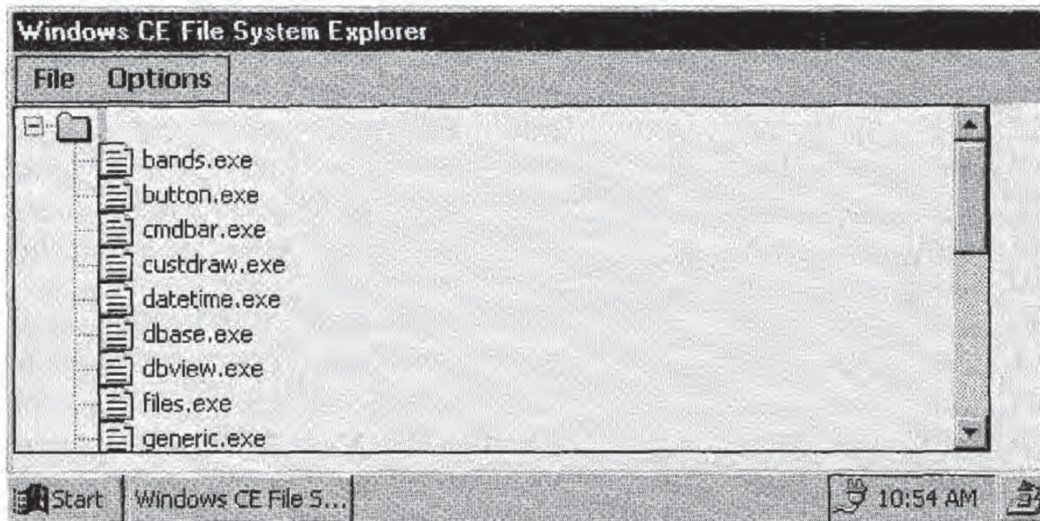


Figure 6.1 The File System Explorer application.

The FILESYS.EXE application allows users to create, delete, and rename files and directories. It also provides very primitive file editing. File properties such as file attributes and file size can also be determined using the application.

Most of these features are accessed through the Options menu shown in Figure 6.2.

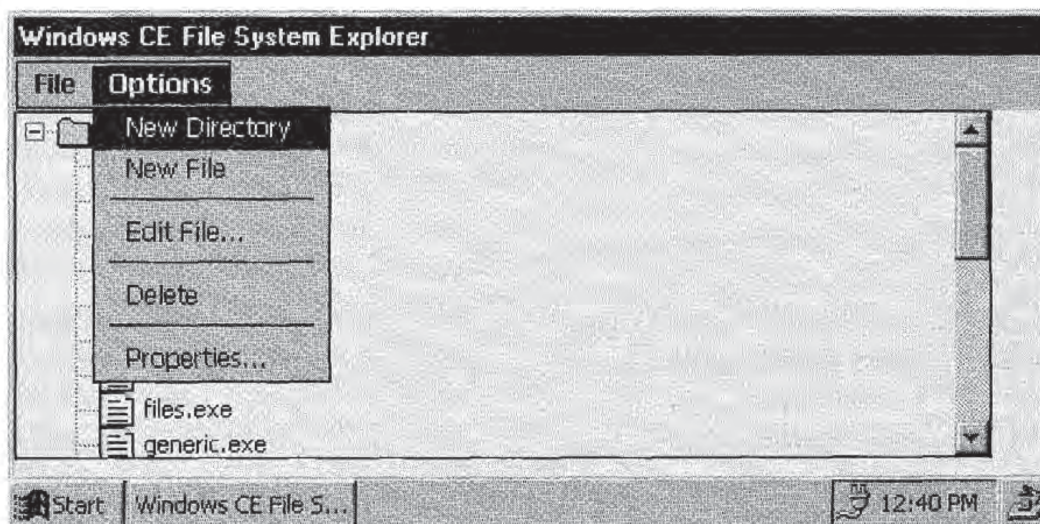


Figure 6.2 The File System Explorer Options menu.

Creating and Deleting Files and Directories

To create a new directory with the FILESYS.EXE application, first select the *parent directory* under which the new directory is to be located. (The term *parent directory* is used to define the directory under which some specific directory or file is located.) To select a file or directory, tap the name or icon of that file or directory in the tree view display.

After selecting the directory that is to contain the new directory, select the New Directory option from the Options menu (see Figure 6.2). For example, to create a new directory under the root directory, select the root directory icon and select the New Directory menu option. This operation results in the creation of a new directory called Empty Folder, as shown in Figure 6.3. Note the Empty Folder icon.

New files are created in much the same way. To create a file under a particular directory, select the directory and then choose the New File option under the Options menu. A file called Empty File will appear. Deleting files and directories is straightforward. Simply select the file or directory you wish to delete, and then select the Delete option from the Options menu.

Renaming Files and Directories

Another common file system operation is renaming files or directories. The names Empty Folder and Empty File are not very useful for real-

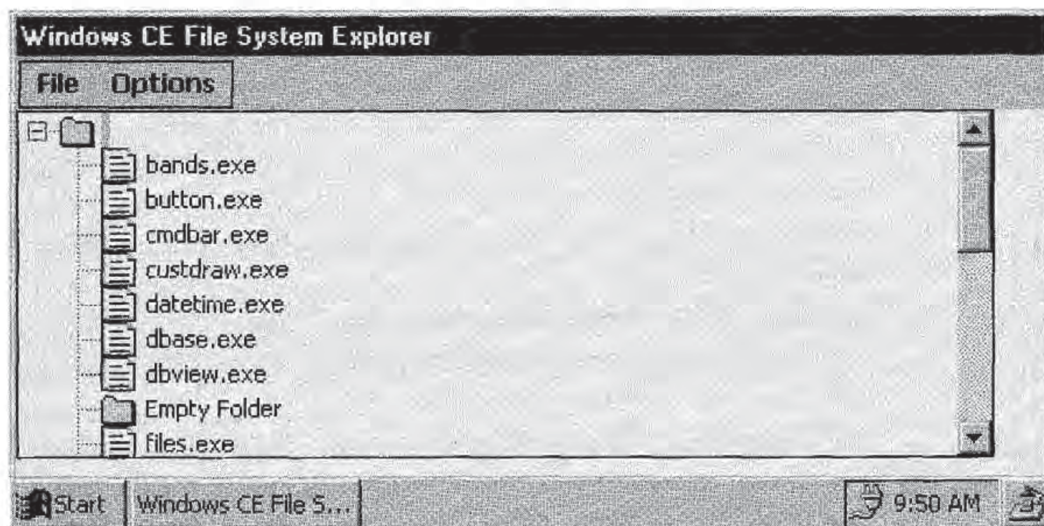


Figure 6.3 Creating a new directory.

world directories and files. To rename a file or directory in the File System Explorer application, double-tap on the file or directory to be renamed. A small text entry field appears containing the name of the selected file or directory. You can then type the new file or directory name in this field. Press the Enter key to make the name change take effect. If the application is running on a Windows CE device that does not have a keyboard, simply tap a different part of the screen and your change will take effect.

As an example, let's say that you want to change the name of the Empty Folder directory in Figure 6.3 to Acme Corp. Simply double-tap on the name Empty Folder, and type Acme Corp in the edit field. The File System Explorer display should then look as shown in Figure 6.4.

Files are renamed the same way. Double-tap the file you wish to rename and type the new name in the edit field that appears.

For example, let's create a file called Expenses under the Acme Corp directory. Tap the Acme Corp directory to select it, and then choose New File from the Options menu. A new file, called Empty File, appears under the Acme Corp directory as shown in Figure 6.5.

Rename this file by tapping it twice and then typing the name Expenses into the edit field that appears. The contents of the Acme Corp directory will then appear as shown in Figure 6.6.

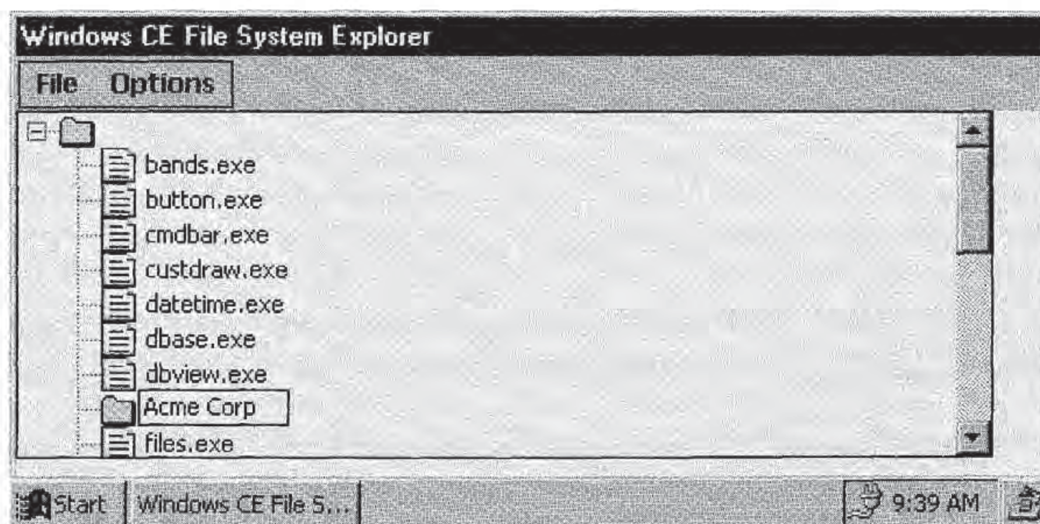


Figure 6.4 Renaming a directory.

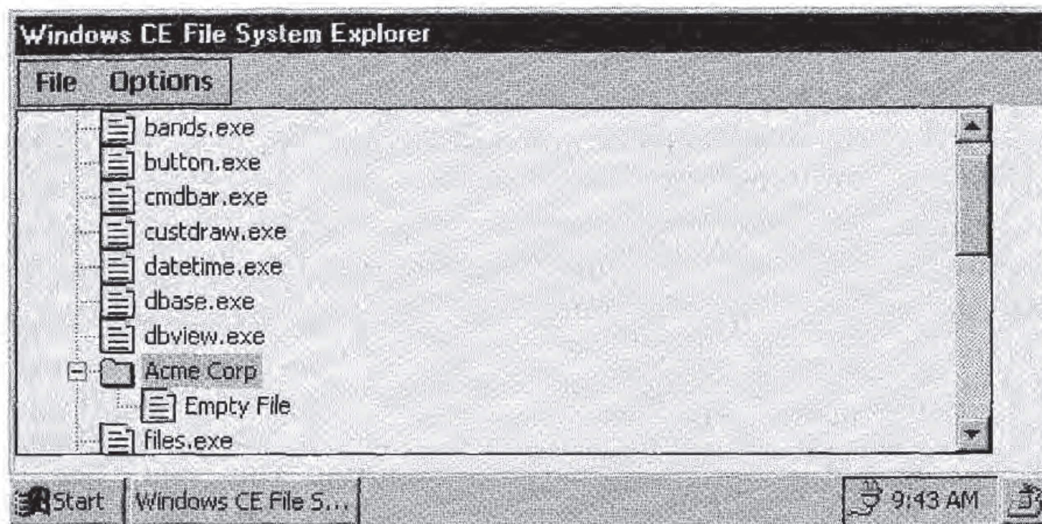


Figure 6.5 Creating a new file.

Editing Files

Another important set of file system functions we will cover in this chapter lets your applications write data to and read data from files. To demonstrate these features, the File System Explorer application provides very rudimentary file editing capabilities. You will not be tempted to delete your current word processing software from your Handheld PC when you see this feature. However, after finishing this

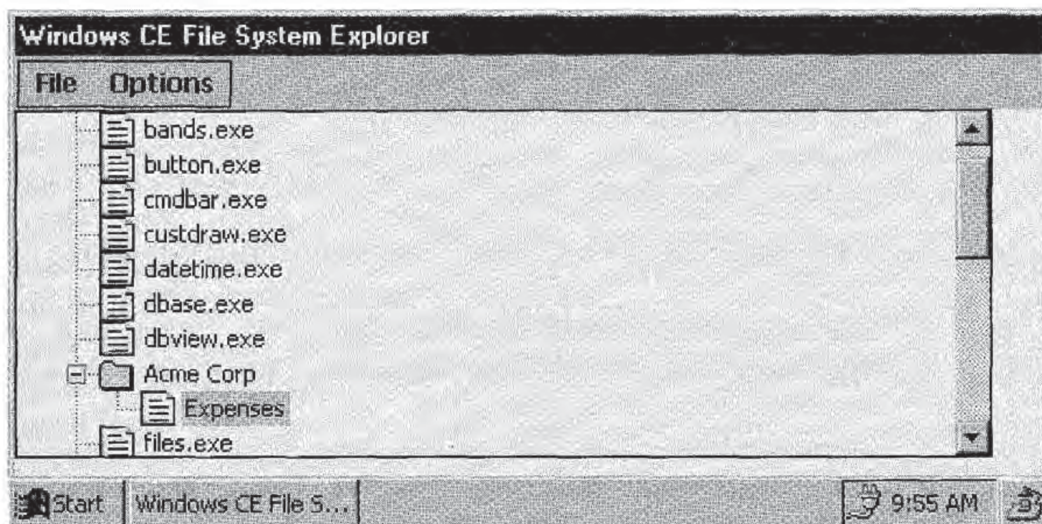


Figure 6.6 Renaming a file.

chapter you will know how to use the file system API to read and write files.

To edit a file in the FILESYS.EXE application, select the file you wish to edit by tapping it once. Then press the Enter key on your keyboard. (For devices that do not include a keyboard, you can use the Edit File menu option in the Options menu to invoke this feature.) The dialog box shown in Figure 6.7 appears.

Text entered in the edit field of this dialog is written to the file if the user presses the Save button. Pressing Cancel aborts file editing.

Examining File Properties

The final notable feature of the File System Explorer application is invoked by the Properties option of the Option menu. Choosing this menu option retrieves and displays various features of the currently selected file. The dialog box shown in Figure 6.8 appears, showing which file attributes are set for the selected file, as well as the size of the file in bytes.

File Handles

The Windows CE file system API functions access files and directories by means of a *file handle*. Like any other Windows CE handle type,

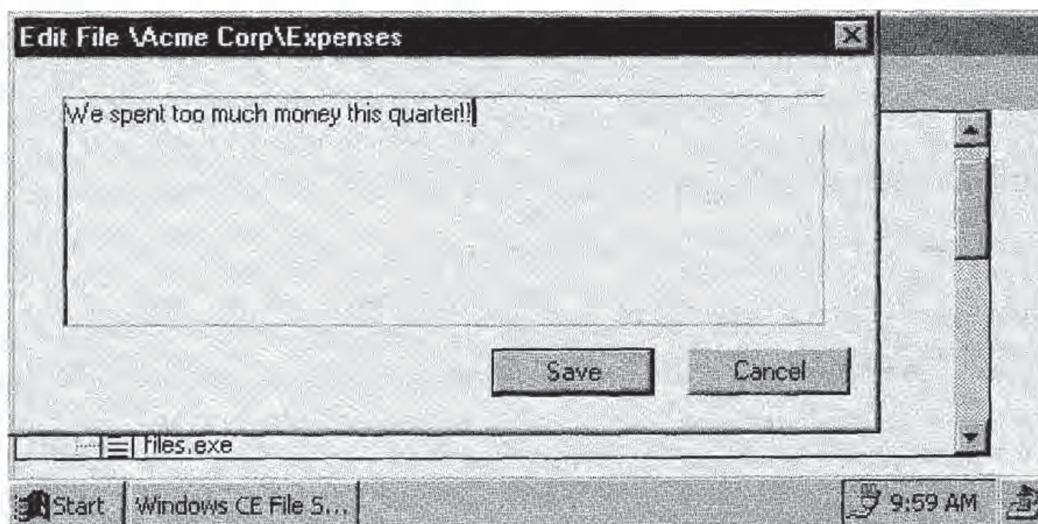


Figure 6.7 Editing a file with the file system explorer.

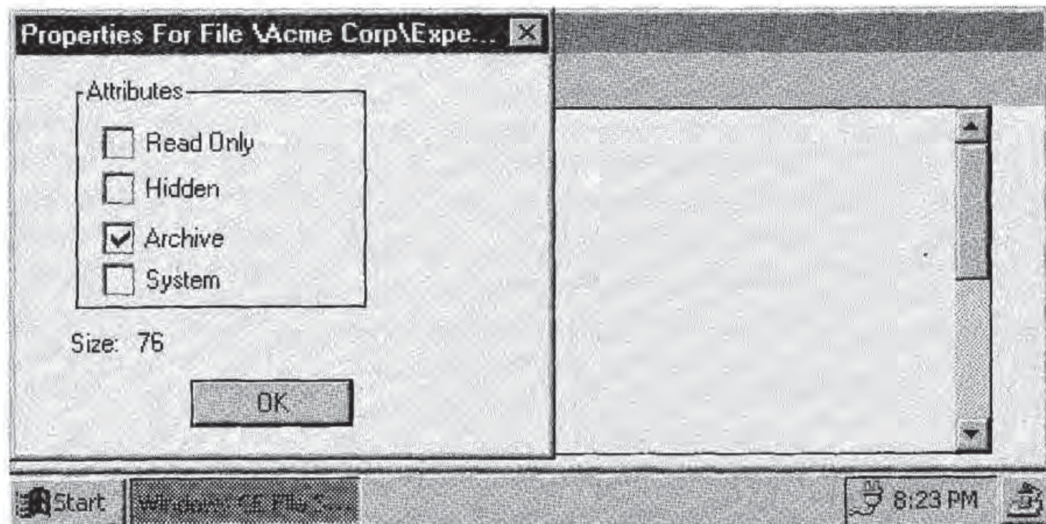


Figure 6.8 Displaying the properties of a file.

such as a window handle, a file handle is an identifier for referencing an object managed by the Windows CE kernel. In this case the object is a file or directory.

As we will see, the function used for opening and creating files returns a handle to the specified file. Functions for reading and writing files require a file handle in order to access the right file. In short, any operation that a Windows CE application may perform on a file or directory requires a valid file handle.

File Attributes

Every file in the Windows CE file system has one or more *attributes*. These attributes are used to distinguish files in terms of characteristics such as how they can be used and by whom. Under Windows CE, files may have one or more of the attributes listed in Table 6.1.

NOTE

File Attributes

Under Windows CE, the FILE_ATTRIBUTE_OFFLINE and FILE_ATTRIBUTE_TEMPORARY file attributes are not supported.

An application typically sets the attributes of a file when the file is created. However, it may be necessary to change or determine the attrib-

Table 6.1 Windows CE File Attributes

ATTRIBUTE	MEANING
FILE_ATTRIBUTE_ARCHIVE	Used by applications to mark a file that has not been backed up.
FILE_ATTRIBUTE_COMPRESSED	File or directory is compressed. For files, this means that all of the data in the file is compressed. For directories, this means that by default all files or subdirectories created in this directory are created with the compressed attribute.
FILE_ATTRIBUTE_HIDDEN	The file is marked as hidden.
FILE_ATTRIBUTE_NORMAL	This attribute cannot be used with any other attribute. Hence, if set, it means that no other attribute is set.
FILE_ATTRIBUTE_READONLY	Applications can only read this file. They cannot write to or delete it.
FILE_ATTRIBUTE_DIRECTORY	Indicates that the particular file is a directory. Note: This attribute cannot be set. It can be returned by GetFileAttributes.
FILE_ATTRIBUTE_SYSTEM	Indicates the file is a system file, i.e., it is intended to be used only by the operating system.
FILE_ATTRIBUTE_INROM	Indicates the file is a read-only operating system file stored in ROM.
FILE_ATTRIBUTE_ROMMODULE	Indicates the file is an in-ROM DLL or EXE.

utes of a file after it has been created. Applications might even need to determine the attributes of files they did not create.

The Windows CE file system API provides two functions to read and modify the attributes of a file, *GetFileAttributes* and *SetFileAttributes*. The first of these functions has the following form:

```
GetFileAttributes(lpFileName);
```

This function takes the Unicode string name of the file of interest in the parameter *lpFileName*. If successful, it returns a DWORD containing the file attributes that are set for the file. The return value is the bitwise OR of one or more of the file attribute values specified in Table 6.1. In addition, Windows CE provides for two additional return values for this function, FILE_ATTRIBUTE_INROM and FILE_ATTRIBUTE_ROMMODULE. The first of these indicates that the file in

question is a read-only operating system file stored in ROM. The second indicates that the file is a DLL or executable (.EXE) file stored in ROM and intended to execute in place. This means that files with the FILE_ATTRIBUTE_ROMMODULE attribute do not need to be copied into RAM in order to run. Files of this type are typically libraries and applications that ship with the Windows CE operating system.

The attributes of a file can be set using the *SetFileAttributes* function:

```
SetFileAttributes(lpFileName, dwFileAttributes);
```

This function returns TRUE if the attributes are successfully set, and FALSE if the function is unsuccessful.

As an example, let's assume that we want to mark as hidden all files that are read-only. The piece of code responsible for testing if a file is read-only and then setting the hidden file attribute would look something like this:

```
//File name is in lpFileName
DWORD dwAttributes;
dwAttributes = GetFileAttributes(lpFileName);
if (dwAttributes & FILE_ATTRIBUTE_READONLY)
{
  dwAttributes |= FILE_ATTRIBUTE_HIDDEN;
  SetFileAttributes(lpFileName, dwAttributes);
}
```

Note that as in Windows NT and Windows 98, *SetFileAttributes* cannot be used to set the FILE_ATTRIBUTE_COMPRESSED attribute of a file. If this attribute is not set when the file is created, you must use the *DeviceIoControl* function to set it.

Searching for Files

It may seem a little strange to discuss searching for files in the Windows CE file system this early in the chapter. Certainly, operations such as creating and deleting files must be more fundamental than file searching!

While this may be true in some sense, it is also the case that many file operations are iterative. For example, deleting a directory requires recursively deleting all files and subdirectories contained by the direc-

tory to be deleted. Such an iterative process entails file searching operations to look for files to delete.

As another example, consider how an application might determine if a particular directory is empty. There is no Windows CE API function *IsDirectoryEmpty*. Writing such an operation from scratch involves searching the directory in question to see if it contains any files.

In fact, file searching is so fundamental to many file system operations that understanding how these features are implemented requires that we first understand file searching under Windows CE.

Windows CE provides three functions for such operations: *FindFirstFile*, *FindNextFile*, and *FindClose*.

The first two of these functions return a data structure containing information about the files that they retrieve. Before discussing the find functions in detail, let's first look at this structure.

NOTE

FINDFIRSTFILEEX

The function *FindFirstFileEx* is not supported under Windows CE.

The WIN32_FIND_DATA Structure

The WIN32_FIND_DATA structure is used by Windows CE to provide information about a file located by one of the find functions.

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

The members of this structure provide all descriptive information about the file, either directly, or by providing a means for extracting more information (such as through the *dwOID* member).

dwFileAttributes contains the attributes of the file as described previously in Table 6.1. *ftCreationTime*, *ftLastAccessTime*, and *ftLastWriteTime* represent the times the file was created, last accessed, and last written to, respectively. *nFileSizeHigh* and *nFileSizeLow* are the high-order and low-order words of the total size of the file. The *dwOID* member contains the object identifier of the file. This means that whenever an application can get WIN32_FIND_DATA about a file, it can also get any of the CEFILEINFO data about the file with a simple call to *CeOidGetInfo*. Finally, *cFileName* is a null-terminated string containing the name of the file.

We will primarily be interested in how to use the *FindFirstFile* and *FindNextFile* functions to get WIN32_FIND_DATA information. It should be noted that Windows CE provides some additional functions for quickly accessing some of the data provided by this structure. In particular, *GetFileTime* retrieves the same information as provided by the FILETIME members of the WIN32_FIND_DATA structure. *GetFileSize* returns the size of a specified file.

In addition, there is a *SetFileTime* function to allow applications to modify the creation time, last access time, and last write time of a file.

Because of its similarity to the WIN32_FIND_DATA structure, I somewhat parenthetically mention the BY_HANDLE_FILE_INFORMATION structure. This is another data structure that contains much the same information about a file as WIN32_FIND_DATA. Given a handle to an open file, an application can get a BY_HANDLE_FILE_INFORMATION structure by calling *GetFileInformationByHandle*.

The *FindFirstFile* and *FindNextFile* Functions

The *FindFirstFile* function is used to locate a specific file or directory. It can also be used to find the first file in a specified directory.

```
FindFirstFile(lpFileName, lpFindFileData);
```

The *lpFileName* parameter contains a path and file name, or a directory name. *lpFindFileData* is used as a return value by the function. It is a pointer to a WIN32_FIND_DATA structure containing information about the located file.

If successful, *FindFirstFile* returns a *search handle*. This handle references an internal structure that is responsible for keeping track of the

progress of a file search. We will see the utility of this search handle a bit later when we discuss the *FindNextFile* function. If *FindFirstFile* fails, it returns INVALID_HANDLE_VALUE.

The *lpFileName* parameter accepts wildcards. This is how you can tell *FindFirstFile* to differentiate between finding the first file in a specified directory and finding the directory itself.

For example, this line of code will try and find a directory called \MyFiles:

```
HANDLE hFile;
WIN32_FIND_DATA fd;
memset(&fd, 0, sizeof(fd));
hFile = FindFirstFile(TEXT("\\MyFiles"), &fd);
```

On the other hand, one subtle change to the *FindFirstFile* call will find the first file in the \MyFiles directory:

```
hFile = FindFirstFile(TEXT("\\MyFiles\\*"), &fd);
```

FindNextFile is used to continue a search started by *FindFirstFile*. For example, if *FindFirstFile* finds the first file in a specified directory, *FindNextFile* will attempt to find the next file in that directory. Each successive call to *FindNextFile* uses the search handle that is updated with each find operation to keep track of the search progress. The syntax of *FindNextFile* is:

```
FindNextFile(hFindFile, lpFindFileData);
```

The first parameter is the search handle returned by a previous call to *FindFirstFile*. The second parameter is the WIN32_FIND_DATA return value, just as in *FindFirstFile*.

FindNextFile returns TRUE if successful and FALSE if it fails. As with all of the file system functions, a call to *GetLastError* can be used to get additional information about why the function call failed if the return value is FALSE.

Creating and Opening Files and Directories

As under Windows NT, creating files and directories under Windows CE is done using the *CreateFile* and *CreateDirectory* functions. Under Windows CE, files are also opened using the *CreateFile* function, as we will soon see.

Creating and Opening Files

To create a file, your application calls the *CreateFile* function:

```
CreateFile(lpFileName,dwDesiredAccess,dwShareMode,
  lpSecurityAttributes, dwCreationDistribution,
   dwFlagsAndAttributes, hTemplateFile);
```

lpFileName is the null-terminated Unicode string file name of the file to be created. Long file names are supported.

dwDesiredAccess is used to indicate the access, or read-write mode, of the file. It can be any combination of the following values:

0. Specifies device query access. This allows an application to query device attributes.

GENERIC_READ. Specifies that the file is created/opened with read access.

GENERIC_WRITE. Specifies that the file is created/opened with write access.

For example, to open a file called "myfile.txt" with read-write access, an application would do the following:

```
CreateFile(TEXT("myfile.txt"),
(GENERIC_READ|GENERIC_WRITE),...);
```

The third parameter to *CreateFile, dwShareMode*, is used to specify if and how the file can be shared. It can be a combination of one or more of the following values:

0. Indicates that the file cannot be shared

FILE_SHARED_READ. Subsequent open operations on the file will only succeed if read access is requested (via the *dwDesiredAccess* parameter).

FILE_SHARED_WRITE. Subsequent open operations on the file will only succeed if write access is requested.

Under Windows CE, file security attributes are ignored. The *lpSecurity-Attributes* parameter should therefore be set to NULL.

The *dwCreationDistribution* parameter controls how the *CreateFile* function behaves when attempting to create existing files, as well as what to do when the function tries to open a nonexistent file. This parameter can be one of the following:

CREATE_NEW. The function creates a new file. If the specified file already exists, the *CreateFile* function fails.

CREATE_ALWAYS. The function creates a new file. If the specified file already exists, it is overwritten by the *CreateFile* operation.

OPEN_EXISTING. The function opens an existing file. If the specified file does not exist, *CreateFile* fails.

OPEN_ALWAYS. The function opens an existing file. If the specified file does not already exist, it is created.

TRUNCATE_EXISTING. The function opens the file, but truncates it to zero length. The file must be opened with GENERIC_WRITE access. *CreateFile* fails if the specified file does not exist.

By taking a look at the allowed *dwCreationDistribution* values, we can see how *CreateFile* can be used to both create new files and open existing files. For example, it is common to want to open a file or have the operating system create a file of that name if it does not exist, as follows:

```
CreateFile(TEXT("myfile.txt"),
GENERIC_READ|GENERIC_WRITE, 0,
  NULL, OPEN_ALWAYS,...);
```

dwFlagsAndAttributes determines the file attributes and several operating modes. We have already discussed file attributes. The flags portion can be any combination of the following values:

FILE_FLAG_WRITE_THROUGH. Instructs Windows CE to write directly to the object store when writing to the specified file, as opposed to writing through any intermediate cache.

FILE_FLAG_RANDOM_ACCESS. The file supports random access.

Most of the flags that are supported under Windows NT are not supported under Windows CE. Also note that the SECURITY_SQOS_PRESENT flag, or any of the other values that can be used with it under Windows NT, are not supported under Windows CE.

Finally, the *hTemplateFile* parameter is ignored under Windows CE and should be set to NULL.

If *CreateFile* is successful, a handle to the open file is returned. If it fails, the return value is INVALID_HANDLE_VALUE.

Open files are closed using the *CloseHandle* function:

```
CloseHandle(hObject);
```

where *hObject* is the handle of the file to close.

Creating Directories

Creating a directory is accomplished with the *CreateDirectory* function. The *CreateDirectoryEx* function available under Windows NT is not supported in Windows CE. The *CreateDirectory* function syntax is:

```
CreateDirectory(lpPathName, lpSecurityAttributes);
```

lpPathName is a null-terminated Unicode string specifying the path of the directory to be created. The maximum allowed length of this name is the operating system–defined value MAX_PATH. The second parameter to this function is ignored, as Windows CE does not support file security attributes. *lpSecurityAttributes* therefore should be set to NULL.

If *CreateDirectory* is successful, it returns TRUE. If unsuccessful, it returns FALSE. An application can get more detailed information about why the function failed by calling *GetLastError*.

An Example

As an example, let's take a look at how the File System Explorer application creates new files and directories. These features are triggered by the New Directory and New File menu options, so the first code to look at is the command handlers in the main window procedure for these two menu options (see Figure 6.2).

The pertinent sections of the window procedure are shown below. Note that *tviCurSel* contains the currently selected tree view item TV_ITEM structure. The *lParam* member of this structure always contains the CEOID object identifier of the file or directory corresponding to the currently selected tree view item.

```
LRESULT CALLBACK WndProc(
  HWND hwnd,
  UINT message,
  WPARAM wParam,
  LPARAM lParam)
{
  CEOID oid;
  CEOIDINFO oidInfo;
```

```
TCHAR *pszFileName, *pszDirectoryName;
switch (message)
{
case WM_COMMAND:
 UINT nID;
 nID = LOWORD(wParam);
 switch(nID)
 {
 case IDC_NEWDIRECTORY:
  //Create a new directory
  oid = (CEOID)tviCurSel.lParam;
  CeOidGetInfo(oid, &oidInfo);
  if (OBJTYPE_DIRECTORY==oidInfo.wObjType)
  {
   pszFileName = NULL;
   pszDirectoryName = TEXT("Empty Folder");
  }
  else
  {
   MessageBox(NULL, TEXT("Files cannot have children"),
    TEXT("New Folder Error"),MB_OK|MB_ICONEXCLAMATION);
   return (0);
  }
  OnNew(pszFileName, pszDirectoryName, tviCurSel.hItem,
   oidInfo, TRUE);
  break;
 case IDC_NEWFILE:
  /Create a new file
  oid = (CEOID)tviCurSel.lParam;
  CeOidGetInfo(oid, &oidInfo);
  if (OBJTYPE_DIRECTORY==oidInfo.wObjType)
  {
   pszDirectoryName = oidInfo.infDirectory.szDirName;
   pszFileName = TEXT("Empty File");
  }
  else
  {
   MessageBox(NULL, TEXT("Files cannot have children"),
    TEXT("New Folder Error"),MB_OK|MB_ICONEXCLAMATION);
   return (0);
  }
  OnNew(pszFileName, pszDirectoryName,
   tviCurSel.hItem, oidInfo, FALSE);
  break;
```

In both the case of creating a new file and creating a new directory, the application first extracts the CEOIDINFO for the currently selected file or directory. A request to create a new file or directory will force the application to try and create the file or directory with the currently selected item as its parent.

Obviously, this only makes sense if the currently selected object is a directory. Files cannot contain other files. Only directories can contain other files or directories. For this reason, both the IDC_NEWDIRECTORY and IDC_NEWFILE case statement code blocks check the *wObjType* member of the object information structure. In either of these cases, if the currently selected file system object is not a directory, a warning message is displayed and the operation is aborted.

If the user is trying to create a new file or directory under an existing directory, however, the appropriate default name is assigned to *pszFileName* or *pszDirectoryName*, and the application defined *OnNew* function is called. In the case of a request to create a new directory, the value "Empty Folder" is assigned to *pszDirectory*. In the case of a new file creation request, the name "Empty File" is assigned to *pszFileName*.

The *OnNew* function contains a lot of code for adding new items to the tree view control in response to new file and directory creations. This code is left out so that we can concentrate on the parts of the function that relate directly to the file system API. Also only the part of this function which creates new files is shown. Since the section that creates new directories is very similar, it was left out for the sake of brevity.

```
BOOL OnNew(TCHAR* pszFileName,
  TCHAR* pszDirectoryName,
  HTREEITEM hParent,
  CEOIDINFO oidInfo,
  BOOL bIsDirectory)
{
  TCHAR pszFullName[MAX_PATH];
  HANDLE hFile;
  wsprintf(pszFullName, TEXT("%s\\%s"),
   pszDirectoryName, pszFileName);
  hFile = FindFirstFile(pszFullName, &fd);
  if (INVALID_HANDLE_VALUE==hFile)
  {
   /File is new
   FindClose(hFile);
   hFile = CreateFile(pszFullName,
    GENERIC_READ|GENERIC_WRITE, 0, NULL,
     CREATE_NEW, FILE_ATTRIBUTE_ARCHIVE, NULL);
  }
  else
  {
   /File already exists
   MessageBox(NULL,
    TEXT("File \"Empty File\" Already Exists"),
```

```
        TEXT("Create New File Error"),
         MB_ICONEXCLAMATION|MB_OK);
     return (FALSE);
   }
   return (TRUE);
}
```

The arguments *pszFileName* and *pszDirectoryName* are the name of the file to be created and the parent directory name, respectively. *hParent* is the tree view item corresponding to the parent directory in the user interface. *oidInfo* is the CEOIDINFO structure containing information about the parent directory. *bIsDirectory* indicates whether a new file or a new directory is to be created by the function.

To create a directory or file, *CreateFile* must be passed the complete path name of the directory or file to be created. *OnNew,* therefore, first constructs the full path name in the variable *pszFullName*. To do this, *OnNew* only needs to concatenate the directory name contained in *pszDirectoryName* with the file name in *pszFileName.* This is the purpose of the *wsprintf* call at the beginning of the function. A "\" character is inserted between the parent directory name and the file name.

After the complete new file path name has been constructed, *OnNew* checks to see if the specified file already exists. It does so by calling *FindFirstFile*:

```
hFile = FindFirstFile(pszFullName, &fd);
```

pszFullName contains the full path name of the file to be created. If this file does not exist, *FindFirstFile* will return INVALID_HANDLE_VALUE. Otherwise it returns the handle of the existing file.

If the file does not exist (i.e., if *hFile* equals INVALID_HANDLE_VALUE,) *OnNew* closes the search handle *hFile* and creates the new file. If the file already exists, a message to this effect is displayed for the user and the function *OnNew* returns without creating a new file.

Reading and Writing File Data

File read and write operations are closely linked to the concept of the *file pointer*. A file pointer marks the current position in a given file. Read operations read data from the file's current position. Write operations write data to the file at the position indicated by the file pointer.

Files also have an *end of file* marker. This marker indicates the last byte of data in the file. As such, the end of file marker also determines the size of the file. As file write operations increase the size of a file, they move this end of file marker. Hence there is no such thing as writing past the end of a file: files grow to accommodate the data being written to them.

Files access can be either *sequential* or *random*. Sequential access means that data is read from the file in order. Random access means that data can be read from the file in any order as determined by the application reading the file. For random access to be possible, there must be a way for applications to manually set the file pointer without requiring read or write operations to occur. We will introduce such functions later in this chapter.

NOTE

ASYNCHRONOUS FILE ACCESS

Under Windows NT, file access operations can be synchronous or asynchronous. Windows CE however does *not* support asynchronous access.

The *ReadFile* Function

Data is read from a file using the *ReadFile* function:

```
ReadFile(hFile, lpBuffer, nNumberOfBytesToRead,
  lpNumberOfBytesRead, lpOverlapped);
```

hFile is the handle of the open file from which the data is to be read. *lpBuffer* is a pointer to the data buffer which receives the data. *nNumberOfBytesToRead* specifies the number of bytes of data to read from the file. *lpNumberOfBytesRead* is a pointer to a DWORD used by *ReadFile* to return the actual number of bytes of data read. As Windows CE does not allow files to be created with the FILE_FLAG_OVERLAPPED flag, *lpOverlapped* is not used. *lpOverlapped* should be set to NULL.

ReadFile returns TRUE if the operation is successful, and FALSE if the operation fails. An application can get additional error information in this case by calling *GetLastError*.

ReadFile returns once the number of bytes specified in *nNumberOfBytesToRead* have been read, or when an error occurs. If an application specifies that more bytes be read than the file actually contains, *ReadFile*

will simply read as many as it can and return the actual number of bytes read in *lpNumberOfBytesRead*.

Random Access Files

The *ReadFile* function advances the file pointer *lpNumberOfBytesRead*. This accounts for the default sequential nature of file access. To implement random file access, your applications must be able to control the position of the file pointer manually. This can be done using the *SetFilePointer* function. *SetFilePointer* can be used to move a file pointer by specifying a 64-bit number representing the number of bytes the pointer is to be moved.

The syntax of *SetFilePointer* is:

```
SetFilePointer(hFile, lDistanceToMove,
  lpDistanceToMoveHigh, dwMoveMethod);
```

hFile is the handle of the open file whose pointer is to be moved. *lDistanceToMove* specifies the low order word of the number of bytes to move the file pointer. This value can be negative, in which case the file pointer is moved backward. *lpDistanceToMoveHigh* is a pointer to the high order word of the number of bytes to move the file pointer. This parameter is also used by *SetFilePointer* to return the high order word of the new file pointer position.

The *dwMoveMethod* parameter indicates the starting point of the move operation. It can be one of the following three values:

FILE_BEGIN. The starting point is the beginning of the file. In this case, the distance to move is interpreted as the unsigned pointer location.

FILE_CURRENT. The starting point is the current file pointer position.

FILE_END. The starting point is the end of file position.

If the function succeeds, it returns the low order word of the new file pointer position. If *lpDistanceToMoveHigh* was not NULL, this parameter will return the high order word of the new position. If *SetFilePointer* fails, the return value is –1 and *lpDistanceToMoveHigh* is NULL.

Random access of Windows CE files can therefore be accomplished by first specifying FILE_FLAG_RANDOM_ACCESS as one of the *dwFlagsAndAttributes* values when creating or opening the file with *CreateFile*. *SetFilePointer* is then called to manually position the file pointer for read and write operations.

An application may also want to change the position of the end of file marker of a particular file. This is done using the *SetEndOfFile* function:

```
SetEndOfFile(hFile);
```

This functions moves the end of file marker of the file specified by the file handle *hFile* to the current file pointer position of that file. If successful, this function returns TRUE. Otherwise it returns FALSE.

For example, to move the end of file marker to the beginning of a file, an application could do this:

```
//Set file pointer to beginning of file
SetFilePointer(hMyFile, 0, 0, FILE_BEGIN);
//Now set end of file marker
SetEndOfFile(hMyFile);
```

The *WriteFile* Function

Writing data to a file in Windows CE is done with the *WriteFile* function. The syntax and use of this function is very similar to *ReadFile*:

```
WriteFile(hFile, lpBuffer, nNumberOfBytesToWrite,
  lpNumberOfBytesWritten, lpOverlapped);
```

The *WriteFile* parameters have the same meanings as the corresponding *ReadFile* parameters except for *nNumberOfBytesToWrite* and *lpNumberOfBytesWritten*. It isn't much of a stretch to realize that *nNumberOfBytesToWrite* specifies the number of bytes of data to write to the file indicated by *hFile*. Similarly, *WriteFile* returns the actual number of bytes written through the *lpNumberOfBytesWritten* parameter. As with *ReadFile*, the *lpOverlapped* parameter is ignored and should be set to NULL.

To write data to a file, the file must have been opened with GENERIC_WRITE access.

Data can be written to any position in a random access file much as data can be randomly read from a random access file. The file must be created or opened with the FILE_FLAG_RANDOM_ACCESS flag set. Then *SetFilePointer* can be used to specify the file location to which data is written.

The File System Explorer example application of this chapter demonstrates *ReadFile* and *WriteFile*, and *SetFilePointer* operations by means of its very rudimentary file editing feature.

An Example

To gain further insight into how to use the *ReadFile* and *WriteFile* functions, let's look at how the File System Explorer application implements its rudimentary file editing capabilities. See Figure 6.7 for a look at the basic file editor.

The two user operations which invoke the editor are selecting the Edit File menu option and pressing the enter key after selecting a file. Both of these operations cause the following application-defined *OnEdit* function to be called:

```
void OnEdit(HWND hwnd, OIDINFO oidInfo)
{
  if (OBJTYPE_FILE==oidInfo.wObjType)
  {
   DialogBox(ghInst, MAKEINTRESOURCE(IDD_EDITFILE),
    hwnd, (DLGPROC)EditDlgProc);
  }
  else
  {
   MessageBox(NULL, TEXT("You May Only Edit Files"),
    TEXT("Directories May Not Be Edited"),
     MB_OK|MB_ICONEXCLAMATION);
  }
}
```

The parameter *hwnd* is the parent of the dialog box which acts as the file editor. *oidInfo* is the CEOIDINFO structure containing information about the currently selected file or directory.

OnEdit checks the type of the currently selected object and displays an error message if the object is a directory. It doesn't make sense to edit a directory.

On the other hand, if the object is a file, the editor is invoked by the *DialogBox* call. All of the rudimentary file editing functionality is coded in this dialog procedure *EditDlgProc*. The dialog procedure looks like this:

```
BOOL CALLBACK EditDlgProc(
  HWND hwndDlg,
  UINT message,
  WPARAM wParam,
  LPARAM lParam)
{
  TCHAR pszText[MAX_FILE_LENGTH];
  HWND hwndEdit;
  DWORD dwBytes;
```

```
CEOID oid;
CEOIDINFO oidInfo;
switch(message)
{
case WM_INITDIALOG:
 oid = (CEOID)tviCurSel.lParam;
 CeOidGetInfo(oid, &oidInfo);
 pszText[0] = 0;
 hFile = CreateFile(oidInfo.infFile.szFileName,
  GENERIC_READ|GENERIC_WRITE, 0,
   NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
    NULL);
 if (INVALID_HANDLE_VALUE!=hFile)
 {
  dwBytes = 0;
  ReadFile(hFile, pszText,
   oidInfo.infFile.dwLength,
    &dwBytes, NULL);
  hwndEdit = GetDlgItem(hwndDlg, IDC_FILETEXT);
  SetWindowText(hwndEdit, pszText);
  wsprintf(pszText, TEXT("Edit File %s"),
   oidInfo.infFile.szFileName);
  SetWindowText(hwndDlg, pszText);
 }
 return (TRUE);
case WM_COMMAND:
 UINT nID;
 nID = LOWORD(wParam);
 switch(nID)
 {
 case IDOK:
  //Save text to file
  DWORD nBytesToWrite;
  hwndEdit = GetDlgItem(hwndDlg, IDC_FILETEXT);
  GetWindowText(hwndEdit, pszText,
   MAX_FILE_LENGTH);
  SetFilePointer(hFile, 0, 0, FILE_BEGIN);
  nBytesToWrite = lstrlen(pszText)*sizeof(TCHAR);
  WriteFile(hFile, pszText, nBytesToWrite,
   &dwBytes, NULL);
  //Deliberate fall-through
 case IDCANCEL:
  CloseHandle(hFile);
  EndDialog(hwndDlg, nID);
  break;
 default:
  break;
 }      //End of switch(nID) statement
 return (FALSE);
default:
```

```
        return (FALSE);
      }       //End of switch(message) statement
    }
```

When the dialog box is opened, the WM_INITDIALOG message handler is executed. This code gets the CEOIDINFO about the currently selected file via the global variable *tviCurSel*. As we mentioned above, this always contains the TV_ITEM of the currently selected file or directory in the tree view user interface. The file name is extracted from this CEOIDINFO data, and the application attempts to open the specified file with a *CreateFile* call.

If the file exists, *CreateFile* returns the handle of the file. This handle is stored in the global variable *hFile* so that the rest of the dialog procedure has access to it. The contents of the file are read with the *ReadFile* call:

```
ReadFile(hFile, pszText, oidInfo.infFile.dwLength,
  &dwBytes, NULL);
```

The number of bytes that *ReadFile* attempts to read is equal to the length of the file. This is specified by *oidInfo.infFile.dwLength*. The file data is read into the string *pszText*.

Next, the contents of the file are placed in the dialog box edit control by the first *SetWindowText* call:

```
SetWindowText(hwndEdit, pszText);
```

Finally, the dialog box caption is changed to show the name of the file being edited.

The FILESYS.EXE user is now free to edit the file by typing text into the edit control.

To save the new text into the file, the user presses the Save button. This action invokes the IDOK command handler in the *EditDlgProc* dialog procedure. This handler code does exactly the opposite of the WM_INITDIALOG code. The contents of the editor are copied into *pszText* by the *GetWindowText* call. Next, the file pointer is set back to the beginning of the file, and the text is written to the file via *WriteFile*.

Execution then falls through to the IDCANCEL handler, which closes the file and the dialog box. This is the same code that is executed when the user presses the Cancel button.

Copying and Renaming Files and Directories

Files and directories on Windows CE–based devices, much like their counterparts on Windows NT, are often used by users of the devices to organize data and documents. Let's say that you are writing a specification (on your Handheld PC, of course!) for a new suite of Windows CE applications. This specification might consist of several files, such as functional and design specifications for each application in the suite.

If you are like most of us, the organization of this specification will change as your understanding of the required behavior of the applications you are designing evolves. Therefore, the locations and names of the files that make up your application suite specification are unlikely to be the same when the specification is complete as they were when the files were created.

Copying files and directories, renaming them, and moving them around in a directory tree are all very common file operations. Windows CE supports these operations with the *CopyFile* and *MoveFile* functions. The *MoveFileEx* function found in Windows NT is not supported under Windows CE.

The *CopyFile* Function

Copying a file in Windows CE simply requires that an application know the name of the file to be copied and the name of the file to which it is to be copied:

```
CopyFile(lpExistingFileName, lpNewFileName, bFailIfExist);
```

The *lpExistingFileName* parameter contains the null-terminated string name of the file to be copied. *lpNewFileName* is the name of the file to which *CopyFile* copies the original file.

bFailIfExist tells *CopyFile* what to do if a file named *lpNewFileName* already exists. If this parameter is TRUE and the new file already exists, *CopyFile* fails and returns FALSE. If *bFailIfExist* is FALSE in this same scenario, *CopyFile* does not fail. In this case it overwrites the existing file named *lpNewFileName*.

Another effect of copying a file is that all of the attributes (FILE_ATTRIBUTE_HIDDEN, etc.) of the file are copied to the new file.

The *MoveFile* Function

At first glance, the *MoveFile* function seems to have been inappropriately named. *MoveFile* actually *renames* files and directories in the Windows CE file system. It might seem that *RenameFile* would be a more accurate name for this function.

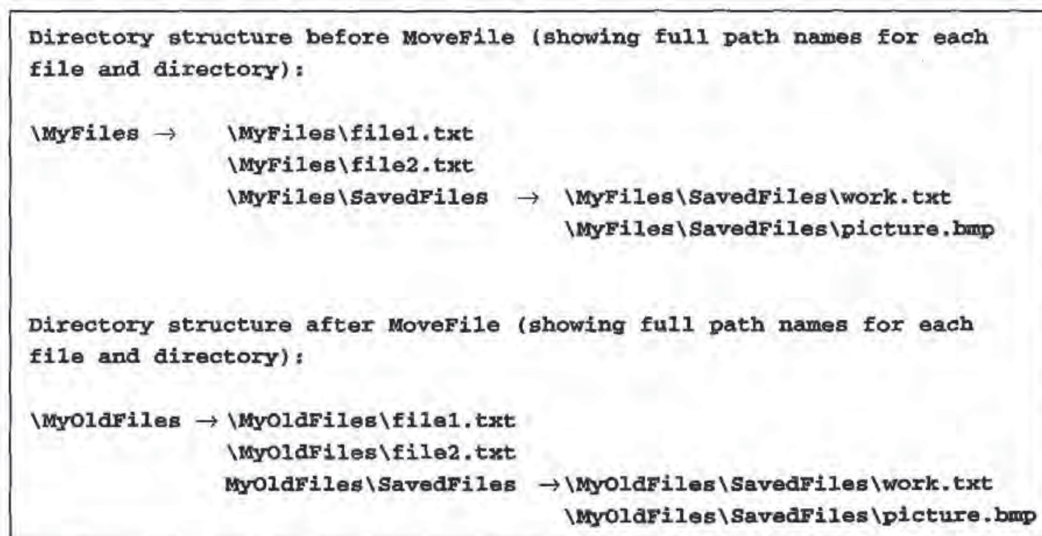
In the case of renaming files, that might be correct. But consider the case of renaming a directory. Let's take the example of a Windows CE directory called MyFiles. When an application renames this directory to MyOldFiles, for example, all of the *children* of this directory are renamed accordingly as well. More specifically, all files and subdirectories that MyFiles contains have their path names changed to reflect the fact that MyFiles has changed to MyOldFiles (Figure 6.9). From this point of view, it's as if the *MoveFile* operation physically moved all the children of MyFiles to a new directory called MyOldFiles.

The syntax of the *MoveFile* function is:

```
MoveFile(lpExistingFileName, lpNewFileName);
```

The *lpExistingFileName* and *lpNewFilename* parameters are the same as in the *CopyFile* function: *lpExistingFileName* is the name of the file or directory to be renamed, and *lpNewFileName* is the name to which the file or directory is renamed.

If *MoveFile* is successful, it returns TRUE. If the function fails, it returns FALSE. In that case, an application can call *GetLastError* to get more



```
Directory structure before MoveFile (showing full path names for each
file and directory):

\MyFiles →    \MyFiles\file1.txt
              \MyFiles\file2.txt
              \MyFiles\SavedFiles  →  \MyFiles\SavedFiles\work.txt
                                      \MyFiles\SavedFiles\picture.bmp


Directory structure after MoveFile (showing full path names for each
file and directory):

\MyOldFiles → \MyOldFiles\file1.txt
              \MyOldFiles\file2.txt
              MyOldFiles\SavedFiles  →\MyOldFiles\SavedFiles\work.txt
                                      \MyOldFiles\SavedFiles\picture.bmp
```

Figure 6.9 Effect of renaming a directory on its children.

information about what went wrong. For example, if an application attempts to rename a file to the name of a file that already exists, *GetLastError* would return ERROR_ALREADY_EXISTS.

Deleting Files and Directories

Deleting a file in Windows CE is a simple matter of calling *DeleteFile*:

```
DeleteFile(lpFileName);
```

Your application calls this function by passing the full path name of the file to be deleted in the *lpFileName* parameter.

If the file is deleted, *DeleteFile* returns TRUE. Otherwise, it will return FALSE, in which case you can call *GetLastError* for more information.

Similarly, directories are deleted using *RemoveDirectory*:

```
RemoveDirectory(lpPathName);
```

lpPathName is the full path name of the directory to be removed. *RemoveDirectory* has the same return values as *DeleteFile*.

RemoveDirectory will fail if an application attempts to delete a directory that is not empty. There is no parameter to force deletion of an empty directory, or to do recursive deletion of an entire directory tree. Such functionality, if required by an application, must be implemented by the application developer. The File System Explorer sample application shows how this is done with its implementation of the *OnDelete* function.

Flash Cards and Persistent Storage

Flash memory cards provide a means for Windows CE–based devices to expand the amount of RAM available. A flash card is a type of *mountable file system*. Both Handheld PCs and Palm-size PCs are equipped to use flash cards. Flash cards can be used to store files just like regular RAM.

Flash cards are assigned object identifiers just like files or directories. We will see some examples of how these identifiers are used in the next section.

Flash cards should include a \My Documents folder. Many of the mountable file system API functions will default to such a folder to perform searches and the like.

To the Windows CE device, a flash card looks like part of the file system. A flash card installed on a Palm-size PC or Handheld PC is assigned the folder name Storage Card by the operating system. Files and directories are created and accessed just as they are in standard RAM using the file system API. The only difference is that the path names of flash card files and directories begin with Storage Card.

For example, to create a directory called "FlashDocs" on a Palm-size PC storage card, an application would simply call *CreateDirectory*:

```
CreateDirectory(TEXT("\\Storage Card\\FlashDocs"), NULL);
```

Flash Card APIs

There are some additional functions provided by Windows CE for enumerating flash cards and files on flash cards. Use of these functions is very similar to their file system API counterparts.

Enumerating Flash Cards

The first set of flash card API functions is used to enumerate flash cards or other mountable file systems attached to a device.

The first of these functions is *FindFirstFlashCard*. This function is used to find the first flash card (or other mountable file system) on a device:

```
FindFirstFlashCard(lpFindFlashData);
```

The only parameter to this function is a pointer to a WIN32_FIND_DATA structure. This function is analogous to *FindFirstFile*. The difference is that instead of returning WIN32_FIND_DATA information about the first specified file or directory, it returns information about the first flash card it finds.

The most important piece of information *FindFirstFlashCard* returns is the object identifier of the flash card. This value is returned in the *dwOID* member of the *lpFindFlashData* parameter.

Also like *FindFirstFile*, *FindFirstFlashCard* returns a search handle that can be used to perform searches for additional flash cards. Additional flash cards can be found with subsequent calls to *FindNextFlashCard*:

```
FindNextFlashCard(hFlashCard, lpFindFlashData);
```

hFlashCard is the search handle returned by *FindFirstFlashCard*. *lpFindFlashData* is a WIN32_FIND_DATA structure pointer containing information about the next flash card.

Note that it is not common for a Windows CE–based device to contain more than one flash card.

Flash card search handles, like file search handles, are closed with the *FindClose* function.

Searching for Flash Card Files

The second set of flash card API functions is used to enumerate files and directories stored on flash cards.

FindFirstProjectFile is the same as *FindFirstFile* except that it can be made to look for files on a specified mountable file system:

```
FindFirstProjectFile(lpFileName, lpFindFileData,
  dwOidFlash, lpszProj);
```

The first two parameters of this function are the same as in *FindFirstFile*. They contain the file or directory name to search for and the returned WIN32_FIND_DATA, respectively.

dwOidFlash identifies the storage card to search on. This value is obtained by a previous call to *FindFirstFlashCard* or *FindNextFlashCard*. This parameter can be set to zero, in which case the main device file system is searched instead of a mountable file system. *FindFirstProjectFile* can thus be used just like *FindFirstFile* on devices that include mountable file systems. In fact, it is recommended that on devices such as the Palm-size PC, *FindFirstProjectFile* be used exclusively.

The final parameter, *lpszProj*, indicates the folder to start the search in. If NULL, the search starts with the \My Documents folder.

FindFirstProjectFile returns a search handle, just like *FindFirstFile*. This search handle is used to perform subsequent searches with the *FindNextProjectFile* function:

```
FindNextProjectFile(hHandle, lpFindProjData);
```

The parameters and behavior of this function are the same as *FindNextFile*.

Concluding Remarks

That's it for the Windows CE file system. You should now be able to add file support to your Windows CE applications in order to store and retrieve data. But more often you will want to store and organize information in a more structured format than a simple flat data file. For this purpose, Windows CE provides a simple database technology. Windows CE databases are the subject of the next chapter.

CHAPTER 7

Windows CE Databases

Long before Microsoft commanded the world of personal computing, another large multinational corporation in a northwestern state far colder and wetter than Washington had its own monopoly on the market for personal information management products. Every time the need arose to record a new phone number, address, or name, or whenever a quick reminder or note had to be jotted down for future reference, millions of people across the country grabbed a pen and wrote this information in (at least in my case) marginally legible handwriting on a one-and-a-half-by-one-inch yellow square piece of sticky paper.

Over time (again in my case, at least), these little pieces of paper were stuck up all over computer monitors, to the inside covers of reference books, kitchen counters, the home office desk; particularly important contacts were even stuck to the back of credit cards in wallets for easy later retrieval. This data storage model naturally led to frequent frantic searches through sock drawers and the front of the refrigerator door for meeting times or important phone numbers. It was somewhat inefficient, but business and our lives managed to move on with few mishaps.

Fortunately, Microsoft and other companies (notably, 3Com Corporation, with its line of PalmPilot organizers) have made all of our lives

easier with the introduction of personal information management devices to help keep track of phone numbers, appointments, to-do lists, and the like in one convenient place. At the tap of a stylus on a touch screen, we can now retrieve the phone numbers of friends and coworkers. With a few simple key or stylus strokes, we can tell these devices to alert us days in advance of impending birthdays and anniversaries. The persistent storage capabilities of Windows CE databases make many of these advances possible.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

Design databases

Create and delete databases

Open and close databases

Add records to and remove them from databases

Read and write database records

Sort databases

Perform database searches

Enumerate databases

Use database notifications

Use the contacts database API

The Phone List Application

To illustrate the features of Windows CE databases and the Windows CE database application programming interface functions, we will look at the example of a phone list database application. This application maintains a database of employee names, phone numbers, and department numbers. It displays the database contents in a user interface based on a simple list view control, where each list view column represents a particular database record property. The main application window is shown in Figure 7.1.

This application will give you a feel for how to write applications that can add and delete database records, sort a database, and perform

Figure 7.1 The phone list application.

database searches. It can also very easily be expanded into a full-fledged contacts or address book application by adding the appropriate properties to the database records. And you can customize the user interface to suit different needs.

The complete source code for the phone list application can be found on the companion CD in the directory \Samples\dbase. The application that is built by the project files is called DBASE.EXE.

Adding and Removing Phone List Database Records

Database entries are added or removed by choosing the appropriate option from the Record menu (Figure 7.2). This menu contains the Add, Delete, and Clear Database options.

For example, to add a new record to the phone list database, a user selects the Add option from the Record menu. When this menu item is

Figure 7.2 The phone list application Record menu.

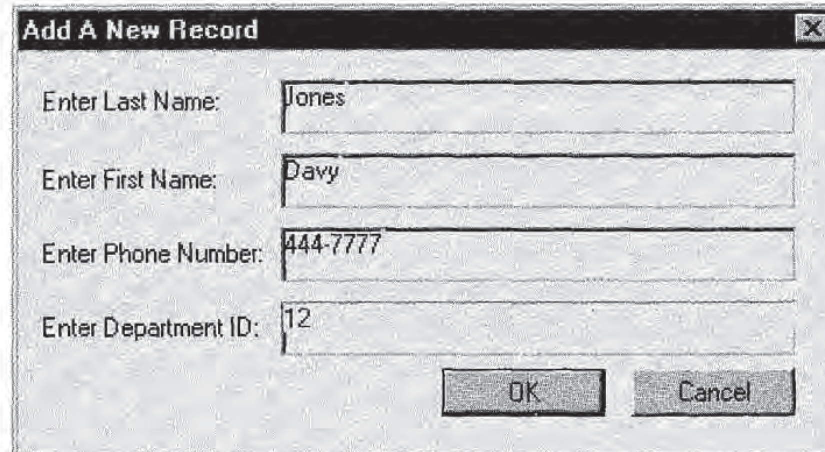


Figure 7.3 The Add A New Record dialog box.

selected, the dialog box shown in Figure 7.3 is displayed. This dialog box allows the user to enter the details of the record to be added to the database. Each text entry field in this dialog corresponds to one of the phone list database record properties.

After all of the properties have been entered, pressing the OK button adds the new record to the phone list database and refreshes the main application window display so that the new record is shown.

To delete a phone list database record, a user simply needs to select the record to be deleted in the main window and select the Delete option from the Record menu. To delete all database records at once, simply select the Clear Database option from the Record menu.

Sorting Phone List Database Records

The phone list application allows users to sort the database by any of the database record properties. Sorting the database by a particular property is done by tapping the column header of the corresponding record property. For example, to sort the database by phone number, the user only needs to tap the phone number column header (Figure 7.4).

File	Record	Options	
Last Name	First Name	Phone Number	Department
Smith	Joe	222-2222	13
Wilson	Will	777-7777	65
Jones	Bill	888-8888	56

Figure 7.4 Sorting the phone list database by phone number.

Searching for Records in the Phone List Database

Database applications, such as phone lists or other personal contacts, are primarily used to look up information pertaining to one or more of the records in the corresponding application database. Users of a phone list application, for example, will often want to look up the phone number of a specific person in the database.

The phone list application provides basic record search capabilities. A search is invoked by selecting the Seek option from the Options menu in the main application window. The Options menu is shown in Figure 7.5. Selecting the Seek option displays the dialog box shown in Figure 7.6.

This dialog lets the user enter a record property value in the text entry field. The record property that this corresponds to (e.g., first name, last name, etc.) depends on the property by which the database is currently sorted. For example, if the phone list database is sorted by last name, the seek record dialog box assumes that the user will enter a last name. This is emphasized by the wording of the caption above the edit field. Entering a last name and pressing the OK button in this case makes the application search the phone list database for a record containing the specified last name.

File Record Options

Seek...

Database Memory...

Last Name		Phone Number	Department
Jones		888-8888	56
Smith	Joe	222-2222	13
Wilson	Will	777-7777	65

Figure 7.5 The phone list application Options menu.

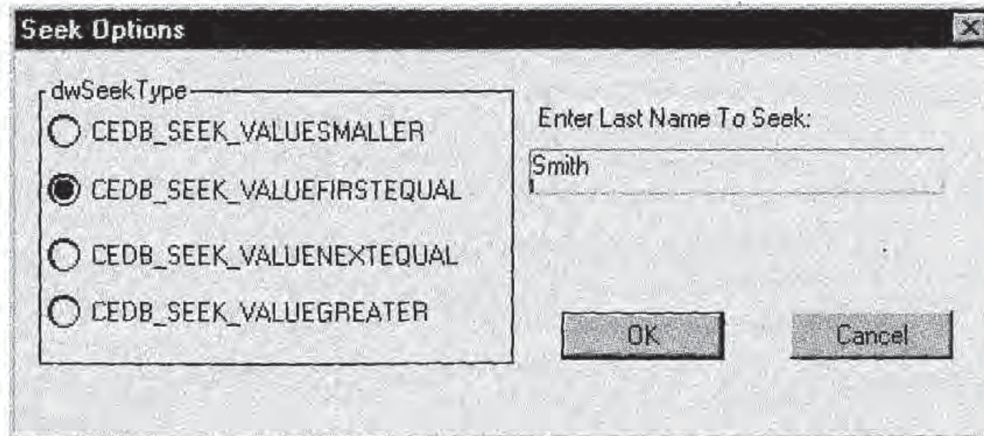


Figure 7.6 The Seek Record dialog box.

The set of radio buttons on the left side of the dialog with the very confusing labels indicate the type of search to perform. As you read the rest of this chapter, you will see that these button labels correspond to one of the parameters of the Windows CE database search function. After reading about this function, you will see that this dialog box is useful for allowing you to see the effect of passing different parameter values to the search function.

For now, from the point of view of highlighting the basic database features provided by Windows CE, it is sufficient to say that these radio buttons control how the search value entered by the user is compared to the corresponding record property for each record in the database.

For example, let's say that a user wants to search the phone list database for the first record with a last name of Smith. The user must first make sure that the database is sorted by last name. As described above, this is done by tapping the Last Name column header.

Next, the user chooses the Seek menu option from the Options menu. In the dialog box that appears, the user types "Smith" in the edit field. The CEDB_SEEK_VALUEFIRSTEQUAL radio button is selected to tell the application to search for the first record containing a last name of Smith (see Figure 7.6). When the user presses the OK button, the phone list application searches for the requested record. If the record is found, the results of the search are displayed in the Search Results field at the bottom of the main application screen as shown in Figure 7.7.

File Record Options

Last Name	First Name	Phone Number	Department
Jones	Bill	888-8888	56
Smith	Joe	222-2222	13
Wilson	Will	777-7777	65

Search Results:

Last Name	First Name	Phone Number	Department
Smith	Joe	222-2222	13

Start 10:35 AM

Figure 7.7 The result of a successful record search.

Determining the Size of the Phone List Database

The final interesting feature of the phone list application is that it allows the user to determine how much object store memory is being used by the phone list database. This feature is invoked by selecting the Database Memory option from the Options menu.

Selecting the Database Memory menu item displays the dialog box shown in Figure 7.8. This dialog simply reports the total number of bytes consumed by the phone list database and the records it contains.

Programming Windows CE Databases

If you are familiar with writing database applications for relational databases, you will find the Windows CE database model very different. Databases under Windows CE do not support any form of structured query language, and provide none of the relational data manipulation techniques such as table joins. Windows CE databases are simply non-hierarchical collections of an arbitrary number of *records*, each of which contains one or more data *properties*. Properties can be the integer, Unicode string, FILETIME, or blob (byte array) data.

As with any object stored in the Windows CE object store, every database and database record is assigned a unique object identifier of type

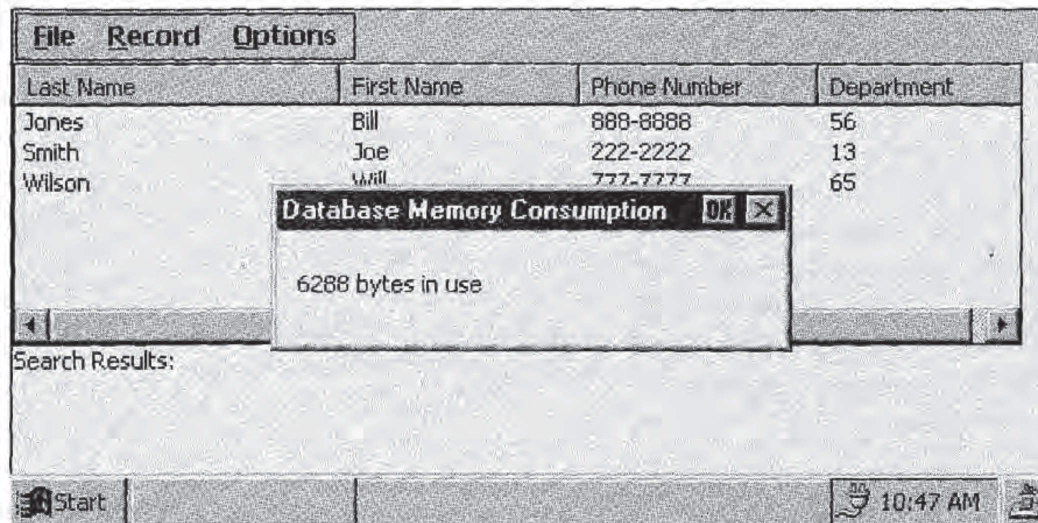


Figure 7.8 The phone list Database Memory Consumption dialog.

CEOID by the Windows CE operating system. This object identifier can be used, for example, when searching for database records or to identify records to delete from a database.

Another important part of a Windows CE database is the *current record pointer*. This is also sometimes called the *seek pointer*. This pointer indicates the record to be read by the next database read operation. The current record is therefore defined to mean the record currently pointed to by the seek pointer. As an application reads records in a database, the current record pointer (or seek pointer) can be thought of as marking the current record position. Seeking and reading records in the database can move the seek pointer position. These operations are discussed in detail later.

Windows CE provides a database application programming interface for creating and managing databases on Windows CE devices. This API provides functionality for the following database operations:

- Creating and deleting databases
- Adding and deleting database records
- Sorting records in a database
- Searching for records in a database
- Enumerating the databases on a Windows CE device

What's All This SQLing I Hear?

The database technology provided with the Windows CE operating system is significantly simpler than the full relational or object-oriented databases that you might be used to programming with in the Windows NT or Windows 98 environments. The Windows CE database technology was originally designed to support basic personal information management (PIM) applications.

The Windows CE database technology is not a relation or object-oriented database. It also does not support Structured Query Language (SQL), the lingua franca of database programmers all over the world. As such, the Windows CE database model is perfectly adequate for relatively simple databases, such as contact lists and collections of e-mail messages. But application developers who need database schema more complex than the simple record-based model of the default Windows CE database have until recently been disappointed. And performing complex searches and queries has been practically impossible due to the lack of SQL.

As Windows CE has matured from its first incarnation over two years ago, so has the level of complexity of software being written for the operating system. Various companies are designing new Windows CE–based consumer and business products, many with database requirements that exceed the capabilities of the basic Windows CE database technology.

Fortunately for these applications, database vendors have recently begun porting their database technologies to Windows CE. Relational as well as object-oriented database solutions are now available from a number of database companies. Oracle, Sybase, and Neoworks Corporations, to name a few, have all begun supporting Windows CE versions of their database software. It is, therefore, now possible to take advantage of traditional database technology, including the power of SQL, when writing software for Windows CE.

Additionally, the Microsoft Foundation Classes library for Windows CE supports Data Access Objects (DAO), providing a SQL-based interface to the Windows CE database technology.

This book, however, will only cover programming for the Windows CE database. Complete coverage of SQL programming and relation and objected-oriented databases is left to the vast number of database programming books on the market today.

For each of the functions in the database API, there is a corresponding function in the Remote Application Programming Interface, or RAPI. RAPI is a part of Windows CE that allows applications on a desktop PC, called the RAPI client, to make function calls on a Windows CE device, the RAPI server. RAPI is covered later in Chapter 15.

Square PEGs in Round Holes

From time to time as you read the Windows CE on-line documentation, you will see references to things like PEGOID when you would expect CEOID, or function names like *PegOpenDatabase* when you would swear we've been talking endlessly about *CeOpenDatabase*. What is this all about?

In Windows CE versions 1.0 and earlier, all of the database types and functions started with *Peg*, instead of *Ce*. This is a leftover from the code name for Windows CE, which was Pegasus. As Windows CE matured, Microsoft realized that function names like *CeOpenDatabase* made infinitely more sense to most software developers than *PegOpenDatabase*. They therefore made the decision (usually considered anathema in most circles) to change the names of the database types and APIs.

This of course left a backward compatibility issue. So in order to make everyone's Windows CE 1.0 database applications compatible with later versions of the operating system, the old Pegasus names are defined to the new names in the public header file WNDBASE.H:

```
#define PEGPROPID            CEPROPID
*.
*.
*.
#define PegOpenDatabase        CeOpenDatabase
etc.
```

In this way, applications written for older versions of Windows CE can be ported to new versions. Any references to the Peg-prefixed symbols get replaced with the Ce-prefixed versions, and the applications will compile and link without complaints.

The Database Design

When designing any database, it is always a good idea to first design the database schema. The database schema is the description of the database and the kinds of information it contains. In our example, this simply means describing what properties each of the phone list records will contain, including the data type of each of the properties. If applicable, you would also define the acceptable range of values that each record property can be assigned. In traditional relational database design, the schema would be more complex, including descriptions of the various relational tables that make up the database.

Taking the time up front to do this design step can save you from rewriting the database definition and management code in your applications because you overlooked an important piece of information required by your application.

The phone list database consists of records that contain these four properties: employee last name, first name, phone number, and department number (Table 7.1). We will allow users of the phone list application to sort the records in the phone list on any of these four properties.

Internal Representation of Record Properties

It is one thing to understand the format of a Windows CE database record in the abstract. It is quite another to understand how Windows CE itself think of records and the properties that they contain.

Windows CE treats each database record as a collection of one or more properties, each of which is of type CEPROPVAL. The definition of the CEPROPVAL structure is:

```
typedef struct _CEPROPVAL
{
  CEPROPID propid;
  WORD wLenData;
  WORD wFlags;
  CEVALUNION val;
} CEPROPVAL;
```

The *propid* member is of type CEPROPID, which is defined as a LONG. The low word of *propid* identifies the data type of the property. The high word is an application-defined index. Typically this index is

Table 7.1 Phone List Database Record Definition

PROPERTY	DATA TYPE
Last Name	LPWSTR
First Name	LPWSTR
Phone Number	LPWSTR
Department Number	short

Table 7.2 CEPROPID Data Type Specifiers

VALUE	DATA TYPE
CEVT_BLOB	A CEBLOB structure
CEVT_FILETIME	A FILETIME structure
CEVT_I2	A 16-bit signed integer
CEVT_I4	A 32-bit signed integer
CEVT_LPWSTR	A null-terminated Unicode string
CEVT_UI2	A 16-bit unsigned integer
CEVT_UI4	A 32-bit unsigned integer

used to represent the zero-based index of the property in the record. The low word must be one of the values listed in Table 7.2.

As an example, the phone list application defines the property identifiers of the phone list database record properties in this way:

```
//First define the indices of the properties within the record
#define PL_LASTNAME_INDEX   0
#define PL_FIRSTNAME_INDEX  1
#define PL_PHONENUMBER_INDEX 2
#define PL_DEPT_INDEX       3*
//Next define the CEPROPID values of the record properties
#define PL_LASTNAME \
  (MAKELONG(CEVT_LPWSTR,PL_LASTNAME_INDEX))
#define PL_FIRSTNAME \*
  (MAKELONG(CEVT_LPWSTR, PL_FIRSTNAME_INDEX))
#define PL_PHONENUMBER \
  (MAKELONG(CEVT_LPWSTR,PL_PHONENUMBER_INDEX))
#define PL_DEPT \
  (MAKELONG(CEVT_I2, PL_DEPT_INDEX))
```

The second member of the CEPROPVAL structure, *wLenData*, is not used. The *wFlags* member is used to define a set of special property flags. This member is typically set to 0. We will discuss the other values of this member and their meanings later when discussing reading and writing database records.

The most interesting of the CEPROPVAL members is the *val* member. As the name indicates, this member contains the actual data associated with the particular record property. This member is of type CEVALUNION, a union defined as follows:

```
typedef union _CEVALUNION
{
  short iVal;
  USHORT uiVal;
  long lVal;
  ULONG ulVal;
  FILETIME filetime;
  LPWSTR lpwstr;
  CEBLOB blob;
} CEVALUNION;
```

Each of the members of this union corresponds to one of the data type identifiers of the *propid* member of the CEPROPVAL structure (see Table 7.2). The member of this union that you would use when setting the value of a particular record property would thus depend on the data type specified in that property's property identifier. For example, to set the first name property of a phone list database record, the phone list application would do the following:

```
CEPROPVAL cepvFirstName;
cepvFirstName.propid = PL_FIRSTNAME;
cepvFirstName.val.lpwstr = TEXT("Some Name");
```

Or to set the department number property:

```
CEPROPVAL cepvDeptNum;
cepvDeptNum.propid = PL_DEPT;
cepvDeptNum.val.iVal = 12; //i.e., the appropriate dept number
```

These examples are meant only to demonstrate the use of the CEPROPVAL *val* member. As we'll see later, applications typically define an array of CEPROPVAL structures to represent the entire record to be read or written.

Creating the Database

Windows CE databases are created using the *CeCreateDatabase* function. The syntax of *CeCreateDatabase* is:

```
CeCreateDatabase(lpszName, dwDbaseType, wNumSortOrder,
  rgSortSpecs);
```

The first parameter is the Unicode string name of the database to be created. This name can be up to 32 characters long (including the null terminator). Database names that exceed this limit are truncated. The next parameter is the database type identifier. This value is defined by

the application to distinguish one type of database from another. For example, let's say that an application needs to manage the phone list databases for three different companies, as well as the payroll databases for those same companies. This application could define the following database types:

```
#define DB_TYPE_PHONE_LIST   0
#define DB_TYPE_PAYROLL     1
```

Each of the phone list databases could be created with a database type identifier DB_TYPE_PHONE_LIST, and the payroll databases could be created with type identifier DB_TYPE_PAYROLL. The application could then use the database type identifier to distinguish between the different types of databases, for example when enumerating all databases on a Windows CE device.

The *wNumSortOrder* indicates the number of sort orders allowed for the database. This is a fancy way of saying how many record properties the database can use as sort keys. The final parameter is an array of SORTORDERSPEC structures defined in more detail below.

The *CeCreateDatabase* function returns the object identifier of the database if the creation is successful. If unsuccessful, *CeCreateDatabase* returns zero.

If we can create a database, we must also be able to delete it. To delete a database, simply call *CeDeleteDatabase*. This function takes one parameter, the object identifier of the database to be deleted.

Sorting and the SORTORDERSPEC

Applications sort databases by specifying up to four *sort orders*. A sort order specifies which property in each database record is to be used as the sorting key, and the order (ascending, descending, etc.) in which the database records are to be sorted when the corresponding property is used as the sorting key. This information can be provided to the database in one of two ways. The first is to pass this sort order information to the database when it is created via the last argument to the *CeCreateDatabase* function, *rgSortSpecs*. The second is to use the *CeSetDatabaseInfo* function and specify the sort orders in a CEDBASEINFO structure. To use either of these techniques, we must first understand how a sort order is represented in Windows CE.

A sort order is defined using the SORTORDERSPEC structure, which is defined as:

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

The first member of this structure is the property identifier of a particular record property. The second member contains sort order flags. These flags define, for example, whether records are sorted in ascending or descending order.

As an example, let's assume that we want to be able to sort the phone list database by last name in ascending order, and at other times to sort by department number in descending order. We would therefore need to define two SORTORDERSPEC structures to convey this information:

```
SORTORDERSPEC sos[2];
sos[0].propid = PL_LASTNAME;    //Specify the last name
                 //property id
sos[0].dwFlags = 0;        //0 indicates ascending order
sos[1].propid = PL_DEPT;      //Specify the deparment number
                 //property id
sos[1].dwFlags = CEDB_SORT_DESCENDING; //Descending sort order
```

Note the value of 0 for the *dwFlags* member of *sos*[0] to indicate ascending sort order. I point this out to save you the same amount of time I wasted searching in vain for a definition of CEDB_SORT_ASCENDING in the Windows CE header files.

Now that we understand how to specify a sort order, we are ready to discuss the two techniques for supplying the database with our sort order information. The first is to pass the array of SORTORDERSPEC structures that you create as the *rgSortSpecs* parameter of the *CeCreateDatabase* function. This would seem to imply that once a database is created, an application has no control over redefining the sort order information associated with that database. This is not the case, however. The second technique for specifying sort orders is with the *CeSetDatabaseInfo* function. The syntax of this function is:

```
CeSetDatabaseInfo(oidDbase, pNewInfo);
```

The first parameter of *CeSetDatabaseInfo* is the object identifier of an open database. The second parameter is a pointer to a CEDBASEINFO structure. This structure is defined as:

```
typedef struct _CEDBASEINFO
{
  DWORD dwFlags;
  WCHAR szDbaseName[CEDB_MAXDBASENAMELEN];
  DWORD dwDbaseType;
  WORD wNumRecords;
  WORD wNumSortOrder;
  DWORD dwSize;
  FILETIME ftLastModified;
  SORTORDERSPEC rgSortSpecs[CEDB_MAXSORTORDER];
} CEDBASEINFO;
```

szDbaseName and *dwDbaseType* are the name and application-defined database type identifier respectively. *dwSize* is used by *CeSetDatabaseInfo* to return the number of bytes of data stored in the database. *ftLastModified* is used to update the time the database was last modified. The *wNumRecords* field is not used.

That leaves *dwFlags*, *wNumSortOrder*, and *rgSortSpecs*. *wNumSortOrder* specifies the new total number of sort orders associated with the database. Remember, Windows CE databases only support up to four sort orders. *rgSortSpecs* is our trusty array of SORTORDERSPEC structures. If you originally created the database with one set of sort orders, you can specify totally different sort orders here.

Finally, the *dwFlags* member is used to indicate to *CeSetDatabaseInfo* which of the other CEDBASEINFO structure members are valid, that is, which characteristics of the database are to be changed by the *CeSetDatabaseInfo* call. *dwFlags* can be one of the four values given in Table 7.3.

It should be noted here that using *CeSetDatabaseInfo* to change the sort order of a database can be a very slow operation for Windows CE to perform, especially if the database being modified contains a large number of records. It is therefore not generally recommended that this be done. The sort order requirements for a particular database should

Table 7.3 CEDBASEINFO *dwFlags* Member Values

VALUE	MEANING
CEDB_VALIDMODTIME	ftLastModified member is valid
CEDB_VALIDNAME	szDbaseName member is valid
CEDB_VALIDTYPE	dwDbaseType member is valid
CEDB_VALIDSORTSPEC	rgSortSpecs member is valid

instead be carefully considered and defined during the application design phase. The sort orders can then be set once and for all when the database is created.

Careful readers who have suffered through this laborious account of how and when to define database sort orders will have noticed that a huge piece of the sorting story is still missing. We have yet to discuss how a Windows CE database is told on which of the properties associated with the various sort orders the database is to be sorted. This is done, mercifully simply, by simply opening the database.

Opening and Closing the Database

Not surprisingly, the function that is used to open a Windows CE database is called *CeOpenDatabase*:

```
CeOpenDatabase(poid, lpszName, propid, dwFlags, hwndNotify);
```

The *poid* argument is the object identifier of the database to open. Alternatively (and more commonly), you will open a database by name. This is done by setting the *poid* argument to zero, and supplying the name of the database to open in the *lpszName* argument. If a database is opened in this way, Windows CE will return the object identifier of the database via the *poid* argument.

The *propid* argument is a CEPROPVAL that specifies which sort order to use when sorting the database. *dwFlags* can be set to CEDB_AUTOINCREMENT, or to zero. If set to CEDB_AUTOINCREMENT, the database record pointer is incremented each time a record is read from the database. If zero, the record pointer is not incremented.

The *hwndNotify* parameter can be used to specify the window to which database notifications are sent. It can be NULL if you are not interested in receiving such notifications. Database notifications are discussed later in the chapter.

CeOpenDatabase returns a handle to the opened database if successful. Otherwise it returns the error code INVALID_HANDLE_VALUE.

To close a database, use the function *CloseHandle*, passing the handle to the open database as the *hObject* parameter.

From the point of view of database sorting, the *propid* argument to *CeOpenDatabase* gets all the glory. An application specifies how the

database is sorted simply by calling *CeOpenDatabase* with the desired sorting property specified in the *propid* parameter. For example, to sort the records in the phone list database by first name, our application simply needs to open the database as follows:

```
CEOID ceoidDBase=0;  //0 because we will open the database
            //by name
CeOpenDatabase(ceoidDBase, TEXT("PhoneList"), PL_FIRSTNAME,
  CEDB_AUTOINCREMENT, NULL);
```

If the application already has the database open when it wants to resort it on a new key, it must first close the database with *CloseHandle* before reopening it.

Writing an *OpenDatabase* Function

You may often find it convenient to combine the processes of creating and opening a database into one function. Consider the following *OpenDatabase* function:

```
HANDLE OpenDatabase(
  LPWSTR lpszName,    /* Database name */
  CEPROPID cepropidSort, /* Sort property */
  DWORD dwFlags,     /*CEDB_AUTOINCREMENT,etc.*/
  HWND hwndNotify,
  CEOID* pceoid)     /*CEOID of opened database*/
{
  HANDLE hdb;
  *pceoid = 0;
  hdb = CeOpenDatabase(pceoid, lpszName,
   cepropidSort, dwFlags, hwndNotify);
  if (INVALID_HANDLE_VALUE==hdb)
  {
   SORTORDERSPEC sos[4];
   sos[0].propid = PL_LASTNAME;
   sos[0].dwFlags = 0;
   sos[1].propid = PL_FIRSTNAME;
   sos[1].dwFlags = 0;
   sos[2].propid = PL_PHONENUMBER;
   sos[2].dwFlags = 0;
   sos[3].propid = PL_DEPT;
   sos[3].dwFlags = 0;
   ceoidDBase = CeCreateDatabase(szDBaseName,0, 4, sos);
   hdb = CeOpenDatabase(pceoid, NULL,cepropidSort,
    CEDB_AUTOINCREMENT, hwndNotify);
  }
  return (hdb);
}
```

One advantage of such a function is that once the function is written, the application programmer can think of creating and opening the database as the same operation. Instead of having to consider whether or not the database exists, the programmer can simply call this *Open-Database* function. If it happens to be the first time that the database is being accessed and it doesn't yet exist, this function will create the database.

Writing and Reading Database Records

We have seen how to create, open, close, and sort the records in a database. But how do the records get into the database to begin with? And how are the records retrieved by an application that needs to use the information that these records contain?

Writing records to a database requires our old friend the CEPROPVAL structure and the *CeWriteRecordProps* function. Basically, an application fills an array of CEPROPVALs with the property information for the record, and then calls *CeWriteRecordProps* to actually write the record to the database. For example, to write a hypothetical record to the phone list database, the phone list application might do something like this:

```
CEPROPVAL cePropVal[4];
//Set the last name
cePropVal[PL_LASTNAME_INDEX].propid = PL_LASTNAME;
cePropVal[PL_LASTNAME_INDEX].val.lpwstr = TEXT("Rubble");
//Set the first name
cePropVal[PL_FIRSTNAME_INDEX].propid = PL_FIRSTNAME;
cePropVal[PL_FIRSTNAME_INDEX].val.lpwstr = TEXT("Barney");
//Set the phone number
cePropVal[PL_PHONENUMBER_INDEX].propid = PL_PHONENUMBER;
cePropVal[PL_PHONENUMBER_INDEX].val.lpwstr = TEXT("888-8888");
//Set the department id
cePropVal[PL_DEPT_INDEX].propid = PL_DEPT;
cePropVal[PL_DEPT_INDEX].val.iVal = 12;
CeWriteRecordProps(hDBase, 0, 4, cePropVal);
```

The first argument of the *CeWriteRecordProps* function is the handle of the open database to be written to. The second argument is of type CEOID and can contain the object identifier of an existing record to be written over, or zero to indicate that a new record with the given properties is to be added to the database. A database record can thus be modified by calling *CeWriteRecordProps* and passing the object identi-

fier of this record in the second parameter. The third parameter indicates the number of properties contained in the fourth parameter, which is the array of CEPROPVALs containing the data to be written to the database. *CeWriteRecordProps* returns the object identifier of the record that was written.

The *wFlags* member of any of the CEPROPVAL structures can either be zero or CEDB_PROPDELETE. If CEDB_PROPDELETE, the write operation deletes the property from the record. In this way *CeWriteRecordProps* can remove selected properties from existing database records.

Reading a record from the database requires *CeReadRecordProps*. This function reads and returns the record as one big block of bytes that actually contains a set of CEPROPVAL structures, one per record property. Once the record data is read, it is the responsibility of your application to unpack the data array, breaking it down into the constituent properties.

The *CeReadRecordProps* function has the following definition:

```
CeReadRecordProps(hDBase, dwFlags, lpcPropID, rgPropID,
  lplpBuffer, lpcbBuffer);
```

The first argument is the handle to the open database. *dwFlags* can either be zero or CEDB_ALLOWREALLOC. Applications will usually use this value, indicating that *CeReadRecordProps* has permission to reallocate the data buffer *lplpBuffer* if it doesn't contain enough space to hold all of the record data. *lpcPropID* is an LPWORD indicating the number of properties to be retrieved. *rgPropID* is an array of CEPROPID values that tells the function which record properties to read.

CeReadRecordProps can thus be used to read any or all of the properties in a given database record. To read all properties, set *lpcPropID* to zero and *rgPropID* to NULL. If these two parameters are used in this way, *lpcPropID* contains the number of properties read once the function returns.

After the function executes, *lplpBuffer* contains the record property data and *lpcbBuffer* contains the total number of bytes in the buffer. If *CeReadRecordProps* succeeds, it return the object identifier of the record that was read. If it fails, it will return zero, in which case you can call *GetLastError* to get more detailed information about what went wrong.

It is possible that a property specified in the *rgPropID* array does not exist in the record retrieved by *CeReadRecordProps*. For example, an

application might accidentally specify an invalid property identifier, or specify a property that has been previously deleted from the record. In these cases, the *wFlags* parameter of the CEPROPVAL structure extracted from *lplpBuffer* for that property will be set to CEDB_PROPNOTFOUND.

At this point, all that is left to do is to convert the data array returned by *CeReadRecordProps* into the property data. Since *lpcPropID* contains the number of properties contained by the record data buffer, the simplest way to do this is to cast the data buffer into an array of CEPROPVAL structures and iterate on each property as follows:

```
WORD cProps = 0;
LPBYTE pBuf=NULL;
DWORD cbByte = 0;
PCEPROPVAL pVals;
CEOID oid;
int i;
//hDBase is a handle to the (previously opened) database
oid = CeReadRecordProps(hDBase, CEDB_ALLOWREALLOC, &cProps,
  NULL, &pBuf, &cbByte);
//Unpack all of the record properties
pVals = (PCEPROPVAL)pBuf;
for (i=0; i<cProps; i++)
{
  switch(HIWORD(pVals[i].propid))
  {
   case PL_LASTNAME_INDEX:
   /* pVals[i].val.lpwstr contains the last name property
     value: The application needs to do something with it.
    */
    break;
   case PL_FIRSTNAME_INDEX:
   /* pVals[i].val.lpwstr contains the first name property */
    break;
   case PL_PHONENUMBER_INDEX:
   /* pVals[i].val.lpwstr contains the phone number
     property
    */
    break;
   case PL_DEPT_INDEX:
   /* pVals[i].val.iVal contains the dept number property */
    break;
   default:
    break;
  }          //End of switch statement*
}            //End of for (i=0; i<cProps; i++) loop
LocalFree(pBuf);
```

The LPBYTE array of raw record data is cast to an array of CEPROPVAL structures (the *pVals* variable). The for loop that follows then executes once for each property that was read from the current record by the *CeReadRecordProps* call. The switch statement checks the application-defined index of the current property in the *pVals* array, and extracts the value of that record accordingly (recall that the high word of each *propid* member of a CEPROPVAL contains the application-defined index of a particular property). The *val* members of the individual CEPROPVALs can be assigned to other variables as needed, or, as we will show later, used to construct a database record structure that is used by the application to more clearly represent the data. Notice that we free (using *LocalFree*) the data buffer returned by *CeReadRecordProps* after using it. This needs to be done whether or not the read operation was successful, as *CeReadRecordProps* might allocate memory in any attempt to read a database record.

Deleting Database Records

Finally, we need to describe how to delete database records. Deleting records from a Windows CE database is done with the *CeDeleteRecord* function:

```
CeDeleteRecord(hDatabase, oidRecord);
```

hDatabase is the handle of the open database from which the record is to be removed. *oidRecord* is the CEOID identifying the record to delete.

CeDeleteRecord returns TRUE if the specified record is successfully deleted from the database. Otherwise the function returns FALSE. In this case, an application can call *GetLastError* for more information about why the delete operation failed.

Managing Records More Cleanly

The preceding examples of how to write and read database records leave something to be desired. In both cases, you might have been left with the impression that Windows CE database applications only work with record data in raw binary form. While it is true that the *CeReadRecordProps* always return an LPBYTE array of data, and *CeWriteRecordProps* always writes record information as a collection of somewhat cumbersome CEPROPVAL structures, it is possible to write your applications in such a way that the rest of your application can treat database records in a more natural form. Specifically, your appli-

cation can define a structure that is used to represent the more abstract notion of records in your database.

For example, in the case of the phone list application, it seems natural to represent an entry in the phone list with a structure like this:

```
typedef struct _PhoneRecord
{
  TCHAR lpszLastName[MAX_NAME_LENGTH];
  TCHAR lpszFirstName[MAX_NAME_LENGTH];
  TCHAR lpszPhoneNumber[MAX_NAME_LENGTH];*
  int nDept;
}PHONERECORD, *LPPHONERECORD;
```

No CEPROPVALSs, CEOIDs, or other abstract database concepts here. A phone record is just a collection of values of C data types that we know and love. When users enter new records through the application's user interface, the application code can assign the values entered into the appropriate fields of an instance of this clear-cut data type. The mechanics of turning the elements of this structure into the form required in order to write the record to the database can be left to the inner workings of one simple *ReadRecord* function:

```
CEOID ReadRecord(LPPHONERECORD lppr)
{
  WORD cProps = 0;
  LPBYTE pBuf=NULL;*
  DWORD cbByte = 0;*
  PCEPROPVAL pVals;*
  CEOID oid;
  int i;
  oid = CeReadRecordProps(hDBase, CEDB_ALLOWREALLOC,
   &cProps, NULL,&pBuf, &cbByte);
  pVals = (PCEPROPVAL)pBuf;
  for (i=0; i<cProps; i++)
  {
   switch(HIWORD(pVals[i].propid))
   {
    case PL_LASTNAME_INDEX:
     lstrcpy(lppr->lpszLastName,
      pVals[i].val.lpwstr);
     break;
    case PL_FIRSTNAME_INDEX:
     lstrcpy(lppr->lpszFirstName,
      pVals[i].val.lpwstr);
     break;
    case PL_PHONENUMBER_INDEX:
     lstrcpy(lppr->lpszPhoneNumber,
      pVals[i].val.lpwstr);
```

```
        break;
      case PL_DEPT_INDEX:
        lppr->nDept = pVals[i].val.iVal;
        break;
      default:
        break;
    }        //End of switch statement
  }          //End of for (i=0; i<cProps; i++) loop
  LocalFree(pBuf);
  return (oid);
}
```

Writing phone list records can be abstracted in much the same way with *WriteRecord*:

```
CEOID WriteRecord(LPPHONERECORD lppr)
{
  CEPROPVAL cePropVal[PROPERTY_COUNT];
  CEOID ceoid;
  HANDLE hDBase;
  WORD wCurrent = 0;
  hDBase = OpenPhoneDatabase(szDBaseName, 0,
   CEDB_AUTOINCREMENT, NULL, &ceoidDBase);
  memset(&cePropVal, 0, sizeof(CEPROPVAL)*PROPERTY_COUNT);
  cePropVal[wCurrent].propid = PL_LASTNAME;
  cePropVal[wCurrent++].val.lpwstr = lppr->lpszLastName;
  cePropVal[wCurrent].propid = PL_FIRSTNAME;
  cePropVal[wCurrent++].val.lpwstr = lppr->lpszFirstName;
  cePropVal[wCurrent].propid = PL_PHONENUMBER;
  cePropVal[wCurrent++].val.lpwstr = lppr->lpszPhoneNumber;
  cePropVal[wCurrent].propid = PL_DEPT;
  cePropVal[wCurrent++].val.iVal = lppr->nDept;
  ceoid = CeWriteRecordProps(hDBase, 0, wCurrent,
   cePropVal);
  CloseHandle(hDBase);   /Close the database
  return (ceoid);
}
```

The rest of the application can now treat phone list data in the way that you would normally model the concept of a record, as a standard C structure.

Searching for Records

The function used for searching for records is *CeSeekDatabase*:

```
CeSeekDatabase(hDatabase, dwSeekType, dwValue, lpdwIndex);
```

hDatabase is a handle to the open database. *dwSeekType* is a DWORD that indicates to the function what kind of database search to perform. It also defines where the database current record pointer is positioned at the end of the seek operation. *lpdwIndex*, the last parameter, is a pointer to a DWORD that *CeSeekDatabase* uses to return the 0-based index of the record that was found by the seek operation. *dwValue* has different meanings depending on the value of *dwSeekType*.

Before looking at the *dwSeekType* parameter more closely, it is important to point out some characteristics of the seek operation. First, *CeSeekDatabase* searches a database in the order specified by the current sort order. Second, a seek can only be performed on a sorted property value. This means that if you are calling *CeSeekDatabase* to search for some record by value, the value specified in the *dwValue* parameter will only be compared to the database property values that correspond to the current sort order property. Recall these points when tracking down bugs in your database searching code. Programmers just starting to use Windows CE databases make the common mistake of searching for a record containing a particular property when the database is sorted on a different property.

dwSeekType can be one of the following values:

- CEDB_SEEK_CEOID
- CEDB_SEEK_VALUESMALLER
- CEDB_SEEK_VALUEFIRSTEQUAL
- CEDB_SEEK_VALUENEXTEQUAL
- CEDB_SEEK_VALUEGREATER
- CEDB_SEEK_BEGINNING
- CEDB_SEEK_CURRENT
- CEDB_SEEK_END

CEDB_SEEK_CEOID implies that *dwValue* is the object identifier of the record to seek in the database. At first glance, this case might not appear to be particularly useful. If an application already knows the object identifier of the record it is seeking, why would a seek even need to be done? It is important to keep in mind that *CeSeekDatabase* repositions the current record pointer, which indicates which record will be read from the database by the next read operation. So, if the phone list application wanted to read the properties of the record with an object

identifier defined as *ceoid*, the application would first have to seek to that record, and then read the record from the database:

```
WORD cProps = 0;
LPBYTE pBuf = NULL;
DWORD cbByte = 0;
if (CeSeekDatabase(hDBase, CEDB_SEEK_CEOID, (DWORD)ceoid,
  &nIndex))
{
  CeReadRecordProps(hBase, CEDB_ALLOWREALLOC, &cProps, NULL,
   &pBuf,&cbByte);
}
```

Calling *CeSeekDatabase* alone will simply point the current record pointer at the record of interest.

The next four *dwSeekType* values indicate that *dwValue* is a pointer to a CEPROPVAL structure that contains the property value for which to seek. CEDB_SEEK_VALUESMALLER says to search the database for the largest value that is smaller than the given value. CEDB_SEEK_VALUEFIRSTEQUAL tells *CeSeekDatabase* to search until it finds the first value equal to that indicated by *dwValue*. CEDB_SEEK_VALUENEXTEQUAL seeks one record forward from the current record position and checks if the property value of that record equals that of *dwValue*. CEDB_SEEK_VALUEGREATER seeks until a record with current sort order property equal to or greater than that of *dwValue* is found. If *CeSeekDatabase* fails with any of these four *dwSeekType* values, the function returns zero and leaves the current record pointer at the end of the database.

If you know the index of the record you are seeking in the database, the CEDB_SEEK_BEGINNING option is the one to use. For example, the user interface of the phone list application displays the phone database, sorted by the current sort order, in a list view control. It is convenient to locate the database record corresponding to the current list view selection by 0-based index. Specifying CEDB_SEEK_BEGINNING for *dwSeekType* implies that *dwValue* is the number of records to seek, that is, the zero-based index of the database record in the current sort order to be retrieved.

CEDB_SEEK_CURRENT moves the current record pointer forward or backward from the current record position the number of records specified by *dwValue*. If *dwValue* is positive, *CeSeekDatabase* seeks forward. The search is backward if *dwValue* is negative. CEDB_SEEK_

END is similar, except that it always seeks backward from the end of the database. It moves the current record pointer backward the number of records specified in *dwValue*.

In any of the above cases, if *CeSeekDatabase* is successful, it returns the object identifier of the record pointed to by the current record pointer.

The phone list application's Seek menu option brings up a dialog box that allows you to experiment with the *CeSeekDatabase* function. It allows you to specify various *dwSeekType* parameter values as well as property values for which to search. The application then displays the record found in the application window in the Search Results field. In the interest of keeping the phone list application to a reasonable size, this feature only allows you to specify *dwSeekType* values that perform database seeks by value.

Database Enumeration

Suppose you wanted to create a list of all the databases currently contained in the object store of a Windows CE device. Or perhaps you need to determine how much object store memory is being used by all the phone list databases available to a phone list management application. In this section we introduce the concept of *database enumeration*. Database enumeration allows applications to find all databases in the object store, or to find a subset of those databases as defined by a particular database type identifier.

Back when we discussed creating Windows CE databases, we introduced the concept of a database type identifier. This was the application-defined index that was passed as the *dwDbaseType* parameter of *CeCreateDatabase*. This index is used to identify all databases of a particular type. This type index is very important to database enumeration operations.

Database enumeration is done with two functions: *CeFindFirstDatabase* and *CeFindNextDatabase*. The enumeration process starts with a call to *CeFindFirstDatabase* to open an *enumeration context* for the type of database to be enumerated. The enumeration context is a handle through which the operating system can reference all databases with a particular database type identifier. *CeFindNextDatabase* is then called repeatedly to get the object identifier of each database of that type. These

functions are analogous to the file system functions *FindFirstFile* and *FindNextFile*.

The function *CeFindFirstDatabase* takes the form:

```
CeFindFirstDatabase(dwDbaseType);
```

The *dwDbaseType* parameter is the database type identifier of interest. This can be any application-defined database type. Note, however, that if you specify zero for *dwDbaseType*, an enumeration context for *all* databases in the object store is returned. If successful, *CeFindFirstDatabase* returns an enumeration context for this database type. If the function fails, it returns INVALID_HANDLE_VALUE.

CeFindNextDatabase looks like this:

```
CeFindNextDatabase(hEnum);
```

hEnum is the enumeration context returned by the *CeFindFirstDatabase* call. This function returns the object identifier of the next enumerated database, or zero if the function fails.

Database enumeration can be used to perform a number of operations in your applications. For example, if you wanted to delete all databases of a particular type, *CeFindNextDatabase* could be used to get the object identifier of each database of that type, and *CeDeleteDatabase* would then delete each of the databases.

At other times, you may wish to determine a particular set of attributes for each database of some type. An example might be getting the name of every database in the object store. Such features would be implemented using the *CeOidGetInfo* function. As an example, let's take a look at how you might get the total amount of memory in bytes in use by databases on a Windows CE device.

```
CEOID oidTemp;
CEOIDINFO oidInfo;
HANDLE hEnum;
DWORD dwBytes;
TCHAR pszText[129];
hEnum = CeFindFirstDatabase(0);
if (INVALID_HANDLE_VALUE!=hEnum)
{
  dwBytes = 0;
  while (oidTemp = CeFindNextDatabase(hEnum))
  {
   CeOidGetInfo(oidTemp, &oidInfo);
   dwBytes += oidInfo.infDatabase.dwSize;
```

```
    }
    CloseHandle(hEnum);
    wsprintf(pszText, TEXT("%ld bytes in use"), dwBytes);
    MessageBox(NULL, pszText,
      TEXT("Database Memory Consumption"), MB_OK);
  }
  else
  {
    MessageBox(NULL, TEXT("Invalid Enumeration Context"),
      TEXT("Enumeration Error"), MB_OK|MB_ICONEXCLAMATION);
  }
```

This sample opens an enumeration context into all of the databases in the object store of the device on which the code is executed by calling *CeFindFirstDatabase* with *dwDbaseType* argument of zero. *CeOidGetInfo* is then called for each database. A running byte count is updated using the size of each enumerated database. The database size is found in the *dwSize* member of the CEDBASEINFO structure returned as part of *oidInfo*. The total byte count is then displayed in a message dialog box.

Database Notifications

The last Windows CE database topic that needs to be covered is database notifications. In all of our examples we have passed NULL to the *hwndNotify* argument of *CeOpenDatabase*. However, this parameter can be used to specify a window that receives notifications whenever another thread of execution modifies the particular database before the thread that opened the database closes it. If the *hwndNotify* parameter is NULL, the thread opening the database is indicating it is not interested in receiving any such notifications.

There are three notifications that can be sent to the *hwndNotify* window. Although called notifications, they are actually Windows CE messages that are posted to the *hwndNotify* window. To respond to them, then, your application needs to include handlers for the ones you are interested in *hwndNotify*'s window procedure. The descriptions of the three notifications (messages) are given in Table 7.4.

The Contacts Database

Perhaps you will recognize this scenario from your college years. (It comes directly from mine.) It's the next to last week of one of your

Table 7.4 Database Notifications

NOTIFICATION NAME	MEANING	WPARAM	LPARAM
DB_CEOID_CHANGED	Object modified	CEOID of	CEOID of modified object modified object's parent
DB_CEOID_CREATED	Object created	CEOID of new object	CEOID of new object's parent
DB_CEOID_RECORD_DELETED	Record deleted	CEOID of deleted	CEOID of Object deleted object's parent

more grueling calculus courses. You've spent the entire time learning how to manually integrate impossibly complex functions that you are convinced you will never encounter in the "real world," using techniques such as the Laplace Transform and integration by parts. Then, almost as an afterthought, your calculus professor makes a brief foray into the subject of how to use an integral table. The chorus of grief is as varied as the students in the classroom, but can be paraphrased something like this: "You mean to tell me we've suffered through this integration business and could have used a cookbook all along?" Prepare for a trip down memory lane. I am about to pull the same thing on you now.

Windows CE provides a predefined database of its own for storing phone number and other personal and business contact information. A number of the applications that are traditionally supplied with Windows CE–based devices use this database. It therefore lives in the Windows CE operating system for all application developers to use. The *contacts database* stores many more useful properties than our phone list database example above. And it provides a complete application programming interface for performing such operations as adding, removing, and modifying information in the database. Given that this rich functionality exists in the operating system for free, why did I just painstakingly guide you through all of the mechanics of programming generic Windows CE databases?

The contacts database is just one example of the type of database that a typical Windows CE application may need to use. Much as an integral table cannot contain all of the cases an engineer might encounter in practice, the built-in features of an operating system like Windows CE cannot anticipate every application that it will be asked to support. It

is therefore crucial to have a well-established understanding of the fundamental capabilities of Windows CE in order to confidently approach any new programming challenge.

Applying the experience of our generic phone list application makes understanding the design and features of the contacts database a straightforward task. Since we have successfully explored the mechanics of generic Windows CE databases, the next sections will only briefly cover the highlights of the contacts database. To extend the classroom metaphor, the full details of using the contacts database are left to the student as an exercise!

NOTE

LINK WITH ADDRSTOR.LIB

To use the contacts database, your applications must link with ADDRSTOR.LIB and include the file ADDRSTOR.H.

Address Cards

Windows CE models the concept of a contact as *address cards*. This name is supposedly meant to conjure up the image of cards in a Rolodex. The address card is implemented as a structure with the following definition:

```
typedef struct _AddressCard
{
  SYSTEMTIME stBirthday;
  SYSTEMTIME stAnniversary;
  TCHAR *pszBusinessFax;
  TCHAR *pszCompany;
  TCHAR *pszDepartment;
  TCHAR *pszEmail;
  TCHAR *pszMobilePhone;
  TCHAR *pszOfficeLocation;
  TCHAR *pszPager;
  TCHAR *pszWorkPhone;
  TCHAR *pszTitle;
  / Other properties such as name, address, fax number, etc.
} AddressCard;
```

This structure is the contacts database analog of the PHONERECORD structure in our phone list database example. Each member of the AddressCard structure represents one of the properties in a particular contacts database record.

The properties of address card records are identified by *property tags*. The concept of property tags comes from the Microsoft Messaging Application Programming Interface (MAPI). In reality, though, a property tag is nothing more than a property identifier like PL_LASTNAME, PL_FIRSTNAME, PL_PHONENUMBER, and PL_DEPT in the phone list application. The property tags for the contacts database all have names of the form HHPR_*. For example, the birthday property has a property identifier HHPR_BIRTHDAY. These identifiers are used to specify the properties that are to be read from or written to records in the contacts database, as we will see a bit later.

Contacts Database Functions

The *ReadRecord* and *WriteRecord* functions in the phone list application were written as function wrappers that hide the internal details of the record data stored in the phone list database. In the same way, the contacts database functions work with AddressCards to allow the application programmer to think of contact information in a more natural way.

For example, to add a new record to the contacts database, an application calls *AddAddressCard*:

```
AddAddressCard(pac, poidCard, pindex);
```

pac is a pointer to the AddressCard structure that contains the contact information to be added to the contacts database. *poidCard* is a pointer to a CEOID that is used by *AddAddressCard* to return the object identifier of the new record if it is successfully added to the contacts database. *pindex* is also a return value indicating the position index of the new record in the database.

In the phone list application example, *WriteRecord* always added a new phone list record that contained data for every property in the record. *AddAddressCard* is more generic in that it allows applications to add records with any subset of AddressCard properties. For this to work, an application must specify which properties are valid for the AddressCard to be added. This is done using the *SetMask* function:

```
SetMask(pac, hhProp);
```

pac is again a pointer to an AddressCard structure. *hhProp* is any of the property tags defined for the contacts database. For example, to add

an address card in which only the company and department fields are valid, an application would do the following:

```
AddressCard ac;
CEOID ceoid;
int nIndex;
memset(&ac, 0, sizeof(AddressCard));
ac.pszCompany = TEXT("Acme Widgets");
ac.pszDepartment = TEXT("Bean Counting");
SetMask(&ac, HHPR_COMPANY_NAME);
SetMask(&ac, HHPR_DEPARTMENT_NAME);
/Now we can add the card
AddAddressCard(&ac, &ceoid, &nIndex);*
```

If all AddressCard properties are valid and are to be written to the database record, your application does not have to call *SetMask* for every property. Simply passing zero in the *hhProp* argument tells *Set-Mask* that all AddressCard properties are valid.

From our understanding of Windows CE databases, we can figure out what is going on inside *AddAddressCard*. *AddAddressCard* uses the mask prepared by the *SetMask* calls to know which values to extract from the AddressCard structure and which property identifiers to use to build up a CEPROPVAL array. As with any Windows CE database, all data read and write transactions ultimately boil down to reading and writing collections of CEPROPVAL structures.

Reading AddressCards is done with the *OpenAddressCard* function:

```
OpenAddressCard(oidCard, pac, uFlags);
```

oidCard is the object identifier of the record to be read. *pac* is a pointer to an AddressCard structure into which *OpenAddressCard* places the contacts properties read from the specified record. *uFlags* can be either OAC_ALLOCATE or zero. OAC_ALLOCATE says that separate memory is allocated for each string property, and the strings are copied from the object store into the particular AddressCard fields. If an application needs to modify the properties in a record, this value must be set.

If *uFlags* is zero, memory is not allocated for the string properties, and the TCHAR* members of the AddressCard record returned by *OpenAddressCard* simply point to the string data in the database. This is the technique to use when the particular AddressCard record is not going to be modified by the application, but simply displayed.

in the application's user interface. Applications can also use the *GetAddressCardProperties* function to read records from the contacts database.

The Complete Contacts Database API

The contacts database API provides functions for opening and closing the contacts database, as well as reading, writing and modifying AddressCards. In addition, you can use the API to sort the database on the various AddressCard properties and enumerate AddressCards. The complete contacts database API is given in Table 7.5.

Concluding Remarks

In this chapter we introduced the various aspects of programming Windows CE databases. You should now be comfortable creating databases and managing database records from your Windows CE applications. We specifically covered the topics of reading and writing database records, sorting databases, and searching for specific records in a database. The subjects of database notifications and database enumeration were also discussed.

The chapter also provided a brief introduction to the Windows CE contacts database and the contacts database API. This database, which is provided as part of the Windows CE operating system, may often come in handy when you write applications such as address books.

We continue our coverage of Windows CE persistent storage in Chapter 8 with a look at the Windows CE registry.

Table 7.5 The Contacts Database API

FUNCTION	PURPOSE
AddAddressCard	Adds an address card to the contacts database.
CloseAddressBook	Closes the contacts database.
CreateAddressBook	Creates the contacts database if it does not already exist.
DeleteAddressCard	Deletes the specified address card from the contacts database.
FreeAddressCard	Frees memory associated with an address card.
GetAddressCardIndex	Returns the position index of the specified address card in the contacts database.
GetAddressCardOid	Retrieves the object identifier of the address card as specified by its position index.
GetAddressCardProperties	Gets the properties of an address card.
GetColumnProperties	Retrieves the property tags corresponding to the columns by which the contacts database can be sorted.
GetMatchingEntry	Searches the contacts database for an address card with a name property containing the specified search string.
GetNumberOfAddressCards	Returns the number of address cards in the contacts database.
GetPropertyDataStruct	Retrieves a PropertyDataStruct for a specified contacts database property.
GetSortOrder	Returns the current contacts database sort order.
ModifyAddressCard	Changes the contents of an address card.
OpenAddressBook	Opens the contacts database if it exists.
RecountCards	Counts the number of address cards. This is necessary if another application modifies the contacts database while your application has it open.
SetColumnProperties	Specifies the properties on which the contacts database can be sorted.
SetMask	Specifies which properties are assigned in an address card.
SetSortOrder	Sets the contacts database sort order.

CHAPTER 8

Using the Windows CE Registry

Thus far in our investigation of Windows CE persistent storage, we have considered two mechanisms typically used for storing large amounts of data. The Windows CE file system is a useful way to store large amounts of data, such as documents, in a hierarchical directory structure. Windows CE databases are useful for storing and managing large numbers of data records such as phone list or contact information.

But what if your application has the need for small amounts of persistent storage? It would be overkill to create an entire database or directory structure just to keep track of a few numbers or strings.

Additionally, a particular database or file format is generally intended for use by the application that creates it. Applications generally are not prevented from accessing data in files or databases created by other applications. But to do so requires knowledge of a specific file format or database record design.

The Windows CE registry provides a generic mechanism for storing persistent information that is intended to be available on a system-wide basis. The registry has a simple hierarchical structure, and provides an application programming interface that makes it easy for any

application on a Windows CE device to find information available to the entire system.

One of the most familiar examples is the use of the registry by Microsoft's Component Object Model (COM) technology. COM uses the registry as a way to, among other things, make information about COM objects available to all interested parties.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

Program the Windows CE registry

Use the Remote Registry Editor

Registry Basics

Although it is part of the Windows CE object store, the registry is different from the Windows CE file system and databases. The registry does not store data as objects with unique object identifiers. The registry functions do not access data in the registry via a particular CEOID associated with a registry entry. The only similarity between the registry and these other two object store entities is that they are all used to store persistent data in object store RAM.

The Windows CE registry is organized as a hierarchical set of *keys*, *subkeys*, and *values*. Keys and subkeys are the registry analog of directories in the Windows CE file system. Keys can contain one or more subkeys. Keys and subkeys can contain one or more values, which are used to store the actual data contained in the registry.

Much as Windows CE databases can be assigned an application-specific database type, registry keys can be given a *class name*. Such a class name can be used to provide further distinction between registry keys.

At the root of the Windows CE registry hierarchy are three *primary keys*: HKEY_LOCAL_MACHINE, HKEY_CLASSES_ROOT, and HKEY_CURRENT_USER. Every registry subkey and value falls under one of these three primary keys.

Just as Windows CE represents files and databases as handles, there is also a handle data type for registry keys called HKEY. Many of the registry functions identify the key or subkey on which they are to operate by means of an HKEY handle.

The Windows CE registry can be used to store data of the following types: binary, DWORD, null-terminated Unicode string, Unicode symbolic link, or resource. The various registry functions refer to these data types by the symbols shown in Table 8.1. We'll see these data type values in the context of the various registry functions later.

Viewing the Windows CE Registry

The Remote Object Viewer allows you to view files and databases on a Windows CE device or in the emulation environment; similarly, the Windows CE Toolkit provides a Remote Registry Editor, which allows you to explore the registry in the emulation environment or on an actual Windows CE device. It also allows you to create, delete, and modify registry subkeys and values.

Table 8.1 Registry API Data Type Symbols

SYMBOL	MEANING
REG_BINARY	Binary data.
REG_DWORD	A 32-bit number.
REG_DWORD_LITTLE_ENDIAN	A 32-bit number in little endian format, i.e., the most significant byte of each word is the high-order byte.
REG_DWORD_BIG_ENDIAN	A 32-bit number in big endian format, i.e., the most significant byte of each word is the low-order byte.
REG_EXPAND_SZ	A null-terminated Unicode string that contains unexpanded references to environment variables, such as %PATH%.
REG_SZ	A null-terminated Unicode string.
REG_MULTI_SZ	An array of null-terminated Unicode strings. The array itself is terminated by two null characters.
REG_LINK	A Unicode symbolic link.
REG_RESOURCE_LIST	A device driver resource list.
REG_NONE	No defined data type.

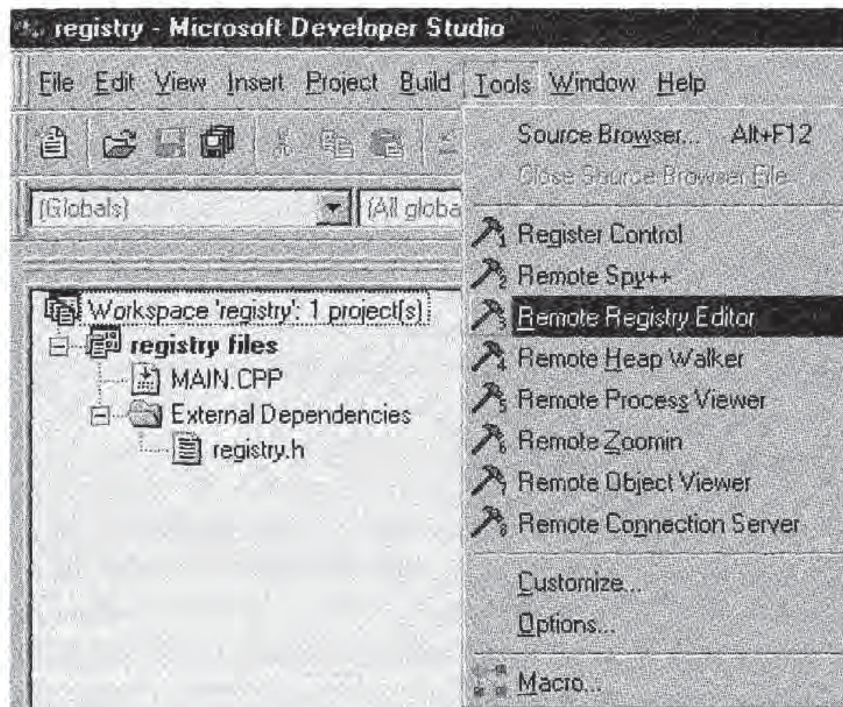


Figure 8.1 Opening the Remote Registry Editor.

You access the remote Registry Editor by choosing the Remote Registry Editor menu option from the Tools menu in the Microsoft Developer Studio development environment (Figure 8.1).

The Remote Registry Editor looks and works much like the Windows NT Registry Editor called regedit. In fact, the remote Registry Editor has all of the functionality of regedit and more.

When the Remote Registry Editor first appears, it contains two tree view nodes in the left-hand pane, labeled My Computer and My Emulation (Figure 8.2). The My Computer item is the root of all of the registry keys on the Windows NT machine on which you are running Microsoft Developer Studio. You can browse these keys and delete, add, or modify subkeys and values just as you would with regedit. Any changes that you make to the registry keys under My Computer are made in the registry of your Windows NT host machine.

The My Emulation tree view node is the root of all the registry keys contained by your Windows CE emulation object store. You can therefore make any modifications you like to your emulation registry by editing the subkeys and values under My Emulation.

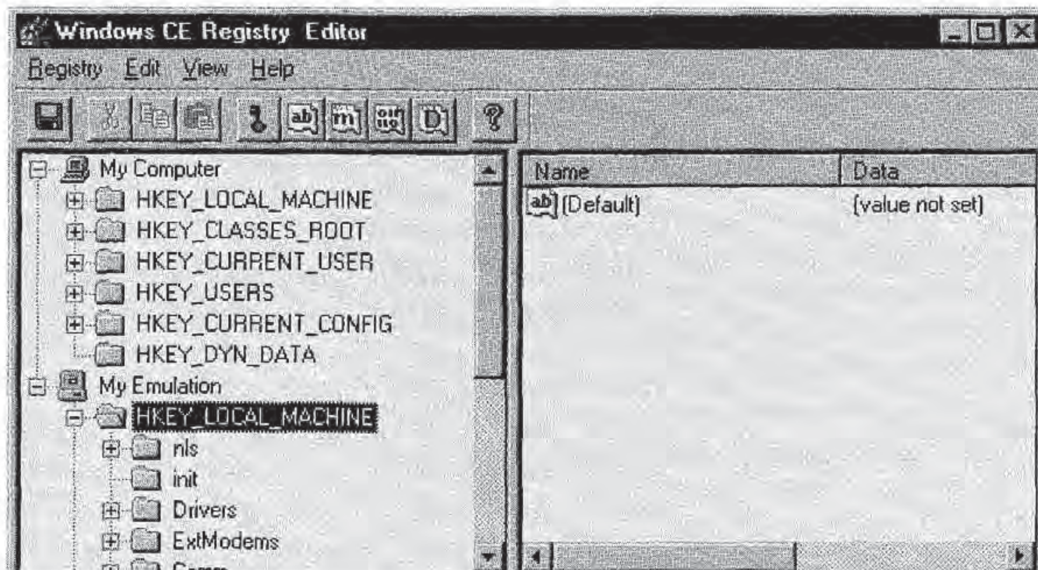


Figure 8.2 The Remote Registry Editor.

The Remote Registry Editor makes it easy for you to modify the Windows CE emulation registry manually. You will very often find yourself wanting to modify the registry in this way, particularly when debugging applications that use the registry. It would be a bit tedious if you could only edit the registry programmatically.

Adding and Removing Subkeys

You add and remove registry subkeys via the Remote Registry Editor just as you would with regedit on Windows NT.

For example, let's say you want to add a subkey called Applications under the My Emulation\HKEY_LOCAL_MACHINE key that is shown in Figure 8.2. To do this, expand the My Emulation node, and then expand the HKEY_LOCAL_MACHINE node. To add the Applications subkey, tap the HKEY_LOCAL_MACHINE key icon so that it is selected, and then select the New Key menu option from the Edit menu as shown in Figure 8.3.

As a result of this operation, the New Key dialog box shown in Figure 8.4 appears. Type "Applications", the name of the new subkey, in the edit field in this dialog box and press OK. The new subkey is then created under the HKEY_LOCAL_MACHINE key as shown in Figure 8.5.

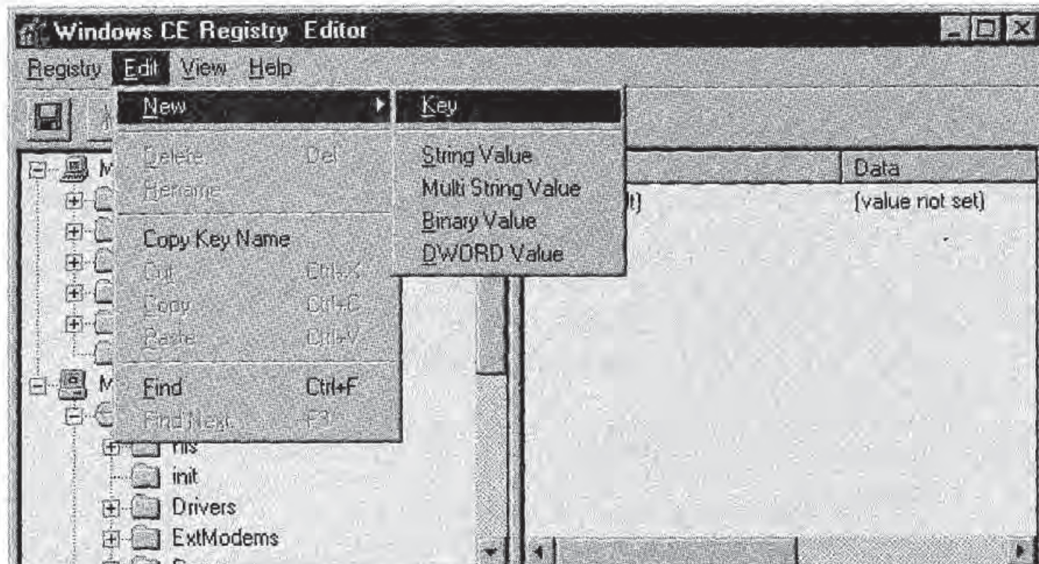


Figure 8.3 The New Key menu option.

To delete a subkey, simply tap on the subkey icon and choose the Delete option from the Remote Registry Editor Edit menu. Alternatively, pressing the Delete key will also delete the selected subkey.

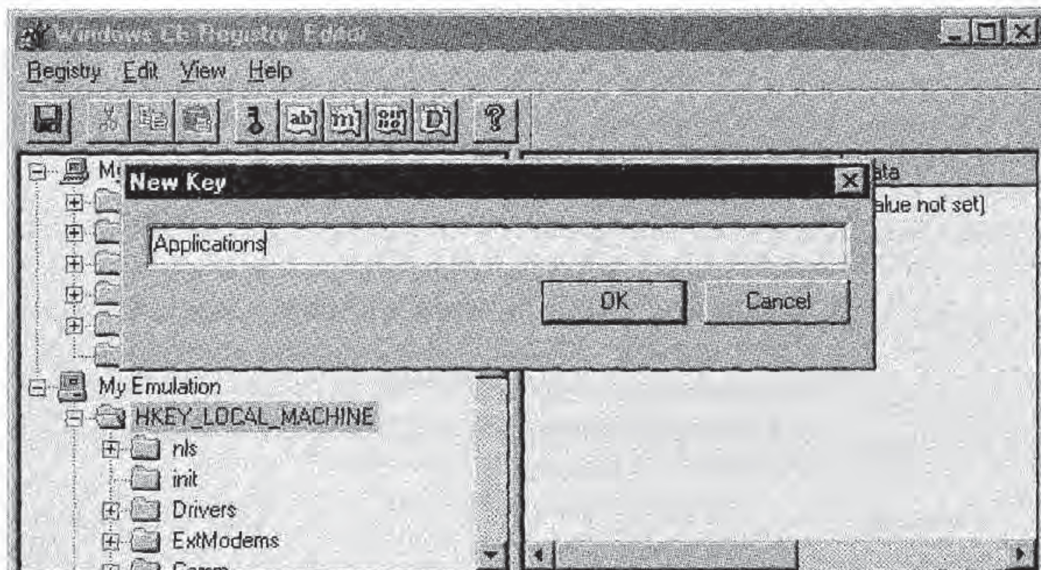


Figure 8.4 The New Key dialog box.

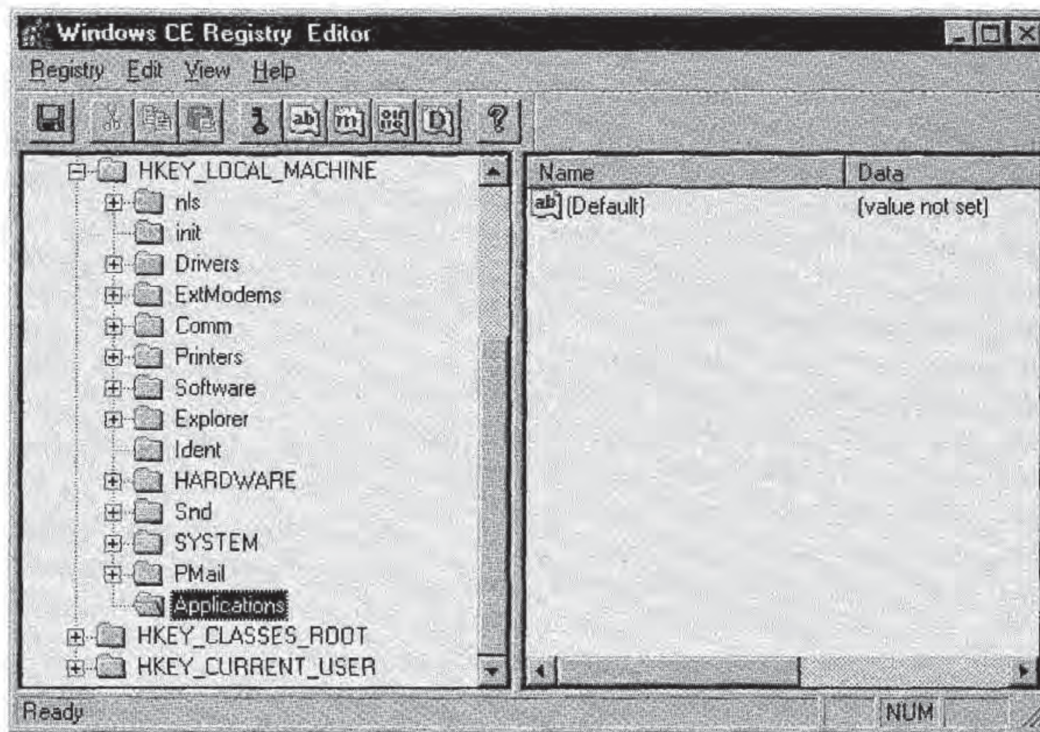


Figure 8.5 The newly created Applications subkey.

Adding and Removing Subkey Values

Registry values are added to the registry (for either Windows NT or CE emulation) in much the same way as subkeys are added. Instead of selecting the New Key menu option, you choose one of the other four New menu options (see Figure 8.3). Each of these options specifies that you want to create a value under the currently selected registry subkey. The data type of the new value is the type specified by the selected menu option.

For example, if you want to add a DWORD value called AppCount to the Applications subkey we created in the previous section, select the Applications and then select the New DWORD Value menu option from the Edit menu. The dialog box shown in Figure 8.6 will appear. Type the name of the new value and the initial value it contains in the corresponding text fields as shown. Press the OK button, and the new value appears in the right-hand pane of the Remote Registry Editor as shown in Figure 8.7.

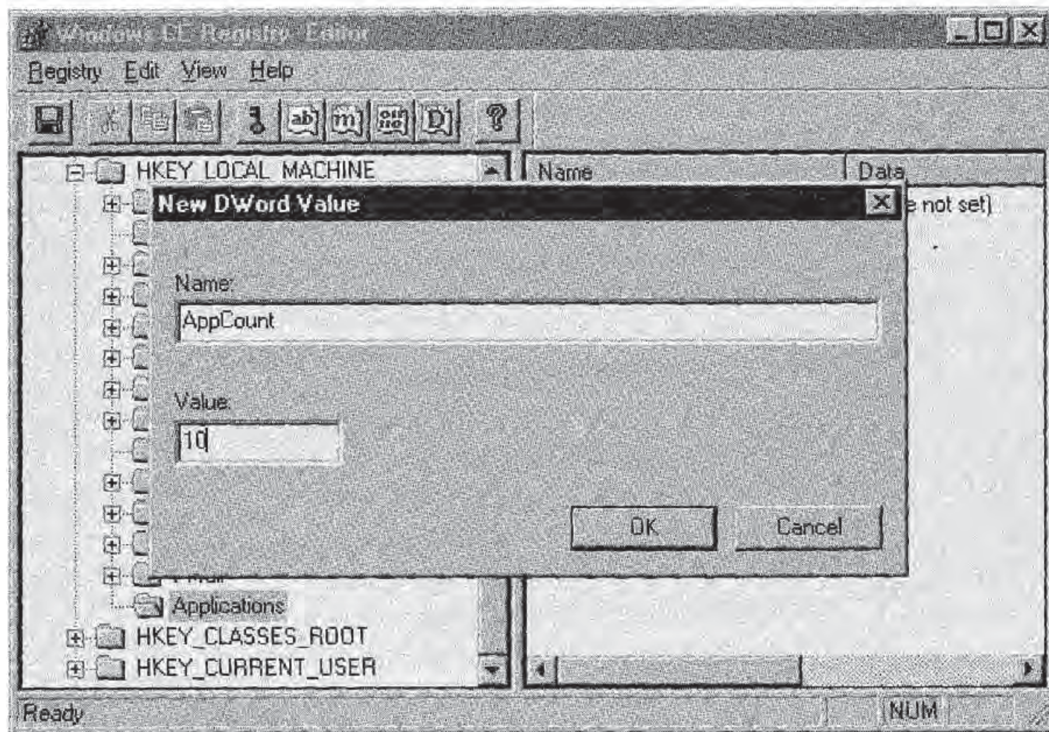


Figure 8.6 The new DWORD value dialog box.

Like registry subkeys, registry values are deleted by selecting the particular value and then choosing the Delete menu option from the Edit menu. Alternatively, pressing the Delete key will also delete the selected value.

The Remote Registry Editor also contains menu options for copying and renaming subkeys and values, as well as for generating registry files from the contents of the registry.

A Note on Registry Function Return Values

The various Windows CE registry API functions return ERROR_SUCCESS if they succeed. If a particular function fails, you might expect to call *GetLastError* in order to get additional clues as to why your function call failed.

Unfortunately, the registry functions do not set the current error code on failure with a call to *SetLastError*. Therefore, calling *GetLastError* tells you nothing in these cases.

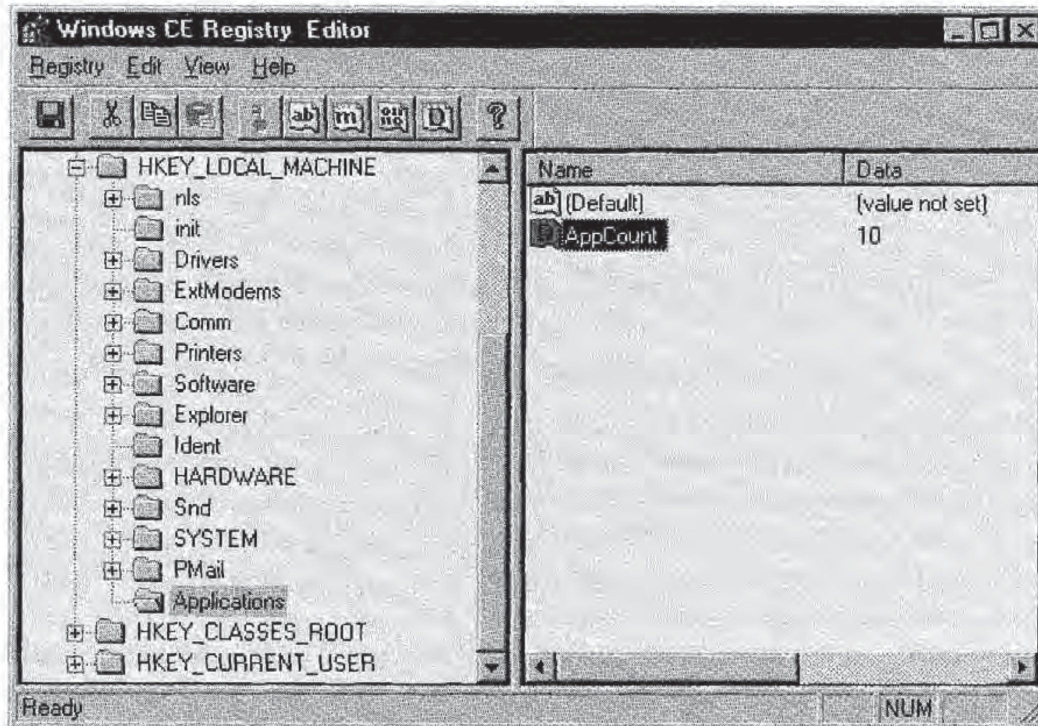


Figure 8.7 The newly created AppCount value.

The possible error return values are those defined in WINERROR.H. You can use the *FormatMessage* function with the FORMAT_MESSAGE_FROM_SYSTEM flag set in the *dwFlags* parameter to get the message text of the error.

Alternatively, you can use my preferred (and quicker) method: Keep WINERROR.H open in your editor while debugging your registry calls and search for the error codes manually!

Creating and Opening Registry Keys

The Windows CE registry API provides functions for both creating new registry keys and opening existing keys.

An application creates a registry key with *RegCreateKeyEx*. This function will create a new key in the registry if the specified key does not exist. If the key already exists, *RegCreateKeyEx* opens the key.

```
RegCreateKeyEx(hKey, lpszSubKey, Reserved, lpszClass,
  dwOptions, samDesired, lpSecurityAttributes,
   phkResult, lpdwDisposition);
```

It is easiest to discuss this function if we start by describing the *phkResult* parameter. This is a pointer to an HKEY, which is returned by *RegCreateKeyEx*. It is the key of the newly created or opened registry key.

The *hKey* parameter is one of the primary keys, such as HKEY_LOCAL_MACHINE, or the HKEY of some other registry key that the application has opened. *lpszSubKey* is the null-terminate string name of the subkey to create or open. This parameter cannot be NULL, implying that the result of *RegCreateKeyEx*, *phkResult*, is always a subkey of *hKey*.

The result of a *RegCreateKeyEx* call can be used to create subkeys of the key created in the first call. For example, assume we wish to create the two subkeys named HKEY_LOCAL_MACHINE\MyKeys and HKEY_LOCAL_MACHINE\MyKeys\Data. The following code would do the trick:

```
HKEY hMyKeys, hData;
//Create HKEY_LOCAL_MACHINE\MyKeys
RegCreateKeyEx(
  HKEY_LOCAL_MACHINE, //Primary key
  TEXT("MyKeys"),    //Subkey Name
  ...,          //Parameters we've
           //yet to discuss
  &hMyKeys,       //i.e., phkResult
  ...);
//Using previous result, create the key
//HKEY_LOCAL_MACHINE\MyKeys\Data
RegCreateKeyEx(
  hMyKeys,       //HKEY_LOCAL_MACHINE\MyKeys
  TEXT("Data"),     //Data subbkey
  ...,
  &hData,        //Handle to Data subkey
  ...);
```

The *Reserved* parameter of *RegCreateKeyEx* is just that, reserved for future use by Windows CE. This parameter must be set to zero.

lpszClassName is a null-terminated string containing the class of the key to be opened or created. If you are not interested in assigning a class name, this parameter can be NULL.

The parameters *dwOptions*, *samDesired*, and *lpSecurityAttributes* are ignored. *dwOptions* and *samDesired* should be zero, and *lpSecurityAttributes* should be NULL.

The final parameter, *lpdwDisposition*, is a DWORD pointer used as a return value which specifies whether the function created a new key or simply opened an existing key. The possible values returned are REG_CREATED_NEW_KEY and REG_OPENED_EXISTING_KEY.

Existing registry keys can alternatively be opened using *RegOpenKeyEx*:

```
RegOpenKeyEx(hKey, lpszSubKey, ulOptions, samDesired,
  phkResult);
```

As with *RegCreateKeyEx*, *hKey*, and *lpszSubKey* are the key and subkey of the key to open. In the case of *RegOpenKeyEx*, however, *lpszSubKey* can be NULL. *ulOptions* and *samDesired* are reserved and must be zero. *phkResult* is the same as in *RegCreateKeyEx*. It returns a handle to the opened key if *RegOpenKeyEx* is successful.

Reading and Writing Registry Values

The real data stored by the Windows CE registry is kept in the various *values* contained in each registry key. As described above, registry values can store data of a variety of types, making registry storage very flexible.

A registry key value is like a data slot. Each key can have one or more values for storing information. An application can read and write data from existing registry values, or it can create new values for its purposes. In either case, the application uses *RegSetValueEx*. This function assigns data to a specified registry value. If the value does not exist, it is created. The syntax of *RegSetValueEx* is:

```
RegSetValueEx(hKey, lpszValueName, Reserved, dwType,
  lpData, cbData);
```

hKey is the key that contains the value to which data is assigned. *lpszValueName* is the null-terminated Unicode string name of the value to set. *Reserved*, again, is reserved for later versions of Windows CE and as such must be set to zero. *dwType* is one of the data type specifiers. It tells *RegSetValueEx* what type of data is being placed in the registry value. *lpData* is a constant BYTE pointer containing the data to be assigned to the value. *cbData* contains the size, or length in bytes, of the data in *lpData*.

As an example, let's say that we wish to create a registry key under the HKEY_LOCAL_MACHINE primary key called "Test." We then wish to create 20 values in that key named Value0, Value1, and so on up to Value19. Additionally, we want to assign each of these values the DWORD integer corresponding to the number in the value name. For example, the number in Value0 will be 0 and the number in Value1 will be 1.

To accomplish this, our application would do the following:

```
HKEY hKeyTest;
DWORD dwDisp, dwSize;
TCHAR pszValue[MAX_STRING_LENGTH];
int i;
if (ERROR_SUCCESS != RegCreateKeyEx(HKEY_LOCAL_MACHINE,
    TEXT("Test"), NULL, NULL, 0, 0,
     NULL, &hKeyTest, &dwDisp))
{
  MessageBox(NULL, TEXT("Could Not Create Key"),
    TEXT("Registry Error"), MB_ICONEXCLAMATION|MB_OK);
}
else
{
  for (i=0; i<20; i++)
  {
   dwSize = sizeof(DWORD);
   wsprintf(pszValue, TEXT("Value%d"), i);
   if (ERROR_SUCCESS != RegSetValueEx(hKeyTest,
    pszValue, NULL, REG_DWORD, (CONST BYTE*)&i,
     dwSize))
   {
    MessageBox(NULL, TEXT("Could Not Set Value"),
     pszValue, MB_ICONEXCLAMATION|MB_OK);
   }
  }    //End of for i loop
}
```

The first thing we do is attempt to create the HKEY_LOCAL_MACHINE\Test registry key. If this *RegCreateKeyEx* call fails, we display a message box to that effect. If the create was successful, we proceed to the for loop, which sets the 20 registry values. *RegSetValueEx* creates each of the registry values if they don't already exist in the registry. The *RegSetValueEx* call passes REG_DWORD as the *dwType* parameter, indicating that the value to be written is a DWORD. The name of each registry value is constructed with the *wsprintf* call.

Reading a registry value is done with the function *RegQueryValueEx*:

```
RegQueryValueEx(hKey, lpszValueName, lpReserved,
  lpType, lpData, lpcbData);
```

hKey and *lpszValueName* have the same meaning as in *RegSetValueEx*. *lpReserved* is a reserved DWORD pointer and must be NULL. *lpType* is a DWORD pointer that contains the registry value's data type. *lpData* is a BYTE pointer in which the function returns the value data. *lpcb-Data* is a DWORD pointer that contains the length in bytes of the data to be read from the registry value.

The *lpcbData* parameter deserves some illumination. Otherwise it will haunt your every registry query. You must assign the number of bytes to be read from the particular registry value to the DWORD pointed to by the *lpcbData* parameter. So far so good. But *RegQueryValueEx* uses this parameter as a return value as well. It returns the actual number of bytes read from the registry key, which may indeed be different from the number you said to read. For example, you may expect a string you are querying to be 50 bytes long. If the string is really 15 bytes long, *RegQueryValueEx* will return 15 in *lpcbData*.

This still sounds OK? Well, maybe, until you try using *RegQueryVal-ueEx* to read multiple registry keys in a loop *and do not reassign lpcs-Data to the number of bytes you want to read for each query.*

Let's look at the following example:

```
int i;
DWORD dwSize;
DWORD dwType;
TCHAR pszText[128];
dwSize = 128;
dwType = REG_SZ;
for (i=0; i<5; I++)
{
  wsprintf(pszText, TEXT("Value%ld"), i);
  RegQueryValueEx(
   hKeyTest,
   pszValue,
   NULL,
   &dwType,
   (LPBYTE)pszText,
   &dwSize);
  //Do something with dwValue
}
```

You expect this code to read five Unicode strings of length 128 bytes from five registry values named Value0 through Value4.

But what if any of the actual strings is less than 128 bytes long? *RegQueryValueEx* will return the real length of that string in *dwSize*. The next *RegQueryValueEx* call will then say to read only as many bytes as were in the last string. At this point, you can count on all the rest of the values read to be completely unreliable. Believe me, the bugs that result from such an oversight are very difficult to track down.

The moral of this story is: Set the *lpcbData* value properly for each and every call to *RegQueryValueEx*.

The *RegQueryInfoKey* Function

There is one more registry function related to reading information about registry keys. Whereas *RegQueryValueEx* reads the actual value data from a specified registry key value, *RegQueryInfoKey* allows your application to determine the number of subkeys and values that a particular registry key contains. This becomes important when you need to iterate over a set of subkeys or values, which is the subject of the next section.

RegQueryInfoKey also does other work for you, such as determining the class name of the particular key and the length of that class name, as well as the maximum subkey, class, and value name lengths of all subkeys and values associated with the queried key.

The syntax of *RegQueryInfoKey* is:

```
RegQueryInfoKey(hKey, lpClass, lpcbClass, lpReserved,
  lpcSubkeys, lpcbMaxSubKeyLen, lpcbMaxClassLen, lpcValues,
   lpcbMaxValueNameLen, lpcbMaxValueData,
    lpcbSecurityDescriptor, lpftLastWriteTime);
```

hKey is the HKEY of the key to be queried. *lpClass* is a Unicode string buffer used by the function to return the class of *hKey*. This parameter can be NULL if your application is not interested in class name information. *lpcbClass* is a DWORD pointer used to return the length of the string returned in *lpClass*. *lpcbClass* should be NULL if *lpClass* is NULL. *lpReserved* is reserved and should be NULL.

The next parameter is *lpcSubkeys*. This is a DWORD pointer in which *RegQueryInfoKey* returns the number of subkeys contained by *hKey*. This parameter can be NULL if this information is not of interest. *lpcbMaxSubKeyLen* returns the length in characters of the longest subkey name. For some mysterious reason, this count does *not* include the

null-terminating character. Compare this with the *lpcbData* parameter of *RegQueryValueEx*, which does include the null-terminator in cases where it is used to read string values from the registry.

lpcbMaxClassLen returns the length of the longest class name of any of the subkeys contained by *hKey*. No null-terminator here, either. Both *lpcbMaxSubKeyLen* and *lpcbMaxClassLen* can be NULL.

lpcValues returns the number of values contained by the queried key. This can be NULL if you are not interested in this information. *lpcb-MaxValueNameLen* returns the length of the longest value name string. This parameter can be NULL, and again, does not include the null-terminator in its string character length count.

lpcbMaxValueData and *lpcbSecurityDescriptor* are not used. They should therefore be set to NULL.

Finally, *lpftlastWriteTime* is not used and can be NULL. Under Windows NT, this parameter could be used to determine the last time a key or any of its values were changed. Windows CE, however, does not provide this feature.

A typical use of *RegQueryInfoKey* is to determine the number of subkeys and values contained by a particular registry key. To continue our example of the HKEY_LOCAL_MACHINE\Test key, let's write the code necessary to find the number of subkeys and values in this key:

```
HKEY hKeyTest;
DWORD dwSubKeys, dwValues;
RegOpenKeyEx(HKEY_LOCAL_MACHINE, TEXT("Test"),
  0, 0, &hKeyTest);
RegQueryInfoKey(hKeyTest, NULL, NULL, NULL,
  &dwSubKeys, NULL, NULL, &dwValues, NULL,
   NULL, NULL, NULL);
```

We first open the HKEY_LOCAL_MACHINE\Test registry key. The *RegQueryInfoKey* call then gets the number of subkeys in *dwSubKeys*, and the number of values in *dwValues*.

Notice all of the NULL parameter values. In this example we are not interested in the class names, class name lengths, value name lengths, and the other sundry things that this function can return. Therefore, the parameters corresponding to these pieces of information are all NULL.

Now that our applications can get subkey and value counts, they have all the information they need to iterate over subkeys and values, read-

ing or writing data as needed. All we need to do is introduce the registry enumeration functions.

Enumerating Registry Keys and Values

Enumeration is the process of iterating over a set of registry keys or values and extracting information about each one as it is iterated.

The *RegEnumValue* Function

The first registry enumeration function is *RegEnumValue*. This function is useful, for example, inside of loops where your application wants to read the data from every value in a subkey. Given the handle to an open key, an application can use this function to iterate over all values of that key, reading their data values, without knowing the names of the values being read. In fact, *RegEnumValue* reads both the value data and value name for you.

The syntax of *RegEnumValue* is:

```
RegEnumValue(hKey, dwIndex, lpValueName,
  lpcbValueName, lpReserverd, lpType,
   lpData, lpcbData);
```

hKey is the handle of the open key whose values are being read. *dwIndex* is the index of the value to retrieve. The name of the value corresponding to *dwIndex* is returned in the *lpValueName* parameter.

lpcbValueName is a pointer to a DWORD that contains the size of the *lpValueName* buffer. This parameter requires all of the caveats pointed out with the *lpcbData* parameter of *RegQueryValueEx*. Specifically, you specify the number of bytes you think the *lpValueName* string will be, and *RegEnumValue* returns the actual length through the same parameter. Hence, you need to reset this value appropriately for every *RegEnumValue* call. To further complicate matters, when you specify a value in *lpcbValueName*, you must take into account the null-terminating character of the *lpValueName* string that will be returned. But the value of *lpcbValueName* returned by *RegEnumValue* does not contain the null-terminating character.

The *lpReserved* parameter should again be NULL. *lpType* returns the type of data in the registry value being enumerated.

lpData points to the data read from the registry value. Finally, *lpcbData* is used both to pass in the expected number of bytes to be read, and to return the actual number of bytes returned in *lpData*. The same caveats apply here as with the *lpcbData* parameter of *RegQueryValueEx*.

You use *RegEnumValue* by initially setting *dwIndex* to zero for the first call of the function, and then incrementing it for each successive *RegEnumValue* call. Let's extend our previous example. The code below shows how to read the number of values associated with the key HKEY_LOCAL_MACHINE\Test as before. It then iterates over all of the registry values and reads their contents:

```
#define MAX_STRING_LENGTH 129
HKEY hKeyTest;
DWORD dwIndex;    //Loop index
DWORD dwValueIndex; //Index of value to read
DWORD dwValues, dwSubKeys; //Number of values, subkeys
DWORD dwSize;      //Size of data returned by
           //RegEnumValue, i.e., the
           //lpcbData parameter
DWORD dwSizeValue;  //Size of the value name string
           //read, i.e., the RegEnumValue
           //lpcbValueName parameter
RegOpenKeyEx(HKEY_LOCAL_MACHINE, TEXT("Test"),
  0, 0, &hKeyText);
if (ERROR_SUCCESS == RegQueryInfoKey(hKeyTest,NULL,
  NULL, NULL, &dwSubKeys, NULL, NULL, &dwValues,
   NULL, NULL, NULL, NULL))
{
  dwValueIndex = 0; //Init to zero to read first value
  for (dwIndex=0; dwIndex<dwValues; dwIndex++)
  {
   dwSizeValue = MAX_STRING_LENGTH;
   dwSize = sizeof(DWORD);
   if (ERROR_SUCCESS==RegEnumValue(
    hKeyTest, dwKeyIndex++,pszValue,
     &dwSizeValue, NULL, &dwType,
      (LPBYTE)&dwValue, &dwSize))
   {
    //Do something with the data
    //read in dwValue
   }
  }      //End of for dwIndex loop
}        //End of if (ERROR_SUCCESS==RegQueryInfoKey)
         //statement
RegCloseKey(hKeyTest);    //Close the key
```

The first part of this example is essentially the same as in the previous example. After opening the registry key HKEY_LOCAL_MACHINE\

Test, we call *RegQueryInfoKey* to determine the number of values in the key.

The code then reads each of the registry values with a call to *RegEnumValue*. The for loop iterates over *dwValues*, the number of registry values as determined by the *RegQueryInfoKey* call. Notice that *dwValueIndex* is initialized to zero, and then incremented with every *RegEnumValue* call. This ensures that each registry key value is read in order.

Also, *dwSizeValue* and *dwSize* are reset after every *RegEnumValue* call. Recall that the *lpcbValueName* and *lpcbData* parameters of *RegEnumValue* are used by the function as return values, and, therefore, the values you initially set may be gone.

At the end we close the key with *RegCloseKey*:

```
RegCloseKey(hKey);
```

The *RegCloseKey* function simply closes the open registry key indicated by the *hKey* parameter.

The *RegEnumKeyEx* Function

The second registry enumeration function is *RegEnumKeyEx*. You can think of it as the subkey analog of *RegEnumValue*. Whereas *RegEnumValue* is used to extract data and other properties from values associated with registry subkeys, *RegEnumKeyEx* extracts information about the subkeys of a specified registry key. The parameters of this function are also analogous to those for *RegEnumValue*.

```
RegEnumKeyEx(hKey, dwIndex, lpName, lpcbName, IpReserved,
  lpClass, lpcbClass, lpftLastWriteTime);
```

hKey is the key whose subkeys are being enumerated. *dwIndex* represents the index of the subkey to enumerate. *lpName* is a Unicode string buffer that will return the name of the subkey enumerated. *lpcbName* works just as it does in all the other registry functions where we've seen it. The expected size of *lpName* goes in, the real size comes out. It's up to your application to make sure it is always initialized properly. *lpReserved* is reserved and must be NULL.

RegEnumKeyEx can return registry subkey class information. The *lpClass* and *lpcbClass* parameters are used for this purpose. *lpClass* is a Unicode string pointer that contains the class name of the enumerated key when the function returns. *lpcbClass* is a pointer to a DWORD con-

taining the length of *lpClass*. The same caveats about *lpcbName* apply to *lpcbClass*: It is used as both an input and a return parameter. If your application does not use class information, simply set *lpClass* and *lpcb-Class* to NULL.

The final parameter, *lpftLastWriteTime*, is not used under Windows CE. You can therefore set it to NULL.

RegEnumKeyEx is typically used to read through a hierarchy of registry keys and subkeys. *RegQueryInfoKey* gets the number of subkeys. Each of these subkeys is then enumerated with *RegEnumKeyEx* and *RegEnumValue*.

NOTE

BE CAREFUL DURING KEY ENUMERATION

Your applications should not perform any operation that changes the number of subkeys or values of a registry key while it is being enumerated. Both *RegEnumValue* and *RegEnumKeyEx* use the index of the key or value being enumerated. Changing the number of subkeys or values will throw any iterative enumeration off and lead to unexpected results.

Deleting Registry Keys and Values

The last thing to know about the registry is how to delete keys and values. The registry API provides two functions, *RegDeleteKey* and *RegDeleteValue*, for these purposes.

RegDeleteKey deletes a specified registry and all of its values. It will not delete all of the subkeys contained in the specified subkey. As with directories in the file system, your applications must iterate through the entire subkey hierarchy of a particular key in order to delete all of its subkeys and their values.

The syntax of *RegDeleteKey* is:

```
RegDeleteKey(hKey, lpSubKey);
```

hKey is the handle of an open key, or one of the three primary keys. *lp-SubKey* is the Unicode string name of the subkey to delete. This parameter cannot be NULL. As described above, *RegDeleteKey* will not delete keys that contain subkeys.

So, in order to delete our HKEY_LOCAL_MACHINE\Test subkey, we would could write the following:

```
RegDeleteKey(HKEY_LOCAL_MACHINE, TEXT("Test"));
```

But note that the following cannot be done (assume the *hKeyTest* is the open HKEY of the HKEY_LOCAL_MACHINE\Test subkey):

```
RegDeleteKey(hKeyTest, NULL);
```

Deleting a value is done with *RegDeleteValue*:

```
RegDeleteValue(hKey, lpValueName);
```

As with *RegDeleteKey*, *hKey* is one of the primary keys or a handle to an open subkey. *lpValueName* is the Unicode string name of the value to be deleted from this subkey.

The Registry Sample Application

The sample application for this chapter is nothing to write home about. It simply packages all of the examples we discussed in this chapter into a Windows CE application. All of the registry functionality it includes is done in *WinMain* before the application even hits its message loop.

The sample creates our favorite HKEY_LOCAL_MACHINE\Test subkey and twenty values. It then queries the subkey for the number of values it contains, and then enumerates each of these values, adding the registry name and value to a list box.

The application also shows how to create and enumerate nested subkeys and values. It also creates the HKEY_LOCAL_MACHINE\Test\SubKey0 and HKEY_LOCAL_ MACHINE\Test\SubKey1 subkeys. It then adds two Unicode string values to each of these subkeys. Finally, the application shows you how to read a hierarchy of nested subkey values using *RegEnumKeyEx* and *RegEnumValue*.

The user interface of this application is very basic (Figure 8.8). It has an Exit button in the main application window for terminating the application. It also includes a list box for displaying the values read by the application from the Windows CE registry.

All of the source code and the project files for building this application can be found on the companion CD in the directory \Samples\registry.

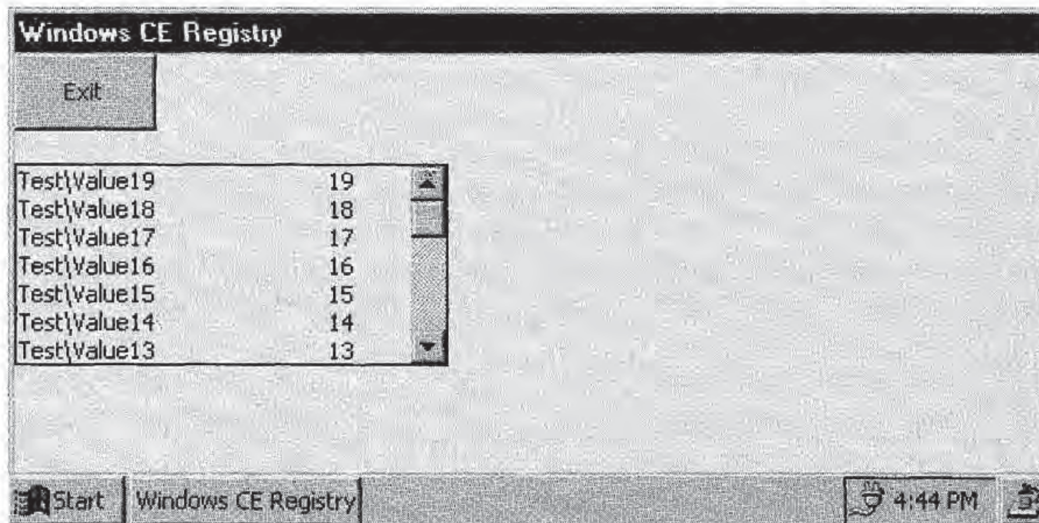


Figure 8.8 The Registry sample application.

Concluding Remarks

In Part II, we have covered how to program the various persistent storage features available under Windows CE. You can now write applications capable of taking advantage of the Windows CE file system and the registry. For more complex data storage needs, your applications can create their own custom databases.

If you stopped reading at this point, you would be well equipped to solve most Windows CE application programming problems. You know all about persistent storage now, and Part I presented the most common application user interface components. You could therefore begin writing applications capable of storing user information and interacting with users.

But most companies building Windows CE–based software and hardware hope to attract customers with features such as nontraditional user interfaces and desktop connectivity. So, up to this point, you really only have half of the Windows CE story. In the next sections we discuss more advanced user interface programming techniques, as well as the area of desktop connectivity.

PART THREE

Windows CE User Interface Programming

Microsoft has big plans for Windows CE. The company hopes to make Windows CE become for consumer electronics what Windows 98 and Windows NT have become for personal computers. The Microsoft vision puts Windows CE on everything from handheld computing devices to Internet-enabled telephones. Although you shouldn't exactly count on (let alone want) Windows CE to toast your bread in the morning, you can expect a growing number of consumer electronics companies to market devices driven by the Windows CE operating system.

Many have compared this phase of the technology revolution to the introduction of the first personal computers. To their way of thinking, the growth of handheld and mobile computing devices and the introduction of new sophisticated consumer devices is the next "paradigm shift" the computer industry has been waiting for.

Along with this wave of innovation in product and software design has come the usual army of designers. These are the folks that are tasked by companies to design the user interfaces for next generation products.

Using such nontechnical personnel who are dedicated exclusively to designing the look and feel of Windows CE applications is usually justifiable. Someone needs to constantly be interacting with a product's potential user community to try and figure out what consumers want from a particular Windows CE device or application.

A second argument usually given for hiring user interface designers goes something like this: Since many companies are designing devices intended for a consumer audience (a polite way of saying non–PC-savvy users), this new breed of Windows CE-based devices must first and foremost *not look like PCs*.

This is generally woefully interpreted to mean, "make the user interface look as different from the traditional Windows user interface as possible." Unfortunately, this often leads companies to release products with user interfaces that make their products *more difficult to use than the PC*.

While many visual improvements on the standard Windows CE interface components are indeed useful in order to more clearly convey the meaning of various user interface elements, many Windows CE user interfaces end up just as cluttered, busy, and confusing as the desktop applications they were meant to improve upon.

Furthermore, those improvements often come at enormous software development cost. One of the largest mistakes being made today by companies pursuing their fortunes through Windows CE is to adamantly insist that the wishes and visions of interface design teams be realized at any cost.

Windows CE is nothing more than a big piece of software. Like any piece of software, there are things Windows CE can do and things it can't. There are things it can do easily, and things it can be made to do with lots and lots of ugly application code. Of all the people in your organization, no one understands the strengths and limitations of Windows CE better than your software engineering staff.

Part III of this book focuses on the vast subject of implementing Windows CE user interfaces and controls that look different from the standard Windows CE model. The focus will be on features provided by Windows CE that allow application programmers to customize the look and feel of the various parts of a user interface. Like all the other chapters in this book, the chapters in this section are primarily intended for Windows CE software developers.

However, if there is one part of this book that I recommend be read by project management, application developers, and user interface designers alike, it is this one. If your entire organization understands the limits and abilities of Windows CE, more realistic user interface designs and more realistic development schedules will result. This ultimately means that you will do what so few companies so far have done: release a Windows CE–based product into the marketplace.

A final word, and then it's off the soapbox and back into programming: If you are in charge of a Windows CE development project,

involve your software engineers in the user interface design process from day one. And heed their words if they say certain things can't be done; what they generally mean is that they can't be done before your competition begins shipping.

What We Will Learn

In the following chapters, we will cover the following Windows CE features for implementing a custom user interface. The order in which they are presented follows the progression of simplest feature to most complex. The features covered are:

- Owner draw controls
- Customizing the application's main window class
- The Windows CE custom draw service
- Implementing custom controls
- Window subclassing

This section also includes chapters on programming the Windows CE HTML Viewer control, and some of the Palm-size PC input techniques such as the rich ink control and the voice API.

We begin in the next chapter with a discussion of owner draw control techniques. This discussion will focus on applying these techniques to owner draw buttons.

CHAPTER 9

Owner Draw Controls and Custom Window Classes

The easiest way to change the appearance of a Windows CE control is to make the control *owner draw*. An owner draw control is a control whose parent window, not the control itself, takes responsibility for creating the physical appearance of the control.

A number of Windows CE controls support the owner draw feature. The techniques for programming owner draw controls is the same for any supported control. This chapter therefore presents these techniques in the context of owner draw buttons.

Why Focus on Owner Draw Buttons?

Of all of the controls that can be used in Windows CE applications, push buttons are probably the most common. Buttons appear everywhere. They fill dialog boxes, letting users choose between various application feature options. Buttons are universal in providing the OK-Cancel choices for committing user input. Buttons send our e-mail and help us navigate around Web pages in Web browser software.

Given how common this control is in applications, it is no wonder that most Windows CE software vendors are interested in changing the

Other Windows CE Owner Draw Controls

The concepts presented in this chapter can be applied to more than just owner draw buttons. Several other controls in Windows CE, such as the list view control and the tab control, to mention a few, support the owner draw functionality.

For example, a tab control created with the TCS_OWNERDRAWFIXED style sends WM_DRAWITEM messages to its parent just like owner draw buttons do. The DRAWITEMSTRUCT structure passed with each message contains information about the individual tab control items. The owner window can use this information to draw the tab items any way it pleases.

The Windows CE controls that support owner draw functionality are the header control, list view control, status bar control, tab bar control, and, of course, the button control.

basic appearance of the button control to differentiate their user interfaces from those of their competitors.

Fortunately, modifying the appearance of button controls is relatively simple. Windows CE, like its Win32 desktop relatives, provides this feature with a programming technique called *owner draw buttons*. With an owner draw button, as with regular push buttons, all of the messaging behavior such as detecting stylus taps and generating WM_COMMAND messages is taken care of for you by Windows CE. But all aspects of the appearance of an owner draw button must be implemented by the owner of the button. Hence the name owner draw button.

The basic concept of owner draw buttons is one that will already be familiar to the more experienced Windows programmers who read this book. But it is presented here nonetheless for a variety of reasons.

First, organizations that are developing Windows CE–based products are almost universally consumed with a passion for making their user interfaces look like anything but a desktop PC. Therefore, it is useful to review even the most basic techniques for customizing an application's look and feel.

Second, the details of owner draw buttons may be unfamiliar to application programmers who come to Windows CE with lots of experience programming with the Microsoft Foundation Classes. This chapter, like Chapters 2 and 3, is partly motivated by a desire to familiarize such programmers with key Windows CE features at the API level.

And third, not everyone is familiar with every traditional Windows programming technique. Many Windows CE software developers come from embedded systems backgrounds. They often need to understand Windows CE programming from both a systems and an applications level. Such readers can benefit from a description of the basics.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

Use owner draw buttons in your applications

Use Windows CE timers

Draw graphics using offscreen bitmaps

Design custom window classes

The Example Application

The example application demonstrating the concepts presented in this chapter is found in \Samples\kiosk on the companion CD. The executable is called KIOSK.EXE.

This application attempts to demonstrate what the front end of a kiosk-style Windows CE device might look like. A common example of a kiosk familiar to most people is the bank automatic teller machine. A kiosk can be described as a dedicated single-purpose device that performs one service for a user. Other examples include the computerized video catalogs that are common in many movie rental chains or those do-it-yourself photo enlargers that are making their way into photo finishing shops.

KIOSK.EXE is an example bank ATM user interface. The user's choices are represented on the screen as owner draw buttons. The main application window and the secondary window that contains the owner draw buttons are made to look different from standard Windows CE windows. This is done by customizing certain aspects of the window classes that govern these windows.

In addition to providing insight into owner draw buttons and custom window classes, this example will also provide a review of some

important graphics functions of the Windows CE Graphics, Windowing, and Event Subsystem.

NOTE

THIS CHAPTER CONTAINS TWO SAMPLE APPLICATIONS

This chapter actually contains two sample applications. KIOSK.EXE is the main one, demonstrating the entire kiosk user interface. \Samples\button contains the source code for the short owner draw button example, BUTTON.EXE, shown in Figures 9.1 and 9.2.

The Anatomy of a Windows CE Control

Understanding owner draw buttons and how to use them will be easier if we first look at how Windows CE controls such as buttons are implemented.

In each of the sample applications that we have encountered in this book, we have created and registered a window class. This window class has described some of the visual aspects of the main application window, as well as the behavior of the main window by means of the window procedure assigned to the window class. In each of the sample applications, we have created just one instance of this window class. But there is nothing stopping us from using this main window's window class to create multiple instances of the window class, each with a different set of window styles, dimensions, window caption text, and the like.

Windows CE controls are used in exactly this way. Each control in a Windows CE application is just a special type of window. Windows CE controls have their own window classes, and hence, their own window procedures controlling their behavior and appearance.

Let's look at the button control class in closer detail. Deep in the implementation of the Windows CE Graphics, Windowing, and Events Subsystem (GWES) lives the implementation of the Windows CE button control class. Somewhere in the GWES code the button class is defined, and registered with a call to *RegisterClass*, just as you register your own window classes in your applications. The button class that gets registered includes a window procedure that implements all of the behavior of every button that appears in any Windows CE application.

The default appearance of buttons, a gray rectangle with text, is implemented by the button class window procedure's WM_PAINT message handler. What happens when you press or release a button is dictated by the WM_LBUTTONDOWN and WM_LBUTTONUP handling code.

When you create a button control, you specify the button class name in the *CreateWindow* (or *CreateWindowEx*) call:

```
HWND hwndButton;
hwndButton = CreateWindow(TEXT("BUTTON"), ...);
```

This tells Windows CE to create an instance of the window class identified by the Unicode string "BUTTON". All messages sent to *hwndButton* are thus handled by the window procedure identified by that window class. Hence, *hwndButton* knows how to walk and talk like a button control.

When a button needs to be repainted, the button class window procedure does all of the work. This is how Windows CE provides the default appearance and behavior of buttons and all other child or common controls.

How Owner Draw Buttons Are Different

Owner draw buttons work almost exactly like other Windows CE controls just described. The only difference is that the button control's parent window, not the button, is responsible for defining the appearance of the button.

An owner draw button behaves in all other ways like a non–owner draw button. For example, it still sends WM_COMMAND messages to its parent when pressed. The difference is that in the case of an owner draw button, the button skips its default WM_PAINT processing and instead sends its parent a WM_DRAWITEM message. The window procedure of the button's parent responds to this message by drawing the button.

The BS_OWNERDRAW Style

An application tells Windows CE that a particular button is an owner draw button by specifying the BS_OWNERDRAW style when the button is created:

```
    #define IDC_BUTTON
HWND hwndButton;

hwndButton = CreateWindow(TEXT("Button"),
             TEXT("Some Caption"),
             WS_VISIBLE|WS_CHILD|
             BS_OWNERDRAW,
             ...);
```

The WM_DRAWITEM Message

To Windows CE, the button *hwndButton* is like any other button except that it has the BS_OWNERDRAW style bit set. The button class window procedure checks for this style when a button is about to be painted. If this style bit is set, the default painting is skipped, and the button sends a WM_DRAWITEM message to its parent.

The WM_DRAWITEM message is sent when an owner draw button must be repainted for any reason. This includes when the button is pressed, released, or receives focus. How a window responds to the WM_DRAWITEM message completely defines how owner draw buttons appear to the user.

A window may contain more than one owner draw button. Each of these buttons may have a completely different appearance. The WM_DRAWITEM message contains information about which button is sending the message so that the parent window can execute the appropriate drawing code. Table 9.1 gives the WM_DRAWITEM message parameter details.

The *wParam* value tells the parent window which of the owner draw buttons that it contains needs to be redrawn. The DRAWITEMSTRUCT pointed to by the *lParam* contains all of the information about the control and why it must be redrawn.

Applications should return TRUE when they finish processing the WM_DRAWITEM message.

Table 9.1 The WM_DRAWITEM Message

PARAMETER	MEANING
(UINT)wParam	Command identifier of the button sending the message.
(LPDRAWITEMSTRUCT)lParam	Pointer to a DRAWITEMSTRUCT structure containing information about the control to be drawn.

The DRAWITEMSTRUCT structure is defined as:

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

The first two members define the type of the control and its identifier.

The *hwndItem* member contains the window handle of the button that sent the WM_DRAWITEM message. Similarly, *hDC* is the button's device context. Any drawing operations performed to render the appearance of the button should be done in this device context.

rcItem contains the rectangular dimensions of the button in client coordinates.

The *itemData* member only has meaning with owner draw list boxes and combo boxes. It therefore has no meaning under Windows CE, which does not support owner draw list boxes or combo boxes.

The two most important members of the DRAWITEMSTRUCT are *itemAction* and *itemState*. These members describe the drawing action that must be performed and the state of the button respectively. An application uses these values to determine how to draw the owner draw button. *itemAction* can be one or more of the following values (combined by a bitwise OR):

ODA_DRAWENTIRE. The entire button must be redrawn.

ODA_FOCUS. The button has lost or gained keyboard focus (as indicated by the *itemState* value).

ODA_SELECT. The button selection status has changed (as indicated by the *itemState* value).

itemState can be one or more of the following:

ODS_CHECKED. Only used for owner draw menus; indicates the item is checked.

ODS_SELECTED. The button is selected/pressed.

ODS_GRAYED. Only used for owner draw menus; indicates the item is to be grayed.

ODS_DISABLED. The button is to be drawn as disabled.

ODS_FOCUS. The button has the keyboard focus.

Application programmers typically just use the *itemState* value to determine how to draw their own draw buttons. Since *itemAction* only indicates which of the *itemState* values to be on the lookout for, it is easiest to just test *itemState*.

An Example

How does all of this get used in practice? Let's take a simple example and demonstrate how a parent window would respond to the WM_DRAWITEM message. Assume that the parent window wants to create an owner draw button with a control identifier defined as IDC_BUTTON and the string "Press Here" as the button text. When the button is unpressed, it appears as shown in Figure 9.1. Figure 9.2 shows the button in the pressed state.

The button is created by the code shown below. *hwndMain* and *hInstance* are the application main window and application instance, respectively.

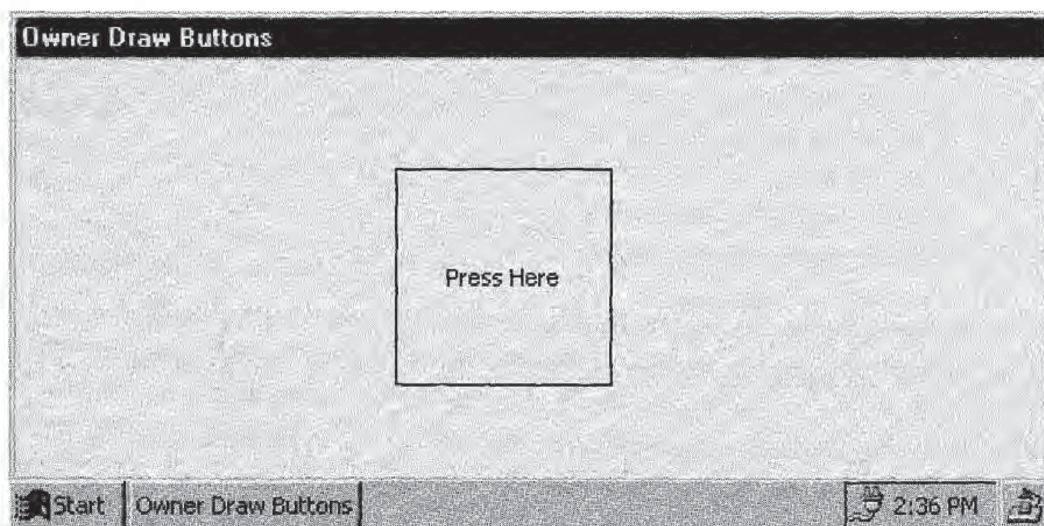


Figure 9.1 Sample owner draw button in the unpressed state.

Figure 9.2 Sample owner draw button in the pressed state.

```
#define IDC_BUTTON  1028
HWND hwndButton;
hwndButton = CreateWindow(TEXT("BUTTON"),
  TEXT("Press Here"),
  WS_VISIBLE|WS_CHILD|BS_OWNERDRAW,
  175,50,100,100,hwndMain,
  (HMENU)IDC_BUTTON,
  hInstance, NULL);
```

The WM_DRAWITEM code to implement the button appearance, which appears in the button parent window's window procedure, is shown below. Only the part of the window procedure relevant to drawing the owner draw buttons is included here.

```
LRESULT CALLBACK WndProc(HWND hwnd,
             UINT message,
             WPARAM wParam,
             LPARAM lParam)
{
  UINT nID;
  switch(message)
  {
  /* Other message handlers here... */
  case WM_DRAWITEM:
   UINT nID;
   LPDRAWITEMSTRUCT lpdis;
   nID = (UINT)wParam;
   switch (nID)
   {
   case IDC_BUTTON:
```

```
        HDC hdc;
        RECT rc;
        HBRUSH hBrushOld;
        HPEN hPenOld;
        int nModeOld;
        TCHAR pszText[129];
        lpdis = (LPDRAWITEMSTRUCT)lParam;
        rc = lpdis->rcItem;
        hdc = lpdis->hDC;
        if (lpdis->itemState & ODS_SELECTED)
        {
         //Invert the button when selected
         PatBlt(hdc, rc.left,rc.top,
          rc.right,rc.bottom, DSTINVERT);
        }
        else
        {
         //Draw the button in its unpressed state
         hPenOld = (HPEN)SelectObject(hdc,
          GetStockObject(BLACK_PEN));
         hBrushOld = (HBRUSH)SelectObject(hdc,
          GetStockObject(WHITE_BRUSH));
         nModeOld = SetBkMode(hdc, TRANSPARENT);
         Rectangle(hdc, rc.left,rc.top,
          rc.right,rc.bottom);
         GetWindowText(lpdis->hwndItem,
          pszText, 129);
         DrawText(hdc, pszText, -1, &rc,
          DT_CENTER|DT_VCENTER);
         SetBkMode(hdc, nModeOld);
         SelectObject(hdc, hBrushOld);
         SelectObject(hdc, hPenOld);
        }
        break;
       default:
        break;
       }    //End of switch(nID) block
       return (TRUE);
      /* Other message handlers here... */
      }       //End of switch(message) block
    }
```

The WM_DRAWITEM handler contains a switch statement for determining which owner draw button is responsible for sending the WM_DRAWITEM message. Although this example only contains one owner draw button, it is a good practice to put such a switch statement in your handler in case you add more owner draw buttons later.

We need to draw the IDC_BUTTON button in the pressed and unpressed states. We check to see if the button is pressed with the following test:

```
if (lpdis->itemState & ODS_SELECTED)
```

In other words, if the ODS_SELECTED flag is set in the DRAWITEMSTRUCT *itemState* member, the button is being pressed. Note that the test is not

```
if (lpdis->itemState == ODS_SELECTED)
```

Since a pressed button also has focus, the second test is too limiting, because the ODS_FOCUS flag will also be set. The *itemState* member of *lpdis* therefore is equal to ODS_SELECTED | ODS_FOCUS when the button is pressed.

If the button is pressed, we simply invert the control rectangle with a call to *PatBlt*. Whatever was drawn in the button in the unpressed state is inverted because *PatBlt* is called with the DSTINVERT.

Much more happens in the unpressed state. A black pen and white brush are selected into the button's device context, and the background mode is set to transparent so that the button surface shows through the background of any text that is drawn. The *Rectangle* call results in the white rectangle with the black outline that you see in Figure 9.1. The *Rectangle* function fills the specified rectangle with the current brush (white, in our case) and draws the rectangle outline with the current pen (in our case, black).

Next the code obtains the button text by calling *GetWindowText*, and draws it centered in the button rectangle with *DrawText*.

After the button is drawn, the original brush and pen are selected back into the device context, and the old background mode is restored.

This example gives us a high-level understanding of how to handle WM_DRAWITEM. The handler checks the command identifier of the control sending the message. It next looks at the *itemState* of the control to determine which state of the control needs to be drawn, and then performs the necessary drawing operations.

This example is a bit simplistic. Real owner draw buttons typically have much fancier graphics that are rendered by drawing custom bitmaps. The KIOSK.EXE example uses bitmaps for richer graphics in

its owner draw buttons. Of course, "richer" is a very subjective term. I make absolutely no claims to artistic ability!

The Kiosk Application

At this point we are ready to present the KIOSK.EXE application. Bear in mind that it is only really a mock-up of a front-end user interface for a kiosk-style Windows CE device. It only demonstrates some owner draw buttons and other user interface customization techniques. There is no real functionality behind this application.

The idea behind the kiosk model is that a user perceives it as a dedicated single-use device, such as an automatic teller machine. It does not provide real-time stock quotes or send e-mail as well.

A kiosk user interface is supposed to be one hundred percent obvious to a user. No on-line help is required, and the user is not faced with myriad confusing user interface components to figure out.

To meet these objectives, the main application window of KIOSK.EXE is entirely blank except for the string "Tap Anywhere To Begin" scrolling continuously across the screen (Figure 9.3). What could be simpler than this? No menus, no buttons, just a black screen and very obvious instructions about how to use the application. The window doesn't even have a border or caption bar.

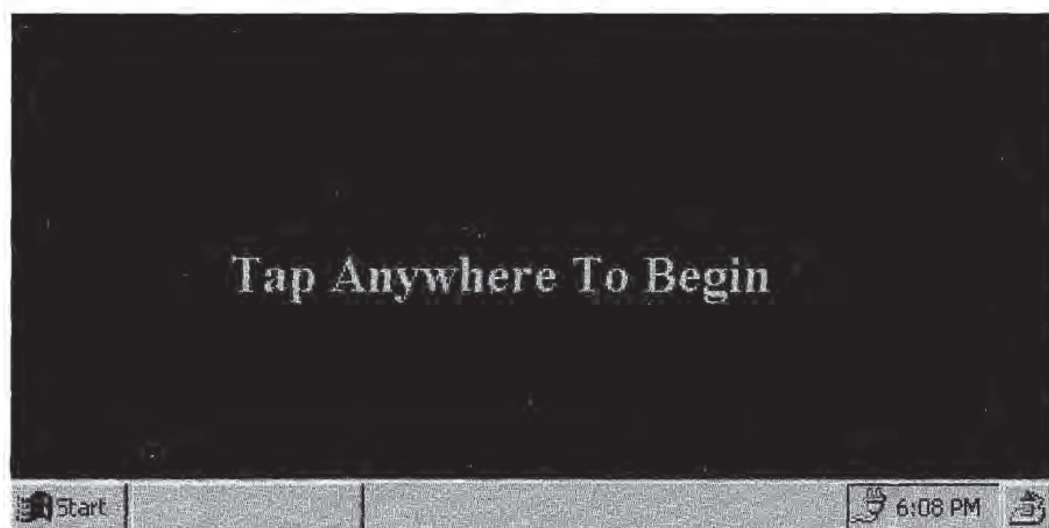


Figure 9.3 The Kiosk application main window.

Creating the Main Application Window

The implementation of this main window is straightforward. Here is how the window class that was used to create the main window is defined and registered. *WndProc* is the main application window's window procedure. *hInstance* is the application HINSTANCE, and *pszAppName* is the window class name.

```
WNDCLASS wndClass;
wndClass.style = 0;
wndClass.lpfnWndProc = WndProc;
wndClass.cbClsExtra = 0;
wndClass.cbWndExtra = 0;
wndClass.hInstance  = hInstance;
wndClass.hIcon    = NULL;
wndClass.hCursor   = NULL;
wndClass.hbrBackground =
  (HBRUSH)GetStockObject(BLACK_BRUSH);
wndClass.lpszMenuName = NULL;
wndClass.lpszClassName = pszAppName;
RegisterClass(&wndClass);
```

This is not much different from most of the other window class declarations that we have seen. The *hbrBackground* member of the WNDCLASS structure is set to the stock black brush. Hence the black background of any instance of this window class.

The main window then gets created:

```
HWND hwndMain;
hwndMain = CreateWindow(pszAppName,
             NULL,
             WS_VISIBLE,...);
```

The only window style that we set is WS_VISIBLE. Therefore the main window has no border and no caption bar. This is why the main application window appears as the plain black background we see in Figure 9.3.

Adding the Scrolling Text

The scrolling banner text in the main application window is implemented using a bitmap and a Windows CE *timer*.

Windows CE Timers

A timer is a device that applications can use to have Windows CE notify them that a specified interval of time has elapsed. In our case, the

timer fires every 0.5 seconds. In response to this timer, the main window produces the effect of scrolling the text by repainting the bitmap in a new position.

Timers are used extensively in a wide variety of Windows CE applications. For example, calendar applications use timers to trigger the alarms that users set to remind them of scheduled appointments..

Each timer that an application creates is associated with a particular window. Windows CE notifies a window that a timer associated with it has elapsed by sending a WM_TIMER message to that window's window procedure. Alternatively, an application-defined callback function can be specified for each timer. In this case, the callback function assigned to the timer is called.

To create the timer, KIOSK.EXE calls the *SetTimer* Windows CE function:

```
SetTimer(hwnd, uIDEvent, uElapse, lpTimerFunc);
```

SetTimer returns the identifier of the new timer (i.e., *uIDEvent*) if the function succeeds. Otherwise the return value is zero.

hwnd is the window that owns the timer. Since a window can own more than one timer, Windows CE needs a way to distinguish between timers. Callers therefore specify the *uIDEvent* parameter. *uIDEvent* is a UINT identifying the timer. *uElapse* defines the *timer interval*, the number of milliseconds that elapse between WM_TIMER messages.

lpTimerFunc is a pointer to a timer callback function. This is the function that gets called by Windows CE whenever the timer interval identified by *uIDEvent* elapses. A timer callback function has the following signature:

```
VOID CALLBACK TimerProc(hwnd, uMsg, idEvent, dwTime);
```

hwnd and *idEvent* identify the window that owns the timer and the timer identifier, respectively. *idEvent* is the same as the *uIDEvent* value that is in the *SetTimer* call.

The *uMsg* parameter is always WM_TIMER for a timer callback. Given that a timer callback is always called because a timer interval has elapsed, it is anyone's guess why this parameter was added to the function definition.

dwTime gives the number of milliseconds since Windows CE was launched on the device hosting the application.

Table 9.2 The WM_TIMER Message

PARAMETER	MEANING
(UINT)wParam	Identifier of the timer whose interval elapsed, causing the WM_TIMER message to be sent.
(TIMERPROC*)lParam	Pointer to the timer callback function.

If you are like me, you might prefer to set *lpTimerFunc* to NULL. In this case, Windows CE sends a WM_TIMER message to the window that owns a timer whose *uElapse* interval has elapsed (Table 9.2). All of the information that you would get from a timer callback is available to any window procedure that handles this message. Perhaps this is why most people don't bother using timer callbacks.

If the *lpTimerFunc* parameter of *SetTimer* is NULL, *lParam* will be NULL for the corresponding WM_TIMER messages.

KIOSK.EXE creates one timer identified as IDT_SCROLL and assigns it to the main application window. The IDT_SCROLL timer fires every 0.5 seconds.

The main window responds to IDT_TIMER by scrolling the banner text. The WM_TIMER handler of the main window's window procedure looks like this:

```
case WM_TIMER:
  if (IDT_SCROLL==wParam)
  {
   //Perform scrolling
  }
  return (0);
```

A WM_TIMER handler typically checks the identity of the timer whose interval has elapsed and performs whatever action that timer was meant to trigger.

An application can destroy a timer by calling *KillTimer*:

```
KillTimer(hwnd, uIDEvent);
```

hwnd identifies the window that owns the timer to be destroyed, and *uIDEvent* is the timer identifier. If the timer specified by *uIDEvent* is successfully destroyed, *KillTimer* returns TRUE. Otherwise it returns FALSE.

Creating the Text: Offscreen Bitmaps

The text that scrolls across the kiosk application's main window is implemented as a bitmap. But you will search in vain if you try to find a bitmap resource somewhere in the project files on the companion CD that has the text "Tap Anywhere To Begin" in it.

This is because the bitmap that is used to draw the scrolling text is created programmatically. It is done using a common Windows CE graphics programming technique known as drawing an *offscreen bitmap*.

An offscreen bitmap is a bitmap like any other. The only difference, as the name implies, is that the bits that constitute the bitmap reside in some portion of program memory that is not owned by the display device. The basic idea is that an application generates the bitmap off screen, and then renders it in some device context when needed.

Note that the actual string "Tap Anywhere To Begin" is stored in a Unicode string variable called *pszText*, defined in the project file KIOSK.H. The offscreen bitmap is generated using this string. So when we say that the text scrolls across the screen, what we really mean is that the offscreen bitmap representing the string is being scrolled.

There are two primary ingredients required to create an offscreen bitmap. The first is a *memory device context*. The second is the bitmap itself.

TIP

STORING THE BANNER TEXT IN THE REGISTRY

Chapter 8 discussed the Windows CE registry as one form of persistent storage. In a complete commercial kiosk application, the scrolling banner text string would most likely be stored in the registry. This would allow the banner text to be changed without requiring the application to be recompiled.

The Memory Device Context

A memory device context is similar to a window device context. The difference is that a memory device context represents a virtual display surface. It is a display surface in memory only. Other than this important distinction, a memory device context is like any other device context.

Page 00283

A memory device context becomes really useful when it has a bitmap selected into it. Then any graphics function call that operates on the memory device context has the effect of producing the result of the function call on the selected bitmap.

Thus, if an application selects a bitmap into a memory device context and then draws a rectangle on that device context, the bitmap will contain that rectangle.

A memory device context is created with the function *CreateCompatibleDC:*

```
CreateCompatibleDC(hdc);
```

This function returns a memory device context with the same attributes as that specified by the parameter *hdc*.

Creating the Bitmap

The second ingredient we need in order to produce an offscreen bitmap is the bitmap object itself. For this purpose an application calls *CreateCompatibleBitmap*:

```
CreateCompatibleBitmap(hdc, nWidth, nHeight);
```

This function returns a handle to a BITMAP object (HBITMAP) *nWidth* pixels wide and *nHeight* pixels tall. The number of bits per pixel and the number of color planes of the bitmap are the same as those of the device context specified in *hdc*. A bitmap created in this way is called an offscreen bitmap.

Once a bitmap has been created in this way, it can be selected into a memory device context with a call to *SelectObject*. Any subsequent graphics operations involving that memory device context are rendered on the offscreen bitmap.

In the case of the KIOSK.EXE application, the offscreen bitmap containing the banner text is produced with the following code. *hdc* is the main application window device context. *nRight* and *nBottom* are the width and height of the main application window.

```
#define BK_COLOR (RGB(0,0,0)) //Black text background
#define TEXT_COLOR (RGB(255,255,0)) //Yellow text
HDC hdcMem;
HBITMAP hBmp;
RECT rc;
```

```
TCHAR* pszBanner[] = TEXT("Tap Anywhere To Begin");
hdcMem = CreateCompatibleDC(hdc);
hBmp = CreateCompatibleBitmap(
  hdc, nRight, nBottom);
SelectObject(hdcMem, hBmp);
SetBkColor(hdcMem, BK_COLOR);
SetTextColor(hdcMem, TEXT_COLOR);
DrawText(hdcMem, pszBanner, -1, &rc, DT_LEFT);
```

The last three statements in the example above operate on the memory device context *hdcMem*. The operations they represent are therefore rendered on the offscreen bitmap currently selected into that device context.

Making the Text Scroll

At this point, the application has the complete offscreen bitmap for displaying the kiosk banner text. Making this text scroll is now very simple.

The main application window simply draws the offscreen bitmap whenever the window gets painted. The scrolling effect is achieved by updating the location at which the bitmap is drawn. This position is updated in response to the IDT_SCROLL timer firing.

Inside the window procedure for the main window, we find this code:

```
case WM_TIMER:
  if (IDT_SCROLL==wParam)
  {
   nScrollX += 50;
   if (nScrollX > nRight)
   {
    nScrollX = -nStringWidth;
   }
   InvalidateRect(hwnd, NULL, TRUE);
  }
  return (0);
```

nStringWidth is the width of the banner text string in pixels. It is calculated at the beginning of the application with a *DrawText* call that uses DT_CALCRECT as a text drawing option.

nScrollX is an integer initialized to zero in *WinMain*. It represents the current x position in pixels of the offscreen bitmap. On every IDT_SCROLL timer tick, this value gets incremented by 50 pixels.

Once *nScrollX* exceeds *nRight*, the right edge of the main window, *nScrollX* is set to the value *-nStringWidth*. This effectively moves the offscreen bitmap off the left edge of the main window.

The *InvalidateRect* call tells Windows CE that the entire client area of the main application window must be redrawn. So, the entire window is redrawn after every IDT_SCROLL timer interval.

The main window is drawn with the WM_PAINT handler code:

```
case WM_PAINT:
  PAINTSTRUCT ps;
  hdc = BeginPaint(hwnd, &ps);
  BitBlt(hdc, nScrollX, 0, nRight, nBottom,
   hdcMem, 0,0, SRCCOPY);
  EndPaint(hwnd, &ps);
  return (0);
```

The *BitBlt* call redraws the offscreen bitmap containing the text at the new *nScrollX* x location.

Implementing the Options Window

The window that appears when a user taps the kiosk application's main window is called the options window. It is the window that the user interacts with to make various banking choices (Figure 9.4).

The options window is another example of a Windows CE user interface designed to look very little like traditional windows. The options

Why Not Just Use *ScrollWindowEx*?

Experienced Windows programmers might question my method of implementing scrolling text with an offscreen bitmap. Why not just draw the text once using *DrawText*, and then call *ScrollWindowEx* in response to the WM_TIMER message?

In addition to giving me an excuse to introduce the offscreen bitmap concept to programmers who may not be familiar with it, my method also makes it easier to produce the scrolling text effect.

***ScrollWindowEx* only scrolls pixels that appear on the window specified by the *hWnd* parameter of *ScrollWindowEx*. After the text scrolls off the right side of the main window, it re-enters the screen from the left. Producing this effect without a bitmap, which has a fixed set of bits, would require clever *DrawText* calls to draw incomplete portions of the text to make it appear to scroll back on the screen.**

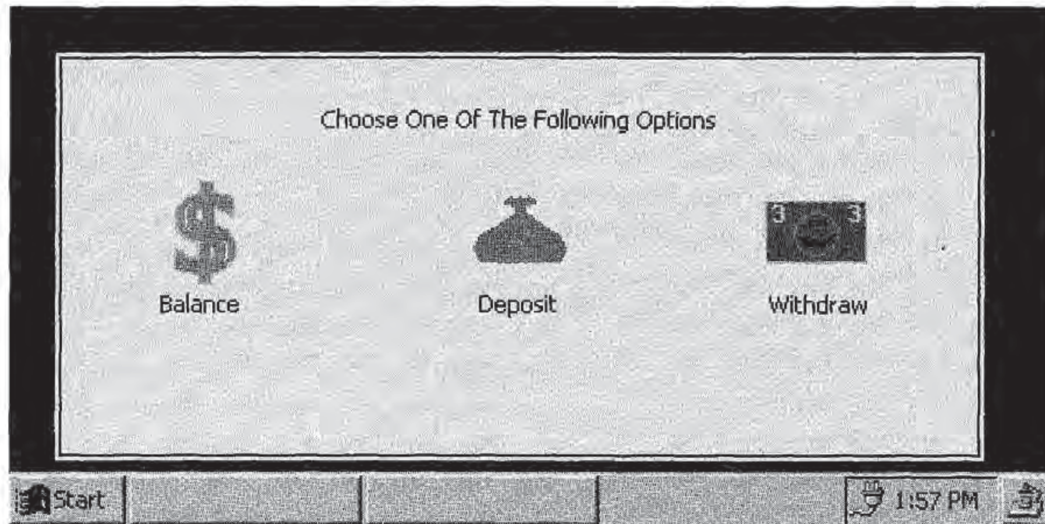


Figure 9.4 The Kiosk application options window.

window allows users to check their bank account balance, make a deposit, or withdraw cash. Of course none of this banking functionality is actually implemented by KIOSK.EXE. The point of the options window and the entire kiosk sample application is to give you insight into the Windows CE options available for implementing non-standard user interfaces.

The options window consists of three owner draw buttons, a custom window border, and some descriptive text. Like the main window, it has no title bar or window caption.

The options window class is defined and registered as follows:

```
TCHAR pszEntryClass[] = TEXT("ENTRYWINDOW");
WNDCLASS wndClassOptions;
wndClassOptions.style = 0;
wndClassOptions.lpfnWndProc = OptionsWndProc;
wndClassOptions.cbClsExtra = 0;
wndClassOptions.cbWndExtra = 0;
wndClassOptions.hInstance  = hInstance;
wndClassOptions.hIcon      = NULL;
wndClassOptions.hCursor    = NULL;
wndClassOptions.hbrBackground=
  (HBRUSH)GetStockObject(WHITE_BRUSH);
wndClassOptions.lpszMenuName = NULL;
wndClassOptions.lpszClassName= pszEntryClass;
RegisterClass(&wndClassOptions);
```

OptionsWndProc is the window procedure for the options window.

Also notice the WHITE_BRUSH background instead of the BLACK_BRUSH background defined for the main window class.

When a user taps the main application screen, an instance of this window class is created. Since the options window appears as a result of tapping the main application window, the options window *CreateWindow* call must appear in the WM_LBUTTONDOWN message handler of the main window's window procedure as shown below. Only the part of the window procedure relevant to options window creation is included here.

```
HWND hwndOptions;
int nOptionsWidth, nOptionsHeight;
LRESULT CALLBACK WndProc(
  HWND hwnd,
  UINT message,
  WPARAM wParam,
  LPARAM lParam)
{
  switch(message)
  {
  /* Other message handlers here... */
  case WM_LBUTTONDOWN:
   nOptionsWidth = (nRight-50);
   nOptionsHeight= (nBottom-50);
   KillTimer(hwndMain, IDT_SCROLL);
   hwndOptions = CreateWindow(pszEntryClass,
    NULL, WS_VISIBLE,
      20, 20, nOptionsWidth, nOptionsHeight,
       hwnd, NULL, ghInst, NULL);
   return (0);
   /* Other message handlers... */
  default:
   return (DefWindowProc(hwnd, message, wParam, lParam));
  }         //End of switch(message) statement
}
```

nRight and *nBottom* are the horizontal and vertical dimensions of the main window's bounding rectangle. *ghInst* is the globally defined application instance.

The first thing to notice in this code is the *KillTimer* call. When the options window is displayed, the text on the main application window stops scrolling. This is done by simply turning the scrolling timer IDT_SCROLL off.

The *CreateWindow* call makes an instance of the options window class that contains only the WS_VISIBLE style. Thus the window does not include the standard Windows CE border or caption bar. The window does appear to have a border, though. This is rendered by the application in response to the WM_ERASEBKGND messages that are sent to the options window.

Drawing the Options Window Border

The WM_ERASEBKGND message is similar to its more well known cousin, WM_PAINT. A WM_PAINT message is sent to a window whenever part or all of the window's client area needs to be repainted. Similarly, WM_ERASEBKGND is sent when part or all of a window's client area needs to be erased.

For example, we've probably all seen applications that have windows with interesting bitmaps as their background. These backgrounds are drawn in response to WM_ERASEBKGND messages (Table 9.3). Including the WM_ERASEBKGND message in the operating system allows applications to break the process of drawing windows into two steps. The WM_ERASEBKGND step can be used to draw the fixed parts of a window's client area display such as custom backgrounds. WM_PAINT is then used to paint the parts of the display that change, such as the text that appears on a page in a word processing application.

The value returned by a window procedure that handles the WM_ERASEBKGND is very important. Returning the wrong value can lead to very subtle bugs in an application which appear as window backgrounds being drawn incorrectly. An application should return a non-zero value if it handles the WM_ERASEBKGND mes-

Table 9.3 The WM_ERASEBKGND Message Parameters

PARAMETER	MEANING
(HDC)wParam	Device context of the window whose background is to be erased.
lParam	Not used.

sage. This tells Windows CE not to perform the default WM_ERASE-BKGND processing. Returning zero tells Windows CE that the default processing should be performed.

This is how the *hbrBackground* member of the window class definition gets used by Windows CE. If an application leaves the processing of WM_ERASEBKGND messages to Windows CE (either by returning zero in response to the message or by calling *DefWindowProc* for WM_ERASEBKGND messages), Windows CE erases the window background itself. It does so by filling the window's client area with the brush specified in the *hbrBackground* member of the window class definition for the particular window. This is how, for example, the background of the kiosk application's main window is painted black.

You can see how telling Windows CE that you erased your window background when you really didn't can cause problems. Incorrectly returning a non-zero value in response to WM_ERASEBKGND can prevent the proper background from being painted.

In the case of the options window in the kiosk application, WM_ERASEBKGND is used to draw the window border. The relevant portion of the *OptionsWndProc* window procedure is shown below:

```
LRESULT CALLBACK OptionsWndProc(
  HWND hwnd,
  UINT message,
  WPARAM wParam,
  LPARAM lParam)
{
  HDC hdc;
  UINT nID;
  RECT rc;
  switch(message)
  {
  //Other message handlers */
  //...
  case WM_ERASEBKGND:
   HBRUSH hBrushOld;
   hdc = (HDC)wParam;
   GetClientRect(hwnd, &rc);
   hBrushOld = (HBRUSH)SelectObject(hdc,
    GetStockObject(WHITE_BRUSH));
   Rectangle(hdc, rc.left, rc.top,
    rc.right, rc.bottom);
   InflateRect(&rc, -3, -3);
   Rectangle(hdc, rc.left, rc.top,
```

```
        rc.right, rc.bottom);
       SelectObject(hdc, hBrushOld);
       return (TRUE);
      default:
       return (DefWindowProc(hwnd, message, wParam, lParam));
      }   //End of switch(message) statement
   }
```

The WM_ERASEBKGND handler first extracts the HDC of the options window from the *wParam* parameter of the window procedure. Next it gets the coordinates of the options window client rectangle by calling *GetClientRect*. It then selects the stock object WHITE_BRUSH into this device context. Any subsequent graphics function calls that fill a rectangle or region will thus use white as the fill color.

The two *Rectangle* function calls result in the border being drawn. The first *Rectangle* call fills the entire client area of the options window with white. The effect of the second *Rectangle* call is to draw the black inset border. This happens for two reasons. First, the *InflateRect* call decreases the dimensions of the rectangle to be drawn. Second, the outline of a rectangle drawn by *Rectangle* is the color of the pen currently selected into the device context specified by the *hdc* parameter. Since this pen is black by default, the rectangle border is black.

Note that we have to make the first *Rectangle* call to fill the entire client area before drawing the inset border. Since the message handler code returns TRUE when it's done, the default WM_ERASEBKGND processing is skipped. If the first *Rectangle* call is not made, only the inset rectangle would ever get drawn by our WM_ERASEBKGND message handler.

Creating and Drawing the Options Buttons

We've seen how the options window is created. But what about the three owner draw buttons that the window contains?

The three owner draw buttons in the options window are created in response to the WM_CREATE message sent to *OptionsWndProc*:

```
/* Global variables and child control
  identifiers defined in kiosk.h */
#define IDC_BALANCE   1028
#define IDC_DEPOSIT   1029
#define IDC_WITHDRAW  1030
HDC hdcButtons;
```

```
HBITMAP hBmpButtons;
HWND hwndBalance;
HWND hwndDeposit;
HWND hwndWithdraw;
LRESULT CALLBACK OptionsWndProc(
  HWND hwnd,
  UINT message,
  WPARAM wParam,
  LPARAM lParam)
{
  HDC hdc;
  UINT nID;
  RECT rc;
  switch(message)
  {
  case WM_CREATE:
   hdc = GetDC(hwnd);
   hdcButtons = CreateCompatibleDC(hdc);
   hBmpButtons = LoadBitmap(ghInst,
    MAKEINTRESOURCE(IDB_BALANCE));
   SelectObject(hdcButtons, hBmpButtons);
   ReleaseDC(hwnd, hdc);
   hwndBalance = CreateWindow(TEXT("BUTTON"), NULL,
    WS_VISIBLE|WS_CHILD|BS_OWNERDRAW,
    ...,
    (HMENU)IDC_BALANCE,...);
   hwndDeposit = CreateWindow(TEXT("BUTTON"), NULL,
    WS_VISIBLE|WS_CHILD|BS_OWNERDRAW,
    ...,
    (HMENU)IDC_DEPOSIT,...);
   hwndWithdraw = CreateWindow(TEXT("BUTTON"), NULL,
    WS_VISIBLE|WS_CHILD|BS_OWNERDRAW,
    ...,
    (HMENU)IDC_WITHDRAW,...);
   return (0);
  //Other message handlers
  //...
  default:
   return (DefWindowProc(hwnd, message, wParam, lParam));
  }    //End of switch(message) statement
}
```

The WM_CREATE handler does more than just create the three owner draw buttons. It also loads the bitmap containing the button images and selects that bitmap into a global memory device context called *hdcButtons*.

The button images are stored in one bitmap as shown in Figure 9.5. Each button has a pair of 48-pixel-wide images. The first image in the

Figure 9.5 Options window button image bitmap.

pair is used to draw the button's unpressed state. The second is used to draw its pressed state.

When the options window draws the buttons in response to the various WM_DRAWITEM messages it is sent, it uses the control identifier of the button sending the WM_DRAWITEM message to determine which set of images to use from the button image bitmap.

```
#define BMP_WIDTH  48
LRESULT CALLBACK OptionsWndProc(
  HWND hwnd,
  UINT message,
  WPARAM wParam,
  LPARAM lParam)
{
  HDC hdc;
  UINT nID;
  RECT rc;
  switch(message)
  {
  //Other message handlers
  //...
  case WM_DRAWITEM:
   LPDRAWITEMSTRUCT lpdis;
   int xBmp, xBmpPressed;
   nID = (UINT)wParam;
   lpdis = (LPDRAWITEMSTRUCT)lParam;
   rc = lpdis->rcItem;
   hdc = lpdis->hDC;
   xBmp = 0;
   xBmpPressed = BMP_WIDTH;
   switch(nID)
   {
    case IDC_DEPOSIT:
     xBmp += 2*BMP_WIDTH;
     xBmpPressed += 2*BMP_WIDTH;
     break;
    case IDC_WITHDRAW:
     xBmp += 4*BMP_WIDTH;
     xBmpPressed += 4*BMP_WIDTH;
     break;
    default:
```

```
        break;
    }   //End of switch(nID) block
```

The WM_DRAWITEM handler first extracts the information it needs to draw the button bitmaps such as the device context of the button and the button's bounding rectangle. Next, it initializes the two offset variables, *xBmp* and *xBmpPressed*, to the x pixel offsets of the first unpressed and pressed button images.

The *nID* switch statement then adjusts these values to correspond to the left edge of the appropriate button bitmaps depending on which button sent the WM_DRAWITEM message.

For example, if the WM_DRAWITEM message was sent by the IDC_DEPOSIT button, *xBmp* and *xBmpPressed* are set to the values 96 and 120, respectively. These values correspond to the leftmost pixels of the images to be used to draw the pressed and unpressed states of the IDC_DEPOSIT button.

Finally, the WM_DRAWITEM handler checks the *itemState* of the button, and displays the proper image with a call to *BitBlt*:

```
/* If the button is pressed... */
if (lpdis->itemState & ODS_SELECTED)
{
  BitBlt(hdc, rc.left,rc.top,
   (rc.right-rc.left),
   (rc.bottom-rc.top),
   hdcButtons,
   xBmpPressed,0,SRCCOPY);
}
/* If the button is not pressed... */
else
{
  BitBlt(hdc, rc.left,rc.top,
   (rc.right-rc.left),
   (rc.bottom-rc.top),
   hdcButtons,
   xBmp,0,SRCCOPY);
}
return (TRUE);
```

Concluding Remarks

In this chapter, you have been introduced to some of the more common techniques for programming custom Windows CE user inter-

faces. But owner draw controls and offscreen bitmaps are but a few of the many ways that you can create user interfaces for your applications that are different from the Windows CE standard.

The next chapter expands on the owner draw concept with the more general subject of the Windows CE custom draw service. The custom draw service, like owner draw controls, allows you to dramatically influence the look and feel of various Windows CE controls—but with much more flexibility.

CHAPTER 10

The Windows CE Custom Draw Service

We have seen how owner draw techniques allow applications to define the appearance of various Windows CE control types. For example, by simply adding the BS_OWNERDRAW style to a button, and responding appropriately to the WM_DRAWITEM message in the parent window procedure, an application can completely redefine the look and feel of the button.

Another programming option that provides even more flexibility for modifying the appearance of Windows CE controls is the *custom draw service*.

In some ways, the custom draw service functionality is very similar to the way that owner draw controls work. With owner draw buttons, the WM_PAINT message handler in the button window procedure sends a WM_DRAWITEM message to the button's parent and skips doing the default button painting operations. Similarly, controls that support the custom draw service send WM_NOTIFY messages to their parents at various times throughout their painting process. The custom draw service is more flexible because it provides more hooks for the parent window to influence the look of the control.

The custom draw service is supported by the following Windows CE common controls, which live in COMMCTRL.DLL. It is not supported by any of the child controls.

- command bands
- header controls
- list view controls
- toolbars
- trackbar controls
- tree view controls

In this chapter, we describe how to use the custom draw service through the example of a custom trackbar control. The control that results is shown in Figure 10.1. (I did not originally set out to try and make this control look like a thermometer!) Compare this to the standard trackbar control shown in Figure 10.2. The custom drawn version of the control demonstrates quite a bit of customization. The trackbar border appears rounded, with nice drop shadowing. The thumb, which is a little black dot, looks totally different from the standard trackbar thumb. And the channel, the area that the thumb gets dragged around in, is different.

The application source code that implements this example is found in \Samples\custdraw on the companion CD. The resulting executable is

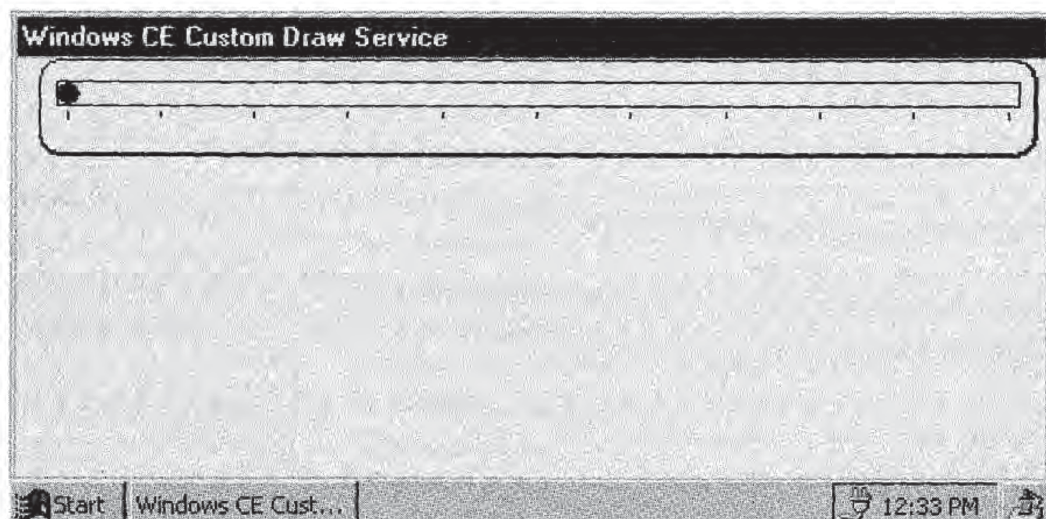


Figure 10.1 A trackbar control drawn using the custom draw service.

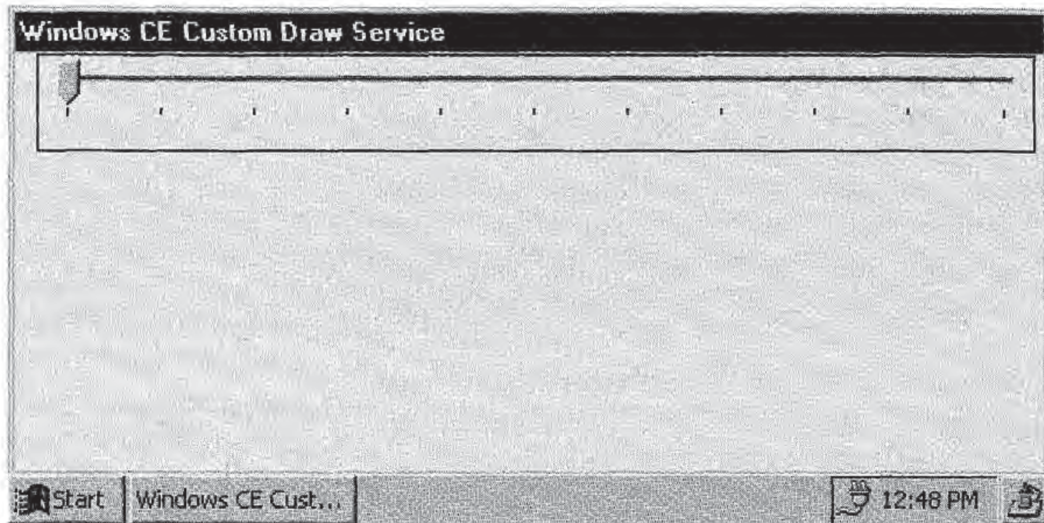


Figure 10.2 A standard trackbar control.

called CUSTDRAW.EXE. To exit the application, tap any part of the main window's client area not covered by the trackbar.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

Use the custom draw service to customize the appearance of Windows CE common controls

Custom Draw Notification

Windows CE controls that support the custom draw service give their parents the opportunity to customize the control drawing process at various times during the control's *paint cycle*.

The paint cycle is defined as all processing that a control (or any window, for that matter) performs in response to the WM_ERASEBKGND and WM_PAINT messages. WM_ERASEBKGND is sent to a window when the window background needs to be erased in preparation for painting. WM_PAINT is sent when a window is asked to repaint itself.

Controls using the custom draw service give their parent windows the opportunity to handle parts of the drawing process by sending the

Table 10.1 Custom Draw Service WM_NOTIFY Message Parameters

PARAMETER	MEANING
wParam	Integer containing the command identifier of the control sending the WM_NOTIFY message.
lParam	Pointer to an NMCUSTOMDRAW structure. If a tree view control sends the message, this parameter is an NMTVCUSTOMDRAW pointer. If the control is a list view control, this parameter is an NMLVCUSTOMDRAW pointer.

NM_CUSTOMDRAW notification. This notification is sent in the form of a WM_NOTIFY message. The parameters sent with the WM_NOTIFY message are described in Table 10.1.

For list view and tree view controls, the first member of the structure pointed to by *lParam* is an NMCUSTOMDRAW structure.

The NMCUSTOMDRAW structure contains information about the control and where the control is in its paint cycle:

```
typedef struct tagNMCUSTOMDRAWINFO
{
  NMHDR hdr;
  DWORD dwDrawStage;
  HDC hdc;
  RECT rc;
  DWORD dwItemSpec;
  UINT uItemState;
  LPARAM lItemlParam;
} NMCUSTOMDRAW, FAR* LPNMCUSTOMDRAW;
```

The *hdr* member of this structure is an NMHDR notify message header structure that always accompanies common control notifications. *hdc* is the device context of the control, and *rc* is supposed to contain the control's bounding rectangle. See the tip "Bogus NMCUSTOMDRAW rc Member" later in this chapter.

dwDrawStage tells your application what stage of the drawing process the custom draw control is in. This member can be one of eight values. The first four specify the state that the paint cycle of the control is in. These are referred to as *global* draw stages:

CDDS_PREPAINT. Sent before the control's WM_PAINT handler begins.

CDDS_POSTPAINT. Sent after the control's WM_PAINT processing is complete.

CDDS_PREERASE. Sent before the control's WM_ERASEBKGND handler begins. Not currently supported.

CDDS_POSTERASE. Sent after the control's WM_ERASEBKGND processing is complete. Not currently supported.

Note that CDDS_PREERASE and CDDS_POSTERASE are not currently supported.

dwDrawStage can also inform the parent window where the control is in the process of drawing individual items within the control. For example, a list view control keeps its parent apprised of the drawing progress for each list view item that is drawn. Trackbar controls tell their parents about trackbar thumb, channel, and tic mark drawing progress. The four possible values of *dwDrawStage* that convey this information are:

CDDS_ITEMPREPAINT. Sent before a control item is painted.

CDDS_ITEMPOSTPAINT. Sent after a control item is painted.

CDDS_ITEMPREERASE. Sent before a control item is erased. Not currently supported.

CDDS_ITEMPOSTERASE. Sent after a control item is erased. Not currently supported.

Note that CDDS_ITEMPREERASE and CDDS_ITEMPOSTERASE are not currently supported.

NOTE

CUSTOM DRAW NOTIFICATIONS

All custom draw service information comes to a control's parent window via the WM_NOTIFY message. This is the way that all common control notifications get to parent windows. But common controls can specify a variety of notification codes when they send WM_NOTIFY messages. This has led to the convention of referring to common control notifications, where the specific code identifies the notification.

Similarly, we will refer to custom draw notifications. Each *dwDrawStage* value specifies a particular custom draw notification. So when we say, for example, the item pre-paint notification, what is specifically meant is a WM_NOTIFY message with notification code NM_CUSTOMDRAW and an NMCUSTOMDRAW *dwDrawStage* member of CDDS_ITEMPREPAINT.

An application's response to the NM_CUSTOMDRAW notification under each of these conditions determines how the custom draw control proceeds with the paint cycle.

The *dwItemSpec* member of NMCUSTOMDRAW specifies the item number to which the notification corresponds. For example, this value would indicate the zero-based index of the particular list view control item being drawn.

For trackbar controls, three unique values are defined for the *dwItemSpec* member. These values specify what part of the trackbar control the parent window is being notified about:

TBCD_CHANNEL. Identifies the trackbar channel.

TBCD_THUMB. Identifies the trackbar thumb.

TBCD_TICS. Identifies the trackbar tic marks.

A separate NM_CUSTOMDRAW notification is sent by a control for each item in that control. This gives the parent window the chance to customize every part of the control's appearance.

For example, assume an NM_CUSTOMDRAW notification sent by a trackbar control with a *dwDrawStage* value of CDDS_ITEMPREPAINT and a *dwItemSpec* value of TBCD_TICS. This means that the trackbar control is notifying its parent that the trackbar is about to draw its tic marks.

The *uItemState* member of the NMCUSTOMDRAW structure specifies the current state of the item indicated by *dwItemSpec*. The following state identifiers can appear in this member. Not all values necessarily have meaning for all custom draw controls. For example, CDIS_CHECKED has no meaning for a trackbar, but does for a list view control that includes the LVS_EX_CHECKBOXES style:

CDIS_CHECKED. The item is checked.

CDIS_DEFAULT. The item is in its default state.

CDIS_DISABLED. The item is disabled.

CDIS_FOCUS. The item has focus.

CDIS_GRAYED. The item is grayed.

CDIS_HOT. The item is under the stylus/pointing device.

CDIS_SELECTED. The item is selected.

The final NMCUSTOMDRAW member, *lItemlParam*, contains any application-defined data that may have been previously assigned to the control item by the application.

For example, list view control items are described by LV_ITEM structures. One of the LV_ITEM members is *lParam*, which can be used by applications to associate data with items. List view items participating in an NM_CUSTOMDRAW notification would send their *lParam* data in the NMCUSTOMDRAW *lItemlParam* member.

TIP

BOGUS NMCUSTOMDRAW RC MEMBER

I have never seen a case where an NM_CUSTOMDRAW notification is sent by a control and the NMCUSTOMDRAW *rc* member had anything but garbage data in it. When implementing responses to custom draw notifications, it is more reliable to get the control's bounding rectangle yourself with a call to *GetClientRect*.

I should point out here that the Windows CE common controls that support the custom draw service do not need to be told by an application to enable their custom draw support. An application programmer might assume that some new control style must be added at creation time so that the control knows that its parent is interested in receiving custom draw notifications. But this is not the case. Controls send custom draw notifications by default. Applications decide to use the custom draw features by responding to these notifications. If the notifications are ignored by the parent window procedure, the service is effectively not used.

NOTE

ERASE NOTIFICATIONS CURRENTLY NOT SUPPORTED

None of the global or item-specific pre- or post-erase notifications are currently supported in Windows CE.

Responding to Custom Draw Notifications

An application's response to the various custom draw notifications controls the custom draw service behavior. Return values can specify how Windows CE completes a particular draw stage. They also can in-

dicate whether or not further custom draw notifications are sent by the control.

We first list all of the defined custom draw notification return values, and then discuss some examples of their use. The values that an application can return in response to an NM_CUSTOMDRAW notification are:

CDRF_DODEFAULT. Tells the control to perform default processing for the particular draw stage.

CDRF_SKIPDEFAULT. Tells the control not to perform default processing for the particular draw stage.

CDRF_NEWFONT. Tells the control that a new font has been selected into the HDC indicated by the *hdc* member of the NMCUSTOMDRAW structure sent with the custom draw notification.

CDRF_NOTIFYPOSTPAINT. Tells the control to send a CDDS_ITEMPOSTPAINT notification after painting an item. This is returned in response to the CDDS_ITEMPREPAINT notification.

CDRF_NOTIFYITEMDRAW. Tells the control to send all item-specific custom draw notifications. NM_CUSTOMDRAW notifications will be sent before and after items are drawn, i.e., CDDS_ITEMPREPAINT and CDDS_ITEMPOSTPAINT notifications will be sent.

CDDS_NOTIFYPOSTERASE. In theory, tells the control to send the parent window post-erase notifications. In reality, this is currently not supported.

The CDDS_NOTIFYITEMDRAW return value is in many ways the most important. An application returns this value in response to the CDDS_PREPAINT notification to request that the control send subsequent notifications for the rest of the current paint cycle.

If, on the other hand, the parent window returns CDRF_DODEFAULT in response to the CDDS_PREPAINT notification, the control will not send any more custom draw notifications for the rest of the current paint cycle.

If your application wants to use a different font to draw a control, select the desired font into the *hdc* member of the NMCUSTOMDRAW structure. You must return CDRF_NEWFONT in that case so that the control knows a new font was selected.

A typical NM_CUSTOMDRAW notification handler looks something like this:

How Are Custom Draw Notifications Ignored by Default?

If you dig through COMMCTRL.H, you'll find the definitions for the various custom draw notification return values. One of these is:

```
#define CDRF_DODEFAULT    0x00000000
```

In the typical window procedure, messages that are not handled are passed to *DefWindowProc* to let Windows CE perform default processing in response to those messages. If a parent window does not respond to custom draw notifications, *DefWindowProc* is called. For WM_NOTIFY messages containing the NM_CUSTOMDRAW notification code, *DefWindowProc* returns 0.

So, by default a control that sends a CDDS_PREPAINT notification at the beginning of its paint cycle will get back 0 if the notification is handled by *DefWindowProc*. Therefore, no more custom draw notifications get sent.

```
case WM_NOTIFY:
  LPNMHDR lpnmhdr;
  lpnmhdr = (LPNMHDR)lParam;
  switch(lpnmhdr->code)
  {
  case NM_CUSTOMDRAW:
   LPNMCUSTOMDRAW lpnmcd;
   lpnmcd = (LPNMCUSTOMDRAW)lParam;
   switch(lpnmcd->dwDrawStage)
   {
   case CDDS_PREPAINT:
    return (CDRF_NOTIFYITEMDRAW);
   case CDDS_ITEMPREPAINT:
   /* Do item-specific painting and
     return a CDRF_ value depending
     on how you want the item painting
     to proceed.
    */
   default:
    return (CDDS_DODEFAULT);
  }     //End of switch(dwDrawStage) block
  default:
   return (0);
  }     //End of switch(code) block
```

This short code sample is the parent window's WM_NOTIFY message handler. It tells the custom draw controls to send all custom draw notifications by returning CDRF_NOTIFYITEMDRAW in response to the CDDS_PREPAINT notification. The CDDS_ITEMPREPAINT handler then performs the custom drawing. The value returned when this is

done depends on whether the application wants the control to continue with default item drawing or not.

A parent window can of course respond to custom draw notifications from various controls in different ways. In that case, the WM_NOTIFY handler would have to look at the *hwndFrom* or *idFrom* member of the NMHDR passed with the NMCUSTOMDRAW structure to determine which control is sending a particular notification.

Other NMCUSTOMDRAW Info Structures

Earlier we said that list view and tree view custom draw controls conveyed information about themselves in custom draw notifications with structures other than NMCUSTOMDRAW. This section describes those control-specific structures.

When list view controls and tree view controls send custom draw notifications, the *lParam* of the corresponding WM_NOTIFY message is a pointer to an NMLVCUSTOMDRAW or NMTVCUSTOMDRAW structure. These structures are identical, so we will discuss just the first. The NMLVCUSTOMDRAW structure is defined as:

```
typedef struct tagNMLVCUSTOMDRAW
{
  NMCUSTOMDRAW nmcd;
  COLORREF clrText;
  COLORREF clrTextBk;
} NMLVCUSTOMDRAW, *LPNMLVCUSTOMDRAW;
```

This structure is very similar to the NMCUSTOMDRAW structure that is sent by other custom draw controls with their custom draw notifications. In fact, the first member of the NMLVCUSTOMDRAW structure, *nmcd*, is an NMCUSTOMDRAW structure.

The NMLVCUSTOMDRAW structure contains two additional members. *clrText* is the color to be used as the foreground color when the particular list view or tree view item's text is drawn. *clrTextBk* is the item's text background color.

These values are useful if an application wants to change the text or background colors that are used when list view or tree view items are drawn. An application can assign new colors to these members and return CDRF_DODEFAULT in response to CDDS_ITEMPREPAINT notifications. The control will then paint its items with the new colors.

A Real Example

Let's look at the real CUSTDRAW.EXE example and see how the custom trackbar of Figure 10.1 is implemented.

The trackbar parent window draws the channel and thumb itself. It also responds to the TBCD_TICS item spec to draw the rounded trackbar outline. The NM_CUSTOMDRAW notification handler looks like this:

```
#define IDC_TRACKBAR  1028
case WM_NOTIFY:
  LPNMHDR lpnmhdr;
  lpnmhdr = (LPNMHDR)lParam;
  switch(lpnmhdr->code)
  {
  case NM_CUSTOMDRAW:
   LPNMCUSTOMDRAW lpnmcd;
   lpnmcd = (LPNMCUSTOMDRAW)lParam;
   /* Respond to notification if it comes
     from the trackbar control.
    */
   if (lpnmcd->hdr.idFrom==IDC_TRACKBAR)
   {
    BOOL bSel;
    bSel = (lpnmcd->uItemState==CDIS_SELECTED);
    switch(lpnmcd->dwDrawStage)
    {
     case CDDS_PREPAINT:
      return (CDRF_NOTIFYITEMDRAW);
     case CDDS_ITEMPREPAINT:
      return (OnDrawTrackbar(
       lpnmcd->hdr.hwndFrom,
       lpnmcd->hdc,
       lpnmcd->dwItemSpec,
       bSel);
     default:
      return (CDRF_DODEFAULT);
    }       //End of switch(dwDrawStage) block
   }        //End of if (hwndTB) block
  default:
   return (0);
  }         //End of switch(lpnmhdr->code) block
```

The trackbar control's parent window tests to see if the NM_CUSTOMDRAW notification is sent by the trackbar. This is done by comparing the trackbar command identifier IDC_TRACKBAR to the NMHDR *idFrom* value:

```
if (lpnmcd->hdr.idFrom==IDC_TRACKBAR)
```

The next thing the code does is obtain the current item state. If you run the CUSTDRAW.EXE application and press the trackbar thumb, the black dot changes to gray. In order to do this, the application needs to know if the thumb is selected.

After that, the code proceeds as in the general NM_CUSTOMDRAW notification handler presented earlier. It returns CDRF_NOTIFYITEM-DRAW in response to the CDDS_PREPAINT notification. This ensures that the trackbar control sends further paint cycle notifications. In response to the individual CDDS_ITEMPREPAINT notifications, the code performs the custom trackbar drawing operations as implemented by the application function *OnDrawTrackbar*:

```
int OnDrawTrackbar(HWND hwnd,
  HDC hdc,
  DWORD dwItemSpec,
  BOOL bSelected)
{
  HBRUSH hBrushOld;
  RECT rc, rcChannel;
  int nRes;
  int nHeight, nCenter;
  /* Calculate the custom channel RECT */
  GetClientRect(hwnd, &rc);
  nHeight = (rc.bottom-rc.top);
  nCenter = rc.top+nHeight/2;
  SendMessage(hwnd, TBM_GETCHANNELRECT, 0,
   (LPARAM)&rcChannel);
  rcChannel.top = nCenter-12;
  rcChannel.bottom = rcChannel.top+12;
  switch(dwItemSpec)
  {
  case TBCD_THUMB:
   SendMessage(hwnd, TBM_GETTHUMBRECT, 0, (LPARAM)&rc);
   rc.top = rcChannel.top+1;
   rc.bottom = rcChannel.bottom-1;
   if (!bSelected)
   {
    hBrushOld = (HBRUSH)SelectObject(hdc,
     GetStockObject(BLACK_BRUSH));
   }
   else
   {
    hBrushOld = (HBRUSH)SelectObject(hdc,
     GetStockObject(LTGRAY_BRUSH));
   }
   RoundRect(hdc, rc.left, rc.top,
    rc.right, rc.bottom, 20, 20);
   nRes = CDRF_SKIPDEFAULT;
```

```
      break;
    case TBCD_CHANNEL:
      hBrushOld = (HBRUSH)SelectObject(hdc,
        GetStockObject(WHITE_BRUSH));
      Rectangle(hdc, rcChannel.left, rcChannel.top,
        rcChannel.right, rcChannel.bottom);
      nRes = CDRF_SKIPDEFAULT;
      break;
    case TBCD_TICS:
      /* Tic marks get drawn first. Therefore draw the
         entire control outline here, so that it doesn't
         wipe out any subsequent painting.
       */
      GetClientRect(hwnd, &rc);
      //First draw a black filled round rectangle
      hBrushOld = (HBRUSH)SelectObject(hdc,
        GetStockObject(BLACK_BRUSH));
      RoundRect(hdc, rc.left, rc.top,
        rc.right, rc.bottom, 20, 20);
      /* Next inset the rectangle slightly, and fill
         it with white to leave behind the black
         "drop shadow" outline.
       */
      SelectObject(hdc,GetStockObject(WHITE_BRUSH));
      rc.bottom--;
      rc.right--;
      RoundRect(hdc, rc.left, rc.top,
        rc.right, rc.bottom, 20, 20);
      nRes = CDRF_DODEFAULT;
      break;
    }  //End of switch(dwItemSpec) statement
    SelectObject(hdc, hBrushOld);
    return (nRes);
}
```

This function is responsible for drawing all of the trackbar control components. As parameters it takes the control HWND and HDC. It also takes the NMCUSTOMDRAW *dwItemSpec* value to specify which part of the control is to be drawn. Finally, it takes a BOOL indicating if the part of the control specified in the *dwItemSpec* parameter is selected.

The first six lines of this function determine the vertical center of the control's bounding rectangle and calculate a new channel rectangle, 24 pixels high and centered around that vertical center point.

The real fun begins with the *dwItemSpec* switch statement. For each of the three trackbar components (thumb, channel, and tic marks), drawing code is implemented to customize the appearance of the control.

The thumb is drawn as a small circle inside the trackbar channel. The color of the thumb depends on whether or not the thumb is pressed. After the thumb is drawn, *OnDrawTrackbar* (and hence the NM_CUSTOMDRAW notification) returns CDRF_SKIPDEFAULT. Since the application has customized the appearance of the thumb, it needs to prevent the control from drawing the default thumb.

The channel is drawn as a white rectangle of dimensions determined at the beginning of the function. This also returns CDRF_SKIPDEFAULT.

The most interesting case is the TBCD_TICS case. We wanted the trackbar to have a rounded border with a thin black outline. *OnDrawTrackbar* draws this outline here because trackbar tic marks are drawn before the thumb and channel. Since drawing the rounded outlines is done with calls to the Windows CE function *RoundRect*, anything inside the specified rectangle dimensions will be drawn over. Therefore, drawing the outline during the tic mark pre-paint notification doesn't erase the custom thumb or channel.

Notice how the application returns CDRF_DODEFAULT after the TBCD_TICS draw processing. The application only drew the control outline here, not the tic marks. To force the control to draw the standard tic marks, CDRF_DODEFAULT is returned.

Concluding Remarks

In this and the previous chapter, we have looked at how to take advantage of the features that Windows CE provides for customizing the appearance of various control classes. But owner draw and custom draw controls only allow your applications to modify the appearance of certain controls. What if you want to design and implement a completely new Windows CE control from scratch? You may not want to be limited to the customization hooks provided by owner draw and custom draw controls.

The next chapter shows you how to implement custom controls in Windows CE. Custom control programming techniques allow you to define completely new controls from scratch. With custom controls, you take complete control of the appearance and even the behavior of the controls you design. As you will see, custom controls provide for great flexibility in Windows CE user interface design.

CHAPTER 11

Designing Windows CE Custom Controls

Windows CE provides a lot of flexibility in designing user interfaces for applications. The child controls and common controls give programmers and designers a wide selection of user input and data presentation options. When the standard controls are not sufficient, Windows CE features such as owner draw buttons and the custom draw service provide ways to customize many of the more commonly used controls.

But sometimes even these options are not flexible enough to implement applications whose user interfaces deviate significantly from the standard Windows CE look and feel. In such cases, application programmers may be compelled to write user interface components from scratch. A *custom control* is any control used in an application that is not part of the Window CE operating system.[1]

A custom control is in many ways like any of the standard Windows CE child or common controls. Like the controls supplied with the Windows CE operating system, custom controls are child windows used in

[1] This definition would imply that other types of controls completely defined by an application programmer, such as ActiveX controls, are custom controls. While technically this is true, this chapter only describes custom controls as they are traditionally defined. Specifically this means any custom HWND-based control whose interface to a client application is the Win32 API.

applications to perform functions like displaying data, editing text, or responding to stylus taps in various ways. Your applications create custom controls by calling *CreateWindow* or *CreateWindowEx*, and respond to WM_COMMAND or WM_NOTIFY messages generated by the controls.

The big difference between the standard Windows CE controls and custom controls is that as the applications programmer, you design and implement the control. All aspects of the appearance of the control and its behavior are implemented by you to satisfy some special user interface needs not satisfied by any of the standard controls that come with Windows CE.

This points out the biggest benefit of using custom controls: As the control programmer, you have complete control over every aspect of the control's appearance and behavior.

In this chapter we will look at how to implement custom controls by implementing a simple custom control in a Windows CE dynamic link library.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

Statically or dynamically link with a dynamic link library

Implement a dynamic link library

Design and implement a custom control as a dynamic link library

The Example Custom Control

Rather than focus on writing fancy control features and graphical appearance, this chapter concentrates on the framework required to implement a custom control. Therefore the custom control implemented in this chapter will look very familiar. We implement the standard button from scratch, and add some new custom control styles in the process.

Yes, this is pretty boring from the point of view of learning how to create user interface components that look nothing like Windows CE. But

the aspects of implementing custom controls requiring the most attention are issues such as supporting control-specific styles and responding to window messages. Also more worth our time is how to package custom controls for the most convenient use by applications.

The custom button defines three styles that are specific to this custom control. These styles are detailed in Table 11.1. Some examples of custom buttons with the various styles are shown in Figures 11.1 and 11.2.

The complete source code for the custom control and the client application that uses the control can be found under \Samples\custom on the companion CD. The workspace (.dsw) file is under the \Samples\custom\control subdirectory and is called control.dsw. This workspace contains the project for the custom control, CONTROL.DLL, as well as the client application, CUSTOM.EXE.

Packaging a Custom Control as a Dynamic Link Library

The first thing to think about when implementing a Windows CE custom control is *not* how to make the control green. Programming the various appearance and behavioral aspects of the control comes second.

Your first concern as a custom control developer should be how to package the control. Your decision to implement a custom control was probably motivated by the specific requirements of one Windows CE application. But if designed properly, your control may find use in a number of different applications. You may even be able to sell it to other software developers.

Table 11.1 Custom Button Control Styles

STYLE	MEANING
CBTN_PUSH	Creates a custom button that acts like a standard Windows CE push-button control.
CBTN_TOGGLE	Creates a custom button that toggles, much like a check box. The button stays pressed when tapped. It must be tapped again to unpress it.
CBTN_LARGEFONT	Tells the control to draw button text with 18 point Times New Roman font. Without this style, the font used is 12 point Times New Roman.

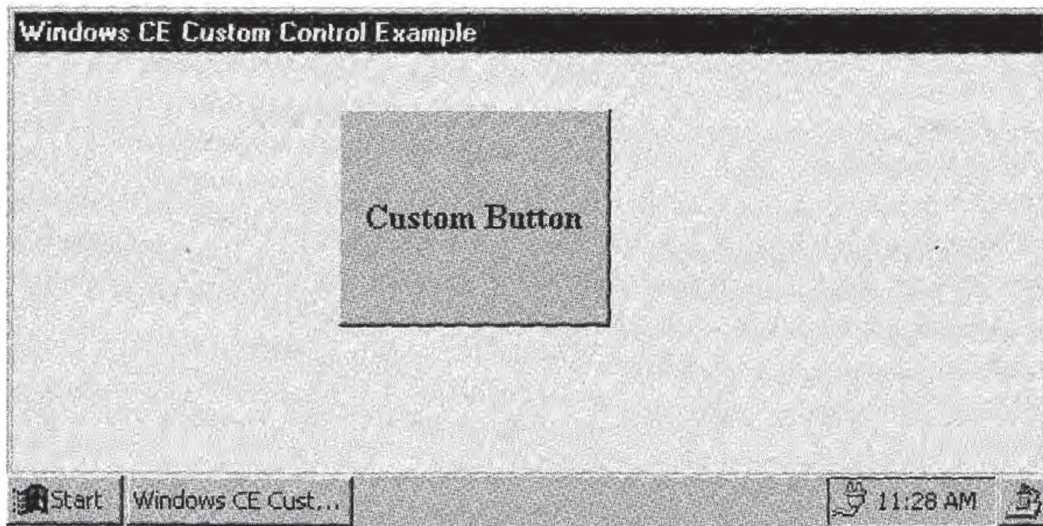


Figure 11.1 A custom button with the CBTN_PUSH style.

If a custom control is implemented inside an application that uses it, it is very hard for another application to use. Therefore, it is highly recommended that any custom control be implemented in a dynamic link library, or DLL. The control is thus packaged as a stand-alone module that can be used by any number of applications.

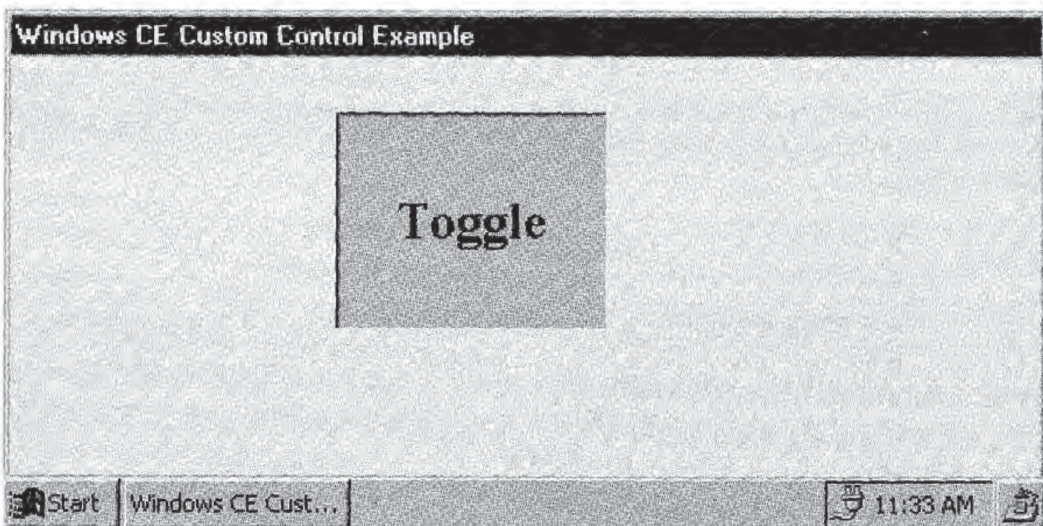


Figure 11.2 A custom button control with the CBTN_LARGEFONT and CBTN_TOGGLE styles in the pressed state.

DLL Basics

There are numerous resources that explore the details of dynamic link libraries in far more depth than can be covered here. This chapter will, however, highlight the main points of DLL programming and point out some of the benefits of using DLLs.

A dynamic link library is very similar to an executable application. A DLL can contain data, resources, and executable code just like an .EXE file. But a DLL is not a program. It is not run like an application. Applications load DLLs and use the data and resources they contain or call functions implemented inside the DLL. Dynamic link libraries are often called *application extensions* because they only become useful within the context of a running application.

Another important feature of dynamic link libraries relates to how functions in the DLL are linked to an application that uses them. Applications can link either *statically* or *dynamically* with the functions in a DLL.

When a DLL is loaded by an application, it technically means that a copy of the DLL is placed in the program memory of the application. The DLL and the application share the same address space. This is why more than one application can use the same DLL at the same time. For example, more than one Windows CE application written using MFC can run at the same time because each application has its own copy of the MFC DLL.

Static Linking

When you compile and link a dynamic link library, the DLL gets created as a file with the extension .DLL. Another file, called the *import library,* is also created with the extension .LIB. This file contains one or more *import records,* one for each function exported by the dynamic link library. Each import record contains the name of the DLL that contains the code implementing the function. It also contains either the name of the function or its *ordinal number*, or both. The ordinal number is simply a number that uniquely identifies a function in a DLL.

When an application statically links with a DLL, it means that the application links with the corresponding import library at link time. The information from the import records corresponding to any DLL

functions referenced by the application get copied into the application's .EXE file.

When the application runs, Windows CE looks for any dynamic links in the .EXE file. Windows CE loads any DLLs referenced in these links that are not already loaded and resolves the import record function reference to the actual address of the function in the DLL. Hence, whenever the application calls a function in a DLL with which it was linked, it can call into the proper function code at run-time.

Dynamic Linking

One disadvantage of static linking is that when an application that statically links with a DLL starts executing, the DLL is loaded as well—even if the application only calls one function in that DLL, and that very rarely. So the DLL takes up room in the application's address space even when it is not in use. This can become problematic if an application uses many DLLs, especially in Windows CE environments where memory is often in short supply.

Dynamic linking offers an alternative. With dynamic linking, an application does not link with the DLL's import library. No import records are copied into the executable, and no dynamic links are established. And if there are no dynamic links in the executable file, no DLLs get loaded when the application begins execution.

Instead, with dynamic linking the application is responsible for loading any DLLs that it uses. Furthermore, the application must determine the address of any function it needs to call, get a pointer to the function, and call the function by de-referencing the pointer. Finally, the application must unload the DLL when it is done using it, in the same way that it frees up resources like fonts or bitmaps.

The benefit of using dynamic linking is that an application can control when a particular DLL is loaded. The application can also better manage its memory resources by deleting DLLs from memory when they are not in use.

Dynamic linking obviously means more work for the application programmer. With static linking, as long as you include the appropriate DLL header files and link with the import library, you can make calls to DLL functions as you would call any other function. Dynamic linking forces the application programmer to load DLLs, get function pointers, and free DLLs.

Let's look at the dynamic linking steps in more detail. An application loads a dynamic link library by calling the Windows CE function *Load-Library*:

```
LoadLibrary(lpLibFileName);
```

lpLibFileName is the null-terminated Unicode string name of the DLL to load. A search path to the DLL name cannot be specified. You must therefore either give the full path name of the DLL to be loaded or depend on Windows CE to find the DLL. If you choose the latter alternative, Windows CE will first look in the root directory of the storage card attached to the device, if any. If there is no such card, or the DLL is not found in the card's root directory, Windows CE proceeds by looking in the \Windows directory. Finally, if that fails, it searches the device's root directory.

If the DLL is found, *LoadLibrary* returns a handle to the DLL as an HINSTANCE. If it fails, the function returns NULL.

Note that *LoadLibrary* can be used to load any Windows CE module (.EXE or .DLL). It is most commonly used for DLLs, though.

When an application is done with a DLL, it calls *FreeLibrary*:

```
FreeLibrary(hLibModule);
```

hLibModule is the DLL instance handle returned by the previous *Load-Library* call.

Note that calling *FreeLibrary* does not necessarily mean that the specified module is removed from the process memory. In multithreaded applications, *LoadLibrary* can be called by one or more threads in a process, incrementing the module's *reference count*. *FreeLibrary* decrements this reference count for the specified module and only removes it from memory once its usage count goes to zero.

Finally, to get a pointer to an exported DLL function, an application uses *GetProcAddress*:

```
GetProcAddress(hModule, lpProcName);
```

hModule is the instance handle of the DLL containing the function of interest. *lpProcName* is the Unicode string name of the function. *GetProcAddress* returns a pointer to the requested function.

We will see an example of how to dynamically link with a DLL a little later in this chapter.

Exporting DLL Functions

The discussion above made references to *exported* DLL functions. A function must be exported by a DLL in order for it to be available to applications or other DLLs.

A dynamic link library can export a function using any one of the following techniques:

- Using the /EXPORT linker option
- Defining functions to be exported with the __declspec(dllexport) modifier
- Specifying the functions to be exported in a module definition file

A module definition file name uses the .DEF extension.

Let's look at an example of a DLL module definition file. The custom button control DLL of this chapter exports the function *InitCustomButton* via this module definition file:

```
LIBRARY   CONTROL.DLL
EXPORTS

  InitCustomButton   @1
```

The first line of the file assigns the name "CONTROL.DLL" to the DLL. The EXPORTS keyword says that the functions that follow are to be exported. Specifically, these are the functions for which import records are included in the import library that is generated when the DLL is linked. The @ sign specifies the ordinal number to assign to the corresponding function in the import record. If an ordinal number is not specified in the .DEF file, one is assigned by the linker.

Any application that appropriately links with CONTROL.DLL, either statically or dynamically, can now call the function *InitCustomButton*.

The DLL Entry Point

When a Windows CE application starts running, a little piece of start-up code added to the beginning of the .EXE file by the linker calls a function known as the application *entry point*. This function is called *WinMain*. It is the function that application programmers think of as the starting point of their applications.

Dynamic link libraries also have an entry point. At various times, such as when a DLL is initially loaded by Windows CE, the operating system calls the function *DllMain*. The signature of *DllMain* is:

```
BOOL WINAPI DllMain(hinstDLL,
  fdwReason, lpvReserved);
```

The WINAPI modifier is simply defined as *__stdcall* in the Windows CE header files.

hinstDLL contains the instance handle of the DLL for which *DllMain* is being called. *fdwReason* and *lpvReserved* can take on various values depending on why *DllMain* is being called. These values and when they are passed to *DllMain* are described in Tables 11.2 and 11.3.

NOTE

CHANGING THE DLL ENTRY POINT FUNCTION NAME

You can freely change the entry point function name on a DLL-by-DLL basis. Simply use the /entry linker flag when linking the particular DLL. For example, many programmers like to use the name *DllEntryPoint* for their DLLs. To do so, add the following to the Project Options under the Link tab of the corresponding Microsoft Developer Studio project settings:

```
/entry:"DllEntryPoint"
```

Of course this new name must then be used in the entry point function implementation.

The value returned by *DllMain* is ignored except when the *fdwReason* parameter is DLL_PROCESS_ATTACH. In this case, *DllMain* should return TRUE if the DLL initialization succeeds and FALSE if it fails.

Table 11.2 DllMain fdwReason Parameter Values

VALUE	MEANING
DLL_PROCESS_ATTACH	DLL is being loaded for the first time by an application.
DLL_PROCESS_DETACH	DLL is being detached from the process that uses it. This happens when the process terminates or a *FreeLibrary* call has forced the DLL usage count to 0.
DLL_THREAD_ATTACH	A new thread has been created in the calling process.
DLL_THREAD_DETACH	A thread has been terminated in the calling process.

Table 11.3 DllMain lpvReserved Parameter Values

VALUE	MEANING
NULL	If *fdwReason* is DLL_PROCESS_ATTACH, this means that *DllMain* was called as a result of a dynamic *LoadLibrary* call. If *fdwReason* is DLL_PROCESS_DETACH, this means that *DllMain* was called as a result of a dynamic *FreeLibrary* call that reduced the DLL reference count to 0.
Non-NULL	If *fdwReason* is DLL_PROCESS_ATTACH, this means that *DllMain* was called as a result of a static DLL load when the process started. If *fdwReason* is DLL_PROCESS_DETACH, this means that *DllMain* was called as a result of process termination.

DLL Benefits

There are numerous reasons for using dynamic link libraries. The most important reasons include:

Maintainability. Applications are broken down into a number of components, each of which is easier to maintain than a larger monolithic application.

Reusability. Functions and resources exported by a DLL are more easily used by multiple applications.

Memory management. Applications have more direct control over memory usage if they choose when to load and free DLLs.

Initializing the DLL in the Client Application

Before we explore the details of programming our custom control DLL, let's look at how the client application initializes the DLL. This provides a real example of how to dynamically link an application with a DLL.

To use our custom button control, an application must do two things. It must first link with the dynamic link library that implements the control. As discussed above, this can be done either statically or dynamically. Next, the application must call the appropriate function to register the custom control window class.

The client application CUSTOM.EXE dynamically links with the custom control library CONTROL.DLL. The code below comes from the

WinMain function found on the companion CD in the file \Samples\custom\main.cpp. *hInstDLL* is an HINSTANCE defined in the file \Samples\custom\custom.h.

The DLL initialization function called *InitCustomButton* is described in detail in the next section.

```
#include <control.h>
typedef void (*LPINITCUSTOMBUTTON)();
LPINITCUSTOMBUTTON lpicb;
HINSTANCE hInstDLL;
HWND hwndExit;
int WINAPI WinMain(HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow)
{
  MSG msg;
  WNDCLASS wndClass;
  /* Save application instance in ghInst for
    possible use by other functions, such as
    the main window's window procedure.
   */
  ghInst = hInstance;
  /* We are dynamically linking with control.dll.
    The application must therefore load the DLL
    and get the address of all exported functions
    that it wishes to call.
   */
  hInstDLL = LoadLibrary(TEXT("control.dll"));
  if (hInstDLL)
  {
   lpicb = (LPINITCUSTOMBUTTON)GetProcAddress(
    hInstDLL, TEXT("InitCustomButton"));
   /* Call the custom control initialization function
     by dereferencing the function pointer extracted by
     the previous line of code.
    */
   (*lpicb)();
  }
  else
  {
   MessageBox(NULL, TEXT("Could not load DLL"),
    TEXT("Custom Control Sample Error"),
    MB_ICONEXCLAMATION|MB_OK);
  }
  /* Application code which registers the application
    main window class, creates the main window, etc.
    not shown
```

```
    */
    hwndExit = CreateWindow(
     CUSTOMBUTTON,
     TEXT("Exit"),
     WS_VISIBLE|WS_CHILD|CBTN_LARGEFONT,
     0,0,100,100,
     hwndMain,
     (HMENU)IDC_EXIT,
     hInstance,
     NULL);
    /* Etc. etc. */
}
```

The first interesting thing in this code sample is the *LoadLibrary* call. This function loads the dynamic link library CONTROL.DLL into the client application's address space. *hInstDLL* contains an instance handle of this DLL.

Next, the client application gets the address of the exported DLL function *InitCustomButton*. It does this by calling *GetProcAddress*:

```
lpicb = (LPINITCUSTOMBUTTON)GetProcAddress(
  hInstDLL, TEXT("InitCustomButton"));
```

lpicb is declared as type LPINITCUSTOMBUTTON. This type is defined by the client application as an alias for pointers to functions with the same signature as *InitCustomButton*. Therefore, after the *GetProcAddress* call, *lpicb* contains a pointer to the *InitCustomButton* function. The application then initializes the custom button control by calling this function by simply de-referencing this function pointer.

After that, the application is free to create instances of the custom button control by calling *CreateWindow* with the CUSTOMBUTTON window class name. The CUSTOMBUTTON symbol is defined in the header file CONTROL.H. We will see this definition a little later.

From the point of view of the client application, that's it. The application can now send window messages or any custom messages defined by the control to *hwndExit*. The application can also respond to notifications or messages (such as WM_COMMAND) that the button may send it.

Implementing the Custom Button Control

We now focus our attention on the details of the custom control DLL implementation.

The *InitCustomButton* Function

The first part of the custom button control implementation we will look at is the *InitCustomButton* function. This is the exported function that client applications call to register the control window class. This function also initializes the two fonts used by the control, but this part of the function is left out for brevity.

```
void InitCustomButton()
{
  WNDCLASS wndClass;
  wndClass.style        = 0;
  wndClass.lpfnWndProc  = ControlWndProc;
  wndClass.cbClsExtra   = 0;
  wndClass.cbWndExtra   = 8;
  wndClass.hInstance    = NULL;
  wndClass.hIcon        = NULL;
  wndClass.hCursor      = NULL;
  wndClass.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
  wndClass.lpszMenuName = NULL;
  wndClass.lpszClassName = CUSTOMBUTTON;
  RegisterClass(&wndClass);
  /* This function also goes on to create the
    two fonts used by the DLL.
   */
}
```

ControlWndProc is the window procedure of the custom button control. We will discuss this function later. Also notice that the *cbWndExtra* member of the WNDCLASS structure is 8. This means that every custom button HWND carries around 8 extra bytes. We will see how these bytes are used later as well.

The header file associated with the custom control DLL, CONTROL.H, includes the following definition:

```
#define CUSTOMBUTTON (TEXT("CUSTOMBUTTON"))
```

This defines the window class name for the control. This name gets assigned to the *lpszClassName* member of the WNDCLASS structure.

Note that since all window classes are global under Windows CE, the *hInstance* member of the window class structure can be set to NULL.

NOTE

A Custom Control Class Can Be Registered in DllMain

Your custom control implementations can do all of the control window class registration in *DllMain* in response to the DLL_PROCESS_ATTACH event. This would eliminate the need for implementing an initialization function. Applications would have one less function to call as well. This chapter chooses to implement the initialization function to more fully demonstrate dynamic linking, and the way *GetProcAddress* is used to obtain pointers to exported functions.

Custom Button Control Styles

The custom button control supports three control styles: CBTN_PUSH, CBTN_TOGGLE, and CBTN_LARGEFONT. Refer to Table 11.1 at the beginning of the chapter for a description of these styles.

In Windows CE, control styles generally occupy the low word of the 32-bit integer that defines the control window styles. The high word is used for the window styles such as WS_CHILD or WS_VISIBLE.

The custom control header file CONTROL.H thus defines the three control styles as the following 16-bit integers:

```
#define CBTN_PUSH     0x0000
#define CBTN_TOGGLE   0x0001
#define CBTN_LARGEFONT 0x0002
```

Assigning CBTN_PUSH a value of zero means that this style is the default custom button control style. Each of the other styles is assigned a unique bit of a 16-bit integer.

An instance of the custom button control class often needs to test one or more of these style bits to see if it is set. Determining if a particular button is toggle style, for instance, requires that the custom control code check for the CBTN_TOGGLE bit.

A window can check its styles by calling the Windows CE *GetWindowLong* function. The function extracts a particular 32-bit integer value that is stored with the window:

```
GetWindowLong(hWnd, nIndex);
```

hWnd specifies the window, and *nIndex* is a value specifying which 32-bit value to retrieve. This parameter can be any one of the values shown in Table 11.4.

As an example, an application can check to see if an instance of the custom button control class has the CBTN_LARGEFONT style with the following code:

```
HWND hwndButton;  //The custom button control HWND
DWORD dwStyle;

dwStyle = GetWindowLong(
  hwndButton,
  GWL_STYLE);
if (dwStyle & CBTN_LARGEFONT)
{
  //Do something
}
```

GetWindowLong has a counterpart, *SetWindowLong*, which can be used to set any of the values described in Table 11.4.

```
SetWindowLong(hWnd, nIndex, dwNewLong);
```

This function sets the window value indicated by *nIndex* to the value specified by *dwNewLong*. The previous value that the window stored for that index is returned by the function.

NOTE

UNSUPPORTED INDICES

Under Windows CE, *GetWindowLong* and *SetWindowLong* do not support the GWL_HINSTANCE, GWL_HWNDPARENT, or GWL_USERDATA indices.

Table 11.4 GetWindowLong and SetWindowLong Index Values

VALUE	MEANING
GWL_EXSTYLE	Retrieves/sets the window extended style.
GWL_STYLE	Retrieves/sets the window style.
GWL_WNDPROC	Retrieves/sets the window procedure.
GWL_ID	Retrieves/sets the window identifier.
DWL_DLGPROC	Retrieves/sets the dialog procedure for a specified dialog box.
DWL_MSGRESULT	Retrieves/sets the return value of a message processed in the dialog box procedure.
DWL_USER	Retrieves/sets an application-specific 32-bit value associated with the specified dialog box.

Performing a Window Brain Transplant: Window Subclassing

The Windows CE functions *GetWindowLong* and *SetWindowLong* provide some very powerful possibilities when used with the GWL_WNDPROC index. With these functions, an application can *subclass* any Windows CE window.

Window subclassing in this sense means replacing the window procedure of a window with a different window procedure. With this technique, an application can give custom behavior to a window on the fly.

This becomes most significant when subclassing Windows CE controls. Subclassing a control lets you take full advantage of the default functionality of the control while adding custom behavior only where needed.

To subclass a control or any other window, you write a window procedure for that control which contains your custom responses to the various Windows CE messages you wish to override. You then make the control use this window procedure by calling *SetWindowLong*:

```
LRESULT CALLBACK myWndProc(HWND hwnd,
  UINT message.
  WPARAM wParam,
  LPARAM lParam);
HWND hwndControl;
WNDPROC wndProcOld;
wndProcOld = (WNDPROC)SetWindowLong(
  hwndControl, GWL_WNDPROC, (LONG)myWndProc);
```

In one step, *SetWindowLong* changes the window procedure to be used by the control *hwndControl* and stores the original control window procedure in *wndProcOld*.

Using Extra Window Words to Maintain the State of the Control

So far we have ignored the significance of the following line of code that is executed when initializing the custom button control's window class:

```
wndClass.cbWndExtra = 8;
```

As discussed back in Chapter 2, this statement means that every instance of the window class will contain eight extra bytes, or two 32-bit values. An application can use these values to store information about an instance of the window class.

Saving the old window procedure makes it easy for your custom window procedure to implement the original default behavior for messages you do not want to override. This is done by calling *CallWindowProc*:

```
CallWindowProc(wndprcPrev, hwnd, uMsg, wParam, lParam);
```

This function passes handling of the message *uMsg* to the window procedure specified by *wndprcPrev*. In other words, *CallWindowProc* allows an application to directly call into a specified window procedure.

As an example, let's say that you only wish to customize the stylus tap logic for a list box control. You could write the custom list box window procedure as follows. *ListWndProcOld* is the original list box window procedure extracted by a call to *SetWindowLong* like the one shown above.

```
LRESULT CALLBACK myWndProc(
  HWND hwnd,
  UINT message.
  WPARAM wParam,
  LPARAM lParam)
{
  switch(message)
  {
   case WM_LBUTTONDOWN:
    //Custom logic here
    return (0);
   default:
    return (CallWindowProc(ListWndProcOld,
     hwnd, message, wParam, lParam));
   }
}
```

These extra words are accessed using *GetWindowLong* and *SetWindowLong*. Instead of using one of the predefined indices in Table 11.4, the application references a particular 32-bit value by its zero-based offset. Since the number of bytes defined by *cbWndExtra* must be a multiple of four, and each extra word is 4 bytes, the index of the first word is 0, the second is 4, and so on.

For example, to access the second of these extra window words, an application would do this:

```
DWORD dwVal;
dwVal = GetWindowLong(hwndControl, 4);
```

The custom button control implementation uses the two extra words it defines to keep information about the state of each instance of the control class. CONTROL.H defines the following two indices:

```
#define GWL_BUTTONINVERT    0
#define GWL_BUTTONFONT      4
```

These are the indices used to access the first and second extra window words, respectively, defined by the custom button control's window class.

The first extra window word keeps track of whether the particular button is currently painted in the inverted state. This information is used by the control's window procedure, for example, to determine which state to repaint a toggle-style control in when a user taps it.

The second word stores the font handle of the font used to draw the button text.

As an example of how this state information is used, let's look at the processing that occurs when a custom button control is created. The following code comes from the control window procedure's WM_CREATE handler. *hwnd* is the HWND of the control, and *lpcs* is a pointer to the CREATESTRUCT. *hFont12Pt* and *hFont18Pt* are the handles to the two fonts available to the control:

```
if (lpcs->style & CBTN_LARGEFONT)
{
  SetWindowLong(
    hwnd,
    GWL_BUTTONFONT,
    (LONG)hFont18Pt
    );
}
else
{
  SetWindowLong(
    hwnd,
    GWL_BUTTONFONT,
    (LONG)hFont12Pt
    );
}
return (TRUE);
```

When an instance of the control is created, the appropriate font is assigned depending on whether the CBTN_LARGEFONT style bit is set.

The control then uses this font when painting itself. The WM_PAINT handler code contains the following line for selecting the font into the control's device context. *hwnd* is the window handle of the control:

```
hFontOld = (HFONT)SelectObject(hdc,
    (HFONT)GetWindowLong(hwnd, GWL_BUTTONFONT));
```

Any text drawn inside the control is thus in the correct font.

Handling Button Presses

Probably the most interesting code in the custom button control implementation is the stylus handling logic. The custom button control class supports the CBTN_PUSH and CBTN_TOGGLE styles, and the stylus code must implement the appropriate behavior for both.

To make it easier for the control source code to determine which of these styles is assigned to a particular button, the CONTROL.H header file defines the following macro:

```
#define IsToggleStyle(hwnd) \
  ((GetWindowLong(hwnd, GWL_STYLE) & CBTN_TOGGLE) \
  ? TRUE : FALSE)
```

The control also uses the first extra window word, indexed by GWL_BUTTONINVERT, to keep track of whether a button instance is in the pressed or unpressed state.

CBTN_PUSH Style Custom Button Controls

Buttons with the CBTN_PUSH style act like regular Windows CE button controls. When pressed, they are painted in the pressed state. Once released, they are repainted in the unpressed state. Also, if a user presses the button and drags the stylus on and off the button without releasing the button, the button changes between the pressed and unpressed states.

When a user presses a CBTN_PUSH-style button, the following WM_LBUTTONDOWN handler code is invoked:

```
SetCapture(hwnd);
SetWindowLong(hwnd, GWL_BUTTONINVERT, (LONG)TRUE);
```

The stylus capture is assigned to the button control window. This forces all subsequent stylus input to be passed to the control. Next, the

GWL_BUTTONINVERT extra window word is set to TRUE. This state information will be used during WM_PAINT handling to determine which state to paint the button in. Specifically, the WM_PAINT handler includes the following code. *hwnd* is the HWND of the control, and *hdc* is the control's device context:

```
RECT rc;
BOOL bInvert;
GetClientRect(hwnd, &rc);
bInvert = (BOOL)GetWindowLong(hwnd, GWL_BUTTONINVERT);
if (bInvert)
{
  DrawEdge(hdc, &rc, EDGE_SUNKEN,
   BF_SOFT|BF_RECT);
}
else
{
  DrawEdge(hdc, &rc, EDGE_RAISED,
   BF_SOFT|BF_RECT);
}
```

In the pressed state the control is drawn with a sunken edge. In the unpressed state, it is drawn in the raised state.

When the user releases the button, the WM_LBUTTONUP code shown in Figure 11.3 is invoked.

Capture is released from the control HWND. The GWL_BUTTONINVERT window word is set to FALSE so that the button is drawn in the unpressed state during the next paint cycle.

The rest of the WM_LBUTTONUP code determines whether the stylus was in the button when it was released. If so, the button sends a WM_COMMAND message to its parent window. WM_COMMAND messages are not sent if a user presses a button and then drags the stylus off of it without releasing the button.

The *PtInRect* call determines if the stylus point is in the control's bounding rectangle. Also note the use of the GWL_ID index in the call to *GetWindowLong*. This extracts the control identifier of the custom button, which must be sent with the WM_COMMAND message.

Finally, the control must handle stylus move messages. The WM_MOUSEMOVE handler implementation is:

```
RECT rc;
BOOL bInvert;
if (GetCapture()==hwnd)
```

```
{
  GetClientRect(hwnd, &rc);
  bInvert = (PtInRect(&rc, pt)) ? TRUE : FALSE;
  if (!IsToggleStyle(hwnd))
  {
   SetWindowLong(hwnd, GWL_BUTTONINVERT, (LONG)bInvert);
  }
}
```

If the control does not have stylus capture, the WM_MOUSEMOVE handler does nothing. This means that if the stylus moves when a button is not pressed, nothing needs to be done.

Otherwise, the code simply toggles the GWL_BUTTONINVERT state depending on whether the stylus is dragged into or out of the pressed button. When the button is repainted, its appearance will alternate between the pressed and unpressed states.

CBTN_TOGGLE Style Custom Button Controls

There are only minor differences in the stylus handling for CBTN_TOGGLE-style buttons. The basic difference is determining when to change the GWL_BUTTONINVERT state so that the button remains pressed until tapped again.

```
HWND hwndParent;
RECT rc;
GetClientRect(hwnd, &rc);
ReleaseCapture();
if (!IsToggleStyle(hwnd))
{
  SetWindowLong(hwnd, GWL_BUTTONINVERT, (LONG)FALSE);
}
  /* Send a WM_COMMAND to the parent if
    stylus went up in the control.
   */
if (PtInRect(&rc, pt))
{
  InvalidateRect(hwnd, NULL, TRUE);
  hwndParent = GetParent(hwnd);
  SendMessage(
   hwndParent,
   WM_COMMAND,
   MAKEWPARAM((WORD)GetWindowLong(hwnd, GWL_ID), 0),
     (LPARAM)hwnd);
}
```

Figure 11.3 Custom button control WM_LBUTTONUP message handling code.

The WM_LBUTTONDOWN code for CBTN_TOGGLE-style controls looks like this:

```
BOOL bInvert;
SetCapture(hwnd);
bInvert = (BOOL)GetWindowLong(hwnd, GWL_BUTTONINVERT);
SetWindowLong(hwnd,
  GWL_BUTTONINVERT, (LONG)(!bInvert));
```

Whereas the CBTN_PUSH style controls always set the GWL_BUTTONINVERT state to TRUE on a stylus tap, the CBTN_TOGGLE buttons toggle this state.

If you refer to the WM_LBUTTONUP code in Figure 11.3, you will see only one difference between CBTN_TOGGLE and CBTN_PUSH controls. The difference is these lines, which force toggle-style buttons to set the invert state extra window word to FALSE:

```
if (!IsToggleStyle(hwnd))
{
  SetWindowLong(hwnd, GWL_BUTTONINVERT, (LONG)FALSE);
}
```

In other words, when the user releases a CBTN_TOGGLE-style button, the toggle state does not change. In this way, a pressed button stays pressed and an unpressed button stays unpressed. The toggle state does not change until the button is pressed again, as shown by the WM_LBUTTONDOWN code above.

Similarly, CBTN_TOGGLE-style buttons do not change state when the stylus moves across them. These lines in the WM_MOUSEMOVE code ensure this:

```
if (!IsToggleStyle(hwnd))
{
  SetWindowLong(hwnd, GWL_BUTTONINVERT, (LONG)bInvert);
}
```

The Complete Sample Application

The complete source code for the custom button control DLL and the client application are included on the companion CD. The directory \Samples\custom contains the client application source code. The directory \Samples\control contains the source code for the custom control DLL, as well as the Microsoft Developer Studio workspace file for both of these components.

Concluding Remarks

In Chapters 9, 10, and 11, we have discussed some of the features provided by Windows CE for designing custom application user interfaces. Owner draw controls let you take over the control painting process in your application code. The custom draw service is similar, but provides more flexibility by giving your application the opportunity to intercede at specific times in the control's paint cycle.

There may be occasions when even more customization is required. For these jobs you can implement complete new controls from scratch using the custom control programming techniques you have just read about.

As you gain experience using these programming techniques, you will come to appreciate the flexibility that Windows CE offers for designing custom user interfaces for your applications.

The next two chapters discuss some additional Windows CE user interface features. Chapter 12 introduces the HTML Viewer control. Chapter 13 describes how to program some of the user interface features specific to the Palm-size PC platform.

CHAPTER 12

The HTML Viewer Control

The advent of the Internet and the World Wide Web have led to dramatic changes in the basic functionality of computer operating systems. Microsoft's Win32-based operating systems are no exception. Windows NT, Windows 98, and Windows CE are all very comfortable with providing users with the ability to browse the Internet and display data in all of the various new formats that this medium has generated.

Applications such as Pocket Internet Explorer, the Microsoft Windows CE Web browser, can display hypertext markup language (HTML) documents containing formatted text and images just like their desktop Web browser counterparts.

HyperText Markup Language, or HTML, has always been an important part of the Internet. One of the simplest ways to display HTML in Windows CE applications is with the HTML viewer control provided with Windows CE. This control is in fact used by Pocket Internet Explorer for viewing HTML pages. As we will see, adding HTML rendering capabilities to your Windows CE applications can be easily done with the HTML viewer control.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

Use the HTML viewer control to display HTML documents

Overview of the HTML Viewer Control

The Windows CE HTML viewer control is not a child control. Therefore it is not part of the Graphics, Windowing, and Event Subsystem. Nor is it part of COMMCTRL.DLL. That means it does not get loaded along with the tree view and trackbar controls when you call *InitCommonControls*.

The HTML viewer control resides in its very own dynamic link library, called HTMLVIEW.DLL. To use the control, applications must either explicitly link with the DLL's import library, HTMLVIEW.LIB, or dynamically link with HTMLVIEW.DLL by loading the DLL at run-time.

Next, before using the HTML viewer control, applications must initialize it by calling *InitHTMLControl*. This is a function, exported by HTMLVIEW.DLL, that is responsible for registering the control's window class. After calling *InitHTMLControl*, applications are free to create instances of the control. In these respects, the design of this control closely resembles the custom control we designed in the previous chapter.

Control Features

The HTML viewer control provides very basic (and I mean *very* basic) functionality for viewing HTML documents. As we will soon see, the control leaves many of the details of navigating between HTML links, displaying images, and playing sounds to the application that contains the control.

First and foremost, it is important to keep in mind that the control is only an HTML viewer. It does not provide HTML editing capabilities.

Next, the HTML viewer control interprets HTML and displays data in the correct format. The control understands all the standard HTML

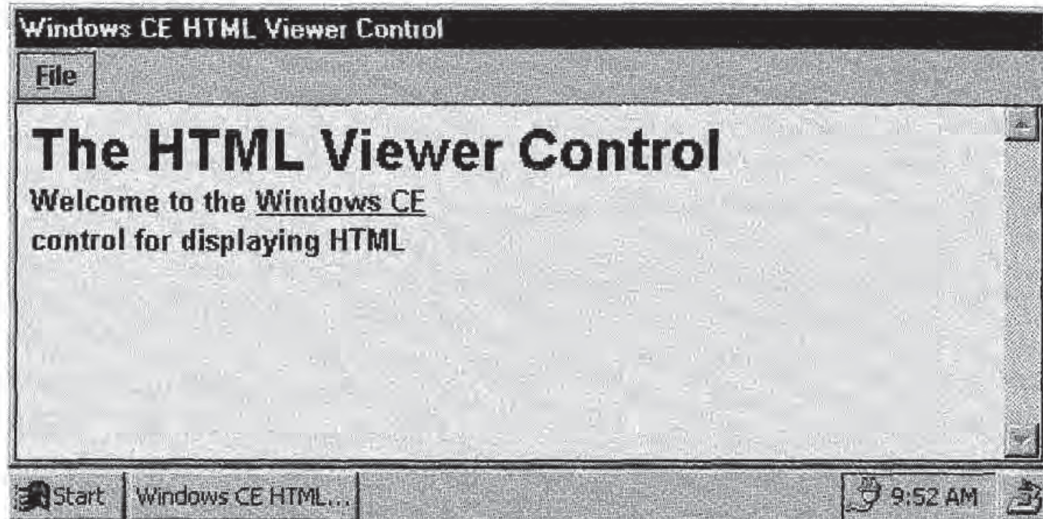


Figure 12.1 The HTML viewer control at work.

tags, and displays text in the correct sizes and styles based on those tags.

As an example, Figure 12.1 shows the HTML viewer control's rendering of the following HTML file:

```
<h1>The HTML Viewer Control</h1>
<h4>Welcome to the <a href="\Windows\wince.htm">Windows
  CE</a></h4>
<h4>control for displaying HTML</h4>
```

The only work the application had to do was send the HTML text to the control. The HTML viewer control takes care of the rest.

The situation is not quite so pleasant in the case of images or sound references in an HTML document. For example, the HTML viewer control does not automatically render GIF files referenced in a document. Nor does it automatically play sound files.

If we add a line to the top of the sample HTML file referencing an inline GIF image, the HTML viewer control displays the file as shown in Figure 12.2:

```
<IMG SRC="\Windows\home.gif">
<h1>The HTML Viewer Control</h1>
<h4>Welcome to the <a href="\Windows\wince.htm">Windows  CE</a> </h4>
<h4>control for displaying HTML</h4>
```

The HTML viewer control only sends a notification to its parent window when it reads a reference to an image from an HTML file. It is the

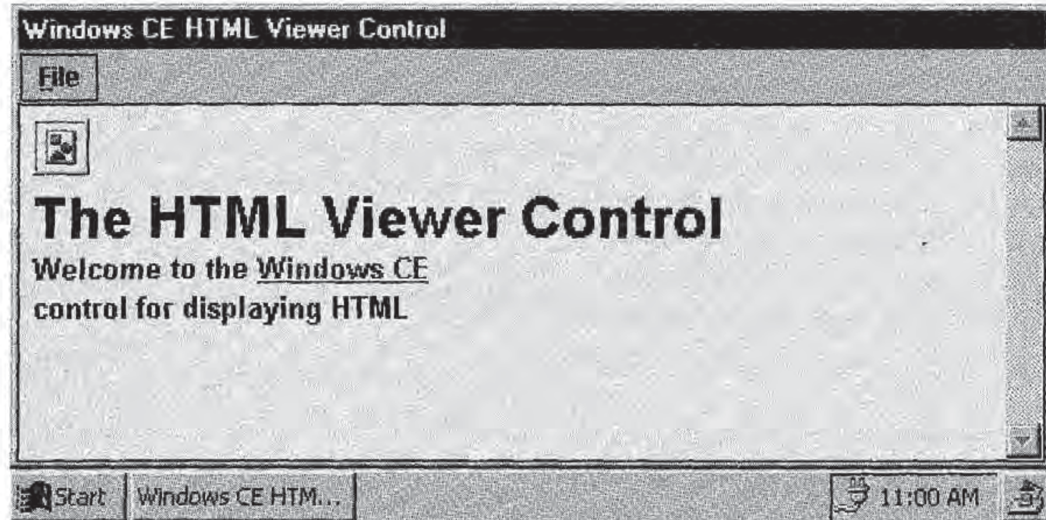


Figure 12.2 The HTML Viewer Control displaying an image.

responsibility of the application to then convert the image file to a Windows CE bitmap. The application must then send the appropriate message to the control telling it to display the bitmap representation of the original image file.

The HTML viewer control will also notify its parent when a user has tapped on a hypertext link. It is again, however, the application's responsibility to follow the link. That is, the application must respond to this notification by loading the referenced file and telling the HTML viewer control to display it.

At first this seems like a major limitation of the control. But it does make sense for the application, not the control, to do this work. A particular instance of the HTML viewer control has no idea if the HTML documents it is being asked to render are stored locally on the Windows CE device or are coming from a live Internet connection. The application has all of this context information and should therefore be responsible for supplying the data to the control in the correct form.

NOTE

THE NAMES SHOULD HAVE BEEN CHANGED TO PROTECT THE INNOCENT

You will notice that all of the HTML viewer control message names begin with DTM_. This is somewhat confusing, because the message names for the date time picker control start with DTM_ as well.

The Sample Application

Our discussion of the HTML viewer control is motivated by the sample application HTML.EXE. This application, which can be found in \Samples\html on the companion CD, illustrates many of the features provided by the HTML viewer control.

HTML.EXE implements a basic HTML file viewer. It allows a user to view any HTML file on the file system of a Windows CE device (or the emulation file system if run in the emulator). Two simple HTML files, sample.htm and wince.htm, are provided. These files must be placed in the device or emulator file system under the \Windows directory. See the section on "Viewing the Windows CE Object Store" in the introduction to Part II for a description of how to transfer files using the Remote Object Viewer.

The main application window was shown in Figure 12.1. When the application first starts, there is no HTML displayed, however. Users can display files by opening them with the Open option of the File menu. Choosing this option presents the user with a standard File Open dialog shown in Figure 12.3. Using this interface, the user can search the file system for a particular HTML file. Selecting the file and pressing the OK button causes the application to display the contents of the selected file.

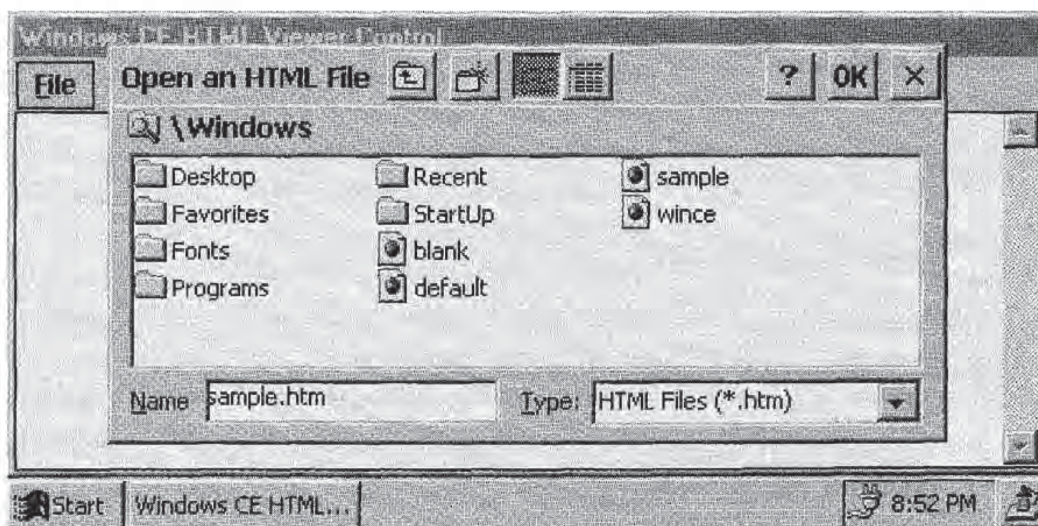


Figure 12.3 The File Open dialog for selecting HTML files.

Preparing to Use the HTML Viewer Control

As mentioned earlier, an application must initialize the HTMLVIEW. DLL module before instances of the HTML viewer control can be created and used. Applications can link with HTMLVIEW.DLL dynamically, or they can link with HTMLVIEW.LIB to avoid loading the DLL at run-time.

To make the application a little less complicated, HTML.EXE links with the import library. It can therefore call *InitHTMLControl* directly. The only thing that must be done that we have not yet mentioned is that the application must include the file HTMLCTRL.H. This is the header file that defines all of the messages and notifications which the HTML viewer control can send, as well as the function *InitHTMLControl.*

InitHTMLControl must be called in order to register the window class that defines the HTML viewer control. This step is analogous to calling *InitCommonControls* before using COMMCTRL.DLL.

HTML.EXE calls *InitHTMLControl* in *WinMain*:

```
#include <htmlctrl.h>
int WINAPI WinMain(HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow)
{
  /* Other application code */
  ...
  InitHTMLControl(hInstance);
  ...
}
```

InitHTMLControl takes only one argument: the HINSTANCE of the application that is using the control. It returns TRUE if the window class is registered properly. Otherwise it returns FALSE.

Creating HTML Viewer Controls

You are now ready to create instances of the HTML viewer control. This step is done with *CreateWindow* or *CreateWindowEx,* just as when creating any other Windows CE control. The only thing left to know is the name of the window class to use.

The HTML viewer control window class is called DISPLAYCLASS. That's an obvious one, isn't it? To create a control with a control identifier defined as IDC_HTML and whose parent is *hwndParent*, an application would do this:

```
HWND hwndHTML;
hwndHTML = CreateWindow(
   TEXT("DISPLAYCLASS"), NULL,
   WS_CHILD|WS_VISIBLE|WS_BORDER,
   0,0,100,100, //i.e., some dimensions
   hwndParent, (HMENU)IDC_HTML,
   hInstance, NULL);
```

The control, referred to by *hwndHTML*, is now ready to display HTML data.

NOTE

HTML Viewer Control Styles

Unlike most of the other Windows CE controls, the HTML viewer control defines no control-specific styles or extended styles to allow programmers to customize the control's appearance or behavior.

Displaying HTML Formatted Text

As an HTML viewer, the HTML viewer control must provide a way for applications to give it HTML text to display.

Text is inserted into the control by sending either the message DTM_ADDTEXT or DTM_ADDTEXTW. DTM_ADDTEXT is for adding non-Unicode text. DTM_ADDTEXTW is the Unicode version.

HTML.EXE uses DTM_ADDTEXT because it assumes that the files it reads are saved as ANSI text, not Unicode.

As the name of these two messages implies, they add text to the specified control. So if there is already text in an HTML viewer control before an application sends one of these messages, the text is added to the bottom of the control's client area. The message does not cause the control to clear its contents before the new text is displayed.

In order to display text by itself, you must first remove the current contents of the control. The easiest way to do this is to send a WM_SETTEXT message to the control with a NULL text string:

```
SendMessage(hwndHTML, WM_SETTEXT, 0, NULL);
```

Closely related to these ADDTEXT messages is the message DTM_ENDOFSOURCE. An application sends this message to an HTML viewer control to tell it that the application is done sending text. Both parameters to this message, *wParam* and *lParam*, are zero.

NOTE

DO YOU REALLY NEED DTM_ENDOFSOURCE?

It appears that not sending DTM_ENDOFSOURCE has no negative side effects.

An Example

The sample application HTML.EXE allows users to pick the HTML file to display using a File Open dialog. Once a file is specified, the application must read the HTML text from the file and send it to the HTML viewer control using the DTM_ADDTEXT message.

The function that HTML.EXE uses to load an HTML file is called *LinkToFile*. (This name will make more sense a bit later when we see that the application uses the same function for following hyperlinks.)

```
void LinkToFile(TCHAR* pszFilename)
{
  DWORD dwSize, dwBytes;
  LPSTR lpszBuf;
  WIN32_FIND_DATA fd;
  CEOIDINFO oidInfo;
  //Clear the control
  SendMessage(hwndHTML, WM_SETTEXT, 0, NULL);
  hFile = FindFirstFile(pszFilename, &fd);
  if (INVALID_HANDLE_VALUE!=hFile)
  {
   CeOidGetInfo(fd.dwOID, &oidInfo);
   dwSize = oidInfo.infFile.dwLength+1;
   FindClose(hFile);
  }
  hFile = CreateFile(pszFilename,
   GENERIC_READ, 0,
   NULL, OPEN_EXISTING,
   FILE_ATTRIBUTE_NORMAL, NULL);
  lpszBuf = (LPSTR)LocalAlloc(LPTR, dwSize+1);
  lpszBuf[dwSize] = 0; //Add NULL terminator
  ReadFile(hFile, lpszBuf, dwSize, &dwBytes, NULL);
  SendMessage(hwndHTML, DTM_ADDTEXT,
   (WPARAM)FALSE, (LPARAM)lpszBuf);
```

```
  SendMessage(hwndHTML, DTM_ENDOFSOURCE, 0, 0);
  CloseHandle(hFile);
}
```

hwndHTML is the window handle of the HTML viewer control. It is defined as a global variable so that the entire application can reference it. *LinkToFile* looks formidable, but it is actually quite straightforward.

First it searches the file system for the file specified by the parameter *pszFilename*. The application already knows the name of the file to display because the user selected a file from the File Open dialog. But the application calls *FindFirstFile* in order to get the object identifier of the selected file. It can then get the size of the file with the *CeOidGetInfo* call, which it needs to allocate space for the file contents. The function then opens the file with the appropriate call to *CreateFile*.

Next, *LinkToFile* allocates enough space in the buffer *lpszBuf* to hold the contents of the entire file by calling *LocalAlloc*. It knows the size it needs to allocate because previously the function determined the byte size of the file. Once the *lpszBuf* is allocated, *ReadFile* fills the buffer with the contents of the file *pszFilename*.

Finally, *LinkToFile* sends a DTM_ADDTEXT message to display the HTML text it just read in the HTML viewer control:

```
SendMessage(hwndHTML, DTM_ADDTEXT,
  (WPARAM)FALSE, (LPARAM)lpszBuf);
```

As noted in Table 12.1, the FALSE *wParam* parameter tells the control to treat the contents of *lpszBuf* as HTML formatted text.

Handling Hyperlinks

The Windows CE HTML viewer control provides some built-in support for hyperlinks in HTML formatted text. The control knows how

Table 12.1 The DTM_ADDTEXT/DTM_ADDTEXTW Messages

PARAMETER	MEANING
(BOOL)wParam	Indicates the type of text to add. If TRUE, the control treats the text as plain text. If FALSE, the control treats the text as HTML formatted text.
(LPSTR)lParam or (LPWSTR)lParam	Pointer to the string to be added. String is ANSI for DTM_ADDTEXT, Unicode for DTM_ADDTEXTW.

to recognize anchor tags. The text corresponding to an anchor is automatically formatted when the HTML viewer control displays the text. Specifically, the text color is changed to blue and the text is underlined.

The HTML viewer control also notifies its parent window whenever a user taps on such a link on the display screen. All of the logic for determining where the stylus taps the screen and decides if that point corresponds to a link is implemented in the viewer control's window procedure.

On the other hand, an application that uses an HTML viewer control is responsible for responding to the fact that a user tapped on a hyperlink. If the link points to another file, the application must load and display it.

The HTML viewer control sends notifications to its parent window when events such as a hyperlink tap occur. These notifications are sent in the form of WM_NOTIFY messages. This is exactly the same notification mechanism used by the Windows CE common controls.

The notification that is sent when a hyperlink is tapped is NM_HOTSPOT. This notification is also sent in response to the user submitting a form, a subject we will not be covering here. The *lParam* sent with this notification is a pointer to an NM_HTMLVIEW structure, In fact, many of the notifications sent by the HTML viewer control pass such a structure.

NM_HTMLVIEW is defined as:

```
typedef struct tagNM_HTMLVIEW
{
  NMHDR hdr;
  LPSTR szTarget;
  LPSTR szData;
  DWORD dwCookie;
} NM_HTMLVIEW;
```

The first member, as with any control notification, is an NMHDR structure.

szTarget contains a NULL-terminated string whose meaning depends on the notification being sent. For example, in the case of the NM_HOTSPOT notification, *szTarget* contains the string that follows the HREF field in the line of HTML text defining the link.

For example, consider again the situation illustrated in Figure 12.1 and the sample HTML file that produced that display. If a user taps the "Windows CE" link in that case, the HTML viewer control that contains the text sends an NM_HOTSPOT notification. The *szTarget* member of the NM_HTMLVIEW structure that is sent with this notification will contains the string "\Windows\wince.htm."

szData and *dwCookie* contain data specific to the particular notification being sent. For example, in the case of an NM_HOTSPOT, *szData* contains the query data sent with a form POST submission. The *dwCookie* value is not used by NM_HOTSPOT.

An Example: How HTML.EXE Follows Links

Now let's see how to respond to the NM_HOTSPOT notification in a real example.

If you run the HTML.EXE sample application, you will see that it properly follows hyperlinks. As long as the links refer to HTML files that are in the emulator file system or the file system of the Windows CE device running the application, everything works fine.

This behavior is implemented by responding to the NM_HOTSPOT notification described above. The relevant part of main application window's window procedure is given below:

```
LRESULT CALLBACK WndProc(
  HWND hwnd,
  UINT message,
  WPARAM wParam,
  LPARAM lParam)
{
  UINT nID;
  int nLen;
  TCHAR* pszFilename;
  switch(message)
  {
  case WM_NOTIFY:
   NM_HTMLVIEW* lpnm;
   lpnm = (NM_HTMLVIEW*)lParam;
   switch(lpnm->hdr.code)
   {
   case NM_HOTSPOT:
    nLen = strlen(lpnm->szTarget);
    pszFilename = (TCHAR*)LocalAlloc(LPTR,
```

```
       sizeof(TCHAR)*nLen);
      AnsiToWide(lpnm->szTarget, pszFilename);
      LinkToFile(pszFilename);
      LocalFree(pszFilename);
      break;
     default:
      break;
    }      //End of switch(lpnm->hdr.code) block
    return (0);
    /* Other message handler code */
    ...
    default:
     return (DefWindowProc(hwnd, message, wParam, lParam));
    }      //End of switch(message) block
  }
```

The NM_HOTSPOT notification handler first extracts the string containing the HREF hyperlink text from the *szTarget* member of the NM_HTMLVIEW structure. HTML.EXE assumes that this contains the file name of an HTML document. *LocalAlloc* is called to allocate enough space in the buffer *pszFilename* to hold the file name.

Then all that is left to do is open, read, and display the contents of the file. This is exactly what the *LinkToFile* function does, so the NM_HOTSPOT notification handler simply needs to call this function. The HTML viewer control responds by displaying the new HTML file.

An important step of the process was skipped, however. The purpose of these two lines of code is not obvious:

```
pszFilename = (TCHAR*)LocalAlloc(LPTR,
  sizeof(TCHAR)*nLen);
AnsiToWide(lpnm->szTarget, pszFilename);
```

The HTML files read by HTML.EXE are assumed to be ANSI text. But the file system API functions called by *LinkToFile* require Unicode file names. Therefore the ANSI string contained by *lpnm->szTarget* (the link file name) must be converted to Unicode.

Each Unicode character requires two bytes (the size of a TCHAR), and there are *nLen* characters in the file name string. So *LocalAlloc* allocates enough bytes to accommodate the Unicode representation of the string *lpnm->szTaget*.

AnsiToWide is an application-defined function that converts ANSI strings to Unicode:

```
void AnsiToWide(LPSTR lpStr, TCHAR* lpTChar)
{
```

```
    while(*lpTChar++ = *lpStr++);
}
```

This is a variation of the classic one-line string copy function.[1] As long as the two string arguments are NULL-terminated, this function copies the ANSI string in *lpStr* to the Unicode string *lpTChar*. Since the ++ operator increments a pointer variable by the size of the type it points to, incrementing *lpTChar* leaves two bytes per character.

Displaying Inline Images

The HTML viewer control provides limited support to displaying images referenced in HTML formatted text. The control sends a notification to its parent window when it encounters an IMG reference in HTML text, but makes no attempt to render the image. As with moving among hyperlinks, this is the responsibility of the application.

An example of including an inline image in an HTML file is:

```
<IMG SRC="image.gif">
```

When an HTML viewer control encounters such a tag, for example while responding to a DTM_ADDTEXT message, it alerts its parent that an image needs to be loaded by sending an NM_INLINE_IMAGE notification.

As with the NM_HOTSPOT notification, the control sends an NM_HTMLVIEW structure in the *lParam* of the WM_NOTIFY message.

The *szTarget* member of this structure contains the text following the SRC parameter in the HTML text. In the example above, *szTarget* would contain the string "image.gif."

In the case of NM_INLINE_IMAGE notifications, the *szData* member of the NM_HTMLVIEW structure is not used. But the *dwCookie* value is. It contains a value that must be passed to the DTM_SETIMAGE message described a little later.

The basic idea to take away from this discussion is that the HTML viewer control only notifies its parent window that a reference to an image has been detected. It is up to the application to load and display

[1]For a more complete explanation of how this function works, see B. Stroustrup, *The C++ Programming Language*, 2nd Ed. (Addison-Wesley, 1991), pp. 92-93.

the image. But the application gets a little help in this from the DTM_SETIMAGE message.

The DTM_SETIMAGE Message

An application tells an HTML viewer control how to display an inline image by sending the control a DTM_SETIMAGE message. This message associates an inline image with a bitmap sent with the message. In other words, after responding to this message, the HTML viewer control displays the specified bitmap in place of the inline image. The parameters for DTM_SETIMAGE are described in Table 12.2.

The application must specify the various attributes of the bitmap to be used for the inline image. This information is specified in an INLINEIMAGEINFO structure:

```
typedef struct tagINLINEIMAGEINFO
{
  DWORD dwCookie;
  int iOrigHeight;
  int iOrigWidth;
  HBITMAP hbm;
  BOOL bOwnBitmap;
} INLINEIMAGEINFO, *LPINLINEIMAGEINFO;
```

The *dwCookie* member of this structure is the same value sent by the control in its NM_INLINE_IMAGE notification. *iOrigHeight* and *iOrigWidth* specify the height and width of the bitmap. The bitmap itself is contained in the *hbm* member.

Finally, *bOwnBitmap* specifies who is responsible for destroying the bitmap resource once it has been displayed. If this member is TRUE, the HTML viewer control must free the resource. If *bOwnBitmap* is FALSE, the application is telling the control that the application will handle destroying the bitmap.

Related to the DTM_SETIMAGE message is DTM_IMAGEFAIL. An application sends this message to an HTML viewer control to indicate

Table 12.2 The DTM_SETIMAGE Message

PARAMETER	MEANING
wParam	Not used.
(LPINLINEIMAGEINFO)lParam	Pointer to an INLINEIMAGEINFO structure that defines the bitmap to use.

that the image specified in an NM_INLINE_IMAGE notification could not be loaded. The control responds by displaying the default "broken image" bitmap for that inline image.

All of this means, of course, that an application that wishes to properly display inline images in an HTML viewer control must have a way of converting the image data in a particular image file into a Windows CE bitmap resource. The NM_INLINE_IMAGE notification tells the application that an inline image has been detected. It does not convert the referenced image file into the required Windows CE bitmap resource.

HTML Viewer Control Messages and Notifications

The HTML viewer control supports many more messages and notifications than those we have detailed. The examples in this chapter should provide enough insight into the use of the HTML viewer to make using the rest of the control features easy.

A complete list of messages and notifications associated with the HTML viewer control are given in Tables 12.3 and 12.4.

Table 12.3 HTML Viewer Control Messages

MESSAGE	BEHAVIOR
DTM_ADDTEXT	Adds the specified ANSI text to the control. The *wParam* indicates if the text is plain or HTML formatted text.
DTM_ADDTEXTW	Unicode version of DTM_ADDTEXT.
DTM_ANCHOR	Tells the control to jump to the specified anchor.
DTM_ANCHORW	Unicode version of DTM_ANCHOR.
DTM_ENABLESHRINK	Toggles the control image shrink mode. The control shrinks images to make the HTML document fit the control window.
DTM_ENDOFSOURCE	Tells the control that the application is done adding text to the control.
DTM_IMAGEFAIL	Used to inform the control that the specified image could not be loaded.
DTM_SETIMAGE	Associates the specified bitmap with an inline image.
DTM_SELECTALL	Selects (highlights) all text displayed in the control.

Table 12.4 HTML Viewer Control Notifications

NOTIFICATION	MEANING
NM_BASE	Sent by the control when it encounters a BASE tag in HTML text.
NM_CONTEXTMENU	Sent by the control when the user taps the client area of the control while pressing the Alt key.
NM_HOTSPOT	Sent by the control when a user taps a hyperlink or submits a form.
NM_INLINE_IMAGE	Sent by the control to tell the application that an image needs to be loaded.
NM_INLINE_SOUND	Sent by the control to tell the application that a sound file needs to be loaded.
NM_META	Sent by the control when it encounters a META tag in HTML text. Notification includes the HTTP-EQUIV and CONTENT parameters of this tag.
NM_TITLE	Sent by the control when it encounters a TITLE tag in HTML text. Notification includes the document title.

CHAPTER 13

Palm-size PC Input Techniques

Traditional computing devices such as personal computers get a large amount of their user input from a keyboard. Composing e-mail, writing documents in a word processor, or even simply entering a password typically requires keyboard input.

But Windows CE-based devices are not required to have a keyboard. For example, users of Palm-size PCs are very comfortable using these devices without a keyboard. Despite the presence of a software keyboard that can be invoked at any time, the majority of user input gets to a Palm-size PC via the stylus, navigation buttons, or even voice.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

Use the rich ink control

Program Palm-size PC navigation buttons

Use the voice recorder control

This chapter also introduces the Palm-size PC emulation environment for the first time. If you have installed the Palm-size PC SDK, you are