows.h"
#include "CeChat.h"                          // Program-specific stuff
//----------------------------------------------------------------------
// Icons and bitmaps
//
ID_ICON ICON   "CeChat.ico"                  // Program icon

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

Figure 9-4. *The CeChat source code.* *(continued)*

Figure 9-4. *continued*

```
//----------------------------------------------------------------------
// Accelerator table
//
ID_ACCEL ACCELERATORS DISCARDABLE
BEGIN
    "S", ID_SENDBTN, VIRTKEY, ALT
    VK_RETURN, ID_SENDBTN, VIRTKEY
END

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

CeChat.h

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

//----------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                             // Structure associates
    UINT Code;                                  // messages
                                                // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                              // Structure associates
    UINT Code;                                  // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);     // function.
};
//----------------------------------------------------------------------
// Generic defines used by application
#define  ID_ICON             1                  // App icon resource ID
#define  ID_MENU             2                  // Menu resource ID
#define  ID_ACCEL            3                  // Accel table ID
#define  IDC_CMDBAR          4                  // Command band ID
#define  ID_RCVTEXT          5                  // Receive text box
#define  ID_SENDTEXT         6                  // Send text box
#define  ID_SENDBTN          7                  // Send button

// Menu item IDs
#define  IDM_EXIT            101

#define  IDM_USECOM          110                // Use COM.
#define  IDM_ABOUT           120                // Help menu

// Command bar IDs
#define  IDC_COMPORT         150                // COM port combo box
#define  IDC_BAUDRATE        151                // Baud rate combo box

#define TEXTSIZE 256
//----------------------------------------------------------------------
// Function prototypes
//
int ReadThread (PVOID pArg);
int SendThread (PVOID pArg);
HANDLE InitCommunication (HWND, LPTSTR);
INT GetIrCommDeviceName (LPTSTR);
INT GetRawIrDeviceName (LPTSTR);
int FillComComboBox (HWND);

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

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

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

(continued)

Figure 9-4. *continued*

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

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

CeChat.c

```
//======================================================================
// CeChat - A Windows CE communication demo
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                 // For all that Windows stuff
#include <commctrl.h>                // Command bar includes
#include "CeChat.h"                  // Program-specific stuff

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

BOOL fContinue = TRUE;
HANDLE hComPort = INVALID_HANDLE_VALUE;
INT nSpeed = CBR_19200;
int nLastDev = -1;

HANDLE g_hSendEvent = INVALID_HANDLE_VALUE;
HANDLE hReadThread = INVALID_HANDLE_VALUE;

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

```
const struct decodeCMD MainCommandItems[] = {
    IDC_COMPORT, DoMainCommandComPort,
    ID_SENDBTN, DoMainCommandSendText,
    IDM_EXIT, DoMainCommandExit,
    IDM_ABOUT, DoMainCommandAbout,
};
//======================================================================
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    HWND hwndMain;
    HACCEL hAccel;
    MSG msg;
    int rc = 0;

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

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

    // Load accelerator table.
    hAccel = LoadAccelerators (hInst, MAKEINTRESOURCE (ID_ACCEL));

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

    // Register application main window class.
```

(continued)

Figure 9-4. *continued*

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

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

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

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

    // Create unnamed auto-reset event initially false.
    g_hSendEvent = CreateEvent (NULL, FALSE, FALSE, NULL);

    // Create main window.
    hWnd = CreateWindow (szAppName, TEXT ("CeChat"),
                         WS_VISIBLE, CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT, NULL,
                         NULL, hInstance, NULL);
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Create write thread. Read thread created when port opened.
    hThread = CreateThread (NULL, 0, SendThread, hWnd, 0, &rc);
    if (hThread)
        CloseHandle (hThread);
    else {
        DestroyWindow (hWnd);
        return 0;
    }
    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//----------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {
    HANDLE hPort = hComPort;

    fContinue = FALSE;

    hComPort = INVALID_HANDLE_VALUE;
    if (hPort != INVALID_HANDLE_VALUE)
        CloseHandle (hPort);

    if (g_hSendEvent != INVALID_HANDLE_VALUE) {
        PulseEvent (g_hSendEvent);
        Sleep(100);
        CloseHandle (g_hSendEvent);
    }
    return nDefRC;
}
//======================================================================
// Message handling procedures for MainWindow
//----------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
```

(continued)

Figure 9-4. *continued*

```
// DoCreateMain - Process WM_CREATE message for window.
//
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB, hC1, hC2, hC3;
    INT  i, j, nHeight;
    TCHAR szFirstDev[32];
    LPCREATESTRUCT lpcs;

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

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);
    CommandBar_InsertMenubar (hwndCB, hInst, ID_MENU, 0);

    // Insert the com port combo box.
    CommandBar_InsertComboBox (hwndCB, hInst, 140, CBS_DROPDOWNLIST,
                               IDC_COMPORT, 1);
    FillComComboBox (hWnd);

    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    nHeight = CommandBar_Height (hwndCB);

    // Create receive text window.
    hC1 = CreateWindowEx (WS_EX_CLIENTEDGE, TEXT ("edit"),
                          TEXT (""), WS_VISIBLE | WS_CHILD |
                          WS_VSCROLL | ES_MULTILINE | ES_AUTOHSCROLL |
                          ES_READONLY, 0, nHeight, lpcs->cx,
                          lpcs->cy - nHeight - 25, hWnd,
                          (HMENU)ID_RCVTEXT, hInst, NULL);
    // Create send text window.
    hC2 = CreateWindowEx (WS_EX_CLIENTEDGE, TEXT ("edit"),
                          TEXT (""), WS_VISIBLE | WS_CHILD,
                          0, lpcs->cy - 25, lpcs->cx-50, 25,
                          hWnd, (HMENU)ID_SENDTEXT, hInst, NULL);
    // Create send text window.
    hC3 = CreateWindowEx (WS_EX_CLIENTEDGE, TEXT ("button"),
                          TEXT ("&Send"), WS_VISIBLE | WS_CHILD |
                          BS_DEFPUSHBUTTON,
                          lpcs->cx-50, lpcs->cy - 25, 50, 25,
                          hWnd, (HMENU)ID_SENDBTN, hInst, NULL);
    // Destroy frame if window not created.
    if (!IsWindow (hC1) || !IsWindow (hC2) || !IsWindow (hC3)) {
        DestroyWindow (hWnd);
```

```
            return 0;
        }
        // Open a com port.
        for (i = 0; i < 3; i++) {
            SendDlgItemMessage (hwndCB, IDC_COMPORT, CB_GETLBTEXT, i,
                                (LPARAM)szFirstDev);
            j = lstrlen (szFirstDev);
            // Really bad hack to determine which is the RAW IR port
            if (InitCommunication (hWnd, szFirstDev) !=
                INVALID_HANDLE_VALUE) {
                SendDlgItemMessage (hwndCB, IDC_COMPORT, CB_SETCURSEL, i,
                                    (LPARAM)szFirstDev);
                break;
            }
        }
        return 0;
}
//----------------------------------------------------------------------
// DoSizeMain - Process WM_SIZE message for window.
//
LRESULT DoSizeMain (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam){
    RECT rect;

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

    SetWindowPos (GetDlgItem (hWnd, ID_RCVTEXT), NULL, rect.left,
                  rect.top, (rect.right - rect.left),
                  rect.bottom - rect.top - 25, SWP_NOZORDER);
    SetWindowPos (GetDlgItem (hWnd, ID_SENDTEXT), NULL, rect.left,
                  rect.bottom - 25, (rect.right - rect.left) - 50,
                  25, SWP_NOZORDER);
    SetWindowPos (GetDlgItem (hWnd, ID_SENDBTN), NULL,
                  (rect.right - rect.left) - 50, rect.bottom - 25,
                  50, 25, SWP_NOZORDER);
    return 0;
}
//----------------------------------------------------------------------
// DoFocusMain - Process WM_SETFOCUS message for window.
//
LRESULT DoSetFocusMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    SetFocus (GetDlgItem (hWnd, ID_SENDTEXT));
    return 0;
```

(continued)

Figure 9-4. *continued*

```
}
//----------------------------------------------------------------------
// DoCommandMain - Process WM_COMMAND message for window.
//
LRESULT DoCommandMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    WORD    idItem, wNotifyCode;
    HWND hwndCtl;
    INT  i;

    // Parse the parameters.
    idItem = (WORD) LOWORD (wParam);
    wNotifyCode = (WORD) HIWORD (wParam);
    hwndCtl = (HWND) lParam;

    // Call routine to handle control message.
    for (i = 0; i < dim(MainCommandItems); i++) {
        if (idItem == MainCommandItems[i].Code)
            return (*MainCommandItems[i].Fxn)(hWnd, idItem, hwndCtl,
                                              wNotifyCode);
    }
    return 0;
}
//----------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
//======================================================================
// Command handler routines
//----------------------------------------------------------------------
// DoMainCommandExit - Process Program Exit command.
//
LPARAM DoMainCommandExit (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {
    SendMessage (hWnd, WM_CLOSE, 0, 0);
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandComPort - Process the COM port combo box commands.
//
LPARAM DoMainCommandComPort (HWND hWnd, WORD idItem, HWND hwndCtl,
                             WORD wNotifyCode) {
```

```
    INT i;
    TCHAR szDev[32];

    if (wNotifyCode == CBN_SELCHANGE) {
        i = SendMessage (hwndCtl, CB_GETCURSEL, 0, 0);
        if (i != nLastDev) {
            nLastDev = i;
            SendMessage (hwndCtl, CB_GETLBTEXT, i, (LPARAM)szDev);
            InitCommunication (hWnd, szDev);
            SetFocus (GetDlgItem (hWnd, ID_SENDTEXT));
        }
    }
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandSendText - Process the Send text button.
//
LPARAM DoMainCommandSendText (HWND hWnd, WORD idItem, HWND hwndCtl,
                              WORD wNotifyCode) {

    // Set event so that sender thread will send the text.
    SetEvent (g_hSendEvent);
    SetFocus (GetDlgItem (hWnd, ID_SENDTEXT));
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandAbout - Process the Help | About menu command.
//
LPARAM DoMainCommandAbout(HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {
    // Use DialogBox to create modal dialog.
    DialogBox (hInst, TEXT ("aboutbox"), hWnd, AboutDlgProc);
    return 0;
}
//======================================================================
// About Dialog procedure
//
BOOL CALLBACK AboutDlgProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                            LPARAM lParam) {
    switch (wMsg) {
        case WM_COMMAND:
            switch (LOWORD (wParam)) {
                case IDOK:
                case IDCANCEL:
                    EndDialog (hWnd, 0);
                    return TRUE;
```

(continued)

Figure 9-4. *continued*

```
            }
            break;
        }
        return FALSE;
}
//----------------------------------------------------------------------
// GetRawIrDeviceName - Returns the device name for the RawIR com port
//
INT GetRawIrDeviceName (LPTSTR pDevName) {
    DWORD dwSize, dwType, dwData;
    HKEY hKey;

    *pDevName = TEXT ('\0');
    // Open the IrDA key.
    if (RegOpenKeyEx (HKEY_LOCAL_MACHINE, TEXT ("Comm\\IrDA"), 0,
                      0, &hKey) == ERROR_SUCCESS) {

        // Query the device number.
        dwSize = sizeof (dwData);
        if (RegQueryValueEx (hKey, TEXT ("Port"), 0, &dwType,
                             (PBYTE)&dwData, &dwSize) == ERROR_SUCCESS)

            // Check for valid port number. Assume buffer > 5 chars.
            if (dwData < 10)
                wsprintf (pDevName, TEXT ("COM%d:"), dwData);

        RegCloseKey (hKey);
    }
    return lstrlen (pDevName);
}
//----------------------------------------------------------------------
// GetIrCommDeviceName - Returns the device name for the IrComm port
//
INT GetIrCommDeviceName (LPTSTR pDevName) {
    DWORD dwSize, dwType, dwData;
    HKEY hKey;

    *pDevName = TEXT ('\0');
    // Open the IrDA key.
    if (RegOpenKeyEx (HKEY_LOCAL_MACHINE,
                      TEXT ("Drivers\\BuiltIn\\IrCOMM"), 0,
                      0, &hKey) == ERROR_SUCCESS) {

        // Query the device number.
        dwSize = sizeof (dwData);
        if (RegQueryValueEx (hKey, TEXT ("Index"), 0, &dwType,
                              (PBYTE)&dwData, &dwSize) == ERROR_SUCCESS)

            // Check for valid port number. Assume buffer > 5 chars.
            if (dwData < 10)
                wsprintf (pDevName, TEXT ("COM%d:"), dwData);

        RegCloseKey (hKey);
    }
    return lstrlen (pDevName);
}
//----------------------------------------------------------------------
// FillComComboBox - Fills the com port combo box
//
int FillComComboBox (HWND hWnd) {
    TCHAR szDev[64];

    lstrcpy (szDev, TEXT ("Serial Port   COM1:"));
    SendDlgItemMessage (GetDlgItem (hWnd, IDC_CMDBAR),
                        IDC_COMPORT, CB_INSERTSTRING,
                        -1, (LPARAM)szDev);

    lstrcpy (szDev, TEXT ("IrComm Port   "));
    GetIrCommDeviceName (&szDev[lstrlen (szDev)]);
    SendDlgItemMessage (GetDlgItem (hWnd, IDC_CMDBAR),
                        IDC_COMPORT, CB_INSERTSTRING,
                        -1, (LPARAM)szDev);

    lstrcpy (szDev, TEXT ("Raw IR Port   "));
    GetRawIrDeviceName (&szDev[lstrlen (szDev)]);
    SendDlgItemMessage (GetDlgItem (hWnd, IDC_CMDBAR),
                        IDC_COMPORT, CB_INSERTSTRING,
                        -1, (LPARAM)szDev);
    SendDlgItemMessage (GetDlgItem (hWnd, IDC_CMDBAR), IDC_COMPORT,
                        CB_SETCURSEL, 0, 0);
    return 0;
}
//----------------------------------------------------------------------
// InitCommunication - Open and initialize selected COM port.
//
HANDLE InitCommunication (HWND hWnd, LPTSTR pszDevName) {
    DCB dcb;
    INT i;
    TCHAR szDbg[128];
    COMMTIMEOUTS cto;
    HANDLE hLocal;
    DWORD dwTStat;
```

(continued)

Figure 9-4. *continued*

```
    hLocal = hComPort;
    hComPort = INVALID_HANDLE_VALUE;

    if (hLocal != INVALID_HANDLE_VALUE)
        CloseHandle (hLocal);  // This causes WaitCommEvent to return.

    // The com port name is the last 5 characters of the string.
    i = lstrlen (pszDevName);
    hLocal = CreateFile (&pszDevName[i-5], GENERIC_READ | GENERIC_WRITE,
                         0, NULL, OPEN_EXISTING, 0, NULL);

    if (hLocal != INVALID_HANDLE_VALUE) {
        // Configure port.
        GetCommState (hLocal, &dcb);
        dcb.BaudRate = nSpeed;
        dcb.fParity = FALSE;
        dcb.fNull = FALSE;
        dcb.StopBits = ONESTOPBIT;
        dcb.Parity = NOPARITY;
        dcb.ByteSize = 8;
        SetCommState (hLocal, &dcb);

        // Set the timeouts.  Set infinite read timeout.
        cto.ReadIntervalTimeout = 0;
        cto.ReadTotalTimeoutMultiplier = 0;
        cto.ReadTotalTimeoutConstant = 0;
        cto.WriteTotalTimeoutMultiplier = 0;
        cto.WriteTotalTimeoutConstant = 0;
        SetCommTimeouts (hLocal, &cto);

        wsprintf (szDbg, TEXT ("Port %s opened\r\n"), pszDevName);
        SendDlgItemMessage (hWnd, ID_RCVTEXT, EM_REPLACESEL, 0,
                            (LPARAM)szDbg);
        // Really bad hack to determine which is the raw IR selection.
        // We need to enable IR on the raw IR port in case port is
        // shared with the standard serial port.
        if (*pszDevName == TEXT ('R')) {
            if (!EscapeCommFunction (hLocal, SETIR)) {
                wsprintf (szDbg, TEXT ("Set IR failed. rc %d\r\n"),
                          GetLastError());
                SendDlgItemMessage (hWnd, ID_RCVTEXT, EM_REPLACESEL,
                                    0, (LPARAM)szDbg);
            }
        }
        // Start read thread if not already started.
        hComPort = hLocal;
        if (!GetExitCodeThread (hReadThread, &dwTStat) ||
            (dwTStat != STILL_ACTIVE)) {
            hReadThread = CreateThread (NULL, 0, ReadThread, hWnd,
                                        0, &dwTStat);
            if (hReadThread)
                CloseHandle (hReadThread);
        }
    } else {
        wsprintf (szDbg, TEXT ("Couldn\'t open port %s. rc=%d\r\n"),
                  pszDevName, GetLastError());
        SendDlgItemMessage (hWnd, ID_RCVTEXT, EM_REPLACESEL,
                            0, (LPARAM)szDbg);
    }
    return hComPort;
}
//======================================================================
// SendThread - Sends characters to the serial port
//
int SendThread (PVOID pArg) {
    HWND hWnd, hwndSText;
    INT cBytes, nGoCode;
    TCHAR szText[TEXTSIZE];

    hWnd = (HWND)pArg;
    hwndSText = GetDlgItem (hWnd, ID_SENDTEXT);
    while (1) {
        nGoCode = WaitForSingleObject (g_hSendEvent, INFINITE);
        if (nGoCode == WAIT_OBJECT_0) {
            if (!fContinue)
                break;
            GetWindowText (hwndSText, szText, dim(szText));
            lstrcat (szText, TEXT ("\r\n"));
            WriteFile (hComPort, szText, lstrlen (szText)*sizeof (TCHAR),
                       &cBytes, 0);
            SetWindowText (hwndSText, TEXT (""));  // Clear out text box
        } else
            break;
    }
    return 0;
}
//======================================================================
// ReadThread - Receives characters from the serial port
//
int ReadThread (PVOID pArg) {
    HWND hWnd;
    INT cBytes, i;
```

(continued)

Figure 9-4. *continued*

```
    BYTE szText[TEXTSIZE], *pPtr;
    TCHAR tch;

    hWnd = (HWND)pArg;
    while (fContinue) {
        tch = 0;
        pPtr = szText;
        for (i = 0; i < sizeof (szText)-sizeof (TCHAR); i++) {

            while (!ReadFile (hComPort, pPtr, 1, &cBytes, 0))
                if (hComPort == INVALID_HANDLE_VALUE)
                    return 0;

            // This syncs the proper byte order for Unicode.
            tch = (tch << 8) & 0xff00;
            tch |= *pPtr++;
            if (tch == TEXT ('\n'))
                break;
        }
        *pPtr++ = 0;  // Avoid alignment probs by addressing as bytes.
        *pPtr++ = 0;

        // If out of byte sync, move bytes down one.
        if (i % 2) {
            pPtr = szText;
            while (*pPtr || *(pPtr+1)) {
                *pPtr = *(pPtr+1);
                pPtr++;
            }
            *pPtr = 0;
        }
        SendDlgItemMessage (hWnd, ID_RCVTEXT, EM_REPLACESEL, 0,
                            (LPARAM)szText);
    }
    return 0;
}
```

When the CeChat window is created, it sniffs out the three port names using the methods I described earlier in the chapter. The combo box is then filled and an attempt is made to open one of the COM ports. Once a port is opened, the read thread is created to wait on characters. The read thread isn't as simple as it should be because it must deal with 2-byte Unicode characters. Because it's quite possible to drop a byte or two in a serial IR link, the receive thread must attempt to resync the proper high bytes with their low byte pair to form a correct Unicode character.

The send thread is actually quite simple. All it does is block on an event that was created when CeChat was started. When the event is signaled, it reads the text from the send text edit control and calls *WriteFile*. Once that has completed, the send thread clears the text from the edit control and loops back to where it blocks again.

In the CeChat window shown in Figure 9-3 on page 561, the program reports that it can't open COM1; this is because COM1 was being used by PC Link to connect to my PC. One of the problems with debugging serial programs on the H/PC or Palm-size PC is that you're generally using the one port that attaches to the PC. In these situations, it helps to have a secondary communication path from the PC to the Windows CE device. While you could put an additional serial PCMCIA Card into the H/PC to add ports, a faster link can be made with a PCMCIA Ethernet Card. Which brings us right to the next chapter, "Windows Networking and IrSock."

Chapter 10

Windows Networking and IrSock

Networks are at the heart of modern computer systems. Over the years, Microsoft Windows has supported a variety of networks and networking APIs. The evolving nature of networking APIs along with the need to keep systems backward compatible has resulted in a huge array of overlapping functions and parallel APIs. As in many places in Windows CE, the networking API is a subset of the vast array of networking functions supported under Windows NT and Windows 98.

Windows CE supports a variety of networking APIs. This chapter covers two. First is the Windows Networking API, WNet. This API supports basic network connections so that a Windows CE device can access disks and printers on a network.

Windows CE also supports a subset of the WinSock 1.1 API. I'm not going to cover the complete WinSock API because plenty of other books do that. I'll spend some time covering what is directly relevant to Windows CE developers. Of particular interest is the fact that that WinSock is the high-level API to the IrDA infrared communication stack. I'll also cover another extension to WinSock, the Internet control message protocol (ICMP) functions that allow Windows CE applications to ping other machines on a TCP/IP network.

WINDOWS NETWORKING SUPPORT

The WNet API is a provider-independent interface that allows Windows applications to access network resources without regard for the network implementation. The Windows CE version of the WNet API has fewer functions but provides the basics so that a Windows CE application can gain access to shared network resources, such as disks and printers. The WNet API is implemented by a "redirector" DLL that translates the WNet functions into network commands for a specific network protocol.

By default, the only network supported by the WNet API is Windows Networking. Support for even this network is limited by the fact that redirector files that implement Windows Networking aren't bundled with most H/PCs or Palm-size PCs. The two files that implement this support, REDIR.DLL and NETBIOS.DLL, are available from Microsoft. As a convenience, I've also included them on the book's companion disc as well. As an aside, the NetBIOS DLL doesn't export a NetBIOS-like interface to applications or drivers.

WNet Functions

Windows CE's support for the WNet functions started with Windows CE 2.0. As with other areas in Windows CE, the WNet implementation under Windows CE is a subset of the same API on the desktop, but support is provided for the critical functions while eliminating the overlapping and obsolete functions. For example, the standard WNet API contains four different and overlapping *WNetAddConnection* functions while Windows CE supports only one, *WNetAddConnection3*.

For the WNet API to work, the redirector DLLs must be installed in the \windows directory. In addition, the network control panel, also a supplementary component on most systems, must be used to configure the network card so that it can access the network. If the redirector DLLs aren't installed, or an error occurs configuring or initializing the network adapter, the WNet functions return the error code ERROR_NO_NETWORK.

Conventions of UNC

Network drives can be accessed in one of two ways. The first method is to explicitly name the resource using the *Universal Naming Convention* (UNC) naming syntax, which is a combination of the name of the server and the shared resource. An example of this is *\\BIGSRVR\DRVC*, where the server name is BIGSERV and the resource on the server is named DRVC. The leading double backslashes immediately indicate that the name is a UNC name. Directories and filenames can be included in the UNC name, as in *\\bigservr\drvc\dir2\file1.ext*. Notice that I changed case in the two names. That doesn't matter because UNC paths are case insensitive.

As long as the WNet redirector is installed, you can use UNC names wherever you use standard filenames in the Windows CE API. You'll have problems, though, with some programs, including, in places, the Windows CE shell, where the application doesn't understand UNC syntax. For example, the Explorer in a Windows CE 2.0 H/PC device understands UNC names, but the File Open dialog box on the same system doesn't.

Mapping a remote drive

To get around applications that don't understand UNC names, you can map a network drive to a local name. When a network drive is mapped on a Windows CE system, the remote drive appears as a folder in the \network folder in the object store. The \network folder isn't a standard folder; in fact, before Windows CE 2.1, it didn't even show up in the Explorer. (For systems based on Windows CE 2.1, the visibility of the \network folder depends on a registry setting.) Instead it's a placeholder name by which the local names of the mapped network drives can be addressed. For example, the network drive *\\BigSrvr\DrvC* could be mapped to the local name JoeBob. Files and directories on *\\BigSrvr\DrvC* would appear under the folder \network\joebob. Since Windows CE doesn't support drive letters, the local name can't be specified in the form of a drive, as in G:.

I mentioned that the \network folder is a virtual folder; this needs further explanation. Before Windows CE 2.1, the network folder wasn't visible to the standard file system functions. If you use the *FindFirstFile/FindNextFile* process to enumerate the directories in the root directory, the \network directory won't be enumerated. However, *FindFirstFile/FindNextFile* enumerates the mapped resources contained in the \network folder. So if the search string is *.* to enumerate the root directory, the network isn't enumerated, but if you use *\network*.** as the search string, any mapped drives will be enumerated.

Starting with Windows CE 2.1, the \network folder can be enumerated by *FindFirstFile* and *FindNextFile* if the proper registry settings are made. However, even though the folder can be enumerated, you still can't place files or create folders within the \network folder. To make the \network folder visible, the DWORD value *RegisterFSRoot* under the key [HKEY_LOCAL_MACHINE]\comm\redir, must be set to a nonzero value.

The most direct way to map a remote resource is to call this function:

```
DWORD WNetAddConnection3 (HWND hwndOwner, LPNETRESOURCE lpNetResource,
                          LPTSTR lpPassword, LPTSTR lpUserName,
                          DWORD dwFlags);
```

The first parameter is a handle to a window that owns any network support dialogs that might need to be displayed to complete the connection. The window handle can be NULL if you don't want to specify an owner window. This effectively turns the *WNetAddConnection3* function into the *WNetAddConnection2* function supported under other versions of Windows.

The second parameter, *lpNetResource*, should point to a NETRESOURCE structure that defines the remote resource being connected. The structure is defined as

```
typedef struct _NETRESOURCE {
    DWORD  dwScope;
    DWORD  dwType;
    DWORD  dwDisplayType;
    DWORD  dwUsage;
    LPTSTR lpLocalName;
    LPTSTR lpRemoteName;
    LPTSTR lpComment;
    LPTSTR lpProvider;
} NETRESOURCE;
```

Most of these fields aren't used for the *WNetAddConnection3* function and should be set to 0. All you need to do is to specify the UNC name of the remote resource in a string pointed to by *lpRemoteName* and the local name in a string pointed to by *lpLocalName*. The local name is limited to 64 characters in length. The other fields in this structure are used by the WNet enumeration functions that I'll describe shortly.

You use the next two parameters in *WNetAddConnection3*, *lpPassword* and *lpUserName*, when requesting access from the server to the remote device. If you don't specify a user name and Windows CE can't find user information for network access already defined in the registry, the system displays a dialog box requesting the user name and password. Finally, the *dwFlags* parameter can be either 0 or the flag CONNECT_UPDATE_PROFILE. When this flag is set, the connection is dubbed *persistent*. Windows CE stores the connection data for persistent connections in the registry. Unlike other versions of Windows, Windows CE doesn't restore persistent connections when the user logs on. Instead, the local name to remote name mapping is tracked only in the registry. If the local folder is later accessed after the original connection was dropped, a reconnection is automatically attempted when the local folder is accessed.

If the call to *WNetAddConnection3* is successful, it returns NO_ERROR. Unlike most Win32 functions, *WNetAddConnection3* returns an error code in the return value if an error occurs. This is a nod to compatibility that stretches back to the Windows 3.1 days. You can also call *GetLastError* to return the error information. As an aside, the function *WNetGetLastError* is supported under Windows CE in that it's redefined as *GetLastError*, so you can call that function if compatibility with other platforms is important.

The other function you can use under Windows CE to connect a remote resource is *WNetConnectionDialog1*. This function presents a dialog box to the user requesting the remote and local names for the connection. The function is prototyped as

```
DWORD WNetConnectionDialog1 (LPCONNECTDLGSTRUCT lpConnectDlgStruc);
```

The one parameter is a pointer to a CONNECTDLGSTRUCT structure defined as the following:

```
typedef struct {
    DWORD cbStructure;
    HWND hwndOwner;
    LPNETRESOURCE lpConnRes;
    DWORD dwFlags;
    DWORD dwDevNum;
} CONNECTDLGSTRUCT;
```

The first field in the structure is the size field and must be set with the size of the CONNECTDLGSTRUCT structure before you call *WNetConnectionDialog1*. The *hwndOwner* field should be filled with the handle of the owner window for the dialog box. The *lpConnRes* field should point to a NETRESOURCE structure. This structure should be filled with zeros except for the *lpRemoteName* field, which may be filled to specify the default remote name in the dialog. You can leave the *lpRemoteName* field 0 if you don't want to specify a suggested remote path.

The *dwFlags* field can either be 0 or set to the flag CONNDLG_RO_PATH. When this flag is specified, the user can't change the remote name field in the dialog box. Of course, this means that the *lpRemoteName* field in the NETRESOURCE structure must contain a valid remote name. Windows CE ignores the *dwDevNum* field in the CONNECTDLGSTRUCT structure.

When the function is called, it displays a dialog box that allows the user to specify a local and, if not invoked with the CONNDLG_RO_PATH flag, the remote name as well. If the user taps on the OK button, Windows attempts to make the connection specified. The connection, if successful, is recorded as a persistent connection in the registry.

If the connection is successful, the function returns NO_ERROR. If the user presses the Cancel button in the dialog box, the function returns –1. Other return codes indicate errors processing the function.

Disconnecting a remote resource

You can choose from three ways to disconnect a connected resource. The first method is to delete the connection with this function:

```
DWORD WNetCancelConnection2 (LPTSTR lpName, DWORD dwFlags,
                             BOOL fForce);
```

The *lpName* parameter points to either the local name or the remote network name of the connection you want to remove. The *dwFlags* parameter should be set to 0 or CONNECT_UPDATE_PROFILE. If CONNECT_UPDATE_PROFILE is set, the entry in the registry that references the connection is removed; otherwise the call won't change that information. Finally, the *fForce* parameter indicates whether the system should continue with the disconnect, even if there are open files or print jobs on the remote device. If the function is successful, it returns NO_ERROR.

You can prompt the user to specify a network resource to delete using this function:

```
DWORD WNetDisconnectDialog (HWND hwnd, DWORD dwType);
```

This function brings up a system provided dialog box that lists all connections currently defined. The user can select one from the list and tap on the OK button to disconnect that resource. The two parameters for this function are a handle to the window that owns the dialog box and *dwType*, which is supposed to define the type of resources—printer (RESOURCETYPE_PRINT) or disk (RESOURCETYPE_DISK)—enumerated in the dialog box. However, some systems ignore this parameter and enumerate both disk and print devices. This dialog, displayed by *WnetDisconnectDialog*, is actually implemented by the network driver. So it's up to each OEM to get this dialog to work correctly.

A more specific method to disconnect a network resource is to call

```
DWORD WNetDisconnectDialog1 (LPDISCDLGSTRUCT lpDiscDlgStruc);
```

This function is misleadingly named in that it won't display a dialog box if all the parameters in DISCDLGSTRUCT are correct and point to a resource not currently being used. The dialog part of this function appears when the resource is being used.

The DISCDLGSTRUCT is defined as

```
typedef struct {
    DWORD cbStructure;
    HWND hwndOwner;
    LPTSTR lpLocalName;
    LPTSTR lpRemoteName;
    DWORD dwFlags;
} DISCDLGSTRUCT;
```

As usual, the *cbStructure* field should be set to the size of the structure. The *hwndOwner* field should be set to the window that owns any dialog box displayed. The *lpLocalName* and *lpRemoteName* fields should be set to the local and remote names of the resource that's to be disconnected. Under current implementations, the *lpLocalName* is optional while the *lpRemoteName* field must be set for the function to work correctly. The *dwFlags* parameter can be either 0 or DISC_NO_FORCE. If this flag is set and the network resource is currently being used, the system simply fails the function. Otherwise, a dialog appears asking the user if he or she wants to disconnect the resource even though the resource is being used. Under the current implementations, the DISC_NO_FORCE flag is ignored.

Enumerating network resources

It's all very well and good to connect to a network resource, but it helps if you know what resources are available to connect to. Windows CE supports three WNet functions used to enumerate network resources: *WNetOpenEnum*, *WNetEnumResource*, and *WNetCloseEnum*. The process is similar to enumerating files with *FileFindFirst*, *FileFindNext*, and *FileFindClose*.

To start the process of numerating network resources, first call the function

```
DWORD WNetOpenEnum (DWORD dwScope, DWORD dwType, DWORD dwUsage,
                    LPNETRESOURCE lpNetResource,
                    LPHANDLE lphEnum);
```

The first parameter *dwScope* specifies the scope of the enumeration. It can be one of the following flags:

- *RESOURCE_CONNECTED* Enumerate the connected resources.
- *RESOURCE_REMEMBERED* Enumerate the persistent network connections.
- *RESOURCE_GLOBALNET* Enumerate all resources on the network.

The first two flags, RESOURCE_CONNECTED and RESOURCE_REMEMBERED, simply enumerate the resources already connected on your machine. The difference is that RESOURCE_CONNECTED returns the network resources that are connected at the time of the call, while RESOURCE_REMEMBERED returns those that are persistent regardless of whether they're currently connected. When using either of these flags, the *dwUsage* parameter is ignored and the *lpNetResource* parameters must be NULL.

The third flag, RESOURCE_GLOBALNET, allows you to enumerate resources—such as servers, shared drives, or printers out on the network—that aren't connected. The *dwType* parameter specifies what you're attempting to enumerate—shared disks (RESOURCETYPE_DISK), shared printers (RESOURCETYPE_PRINT), or both (RESOURCETYPE_ANY).

You use the third and fourth parameters only if the *dwScope* parameter is set to RESOURCE_GLOBALNET. The *dwUsage* parameter specifies the usage of the resource and can be 0 to enumerate any resource, RESOURCEUSAGE_CONNECTABLE to enumerate only connectable resources, or RESOURCEUSAGE_CONTAINER to enumerate only containers such as servers.

If the *dwScope* parameter is set to RESOURCE_GLOBALNET, the fourth parameter, *lpNetResource* must point to a NETRESOURCE structure; otherwise the parameter must be NULL. The NETRESOURCE structure should be initialized to specify the starting point on the network for the enumeration. The starting point is specified by a UNC name in the *lpRemoteName* field of NETRESOURCE. The *dwUsage* field of the NETRESOURCE structure must be set to RESOURCETYPE_CONTAINER. For example, to enumerate the shared resources on the server BIGSERV the *lpRemoteName* field would point to the string *\\BIGSERV*. To enumerate all servers in a domain, the *lpRemoteName* should simply specify the domain name. For the domain EntireNet, the *lpRemoteName* field should point to the string *EntireNet*. Because Windows CE doesn't allow you to pass a NULL into *lpRemoteName* when you use the RESOURCE_GLOBALNET flag, you can't enumerate all resources in the network namespace as you can under Windows 98 or Windows NT. This restriction exists because Windows CE doesn't support the concept of a Windows CE device belonging to a specific network context.

The final parameter of *WNetOpenEnum*, *lphEnum*, is a pointer to an enumeration handle that will be passed to the other functions in the enumeration process. *WNetOpenEnum* returns a value of NO_ERROR if successful. If the function isn't successful, you can call *GetLastError* to query the extended error information.

Once you have successfully started the enumeration process, you actually query data by calling this function:

```
DWORD WNetEnumResource (HANDLE hEnum, LPDWORD lpcCount,
                        LPVOID lpBuffer,
                        LPDWORD lpBufferSize);
```

The function takes the handle returned by *WNetOpenEnum* as its first parameter. The second parameter is a pointer to a variable that should be initialized with the number of resources you want to enumerate in each call to *WNetEnumResource*. You can specify a –1 in this variable if you want *WNetEnumResource* to return the data for as many resources as will fit in the return buffer specified by the *lpBuffer* parameter. The final parameter is a pointer to a DWORD that should be initialized with the size of the buffer pointed to by *lpBuffer*. If the buffer is too small to hold the data for even one resource, *WNetEnumResource* sets this variable to the required size for the buffer.

The information about the shared resources returned by data is returned in the form of an array of NETRESOURCE structures. While this is the same structure I described when I talked about the *WNetAddConnection3* function, I'll list the elements of the structure here again for convenience:

```
typedef struct _NETRESOURCE {
    DWORD  dwScope;
    DWORD  dwType;
    DWORD  dwDisplayType;
    DWORD  dwUsage;
    LPTSTR lpLocalName;
    LPTSTR lpRemoteName;
    LPTSTR lpComment;
    LPTSTR lpProvider;
} NETRESOURCE;
```

The interesting fields in the context of enumeration start with the *dwType* field, which indicates the type of resource that was enumerated. The value can be RESOURCETYPE_DISK or RESOURCETYPE_PRINT. The *dwDisplayType* field provides even more information about the resource, demarcating domains (RESOURCE-DISPLAYTYPE_DOMAIN) from servers (RESOURCEDISPLAYTYPE_SERVER) and from shared disks and printers (RESOURCEDISPLAYTYPE_SHARE). A fourth flag, RESOURCEDISPLAYTYPE_GENERIC, is returned if the display type doesn't matter.

The *lpLocalName* field points to a string containing the local name of the resource if the resource is currently connected or is a persistent connection. The *lpRemoteName* field points to the UNC name of the resource. The *lpComment* field contains the comment line describing the resource that's provided by some servers.

WNetEnumResource either returns NO_ERROR, indicating the function passed (but you need to call it again to enumerate more resources), or ERROR_NO_MORE_ITEMS, indicating that you have enumerated all resources matching the specification passed in *WNetOpenEnum*. With any other return code, you should call *GetLastError* to further diagnose the problem.

You have few strategies when enumerating the network resources. You can specify a huge buffer and pass a –1 in the variable pointed to by *lpcCount*, telling *WNetEnumResource* to return as much information as possible in one shot. Or you can specify a smaller buffer and ask for only one or two resources for each call to *WNetEnumResource*. The one caveat on the small buffer approach is that the strings that contain the local and remote names are also placed in the specified buffer. The name pointers inside the NETRESOURCE structure then point to those strings. This means that you can't specify the size of the buffer to be exactly the size of the NETRESOURCE structure and expect to get any data back. A third possibility is to call *WNetEnumResource* twice, the first time with the *lpBuffer* parameter 0, and have Windows CE tell you the size necessary for the buffer. Then you allocate the buffer and call *WNetEnumResource* again to actually query the data. However you use *WnetEnumResource*, you'll need to check the return code to see whether it needs to be called again to enumerate more resources.

When you have enumerated all the resources, you must make one final call to the function:

```
DWORD WNetCloseEnum (HANDLE hEnum);
```

The only parameter to this function is the enumeration handle first returned by *WNetOpenEnum*. This function cleans up the system resources used by the enumeration process.

Following is a short routine that uses the enumeration functions to query the network for available resources. You pass to a function a UNC name to use as the root of the search. The function returns a buffer of zero-delimited strings that designate the local name, if any, and the UNC name of each shared resource found.

```
// Helper routine
int AddToList (LPTSTR *pPtr, INT *pnListSize, LPTSTR pszStr) {
    INT nLen = lstrlen (pszStr) + 1;

    if (*pnListSize < nLen) return -1;
    lstrcpy (*pPtr, pszStr);
    *pPtr += nLen;
    *pnListSize -= nLen;
    return 0;
}
//----------------------------------------------------------------------
// EnumNetDisks - Produces a list of shared disks on a network
//
int EnumNetDisks (LPTSTR pszRoot, LPTSTR pszNetList, int nNetSize){
    INT i = 0, rc, nBuffSize = 1024;
    DWORD dwCnt, dwSize;
    HANDLE hEnum;
    NETRESOURCE nr;
    LPNETRESOURCE pnr;
    PBYTE pPtr, pNew;

    // Allocate buffer for enumeration data.
    pPtr = (PBYTE) LocalAlloc (LPTR, nBuffSize);
    if (!pPtr)
        return -1;

    // Initialize specification for search root.
    memset (&nr, 0, sizeof (nr));
    nr.lpRemoteName = pszRoot;
    nr.dwUsage = RESOURCEUSAGE_CONTAINER;

    // Start enumeration.
    rc = WNetOpenEnum (RESOURCE_GLOBALNET, RESOURCETYPE_DISK, 0, &nr,
                       &hEnum);
    if (rc != NO_ERROR)
        return -1;

    // Enumerate one item per loop.
    do {
        dwCnt = 1;
        dwSize = nBuffSize;
        rc = WNetEnumResource (hEnum, &dwCnt, pPtr, &dwSize);

        // Process returned data.
        if (rc == NO_ERROR) {
            pnr = (NETRESOURCE *)pPtr;
            if (pnr->lpRemoteName)
                rc = AddToList (&pszNetList, &nNetSize,
                                pnr->lpRemoteName);
            // If our buffer was too small, try again.
            } else if (rc == ERROR_MORE_DATA) {
                pNew = LocalReAlloc (pPtr, dwSize, LMEM_MOVEABLE);
                if (pNew) {
                    pPtr = pNew;
                    nBuffSize = LocalSize (pPtr);
                    rc = 0;
                }
            }
        } while (rc == 0);

        // If the loop was successful, add extra zero to list.
        if (rc == ERROR_NO_MORE_ITEMS) {
            rc = AddToList (&pszNetList, &nNetSize, TEXT (""));
            rc = 0;
        }

        // Clean up.
        WNetCloseEnum (hEnum);
        LocalFree (pPtr);
        return rc;
}
```

While the enumeration functions work well to query what's available on the net, you can use another strategy for determining the current connected resources. At the simplest level, you can use *FileFindFirst* and *FileFindNext* to enumerate the locally connected network disks by searching the folders in the \network directory. Once you have the local name, a few functions are available to you for querying just what that local name is connected to.

Querying connections and resources

The folders in the \network directory represent the local names of network shared disks that are persistently connected to network resources. To determine which of the folders are currently connected, you can use the function

```
DWORD WNetGetConnection (LPCTSTR lpLocalName,
                         LPTSTR lpRemoteName,
                         LPDWORD lpnLength);
```

WNetGetConnection returns the UNC name of the network resource associated with a local device or folder. The *lpLocalName* parameter is filled with the local name of a shared folder or printer. The *lpRemoteName* parameter should point to a buffer that can receive the UNC name for the device. The *lpnLength* parameter points to a DWORD value that initially contains the length in characters of the remote name buffer. If the buffer is too small to receive the name, the length value is loaded with the number of characters required to hold the UNC name.

One feature (or problem, depending on how you look at it) of *WNetGetConnection* is that it fails unless the local folder or device has a current connection to the remote shared device. This allows us an easy way to determine which local folders are currently connected and which are just placeholders for persistent connections that aren't currently connected.

Sometimes you need to transfer a filename from one system to another and you need a common format for the filename that would be understood by both systems. The *WNetGetUniversalName* function translates a filename that contains a local network name into one using the UNC name of the connected resource. The prototype for *WNetGetUniversalName* is the following:

```
DWORD WNetGetUniversalName (LPCTSTR lpLocalPath, DWORD dwInfoLevel,
                            LPVOID lpBuffer, LPDWORD lpBufferSize);
```

Like *WNetGetConnection*, this function returns a UNC name for a local name. There are two main differences between *WNetGetConnection* and *WNetGetUniversalName*. First, *WNetGetUniversalName* works even if the remote resource isn't currently connected. Second, you can pass a complete filename to *WNetGetUniversalName* instead of simply the local name of the shared resource, which is all that is accepted by *WNetGetConnection*.

WNetGetUniversalName returns the remote information in two different formats. If the *dwInfoLevel* parameter is set to UNIVERSAL_NAME_INFO_LEVEL, the buffer pointed to by *lpBuffer* is loaded with the following structure:

```
typedef struct _UNIVERSAL_NAME_INFO {
    LPTSTR  lpUniversalName;
} UNIVERSAL_NAME_INFO;
```

The only field in the structure is a pointer to the UNC name for the shared resource. The string is returned in the buffer immediately following the structure. So, if a server \*BigServ**DriveC* was attached as LocC and you pass *WnetGetUniversalName* the filename \network\LocC\win32\filename.ext, it returns the UNC name \\*BigServ**DriveC**win32**filename.ext*.

If the *dwInfoLevel* parameter is set to REMOTE_NAME_INFO_LEVEL, the buffer is filled with the following structure:

```
typedef struct _REMOTE_NAME_INFO
    LPTSTR  lpUniversalName;
    LPTSTR  lpConnectionName;
    LPTSTR  lpRemainingPath;
} REMOTE_NAME_INFO;
```

This structure returns not just the UNC name, but also parses the UNC name into the share name and the remaining path. So, using the same filename as in the previous example, \network\LocC\win32\filename.ext, the REMOTE_NAME_INFO fields would point to the following strings:

lpUniveralName:	*\\BigServ\DriveC\win32\filename.ext*
lpConnectionName:	*\\BigServ\DriveC*
lpRemainingPath:	*\win32\filename.ext*

One more thing: you don't have to prefix the local share name with \network. In the preceding example, the filename \LocC\Win32\filename.ext would have produced the same results.

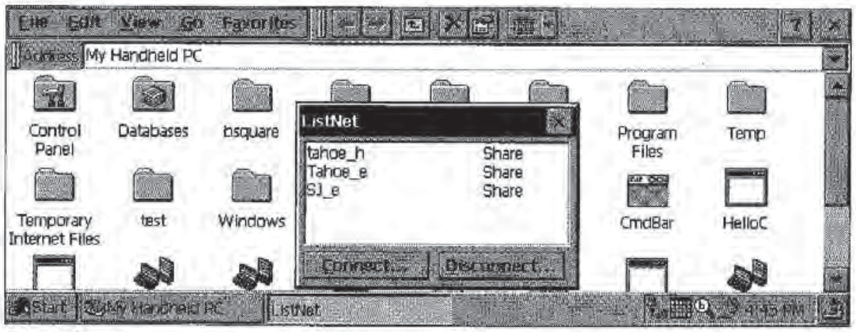
One final WNet function supported by Windows CE is

```
DWORD WnetGetUser (LPCTSTR lpName, LPTSTR lpUserName,
                   LPDWORD lpnLength);
```

This function returns the name the system used to connect to the remote resource. *WnetGetUser* is passed the local name of the shared resource and returns the user name the system used when connecting to the remote resource in the buffer pointed to by *lpUserName*. The *lpnLengh* parameter should point to a variable that contains the size of the buffer. If the buffer isn't big enough to contain the user name, the variable pointed to by *lpnLength* is filled with the required size for the buffer.

The ListNet Example Program

ListNet is a short program that lists the persistent network connections on a Windows CE machine. The program's window is a dialog box with three controls: a list box that displays the network connections, a Connect button that lets you add a new persistent connection, and a Disconnect button that lets you delete one of the connections. Double-clicking on a connection in the list box opens an Explorer window to display the contents of that network resource. Figure 10-1 shows the ListNet window while Figure 10-2 on the next page shows the ListNet source code.



```
tahoe_h     Share
Tahoe_e     Share
SJ_e        Share
```

Figure 10-1. *The ListNet window containing a few network folders.*

Page 00614

ListNet.rc

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

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

//----------------------------------------------------------------------
// Main window dialog template
//
ListNet DIALOG discardable 10, 10, 120, 65
STYLE  WS_OVERLAPPED | WS_VISIBLE | WS_CAPTION | WS_SYSMENU |
       DS_CENTER | DS_MODALFRAME
CAPTION "ListNet"
BEGIN
    LISTBOX                     IDD_NETLIST,   2,   2, 116,  46,
                             WS_TABSTOP | WS_VSCROLL |
                             LBS_NOINTEGRALHEIGHT | LBS_USETABSTOPS
    PUSHBUTTON "&Connect...",   IDD_CNCT,    2,  50,  55,  12, WS_TABSTOP
    PUSHBUTTON "&Disconnect...",
                                IDD_DCNCT,  61,  50,  55,  12, WS_TABSTOP
END
```

ListNet.h

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

//----------------------------------------------------------------------
// Generic defines and data types
```

Figure 10-2. *The ListNet source.*

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

#define  ID_ICON           1

#define  IDD_NETLIST       100              // Control IDs
#define  IDD_CNCT          101
#define  IDD_DCNCT         102

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

// Dialog window procedure
BOOL CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Dialog window Message handlers
BOOL DoCommandMain (HWND, UINT, WPARAM, LPARAM);

// Command functions
LPARAM DoMainCommandExit (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandViewDrive (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandMapDrive (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandFreeDrive  (HWND, WORD, HWND, WORD);
```

ListNet.c

```
//======================================================================
// ListNet - A network demo application for Windows CE
//
// Written for the book Programming Windows CE
```

(continued)

Figure 10-2. *continued*

```
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                  // For all that Windows stuff
#include <winnetwk.h>                 // Network includes
#include "ListNet.h"                  // Program-specific stuff

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

// Command Message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDOK, DoMainCommandExit,
    IDCANCEL, DoMainCommandExit,
    IDD_NETLIST, DoMainCommandViewDrive,
    IDD_CNCT, DoMainCommandMapDrive,
    IDD_DCNCT, DoMainCommandFreeDrive,
};
//======================================================================
//
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.
    DialogBox (hInst, szAppName, NULL, MainWndProc);
    return 0;
}
//======================================================================
// Message handling procedures for main window
//----------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
BOOL CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                           LPARAM lParam) {
    INT i;
    // With only two messages, do it the old-fashioned way.
    switch (wMsg) {
    case WM_INITDIALOG:
        i = 75;
```

```
            SendDlgItemMessage (hWnd, IDD_NETLIST, LB_SETTABSTOPS, 1,
                                (LPARAM)&i);
            RefreshLocalNetDrives (hWnd);
            break;

        case WM_COMMAND:
        return DoCommandMain (hWnd, wMsg, wParam, lParam);
        }
        return FALSE;
}
//----------------------------------------------------------------------
// DoCommandMain - Process WM_COMMAND message for window.
//
BOOL 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) {
            (*MainCommandItems[i].Fxn)(hWnd, idItem, hwndCtl,
                                       wNotifyCode);
            return TRUE;
        }
    }
    return FALSE;
}
//======================================================================
// Command handler routines
//----------------------------------------------------------------------
// DoMainCommandExit - Process Program Exit command
//
LPARAM DoMainCommandExit (HWND hWnd, WORD idItem, HWND hwndCtl,
                          WORD wNotifyCode) {
    EndDialog (hWnd, 0);
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandViewDrive - Process list box double clicks
//
```

(continued)

Figure 10-2. *continued*

```
LPARAM DoMainCommandViewDrive (HWND hWnd, WORD idItem, HWND hwndCtl,
                               WORD wNotifyCode) {
    TCHAR szCmdLine[128], szFolder[MAX_PATH];
    PROCESS_INFORMATION pi;
    HCURSOR hOld;
    INT i, rc;

    // We're only interested in list box double-clicks.
    if (wNotifyCode != LBN_DBLCLK)
        return 0;

    i = SendMessage (hwndCtl, LB_GETCURSEL, 0, 0);
    if (i == LB_ERR) return 0;
    i = SendMessage (hwndCtl, LB_GETTEXT, i, (LPARAM)szFolder);

    hOld = SetCursor (LoadCursor (NULL, IDC_WAIT));
    lstrcpy (szCmdLine, TEXT ("\\network\\"));
    lstrcat (szCmdLine, szFolder);

    rc = CreateProcess (TEXT ("Explorer"), szCmdLine, NULL, NULL,
                        FALSE, 0, NULL, NULL, NULL, &pi);
    if (rc) {
        CloseHandle (pi.hProcess);
        CloseHandle (pi.hThread);
    }
    SetCursor (hOld);
    return TRUE;
}
//----------------------------------------------------------------------
// DoMainCommandMapDrive - Process map network drive command.
//
LPARAM DoMainCommandMapDrive (HWND hWnd, WORD idItem, HWND hwndCtl,
                              WORD wNotifyCode) {
    DWORD rc;
    CONNECTDLGSTRUCT cds;
    NETRESOURCE nr;
    TCHAR szRmt[256];

    memset (&nr, 0, sizeof (nr));
    nr.dwType = RESOURCETYPE_DISK;
    memset (szRmt, 0, sizeof (szRmt));

    cds.cbStructure = sizeof (cds);
    cds.hwndOwner = hWnd;
    cds.lpConnRes = &nr;
    cds.dwFlags = CONNDLG_PERSIST;
    // Display dialog box.
    rc = WNetConnectionDialog1 (&cds);

    if (rc == NO_ERROR)
        RefreshLocalNetDrives (hWnd);
    return 0;
}
//----------------------------------------------------------------------
// DoMainCommandFreeDrive - Process disconnect network drive command.
//
LPARAM DoMainCommandFreeDrive (HWND hWnd, WORD idItem, HWND hwndCtl,
                               WORD wNotifyCode) {
    WNetDisconnectDialog (hWnd, RESOURCETYPE_DISK);
    RefreshLocalNetDrives (hWnd);
    return 0;
}
//======================================================================
// Network browsing functions
//----------------------------------------------------------------------
// EnumerateLocalNetDrives - Add an item to the list view control.
//
INT RefreshLocalNetDrives (HWND hWnd) {
    HWND hwndCtl = GetDlgItem (hWnd, IDD_NETLIST);
    INT rc, nBuffSize = 1024;
    DWORD dwCnt, dwSize;
    HANDLE hEnum;
    LPNETRESOURCE pnr;
    NETRESOURCE nr;
    PBYTE pPtr, pNew;
    TCHAR szText[256];

    SendMessage (hwndCtl, LB_RESETCONTENT, 0, 0);

    // Allocate buffer for enumeration data.
    pPtr = (PBYTE) LocalAlloc (LPTR, nBuffSize);
    if (!pPtr)
        return -1;

    // Initialize specification for search root.
    memset (&nr, 0, sizeof (nr));
    lstrcpy (szText, TEXT ("\\sjdev"));
    nr.lpRemoteName = szText;
    nr.dwUsage = RESOURCEUSAGE_CONTAINER;

    // Start enumeration.
    rc = WNetOpenEnum (RESOURCE_REMEMBERED, RESOURCETYPE_ANY, 0, 0,
                       &hEnum);
```

(continued)

Figure 10-2. *continued*

```
    if (rc != NO_ERROR) return -1;

    // Enumerate one item per loop.
    do {
        dwCnt = 1;
        dwSize = nBuffSize;
        rc = WNetEnumResource (hEnum, &dwCnt, pPtr, &dwSize);
        pnr = (NETRESOURCE *)pPtr;
        lstrcpy (szText, pnr->lpLocalName);
        // Process returned data.
        if (rc == NO_ERROR) {
            switch (pnr->dwType) {
            case RESOURCETYPE_ANY:
                lstrcat (szText, TEXT ("\t Share"));
                break;
            case RESOURCETYPE_PRINT:
                lstrcat (szText, TEXT ("\t Printer"));
                break;
            case RESOURCETYPE_DISK:
                lstrcat (szText, TEXT ("\t Disk"));
                break;
            }
            SendMessage (hwndCtl, LB_ADDSTRING, 0, (LPARAM)szText);

        // If our buffer was too small, try again.
        } else if (rc == ERROR_MORE_DATA) {
            pNew = LocalReAlloc (pPtr, dwSize, LMEM_MOVEABLE);
            if (pNew) {
                pPtr = pNew;
                nBuffSize = LocalSize (pPtr);
                rc = 0;
            } else
                break;
        }
    } while (rc == 0);
    // Clean up.
    WNetCloseEnum (hEnum);
    LocalFree (pPtr);
    return 0;
}
```

The heart of the networking code is at the end of ListNet, in the routine *RefreshLocalNetDrives*. This routine uses the WNet enumerate functions to determine the persistent network resources mapped to the system. Network connections and disconnections are accomplished with calls to *WNetConnectionDialog1* and

WnetDisconnectDialog respectively. You open an Explorer window containing the shared network disk by launching EXPLORER.EXE with a command line that's the path of the folder to open.

BASIC SOCKETS

WinSock is the name for the Windows socket API. WinSock is the API for Windows CE TCP/IP networking stack as well as the IrDA infrared communication stack. Windows CE implements a subset of WinSock version 1.1. What's left out of the Windows CE implementation of WinSock is the ever-so-handy *WSAAsyncSelect* function that enables (under other Windows systems) an application to be informed when a WinSock event occurred. Actually, most of the *WSAxxx* calls that provide asynchronous actions are missing from Windows CE. Instead, the Windows CE implementation is more like the original "Berkeley" socket API. Windows CE's developers decided not to support these functions to reduce the size of the WinSock implementation. These functions were handy, but not required because Windows CE is multithreaded.

The lack of asynchronous functions doesn't mean that you're left with calling socket functions that block on every call. You can put a socket in nonblocking mode so that any function that can't accomplish its task without waiting on an event will return with a return code indicating that the task isn't yet completed.

Windows CE has extended WinSock in one area. As I mentioned in Chapter 9, WinSock is also the primary interface for IrDA communication. To do this, Windows CE extends the socket addressing scheme, actually providing an entirely different addressing mode designed for the transitory nature of IrDA communication.

In this section, I'm not going to dive into a complete explanation of socket-based communication. Instead, I'll present an introduction that will get you started communicating with sockets. In addition, I'll spend time with the IrSock side because this interface is so significant for Windows CE devices.

Initializing the WinSock DLL

Like other versions of WinSock, the Windows CE version should be initialized before you use it. You accomplish this by calling *WSAStartup*, which initializes the WinSock DLL. It's prototyped as

```
int WSAStartup (WORD wVersionRequested, LPWSADATA lpWSAData );
```

The first parameter is the version of WinSock you're requesting to open. For all current versions of Windows CE, you must indicate version 1.1. An easy way to do this is to use the MAKEWORD macro as in MAKEWORD (1,1). The second parameter must point to a WSAData structure, shown in the code on the next page.

```
struct WSAData {
    WORD wVersion;
    WORD wHighVersion;
    char szDescription[WSADESCRIPTION_LEN+1];
    char szSystemStatus[WSASYSSTATUS_LEN+1];
    unsigned short iMaxSockets;
    unsigned short iMaxUdpDg;
    char FAR * lpVendorInfo;
};
```

This structure is filled in by *WSAStartup*, providing information about the specific implementation of this version of WinSock. Currently, the first two fields return *0x0101*, indicating support for version 1.1. The *szDescription* and *szSystemStatus* fields can be used by WinSock to return information about itself. In the current Windows CE version of WinSock, these fields aren't used. The *iMaxSockets* parameter suggests a maximum number of sockets that an application should be able to open. This number isn't a hard maximum but more a suggested maximum. Finally, the *iMaxUdpDg* field indicates the maximum size of a datagram packet. A 0 indicates no maximum size for this version of WinSock.

WSAStartup returns 0 if successful; otherwise the return value is the error code for the function. Don't call *WSAGetLastError* in this situation because the failure of this function indicates that WinSock, which provides *WSAGetLastError*, wasn't initialized correctly.

Windows CE also supports *WSACleanup*, which is traditionally called when an application has finished using the WinSock DLL. For Windows CE, this function performs no action but is provided for compatibility. Its prototype is

```
int WSACleanup ();
```

ASCII vs. Unicode

One issue that you'll have to be careful of is that almost all the string fields used in the socket structures are char fields, not Unicode. Because of this, you'll find yourself using the functions

```
int WideCharToMultiByte(UINT CodePage, DWORD dwFlags,
                        LPCWSTR lpWideCharStr, int cchWideChar,
                        LPSTR lpMultiByteStr, int cchMultiByte,
                        LPCSTR lpDefaultChar, LPBOOL lpUsedDefaultChar);
```

to convert Unicode strings into multibyte strings and

```
int MultiByteToWideChar (UINT CodePage, DWORD dwFlags,
                        LPCSTR lpMultiByteStr, int cchMultiByte,
                        LPWSTR lpWideCharStr, int cchWideChar);
```

to convert multibyte characters into Unicode. The functions refer to multibyte characters instead of ASCII because on double-byte coded systems, they convert double-byte characters into Unicode.

Stream Sockets

Like all socket implementations, WinSock under Windows CE supports both stream and datagram connections. In a stream connection, a socket is basically a data pipe. Once two points are connected, data is sent back and forth without the need for additional addressing. In a datagram connection, the socket is more like a mailslot, with discrete packets of data being sent to specific addresses. In describing the WinSock functions, I'm going to cover the process of a creating a *stream* connection (sometimes called a *connection-oriented* connection) between a client and server application. I'll leave explanation of the datagram connection to other, more network-specific books.

The life of a stream socket is fairly straightforward: it's created, bound, or connected to an address; read from or written to; and finally closed. A few extra steps along the way, however, complicate the story slightly. Sockets work in a client/server model. A client initiates a conversation with a known server. The server, on the other hand, waits around until a client requests data. When setting up a socket, you have to approach the process from either the client side or the server side. This decision determines which functions you call to configure a socket. Figure 10-3 illustrates the process from both the client and the server side. For each step in the process, the corresponding WinSock function is shown.

Server	***Function***	***Client***	***Function***
Create socket	*socket*	Create socket	*socket*
Bind socket to an address	*bind*	Find desired server	(many functions)
Listen for client connections	*listen*	Connect to server	*connect*
Accept client's connection	*accept*		
Receive data from client	*recv*	Send data to server	*send*
Send data to client	*send*	Receive data from server	*recv*

Figure 10-3. *The process for producing a connection-oriented socket connection.*

Both the client and the server must first create a socket. After that, the process diverges. The server must attach, or to use the function name, *bind,* the socket to an address so that another computer or even a local process, can connect to the socket. Once an address has been bound, the server configures the socket to listen for a connection from a client. The server then waits to accept a connection from a client. Finally, after all this, the server is ready to converse.

The client's job is simpler: the client creates the socket, connects the socket to a remote address, and then sends and receives data. This procedure, of course, ignores the sometimes not-so-simple process of determining the address to connect to. I'll leave that problem for a few moments while I talk about the functions behind this process.

Creating a socket

You create a socket with the function

```
SOCKET socket (int af, int type, int protocol);
```

The first parameter, *af*, specifies the addressing family for the socket. Windows CE supports two addressing formats; AF_INET and AF_IRDA. You use the AF_IRDA constant when you're creating a socket for IrDA use, and you use AF_INET for TCP/IP communication. The type parameter specifies the type of socket being created. For a TCP/IP socket, this can be either SOCK_STREAM for a stream socket or SOCK_DGRAM for a datagram socket. For IrDA sockets, the type parameter must be SOCK_STREAM. Windows CE doesn't currently expose a method to create a raw socket, which is a socket that allows you to interact with the IP layer of the TCP/IP protocol. Among other uses, raw sockets are used to send an echo request to other servers, in the process known as pinging. However, Windows CE does provide a method of sending an ICMP echo request. I'll talk about that shortly.

The protocol parameter specifies the protocol used by the address family specified by the *af* parameter. The function returns a handle to the newly created socket. If an error occurs, the socket returns INVALID_SOCKET. You can call *WSAGetLastError* to query the extended error code.

Server side: binding a socket to an address

For the server, the next step is to bind the socket to an address. You accomplish this with the function

```
int bind (SOCKET s, const struct sockaddr FAR *addr, int namelen);
```

The first parameter is the handle to the newly created socket. The second parameter is dependent on whether you're dealing with a TCP/IP socket or an IrDa socket. For a standard TCP/IP socket, the structure pointed to by *addr* should be SOCKADDR_IN, which is defined as

```
struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    IN_ADDR sin_addr;
    char sin_zero[8];
};
```

The first field, *sin_family* must be set to AF_INET. The second field is the IP port while the third field specifies the IP address. The last field is simply padding to fit the standard SOCKADDR structure. The last parameter of bind, *namelen*, should be set to the size of the SOCKADDR_IN structure.

When you're using IrSock, the address structure pointed to by *sockaddr* is SOCKADDR_IRDA, which is defined as

```
struct sockaddr_irda {
    u_short irdaAddressFamily;
    u_char irdaDeviceID[4];
    char irdaServiceName[25];
};
```

The first field, *irdaAddressFamily*, should be set to AF_IRDA to identify the structure. The second field, *irdaDeviceID*, is a 4-byte array that defines the address for this IR socket. This can be set to 0 for an IrSock server. The last field should be set to a string to identify the server.

You can also use a special, predefined name in the *irdaServiceName* field to bypass the IrDA address resolution features. If you specify the name LSAP-SEL*xxx* where *xxx* is a value from 001 through 127, the socket will be bound directly to the LSAP (Logical Service Assess Point) selector defined by the value. Applications should not, unless absolutely required, bind directly to a specific LSAP selector. Instead, by specifying a generic string, the IrDA Address resolution code determines a free LSAP selector and uses it.

Listening for a connection

Once a socket has been bound to an address, the server places the socket in listen mode so that it will accept incoming communication attempts. You place the socket in listen mode by using the aptly named function

```
int listen (SOCKET s, int backlog);
```

The two parameters are the handle to the socket and the size of the queue that you're creating to hold the pending connection attempts. This value can be set to SOMAXCONN to set the queue to the maximum supported by the socket implementation. For Windows CE, the only supported queue sizes are 1 and 2. Values outside this range are rounded to the closest valid value.

Accepting a connection

When a server is ready to accept a connection to a socket in listen mode, it calls this function:

```
SOCKET accept (SOCKET s, struct sockaddr FAR *addr,
               int FAR *addrlen);
```

The first parameter is the socket that has already been placed in listen mode. The next parameter should point to a buffer that receives the address of the client socket that has initiated a connection. The format of this address is dependent on the protocol used by the socket. For Windows CE, this is either a SOCKADDR_IN or a SOCKADDR_IRDA structure. The final parameter is a pointer to a variable that contains the size of the buffer. This variable is updated with the size of the structure returned in the address buffer when the function returns.

The *accept* function returns the handle to a new socket that's used to communicate with the client. The socket that was originally created by the call to *socket* will remain in listen mode, and can potentially accept other connections. If *accept* detects an error, it returns INVALID_SOCKET. In this case, you can call *WSAGetLastError* to get the error code.

The *accept* function is the first function I've talked about so far that blocks. That is, it won't return until a remote client requests a connection. You can set the socket in nonblocking mode so that, if no request for connection is queued, *accept* will return INVALID_SOCKET with an extended error code of WSAEWOULDBLOCK. I'll talk about blocking vs. nonblocking sockets shortly.

Client side: connecting a socket to a server

On the client side, things are different. Instead of calling the *bind* and *accept* functions, the client simply connects to a known server. I said simply, but as with most things, we must note a few complications. The primary one is addressing—knowing the address of the server you want to connect to. I'll put that topic aside for a moment and assume the client knows the address of the server.

To connect a newly created socket to a server, the client uses the function

```
int connect (SOCKET s, const struct sockaddr FAR *name,
             int namelen);
```

The first parameter is the socket handle that the client created with a call to *socket*. The other two parameters are the address and address length values we've seen in the *bind* and *accept* functions. Here again, Windows CE supports two addressing formats: SOCKADDR_IN for TCP/IP–based communication and SOCKADDR_IRDA for IrDA communication.

If connect is successful, it returns 0. Otherwise it returns SOCKET_ERROR, and you should call *WSAGetLastError* to get the reason for the failure.

Sending and receiving data

At this point, both the server and client have socket handles they can use to communicate with one another. The client uses the socket originally created with the call to *socket*, while the server uses the socket handle returned by the *accept* function.

All that remains is data transfer. You write data to a socket this way:

```
int send (SOCKET s, const char FAR *buf, int len, int flags);
```

The first parameter is the socket handle to send the data. You specify the data you want to send in the buffer pointed to by the *buf* parameter while the length of that data is specified in *len*. The *flags* parameter must be 0.

You receive data by using the function

```
int recv (SOCKET s, char FAR *buf, int len, int flags);
```

The first parameter is the socket handle. The second parameter points to the buffer that receives the data, while the third parameter should be set to the size of the buffer. The flags parameter can be 0, or it can be MSG_PEEK if you want to have the current data copied into the receive buffer but not removed from the input queue or if this is a TCP/IP socket (MSG_OOB) for receiving any out-of-band data that has been sent.

Two other functions can send and receive data; they are the following:

```
int sendto (SOCKET s, const char FAR *buf, int len, int flags,
            const struct sockaddr FAR *to, int token);
```

and

```
int recvfrom (SOCKET s, char FAR *buf, int len, int flags,
              struct sockaddr FAR *from, int FAR *fromlen);
```

These functions enable you to direct individual packets of data using the address parameters provided in the functions. They're used for connectionless sockets, but I mention them now for completeness. When used with connection-oriented sockets such as those I've just described, the addresses in *sendto* and *recvfrom* are ignored and the functions act like their simpler counterparts, *send* and *recv*.

Closing a socket

When you have finished using the sockets, call this function:

```
int shutdown (SOCKET s, int how);
```

The *shutdown* function takes the handle to the socket and a flag indicating what part of the connection you wish to shut down. The *how* parameter can be SD_RECEIVE to prevent any further *recv* calls from being processed, SD_SEND to prevent any further *send* calls from being processed, or SD_BOTH to prevent either *send* or *recv* calls from being processed. The *shutdown* function affects the higher level functions *send* and *recv* but doesn't prevent data previously queued from being processed. Once you have shut down a socket, it can't be used again. It should be closed and a new socket created to restart a session.

Once a connection has been shut down, you should close the socket with a call to this function:

```
int closesocket (SOCKET s);
```

The action of *closesocket* depends on how the socket is configured. If you've properly shut down the socket with a call to *shutdown*, no more events will be pending and *closesocket* should return without blocking. If the socket has been configured into "linger" mode and configured with a timeout value, *closesocket* will block until any data in the send queue has been sent or the timeout expires.

IrSock

I've alluded to IrSock a number of times as I've described functions. IrSock is essentially a socketlike API built over the top of the IrDA stack used for infrared communication. IrSock is the only high-level interface to the IrDA stack. Even the IrComm virtual comm port described in Chapter 9 uses the IrSock API underneath the covers.

The major differences between IrSock and WinSock are that IrSock doesn't support datagrams, it doesn't support security, and the method used for addressing it is completely different from that used for WinSock. What IrSock does provide is a method to query the devices ready to talk across the infrared port, as well as arbitration and collision detection and control.

From a programmer's perspective, the main difference in programming IrSock and WinSock is that the client side needs a method of detecting what infrared capable devices are within range and are ready to accept a socket connection. This is accomplished by calling *getsockopt* with the level parameter set to SOL_IRLMP and the *optname* parameter set to IRLMP_ENUMDEVICES, as in the following:

```
dwBuffSize = sizeof (buffer);
rc = getsockopt (hIrSock, SOL_IRLMP, IRLMP_ENUMDEVICES,
                 buffer, &dwBuffSize);
```

When called with IRLMP_ENUMDEVICES, *getsockopt* returns a DEVICELIST structure in the buffer. DEVICELIST is defined as

```
typedef struct _DEVICELIST {
    ULONG numDevice;
    IRDA_DEVICE_INFO Device[1];
} DEVICELIST;
```

The DEVICELIST structure is simply a count followed by an array of IRDA_DEVICE_INFO structures, one for each device found. The IRDA_DEVICE_INFO structure is defined as

```
typedef struct _IRDA_DEVICE_INFO {
    u_char irdaDeviceID[4];
    char irdaDeviceName[22];
    u_char Reserved[2];
} IRDA_DEVICE_INFO;
```

The two fields in the IRDA_DEVICE_INFO structure are a device ID and a string that can be used to identify the remote device.

Following is a routine that opens an IR socket and uses *getsockopt* to query the remote devices that are in range. If any devices are found, their names and IDs are printed to the debug port.

```
//
// Poll for IR devices.
//
DWORD WINAPI IrPoll (HWND hWnd) {
    INT rc, nSize, i, j;
    char cDevice[256];
    TCHAR szName[32], szOut[256];
    DEVICELIST *pDL;
    SOCKET irsock;

    // Open an infrared socket.
    irsock = socket (AF_IRDA, SOCK_STREAM, 0);
    if (irsock == INVALID_SOCKET)
        return -1;

    // Search for someone to talk to, try 10 times over 5 seconds.
    for (i = 0; i < 10; i++) {

        // Call getsockopt to query devices.
        memset (cDevice, 0, sizeof (cDevice));
        nSize = sizeof (cDevice);
        rc = getsockopt (irsock, SOL_IRLMP, IRLMP_ENUMDEVICES,
                         cDevice, &nSize);
        if (rc)
            break;

        pDL = (DEVICELIST *) cDevice;
        if (pDL->numDevice) {
            Add2List (hWnd, TEXT ("%d devices found."), pDL->numDevice);

            for (j = 0; j < (int)pDL->numDevice; j++) {
                // Convert device ID.
                wsprintf (szOut,
                          TEXT ("DeviceID \t%02X.%02X.%02X.%02X"),
                          pDL->Device[j].irdaDeviceID[0],
                          pDL->Device[j].irdaDeviceID[1],
                          pDL->Device[j].irdaDeviceID[2],
                          pDL->Device[j].irdaDeviceID[3]);
                OutputDebugString (szOut);

                // Convert device name to Unicode.
                mbstowcs (szName, pDL->Device[j].irdaDeviceName,
                          sizeof (pDL->Device[j].irdaDeviceName));

                wsprintf (szOut, TEXT ("irdaDeviceName \t%s"),
                          szName);
                OutputDebugString (szOut);
            }
```

(continued)

```
            }
            Sleep(500);
        }
        closesocket (irsock);
        return 0;
}
```

Just having a device with an IR port in range isn't enough; the remote device must have an application running that has opened an IR socket, bound it, and placed it into listen mode. This requirement is appropriate because these are the steps any server using the socket API would perform to configure a socket to accept communication.

Querying and setting IR socket options

IrSock supports the *getsockopt* and *setsockopt* functions for getting and setting the socket options, but the options supported have little overlap with the socket options supported for a standard TCP/IP socket. To query socket options, use this function:

```
int getsockopt (SOCKET s, int level, int optname,
                char FAR *optval, int FAR *optlen);
```

The first parameter is the handle to the socket while the second parameter is the level in the communications stack for the specific option. The level can be at the socket level SO_SOCKET or a level unique to IrSock, SOL_IRLMP. The options supported for IrSock are shown in the lists below.

For the SOL_SOCKET level, your option is

- *SO_LINGER* It queries the linger mode.

For the SOL_IRLMP level, your options are

- *IRLMP_ENUMDEVICES* which enumerate remote IrDA devices
- *IRLMP_IAS_QUERY* which queries IAS attributes
- *IRLMP_SEND_PDU_LEN* which queries the maximum size of send packet for IrLPT mode.

The corresponding function with which to set the options is

```
int setsockopt (SOCKET s, int level, int optname,
                const char FAR *optval, int optlen);
```

The parameters are similar to *getsockopt*. The allowable options are shown below.

For the SOL_SOCKET level, your option is

- *SO_LINGER* which delays the close of a socket if unsent data remains in the outgoing queue

For the SOL_IRLMP level, your options are

- *IRLMP_IAS_SET* which sets IAS attributes
- *IRLMP_IRLPT_MODE* which sets the IrDA protocol to IrLPT
- *IRLMP_9WIRE_MODE* which sets the IrDA protocol to 9-wire serial mode
- *IRLMP_SHARP_MODE* which sets the IrDA protocol to Sharp mode

Blocking vs. nonblocking sockets

One issue I briefly touched on as I was introducing sockets is blocking. Windows programmers are used to the quite handy asynchronous socket calls that are an extension of the standard Berkeley socket API. By default, a socket is in blocking mode so that, for example, if you call *recv* to read data from a socket and no data is available, the call blocks until some data can be read. This isn't the type of call you want to be making with a thread that's servicing the message loop for your application.

Although Windows CE doesn't support the *WSAAsync* calls available to desktop versions of Windows, you can switch a socket from its default blocking mode to nonblocking mode. In nonblocking mode, any socket call that might need to wait to successfully perform its function instead returns immediately with an error code of WSAEWOULDBLOCK. You are then responsible for calling the would-have-blocked function again at a later time to complete the task.

To set a socket into blocking mode, use this function:

```
int ioctlsocket (SOCKET s, long cmd, u_long *argp);
```

The parameters are the socket handle, a command, and a pointer to a variable that either contains data or receives data depending on the value in *cmd*. The allowable commands for Windows CE IrSock sockets are the following:

- *FIONBIO* Set or clear a socket's blocking mode. If the value pointed to by *argp* is nonzero, the socket is placed in blocking mode. If the value is zero, the socket is placed in nonblocking mode.
- *FIONREAD* Returns the number of bytes that can be read from the socket with one call to the *recv* function.

So to set a socket in blocking mode, you should make a call like this one:

```
fBlocking = FALSE;
rc = ioctlsocket (sock, FIONBIO, &fBlocking);
```

Of course, once you have a socket in nonblocking mode, the worst thing you can do is continually poll the socket to see if the nonblocked event occurred. On a battery-powered system, this can dramatically lower battery life. Instead of polling, you can use the *select* function to inform you when a socket or set of sockets is in a nonblocking state. The prototype for this function is

```
int select (int nfds, fd_set FAR *readfds, fd_set FAR *writefds,
            fd_set FAR *exceptfds,
            const struct timeval FAR *timeout);
```

The parameters for the *select* function look somewhat complex, which, in fact, they are. Just to throw a curve, the function ignores the first parameter. The reason it exists at all is for compatibility with the Berkeley version of the *select* function. The next three parameters are pointers to sets of socket handles. The first set should contain the sockets that you want to be notified when one or more of the sockets is in a nonblocking read state. The second set contains socket handles of sockets that you want informed when a write function can be called without blocking. Finally, the third set, pointed to by *exceptfds*, contains the handles of sockets that you want notified when an error condition exists in that socket.

The final parameter is a timeout value. In keeping with the rather interesting parameter formats for the *select* function, the timeout value isn't a simple millisecond count. Rather, it's a pointer to a TIMEVAL structure defined as

```
struct timeval {
    long    tv_sec;
    long    tv_usec;
};
```

If the two fields in TIMEVAL are 0, the *select* call returns immediately even if none of the sockets has had an event occur. If the pointer, *timeout*, is NULL instead of pointing to a TIMEVAL structure, the select call won't time out and returns only when an event occurs in one of the sockets. Otherwise, the timeout value is specified in seconds and microseconds in the two fields provided.

The function returns the total number of sockets for which the appropriate events occur, 0 if the function times out, or SOCKET_ERROR if an error occurred while processing the call. If an error does occur, you can call *WSAGetLastError* to get the error code. The function modifies the contents of the sets so that, on returning from the function, the sets contain only the socket handles of sockets for which events occur.

The sets that contain the events should be considered opaque. The format of the sets doesn't match their Berkeley socket counterparts. Each of the sets is manipulated by four macros defined in WINSOCK.H. These are the four macros:

- *FD_CLR* Removes the specified socket handle from the set
- *FD_ISSET* Returns true if the socket handle is part of the set

- *FD_SET* Adds the specified socket handle to the set
- *FD_ZERO* Initializes the set to 0

To use a set, you have to declare a set of type *fd_set*. Then initialize the set with a call to FD_ZERO and add the socket handles you want with FD_SET. An example would be

```
fd_set fdReadSocks;

FD_ZERO (&fdReadSocks);
FD_SET (hSock1, &fdReadSocks);
FD_SET (hSock2, &fdReadSocks);

rc = select (0, &fdReadSocks, NULL, NULL, NULL);
if (rc != SOCKET_ERROR) {
    if (FD_ISSET (hSock1, &fdReadSocks))
        // A read event occurred in socket 1.
    if (FD_ISSET (hSock2, &fdReadSocks))
        // A read event occurred in socket 2.
}
```

In this example, the *select* call waits on read events from two sockets with handles of *hSock1* and *hSock2*. The write and error sets are NULL as is the pointer to the *timeout* structure, so the call to *select* won't return until a read event occurs in one of the two sockets. When the function returns, the code checks to see if the socket handles are in the returned set. If so, that socket has a nonblocking read condition.

The last little subtlety concerning the *select* function is just what qualifies as a read, write, and error condition. A socket in the read set is signaled when one of the following events occur:

- There is data in the input queue so that *recv* can be called without blocking.
- The socket is in listen mode and a connection has been attempted so that a call to *accept* won't block.
- The connection has been closed, reset, or terminated. If the connection was gracefully closed, *recv* returns with 0 bytes read; otherwise the *recv* call returns SOCKET_ERROR. If the socket has been reset, the *recv* function returns the error WSACONNRESET.

A socket in the write set is signaled under the following conditions:

- Data can be written to the socket. A call to send still might block if you attempt to write more data than can be held in the outgoing queue.

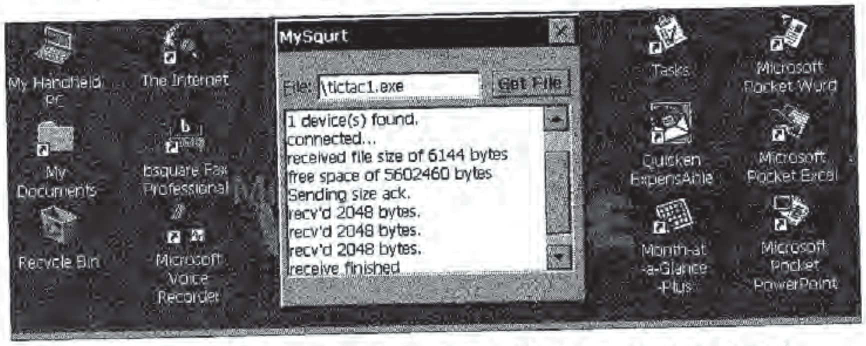
- A socket is processing a *connect* and the connect has been accepted by the server.

A socket in the exception set is signaled under the following condition:

- A socket is processing a *connect* and the connect failed.

The MySqurt Example Program

To demonstrate IrSock, the following program, MySqurt, shows how to transfer files from one Windows CE device to another. It's similar to the IrSquirt program provided with the H/PC and Palm-size PC. The difference is that instead of sending a file across the infrared link and having the receiving side accept whatever file is sent, MySqurt has the receiving side specify the file that's sent from the serving side of the application. In addition, MySqurt has a window that displays a list of status messages as the handshaking takes place between the two Windows CE systems. To use MySqurt, you'll need to have it running on both the Windows CE systems. To transfer a file, enter the name of the file you want from the other system and tap on the Get File button. The system transmits the request to the system and, if the file exists, it will be sent back to the requesting system. The MySqurt window is shown in Figure 10-4. The source code for the example is shown in Figure 10-5.



```
\tictac1.exe
```

```
1 device(s) found.
connected...
received file size of 6144 bytes
free space of 5602460 bytes
Sending size ack.
recv'd 2048 bytes.
recv'd 2048 bytes.
recv'd 2048 bytes.
receive finished
```

Figure 10-4. *The MySqurt window after a file has been transferred.*

MySqurt.rc

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

Figure 10-5. *The MySqurt source.*

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

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

//----------------------------------------------------------------------
// Main window dialog template
//
MySqurt DIALOG discardable 10, 10, 130, 110
STYLE  WS_OVERLAPPED | WS_VISIBLE | WS_CAPTION | WS_SYSMENU |
       DS_CENTER | DS_MODALFRAME
CAPTION "MySqurt"
CLASS "MySqurt"
BEGIN
    LTEXT "&File:"                      -1,   2,  11,  15,  12
    EDITTEXT                  IDD_OUTTEXT,  17,  10,  71,  12,
                                          WS_TABSTOP | ES_AUTOHSCROLL
    PUSHBUTTON "&Get File"    IDD_GETFILE,  92,  10,  34,  12, WS_TABSTOP

    LISTBOX                   IDD_INTEXT,   2,  25, 124,  80,
                                          WS_TABSTOP | WS_VSCROLL
END
```

MySqurt.h

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

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

(continued)

Figure 10-5. *continued*

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

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

#define  ID_ICON             1

#define  IDD_INTEXT          10                   // Control IDs
#define  IDD_GETFILE         11
#define  IDD_OUTTEXT         12

// Error codes used by transfer protocol
#define GOOD_XFER          0
#define BAD_FILEOPEN       -1
#define BAD_FILEMEM        -2
#define BAD_FILEREAD       -3
#define BAD_FILEWRITE      -3
#define BAD_SOCKET         -4
#define BAD_SOCKETRECV     -5
#define BAD_FILESIZE       -6
#define BAD_MEMORY         -7

#define BLKSIZE            2048                     // Transfer block size

//----------------------------------------------------------------------
// Function prototypes
//
int ServerThread (PVOID pArg);
int SenderThread (PVOID pArg);
int GetFile (HWND hWnd, TCHAR *szFileName);

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

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

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

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

// Thread functions
int SenderThread (PVOID pArg);
int ReaderThread (PVOID pArg);
```

MySqurt.c

```
//======================================================================
// MySqurt - A simple IrSock application for Windows CE
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//======================================================================
#include <windows.h>                   // For all that Windows stuff
#include <winsock.h>                   // socket includes
#include <af_irda.h>                   // IrDA includes

#include "MySqurt.h"                   // Program-specific stuff
//----------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("MySqurt");
const char chzAppName[] = "MySqurt";
HINSTANCE hInst;                       // Program instance handle
HWND hMain;                            // Main window handle
BOOL fContinue = TRUE;                 // Server thread cont. flag

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_COMMAND, DoCommandMain,
    WM_DESTROY, DoDestroyMain,
};
// Command Message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDOK, DoMainCommandGet,
    IDCANCEL, DoMainCommandExit,
    IDD_GETFILE, DoMainCommandGet,
};
//======================================================================
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
```

(continued)

Figure 10-5. *continued*

```
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;

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

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

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        if ((hMain == 0) || !IsDialogMessage (hMain, &msg)) {
            TranslateMessage (&msg);
            DispatchMessage (&msg);
        }
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//----------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;
    HWND hWnd;

    // If previous instance, activate it instead of us.
    hWnd = FindWindow (szAppName, NULL);
    if (hWnd) {
        SetForegroundWindow (hWnd);
        return -1;
    }
    // Register application main window class.
    wc.style = 0;                             // Window style
    wc.lpfnWndProc = MainWndProc;             // Callback function
    wc.cbClsExtra = 0;                        // Extra class data
    wc.cbWndExtra = DLGWINDOWEXTRA;           // Extra window data
    wc.hInstance = hInstance;                 // Owner handle
    wc.hIcon = NULL,                          // Application icon
    wc.hCursor = NULL;                        // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (LTGRAY_BRUSH);
    wc.lpszMenuName =  NULL;                  // Menu name
    wc.lpszClassName = szAppName;              // Window class name

    if (RegisterClass (&wc) == 0) return 1;
    return 0;
}
//----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow){
    HWND hWnd;
    HANDLE hThread;
    INT rc;

    hInst = hInstance;                    // Save program instance handle.

    // Create main window.
    hWnd = CreateDialog (hInst, szAppName, NULL, NULL);
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Create secondary threads for interprocess comm.
    hThread = CreateThread (NULL, 0, ServerThread, hWnd, 0, &rc);
    if (hThread == 0) {
        DestroyWindow (hWnd);
        return 0;
    }
    CloseHandle (hThread);

    ShowWindow (hWnd, nCmdShow);          // Standard show and update calls
    UpdateWindow (hWnd);
    SetFocus (GetDlgItem (hWnd, IDD_OUTTEXT));
    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) {
```

(continued)