Writing DLLs for Windows Help

A dynamic-link library (DLL) is an executable module containing functions that Windows-based applications (like Windows Help) can call to perform useful tasks. Windows Help accesses DLLs in two ways:xe "DLLs:Help"§

- Through DLL functions registered as Help macros in a Help Project file. These functions can then be used in hot spots and macro footnotes in topic files.
- Through embedded window (**ew**) references in topic files.

If Windows Help's internal macro set (described in Chapter 15, "Help Macro Reference") doesn't provide all the functionality you need for a Help file, you can write your own DLLs to add extensions to Windows Help. In this chapter, you'll learn two ways to extend Windows Help: by providing custom DLLs containing author-defined Windows Help macros and by providing DLL access through embedded-window references.xe "DLLs:creating custom"§

When creating DLLs for embedded windows, you must follow all the standard design requirements for Windows DLLs. This section assumes that you're familiar with these requirements, which are described in Chapter 20, "Dynamic-Link Libraries," of the Microsoft Windows version 3.1 Software Development Kit (SDK), *Guide to Programming manual*..

To create DLLs for Windows Help, you need the Microsoft Windows Version 3.1 SDK and the Microsoft C Optimizing Compiler, version 6.0 or greater, or Microsoft Quick C for Windows version 1.0.xe "Software:to create DLLs"§

Included with this Authoring Guide is the source code for a sample DLL and a MAKEFILE for building the sample DLL. The source code is contained in the files DLLDEMO.C (for the Help macro DLL) and EWDEMO.C (for the embedded window DLL). You can use this sample code as a template for creating your own DLL. Be sure you study these source files and understand how they work before you continue in this chapter, since the sample DLL illustrates the concepts presented in this chapter and the remainder of this chapter assumes you are familiar with these files.

Creating Custom Help Macros

As explained in Chapter 14, “Help Macros,” you can use the **RegisterRoutine** macro to register any DLL function as a Windows Help macro. You might want to create a DLL whose sole purpose is to provide new Windows Help macros for your Help authors. This section will show you how to do that.xe "**RegisterRoutine** Windows Help macro:DLL functions"§xe "DLLs:registering as Windows Help macro"§

Examples in this section use the sample source code in the DLLDEMO.C file. DLLDEMO.C contains a **HelloWorld** function, which plays a beep on the speaker.xe "DLLs:Windows Help-command sample source code"§xe "Sample application:DLLDEMO sample DLL"§xe "DLLs:embedded-window sample source code"§

The examples also refer to a small Windows Help file called DLLDEMO.HLP, along with the DLLDEMO.RTF source file and DLLDEMO.HPJ Help project File used to build it. **HelloWorld** is registered as a Help macro in DLLDEMO.HPJ, and the following hot spot in the DLLDEMO.HLP Help file executes **HelloWorld** to play a beep.

Call HelloWorld.**!HelloWorld(hwndapp,qchPath)**

Note

To debug your DLL or observe any of the operations described in this section, you can compile DLLDEMO for the CodeView for Windows symbolic debugger and load the DLL into CodeView while you view the sample title. See the DLLDEMO.C file for specific instructions.xe "DLLs:creating custom"§

Registering DLL Functions as Help Macros

When registering a DLL function, you provide Help the following information:

- DLL filename
- Function name
- Data type returned by the function
- Number and type of function parameters

To register the DLL function, you enter a **RegisterRoutine** macro in the [CONFIG] section of the Help project file. (If you don't register the DLL function in the [CONFIG] section, you must register it another way before using the function.) The **RegisterRoutine** macros are executed when the Help file is opened, so the registered functions are available during the entire Help session. You must register a DLL routine before using it, or the Help compiler will report an error when it encounters the unregistered macro in the RTF source files, and the macro will not work when executed in the built Help file.

The **RegisterRoutine** macro has the following syntax:

RegisterRoutine("*DLL-name*"**,** "*function-name*"**,** "*parameter-spec*"**)**

Parameter	Description
DLL-name	String specifying the name of the DLL in which the function resides. The filename must be enclosed in quotation marks. You can omit the .DLL filename extension. Specify the directory only if necessary. Generally, DLLs are installed in the directory where Windows Help resides. See the next section, "How Help Locates .DLLs," for more information.
function-name	String specifying the name of the function to use as a Help macro. The function name must be enclosed in quotation marks.
parameter-spec	String specifying the formats of parameters passed to the function. Characters in the string represent C parameter types. Valid parameter types include the following:

Character	Data Type	Equivalent Windows Data Type
u	Unsigned short integer	UINT, WORD, WPARAM
U	Unsigned long integer	DWORD, RGBQUAD
i	Signed short integer	BOOL (also C int or short)
I	Signed long integer	LONG, LPARAM, LRESULT
S	Far pointer to a null-terminated text string	LPSTR, LPCSTR
v	Void (means no type; used only with return values)	None. Equivalent to C void data type.

The parameter-spec must be enclosed in quotation marks. Windows Help checks the format string to ensure that it matches the function prototype defined in the DLL. (For more information about the **RegisterRoutine** macro, see Chapter 15, "Help Macro Reference.")

To determine the data type of the function's parameters, consult the application programming interface (API) documenation for the DLL, or ask the person who developed the DLL. For information on parameter types of Windows functions, see the Windows version 3.1 SDK.

The **HelloWorld** function is defined as follows in the DLLDEMO.C file:

```
PUBLIC BOOL PASCAL EXPORT HelloWorld( LONG hwndContext, LPSTR lszHlpFile)
{
   MessageBeep(0);
}
```


HelloWorld takes, as parameters, two internal Windows Help variables: a handle to the Windows Help context window and a pointer to the name of the .HLP file that Windows Help has opened. Although **HelloWorld** doesn't do anything with these parameters, they illustrate the types of internal Windows Help variables you might pass to a DLL. For a complete list of these variables, see the "Windows Help Internal Variables section," later in this chapter.xe "Internal Windows Help variables"§

HelloWorld is registered in the DLLDEMO.HPJ Help project file as follows:

```
[CONFIG]
RegisterRoutine("dlldemo", "HelloWorld", "US")
```

As described above, the first parameter to **RegisterRoutine** is the DLL name (DLLDEMO); the second is the name of the function being registered; and the third is a format string representing the types of parameters in the function. Windows Help compares the parameter-spec with the function's parameter types and issues an error if they don't match.

The format string for the parameters to **HelloWorld** are **U** (unsigned long) for the LONG parameter and **S** for the LPSTR parameter. All internal variables that are handles should be declared in **RegisterRoutine** with a **U** specifier, even though they are normally WORDs, for upward compatibility with future versions of the Windows graphical environment. All internal string variables should be declared with an **S** specifier.xe "**RegisterRoutine** Windows Help macro:DLL functions"§xe "DLLs:registering as Windows Help macro"§

How Help Locates DLLs

When executing custom DLLs using the **RegisterRoutine** macro, Windows Help loads the DLL only when it is needed by the Help file. To load a DLL , Help must be able to find it on the user's system. When preparing to use a .DLL, Help looks in the following locations, in the following order:

- Help's current directory
- The MS-DOS current directory
- The user's Windows directory
- The Windows SYSTEM directory
- The directory containing WINHELP.EXE

- The directories listed in the user's PATH environment variable
- The directories specified in WINHELP.INI

If Help cannot find the DLL after searching in all these locations, it displays an error message.

To increase the likelihood that Help will locate the DLL quickly, you should also observe the following guidelines:

- Use unique names for all DLLs accessed by the Help file.
- When installing your application on a user's hard disk drive, your setup program should copy all custom DLLs to the directory where Help is located.
- If your product is distributed on CD-ROM, copy WINHELP.EXE and any custom DLLs to the user's hard disk drive.
- Define a WINHELP.INI entry for each custom DLL that your Help file is using so that Help knows where to locate them.

For an explanation of the WINHELP.INI file, see the "Creating Links Between Help Files" section in Chapter 8, "Creating Links and Hot Spots."

Windows Help Internal Variables

In the hot-spot example earlier in this section, the **HelloWorld** function used the internal Windows Help variables **hwndContext** (a handle to the Windows Help context window) and **qchPath** (a pointer to the Help filename). In general, after you register a DLL function as a Help macro, you can specify Windows Help internal variables as parameters to that function when the function appears in hot spots or macro footnotes.xe "DLLs:internal Windows Help variables"§

In this section, you'll learn about other Windows Help internal variables that you can pass to DLL functions registered as Windows Help macros. This section also examines in detail one variable that controls how the DLL handles Windows Help errors.

List of Variables

You can use any of the following Windows Help internal variables in DLL functions.

Variable	Format Spec	Description
hwndApp	U	32-bit handle to the main Help window. This variable is guaranteed to be valid only while the function is executing.
hwndContext	U	Handle to the current active window (either the main Help window or a secondary window).
qchPath	S	Fully qualified path of the currently open Help (.HLP) file.
qError	S	Long pointer to a structure containing information about the most recent Windows Help error (described later in this section).
lTopicNo	U	Topic number. This number is relative to the order of topics in the RTF files used to build the Help file.
hfs	U	Handle to the file system for the currently open Help (.HLP) file.
coForeground	U	Current foreground color.
coBackground	U	Current background color.xe "Internal Windows Help variables"§

Error Handling

The **qError** internal variable points to a structure containing information about the most recent Windows Help error. The error structure is defined as follows:xe "Error handling:DLL errors"§xe "DLLs:error handling"§

```
struct
{   WORD  fwFlags;
    WORD  wError;
    char  rgchError[wMACRO_ERROR];
} QME;
```

fwFlags

The **fwFlags** field contains flags indicating how Windows Help responds to errors. The following are possible error flags and their values.

Flag	Value	Description
fwMERR_ABORT	0x0001	Allows the Abort option. This flag is set by default.
fwMERR_CONTINUE	0x0002	Allows the Continue option.
fwMERR_RETRY	0x0004	Allows the Retry option.

wError

The **wError** field is a number indicating the type of error that occurred. The following are possible error numbers and their values.

Error	Number	Description
wMERR_NONE	0	No error (initial value)

wMERR_MEMORY	1	Out of memory (local)
wMERR_PARAM	2	Invalid parameter passed
wMERR_FILE	3	Invalid file parameter
wMERR_ERROR	4	General Help macro error
wMERR_MESSAGE	5	Help macro error with message

rgchError

If the **wError** field is **wMERR_MESSAGE**, the **rgchError** field contains the error message that Windows Help displays.xe "DLLs:internal Windows Help variables"§xe "DLLs:error handling"§xe "Error handling:DLL errors"§

Notifying DLLs of Windows Help Events

You might want your DLL to receive notification of Windows Help events (for example, when the user selects a jump or changes input focus to an application other than Windows Help). To do this, you add a **LDLLHandler** function to the DLL. The **LDLLHandler** function has the job of processing messages sent from

Windows Help to the DLL.xe "DLLs:Windows Help event notification"§xe "**LDLLHandler** function:Windows Help event notification"§

When the user clicks the Call HelloWorld hot spot in DLLDEMO.HLP, Windows Help loads DLLDEMO.DLL and looks for a function named **LDLLHandler**. If Windows Help finds **LDLLHandler**, it calls **LDLLHandler**. A sample **LDLLHandler** function is defined in the DLLDEMO.C file.

Note

To debug your DLL or observe any of the operations described in this section, you can compile DLLDEMO for the CodeView for Windows symbolic debugger and open the DLL in CodeView while you view the sample title. See the DLLDEMO.C file for specific instructions.

The **LDLLHandler** function is defined in DLLDEMO.C as illustrated in the following example:

```
PUBLIC    LONG PASCAL EXPORT LDLLHandler(
   WORD    wMsg,
   LONG    lParam1,
   LONG    lParam2)
   {
      switch(wMsg) {
         case DW_WHATMSG:
            return DC_INITTERM | DC_JUMP;
         case DW_INIT:
            return TRUE;
         case DW_TERM:
            return TRUE;
         case DW_ACTIVATE:
            return TRUE;
         case DW_CHGFILE:
            return TRUE;
   }
   return FALSE;
}
```

In this definition, **wMsg** identifies the message Windows Help has sent to the DLL. The two LONG parameters are passed with the message and depend on the message type.

When the user first clicks the Call HelloWorld hot spot, Windows Help calls **LDLLHandler** with the **wMsg** parameter set to **DW_WHATMSG**. This message asks the DLL what types of Windows Help messages it wants to receive. **LDLLHandler** should return one or more of the flags in the following table with this information. Windows Help uses the flags returned by **LDLLHandler** to determine which messages the DLL wants to receive in the **LDLLHandler**

function.

LDLLHandler returns	Windows Help sends	Description
DC_MINMAX	**DW_MINMAX**, **DW_SIZE**	Minimize, maximize, or resize Windows Help
DC_INITTERM	**DW_INIT**, **DW_TERM**	Initialize or terminate DLL
DC_JUMP	**DW_STARTJUMP**, **DW_ENDJUMP**, **DW_CHGFILE**	Jump or change .HLP file
DC_ACTIVATE	**DW_ACTIVATE**	Give Windows Help or another application input focus
DC_CALLBACKS	**DW_CALLBACKS**	Give DLL access to Windows Help entry points (see Calling Windows Help Internal Functions, later in this chapter, for more information)

You combine these flags using the standard C bitwise-OR (|) operator. For example, many DLLs request notification when Windows Help is about to initialize or terminate them. The sample **LDLLHandler** function requests this notification and asks to be notified when the user has moved the input focus to or from Windows Help. It processes the **DW_WHATMSG** message as follows:

```
case DW_WHATMSG:
   return DC_INITTERM | DC_ACTIVATE;
```

The remainder of this section looks at the different Windows Help messages that **LDLLHandler** processes.

DC_INITTERM Flag

The **DC_INITTERM** flag tells Windows Help that the DLL should receive initialization and termination messages.

DW_INIT

After **LDLLHandler** returns a **DC_INITTERM** flag, Windows Help sends a DW_INIT message. This message tells **LDLLHandler** to perform any required initialization operations (such as initializing variables or loading strings). The *lParam1* and *lParam2* parameters are not used.

If **LDLLHandler** returns FALSE, Windows Help unloads the DLL and stops sending messages. If **LDLLHandler** returns TRUE, the DLL remains loaded.

In the DLLDEMO sample, **LDLLHandler** returns TRUE for this message.

DW_TERM

When the user ends a Windows Help session, Windows Help sends a DW_TERM message. This message tells **LDLLHandler** to perform any required cleanup operations before the DLL is unloaded from memory. The *lParam1* and *lParam2* parameters are not used, and the return value has no meaning.

DC_MINMAX Flag

The **DC_MINMAX** flag tells Windows Help that the DLL should receive messages when the user minimizes, maximizes, or resizes the Windows Help window.

DW_MINMAX

Windows Help sends the DLL a **DW_MINMAX** message if the user minimizes or maximizes the context window.

The *lParam1* parameter to **DW_MINMAX** indicates which operation was performed: 1L for minimized or 2L for maximized.

The *lParam2* parameter is not used.

If **LDLLHandler** returns TRUE for this message, Windows Help continues sending this message throughout the remainder of the session; if **LDLLHandler**

returns FALSE, Windows Help stops sending it.

DW_SIZE

Windows Help sends the DLL a DW_SIZE message if the user resizes the context window.

The *lParam1* message specifies the horizontal size of the window in the low-order word and the vertical size in the high-order word.

The *lParam2* parameter is not used.

If **LDLLHandler** returns TRUE for this message, Windows Help continues sending this message throughout the remainder of the session; if **LDLLHandler** returns FALSE, Windows Help stops sending it.

DC_JUMP Flag

The **DC_JUMP** flag tells Windows Help that the DLL should receive messages when the user executes a jump in the Help file.

DW_STARTJUMP

Windows Help sends the DLL a DW_STARTJUMP message after the user executes a jump. The parameters to **DW_STARTJUMP** are not used.

If **LDLLHandler** returns TRUE for this message, Windows Help continues sending this message throughout the remainder of the session; if **LDLLHandler** returns FALSE, Windows Help stops sending it.

DW_ENDJUMP

Windows Help sends the DLL a DW_ENDJUMP message after it displays the jump-destination topic.

The *lParam1* parameter specifies the byte offset of that topic within the Help file's file system. This value can be passed to the **LSeekHf** function (defined in the DLL.H file) to perform a seek to that topic within the file.

The *lParam2* parameter specifies the position of the scroll box in the topic. This value can be used in any of the standard Windows scrolling functions that accept scroll-position parameters.

If **LDLLHandler** returns TRUE for this message, Windows Help continues sending this message throughout the remainder of the session; if **LDLLHandler** returns FALSE, Windows Help stops sending it.

DW_CHGFILE

Windows Help sends the DLL a **DW_CHGFILE** message if the user selects Open from the File menu to change .HLP files or jumps to a topic in a different .HLP file. If the message is the result of a jump to a new file, Windows Help sends the **DW_CHGFILE** message between the DW_STARTJUMP and **DW_ENDJUMP** messages.

The *lParam1* parameter specifies a long pointer to the .HLP filename. The *lParam2* parameter is not used.

If **LDLLHandler** returns TRUE for this message, Windows Help continues sending this message throughout the remainder of the session; if **LDLLHandler** returns FALSE, Windows Help stops sending it.

DC_ACTIVATE Flag

The **DC_ACTIVATE** flag tells Windows Help that the DLL should receive a DW_ACTIVATE message when the user gives either Windows Help or another application the input focus.

DW_ACTIVATE

The *lParam1* parameter to the DW_ACTIVATE message specifies whether Windows Help received (nonzero LONG value) or lost (0L) the input focus. The *lParam2* parameter is not used.

If **LDLLHandler** returns TRUE for this message, Windows Help continues sending this message throughout the remainder of the session; if **LDLLHandler** returns FALSE, Windows Help stops sending it. xe "DLLs:Windows Help event notification"§xe "**LDLLHandler** function:Windows Help event notification"§

Calling Windows Help Internal Functions

Windows Help provides DLL authors with access to 16 of its internal functions. These functions perform operations in the internal .HLP file system, get global information about the currently open Help file, or display information in a standard Windows Help dialog box. They are documented in the DLL.H header file.xe "DLL.H header file"§xe "DLLs:calling internal Windows Help functions"§

In this section, you'll learn how to access six of these internal functions. An overview of the mechanism is as follows:

- Windows Help sends a DW_WHATMSG message to the DLL.
- The **LDLLHandler** function in the DLL processes the DW_WHATMSG message by returning a **DC_CALLBACKS** flag.
- Windows Help sends a **DW_CALLBACKS** message specifying a pointer to an array of Windows Help internal functions.
- The **LDLLHandler** function uses the pointer passed in the DW_CALLBACKS message to obtain pointers to the Windows Help functions it will use.

The **ExportBag** function in the DLLDEMO.C sample file is used for the examples in this section. **ExportBag** copies a file from the .HPJ internal file system to an
MS-DOS file. (The file from the .HPJ file system is initially stored using an entry in the [BAGGAGE] section of the DLLDEMO.HPJ project file.) You can use the mechanism illustrated in this example to access files stored in any .HLP file system.

Sample Code

ExportBag is executed from a two topic entry macros in the sample DLLDEMO.HLP Help file. Like the **HelloWorld** function earlier in this chapter, **ExportBag** is registered as a Help macro in the DLLDEMO.HPJ Help project file as follows:

```
RegisterRoutine("dlldemo","ExportBag","SSSS")
```

The two entry macros in DLLDEMO.RTF that execute **ExportBag** are coded as follows:

```
! ExportBag(qchPath, "dlldemo.hpj", "decomp.hpj", qError )
! ExportBag(qchPath, "dlldemo.rtf", "decomp.rtf", qError )
```


Note

To debug your DLL or observe any of the operations described in this section, you can compile DLLDEMO for the CodeView for Windows symbolic debugger and open the DLL in CodeView while you view the sample title. See the DLLDEMO.C file for specific instructions.

Accessing Windows Help Functions (DW_CALLBACKS)

If a DLL wants access to Windows Help internal functions, its **LDLLHandler** function handles the DW_WHATMSG message by returning a **DC_CALLBACKS** flag. In DLLDEMO.C, the **LDLLHandler** function requests access as follows:xe "**LDLLHandler** function:accessing internal Windows Help function"§

```
case DW_WHATMSG:
      return DC_INITTERM | DC_CALLBACKS;
```

When it gets this flag, Windows Help sends a DW_CALLBACKS message to the DLL. The *lParam1* parameter is a long pointer to an array containing pointers to each of the 16 internal Windows Help functions. The DLL.H file defines symbolic names for indexing each function in the array.

In processing the DW_CALLBACKS message, the **LDLLHandler** function calls a function named **GetCallBacks** to specify which functions it wants to access. **GetCallBacks** is defined as follows in DLLDEMO:

```
PUBLIC    BOOL PASCAL EXPORT GetCallBacks(
VPTR    VPtr,
LONG    lVersion)
{

   // hfs level:
   lpfn_HfsOpenSz      = VPtr[HE_HfsOpenSz];
   lpfn_RcCloseHfs     = VPtr[HE_RcCloseHfs];
   lpfn_RcLLInfoFromHfs = VPtr[HE_RcLLInfoFromHfs];

   // bag level routines
   lpfn_FAccessHfs =  VPtr[HE_FAccessHfs];
   lpfn_HfOpenHfs  =  VPtr[HE_HfOpenHfs];
   lpfn_LcbReadHf  =  (LPFN_LCBREADHF) VPtr[HE_LcbReadHf];
   lpfn_RcCloseHf  =  VPtr[HE_RcCloseHf];
   return TRUE;
```

}

The **VPtr** parameter in **GetCallBacks** is the *lParam1* pointer passed by the DW_CALLBACKS message from Windows Help. The **GetCallBacks** function gets pointers to the following Windows Help internal functions, which it uses later in the example.

Function	**What it does**
HfsOpenSz	Opens the .HLP file system (the compound file containing the files internal to the Help file)
RcCloseHfs	Closes the open .HLP file system
RcLLInfoFromHfs	Maps a handle to the open .HLP file system (returned by **HfsOpenSz**) to low-level file information
FAccessHfs	Determines whether a file within the .HLP file system is accessible
HfOpenHfs	Opens a file within the .HLP file system
LcbReadHf	Reads bytes from a file within the .HLP file systemxe "**LDLLHandler** function:accessing internal Windows Help function"§

Note

This sample code does not take multiple instances of Windows Help into account. If more than one instance is started, the DLL receives different pointers to these callback functions—one pointer for each instance. You must keep track of which pointer is associated with each Windows Help instance. One way to do this is to create an array associating the task with the callback pointer.

Copying Files from the .HLP File System

Now that DLLDEMO has access to the Windows Help functions it needs, it is ready to copy a file from the .HLP file system. The **ExportBag** function performs this operation. **ExportBag** uses the following variables:

```
PUBLIC          BOOL PASCAL EXPORT ExportBag(
LPSTR           lszHLPname,
LPSTR   lszBagFName,
LPSTR   lszExportName,
QME     qError)
{
        HANDLE      hfsHlp;             // handle to .HLP file
        HANDLE      hfBag;              // file handle to bag file
        HANDLE      hFile;              // handle to output file
        BOOL    fClean = TRUE;          // if set, Delete file in Error state Machine.
        DWORD  dwBytesRead;             // input bytes read
        HANDLE      hMem;               // handle to copy buffer
        LPBYTE      lpMem;              // ptr to copy buffer
        OFSTRUCT ofs;
```

Getting a Handle to the .HLP File System

First, **ExportBag** uses the **HfsOpenSz** function to get a handle to the .HLP file system, as follows:

```
if ((hfsHlp = (*lpfn_HfsOpenSz)(lszHLPname, fFSOpenReadOnly)) == NULL)
   {
   qError->fwFlags = fwMERR_ABORT;
   qError->wError = wMERR_PARAM;
   goto ExitBag;
   }
```

The **lszHLPname** parameter identifies the .HLP file for which it wants the handle. This parameter takes its value from Windows Help's **qchPath** internal variable.

As you'll recall, **ExportBag** is executed from two entry macros in the

DLLDEMO.HLP Help file. The DLLDEMO.HPJ project file for that Help file includes a **RegisterRoutine** macro that registers **ExportBag** as a Help authoring macro. When **ExportBag** is run from the macros, **qchPath** is given as the first parameter to **ExportBag**.

Verifying the .HLP File

Next, **ExportBag** makes sure the baggage file within the .HPJ file system is valid, as follows:

```
if (((*lpfn_FAccessHfs)(hfsHlp, lszBagFName, NULL)) == FALSE)
   {
   qError->fwFlags = fwMERR_RETRY;
   qError->wError = wMERR_MESSAGE;
   lstrcpy(qError->rgchError, "Could not open baggage file `");
   lstrcat(qError->rgchError, lszBagFName);
   lstrcat(qError->rgchError, "'.");
   goto ExitBag;
   }
```

The **lszBagFName** parameter to **ExportBag** gives this filename. Like the **lszHLPname** parameter, the baggage filename in **lszBagFName** is passed from the entry macros encoded in the DLLDEMO.HLP file. However, the baggage filename is hard-coded in the macro rather than obtained from a Windows Help internal variable.

In this and other examples, the return code **rc** would normally be serviced in a common error routine at label **UNDOstateEXIT**. This error routine would inform the user of the problem. For simplicity, this error routine has been omitted from DLLDEMO.C.

Copying the Baggage File

ExportBag now opens the baggage file for reading, as folows:

```
if ((hfBag = (*lpfn_HfOpenHfs)(hfsHlp, lszBagFName, fFSOpenReadOnly))
   == NULL)
   {
   qError->fwFlags = fwMERR_ABORT;
   qError->wError = wMERR_ERROR;
   goto ExitBag;
   }
```

It creates a buffer to improve efficiency in the copy:

```
if ((hMem = GlobalAlloc(GMEM_MOVEABLE,
      (DWORD)wCOPY_SIZE)) == NULL)
   {
   qError->fwFlags = fwMERR_ABORT;
```

```
   qError->wError = wMERR_MEMORY;
   goto ExitBag;
   }
lpMem = GlobalLock(hMem);
```

To make sure the MS-DOS file it's copying to is not already present, **ExportBag** calls the Windows API **OpenFile**. This function executes an MS-DOS function to purge the filename **ExportBag** is using:

```
OpenFile(lszExportName, &ofs, OF_DELETE);
```

ExportBag creates the MS-DOS file it's copying to and obtains a handle to the new file, as follows:

```
if ((hFile = _lcreat(lszExportName,0)) == -1)
   {
   qError->fwFlags = fwMERR_ABORT;
   qError->wError = wMERR_ERROR;
   goto ExitBag;
   }
```

Now **ExportBag** starts the write loop that copies the baggage file to the new MS-DOS file, as follows:

```
do  {
   if ((dwBytesRead =
      (*lpfn_LcbReadHf)(hfBag, lpMem, wCOPY_SIZE)) == -1L)
      {
      qError->fwFlags = fwMERR_ABORT;
      qError->wError = wMERR_ERROR;
      goto ExitBag;
      }
   if (_lwrite(hFile, lpMem,(WORD) dwBytesRead) != (WORD) dwBytesRead)
      {
      qError->fwFlags = fwMERR_ABORT | fwMERR_RETRY;
      qError->wError = wMERR_MESSAGE;
      lstrcpy(qError->rgchError, "Out of disk space.");
      goto ExitBag;
      }
   } while (dwBytesRead == wCOPY_SIZE);
```

ExportBag reads the size of the buffer it created until it reaches the end of the baggage file. The end-of-file condition is indicated when the **LcbReadhf** function returns –1L. **ExportBag** writes to the new MS-DOS file using the standard Windows **_lwrite** function.

When **ExportBag** has finished writing the MS-DOS file, it closes the baggage and MS-DOS files and cleans up its memory, as follows:

```
ExitBag:
      if (hfBag != NULL)
   (*lpfn_RcCloseHf)(hfBag);
```

```
if (hfsHlp != NULL)
   (*lpfn_RcCloseHfs)(hfsHlp);
if (hMem != NULL)
   {
   GlobalUnlock(hMem);
   GlobalFree(hMem);
   }
if (hFile != -1)
   _lclose(hFile);
if (qError->wError != wMERR_NONE && fClean)
```


Writing DLLs for Embedded Windows

```
   OpenFile(lszExportName, &ofs, OF_DELETE);
```

In Windows Help version 3.1 you can create embedded windows to extend the functionality of Windows Help by placing objects in a fixed-size window under the control of a DLL. For example, you can create a DLL to display animation sequences or to play audio segments in an embedded window. However, in version 3.1 you cannot create an embedded window that functions as a hot spot. This section explains how to write custom DLLs for Windows Help.xe "Embedded window:controlling with DLLs:creating window"§xe "DLLs:calling internal Windows Help functions"§

Creating Embedded Windows

To create an embedded window, authors insert an embedded window reference in the RTF source file. This reference has the following general form:xe "DLLs:embedded-window"§

{ew*x DLL-name, window-class, author-data***}**

Parameter	Description
x	A character specifying the alignment of the window: **l** for a left-aligned window, **r** for a right-aligned window, or **c** for a character-aligned window.
DLL-name	The name of the DLL that controls the embedded window. The file name should not include an extension or be fully qualified, but it can include a relative path. Windows Help assumes .DLL or .EXE to be the default extension.

window-class	The name of the embedded window class as defined in the source code for the DLL.
author-data	An arbitrary string, which Windows Help passes to the embedded window when it creates the window. This string can be one or more substrings separated by any punctuation mark except a comma. The DLL is responsible for parsing this string. The string is terminated by the closing brace (})xe "Embedded window:controlling with DLLs:creating window"§

The following example shows a valid embedded window reference:

{ewl FADE, AmfWnd, clipbrd.amf}

How Embedded Windows Work in Help

Windows Help displays embedded windows within topic windows. To Windows Help, an embedded window is simply a child window of that topic window. Users cannot minimize, maximize, or resize an embedded window. Embedded windows cannot be used as hot spots. However, you can include hot spots in an embedded window if they are controlled by the DLL, but users will not be able to use the keyboard equivalents to access the hot spots. That is because embedded windows cannot receive the input focus, which means they cannot process keystrokes. So, you should not place anything in an embedded window that requires keyboard input from users.

Windows Help positions the embedded window using the justification character (left, right, or character) specified by the author in the embedded window reference. The embedded window DLL is expected to display the information in the window and resize the window appropriately. Help expects the window size to remain fixed as long as the topic is displayed. The window element and DLL determine the size and content of the embedded window, and Help arranges the other elements of the topic around the embedded window.

Windows Help displays embedded windows only when necessary—while the topic containing the embedded window is being displayed. However, an embedded window may exist while it is not being displayed (if the user scrolls the topic past the embedded window, for example). Because it is part of a specific

topic, an embedded window goes away when the user displays a different topic.

Initialization

Windows Help creates embedded windows with window style WS_CHILD and a default size. It initializes the DLL for an embedded window before it displays the window.xe "Embedded window:controlling with DLLs:initializing DLL"§xe "DLLs:initializing for embedded window"§

When Windows Help creates the embedded window, it passes two strings in the WM_CREATE message. The *lparam* parameter points to a **CREATESTRUCT** structure, and the **lpCreateParams** field of the **CREATESTRUCT** structure points to a second structure defined as follows in the DLL.H header file:

```
typedef  struct tagCreateInfo
         {
         short          idMajVersion;
         short          idMinVersion;
         LPSTR          lpstrFileName;
         LPSTR          lpstrAuthorData;
         HANDLE             hfs;
         DWORD          coForeground;
         DWORD          coBackground;
         }   CREATEINFO;
```

The fields in the structure are defined in the following table.

Field	Use
idMajVersion, idMinVersion	Identifies the version of Windows Help (and the version of the **CREATESTRUCT** structure used for embedded windows in that version of Windows Help). Currently, these fields are 0. If the **idMinVersion** field is a value other than 0, additional interfaces may be available to DLLs, but all the interfaces described in this chapter are still supported. The DLL should always verify that the value in the **idMajVersion** and **idMinVersion** fields of the **CREATESTRUCT** structure are 0. If they are not, the DLL might not work with that version of Windows Help.
lpstrFileName	Points to the fully qualified name of the .HLP file containing the embedded window. This field will be NULL if the data is not available. If the DLL uses the string contained in this field, it should make a copy of the string.

lpstrAuthorData	Points to the *author-data* parameter in the **ewl**, **ewc**, or **ewr** reference from the source RTF file. This field will be NULL if the data is not available. If the DLL uses the string contained in this field, it should make a copy of the string.
hfs	Specifies a handle to the file system for the .HLP file.
CoForeground, CoBackground	Specify the foreground and background colors of the main Windows Help window. If the embedded window uses the same colors as the main Windows Help window, the DLL can use the color values in these fields to set those colors in the embedded window.

Embedded Window Behavior

While a topic is being displayed, Windows Help creates embedded windows when it needs to lay them out and destroys them when they are no longer needed. The window can be created and destroyed, then created again if necessary, while Windows Help tries to lay out the embedded window object. If you load information, you might want to do so during WM_SHOWWINDOW processing so that you don’t have to load this information twice.

An embedded window may exist while it is not being displayed (for example, if the topic requires scrolling and the embedded window resides in the portion not currently displayed in the Help window). Do not set the window style to WS_VISIBLE or call the **ShowWindow** function during WM_CREATE processing. Windows Help displays the window when necessary.

As an embedded window DLL processes a WM_CREATE message, it is expected to set up any window styles the window uses and resize the window appropriately.xe "Embedded window:controlling with DLLs:initializing"§

Windows Help positions the window using the justification character (left, right, or character) authored into the RTF file. Windows Help expects the window size to remain fixed during the window’s lifetime.

Because it is a child window of the main topic window, an embedded window does not need to destroy itself; it will be destoyed when the parent window is destroyed.xe "Embedded window:controlling with DLLs:destroying window"§xe

"Embedded window:controlling with DLLs:initializing DLL"§xe "DLLs:initializing for embedded window"§

Message Processing for Embedded Windows

Embedded windows receive most standard Windows messages, including mouse and WM_PAINT messages. Embedded windows should not take the input focus; therefore, they do not need to process keystrokes.

Embedded window DLLs must also process the following messages, which are specifically defined for use with embedded windows:xe "Embedded window:controlling with DLLs:message processing"§xe "DLLs:processing embedded window messages"§

- EWM_RENDER (0x706A)
- EWM_QUERYSIZE (0x706B)
- EWM_ASKPALETTE (0x706C)

EWM_RENDER

Windows Help sends the EWM_RENDER message to an embedded window to get information about the image in the window. It uses this information when printing the image or placing it in the Clipboard. The *wParam* parameter to EWM_RENDER indicates the type of information returned by the message, and the LOWORD of the return value contains a handle to this information. Possible *wParam* values and return values are shown in the following table.xe "Embedded window:controlling with DLLs:getting rendering information"§

wParam	**Return Value**
CF_TEXT	Sharable global handle to a null-terminated ASCII string. Windows Help frees this handle.
CF_BITMAP	Handle to a bitmap. Windows Help removes this handle.xe "EWM_RENDER window message:in embedded windows"§

EWM_QUERYSIZE

Windows Help sends the EWM_QUERYSIZE message to an embedded window to obtain the size of the window. The *wParam* parameter is a handle to the device context for the window. The *lParam* parameter points to a **POINT** structure. The embedded window DLL should fill in the **x** field of the structure with the window's height and the **y** field of the structure with the window's width. Windows Help uses this information when laying out the topic in the topic window, when printing the topic, or when placing it in the Clipboard.xe "EWM_QUERYSIZE window message:in embedded windows"§xe "Embedded window:controlling with DLLs:getting window size"§

EWM_ASKPALETTE

Windows Help sends the EWM_ASKPALETTE message to all embedded windows displayed within a topic to get information about the palettes used in each window. The *wParam* and *lParam* parameters are not used.xe "EWM_PALETTE window message"§xe "Palettes:querying information in DLL"§xe "Embedded window:controlling with DLLs:painting"§

When an embedded window receives an EWM_ASKPALETTE message, it should return a handle to the palette that it uses. Windows Help then selects the most appropriate palette for use in that topic. Usually, this palette is the one used for the first embedded window in the currently displayed part of the topic.

Before the DLL repaints an embedded window, it should send an EWM_ASKPALETTE message to the parent window to determine which palette to use. The following code fragment illustrates how to do this:

hpal = (HPALETTE)SendMessage(GetParent(hwnd), EWM_ASKPALETTE, 0, 0L)xe "EWM_PALETTE window message"§xe "Palettes:querying information in DLL"§xe "Embedded window:controlling with DLLs:message processing"§xe "DLLs:processing embedded window messages"§xe "Embedded window:controlling with DLLs:painting"§

Sample Embedded Window DLL

In this section, we'll analyze more source code in the sample DLL, and point out the functions and definitions required for embedded windows. The EWDEMO.C file contains the source code for a sample embedded window. The sample DLL displays a dynamic list of network printers in an embedded window. You can use this sample code as a template for creating your own embedded window DLLs. xe "Embedded window:controlling with DLLs:example of"§xe "DLLs:embedded window, example of"§

Functions

The DLL consists of four functions: **LibMain**, **WEP**, **PrinterListProc**, and **InitPrinterList**. **LibMain** is the standard DLL function that registers the window class, and **WEP** is the standard function that terminates the DLL. **PrinterListProc** performs the message processing, including processing for the embedded window messages **EWM_RENDER** and **EWM_QUERYSIZE**. It also calls **InitPrinterList** to initialize the printer list after it receives a WM_CREATE message from Windows Help.

Initialization (WM_CREATE)

The **PrinterListProc** function is responsible for creating an embedded window when the window first receives a WM_CREATE message. The type for the structure passed in the WM_CREATE message is defined in DLL.H, as shown below:

```
typedef struct
 {
    short   idMajVersion;
    short   idMinVersion;
    LPSTR   szFileName;
    LPSTR   szAuthorData;
    HANDLE  hfs;
    DWORD   coFore;
    DWORD   coBack;  }
 EWDATA, FAR * QEWDATA;
```

As noted earlier, the **szFileName** field points to the relative path of the .HLP file, and the **szAuthorData** field points to the *author-data* parameter in the **ewl**, **ewc**, or **ewr** command from the source RTF file. The local variable **qci** is defined using this type.

PrinterListProc first takes the information passed in the *lParam* parameter to **WM_CREATE** and copies it into the fields of the **qci** structure, as follows:

```
case WM_CREATE:
    qci = (QCI)((CREATESTRUCT FAR *)lParam)->lpCreateParams;

    /*-------------------------------------------------------------*\
    | Save the WM_CREATE information.
    \*-------------------------------------------------------------*/
    lstrcpy( rgchFileName, qci->szFileName );
    lstrcpy( rgchAuthorText, qci->szAuthorData );
```

Next, **PrinterListProc** adds a border to the embedded window and initializes the printer list, as follows:

```
/*-------------------------------------------------------------*\
    | Add a border to this window.
    \*-------------------------------------------------------------*/
```

```
    SetWindowLong( hwnd, GWL_STYLE,
            GetWindowLong( hwnd, GWL_STYLE ) | WS_BORDER );
    /*-------------------------------------------------------------*\
    | Initialize the printer list.  This could be updated more
    | dynamically, say on each WM_PAINT message.
    \*-------------------------------------------------------------*/
    InitPrinterList();

    return 0L;
```

Painting (WM_PAINT)

PrinterListProc is also responsible for painting the embedded window. When the window receives a WM_PAINT message, **PrinterListProc** lists the names of the printers in the list initialized by **InitPrinterList**, as follows:

```
case WM_PAINT:
    BeginPaint( hwnd, &ps );
    GetTextMetrics( ps.hdc, &tm );
    SetTextColor( ps.hdc, GetSysColor( COLOR_WINDOWTEXT ) );
    SetBkColor( ps.hdc, GetSysColor( COLOR_WINDOW ) );
    for (irgch = 0; irgch  MAX_PRINTERS && rgrgchPrinters[irgch][0];
        irgch++)
        TextOut( ps.hdc, tm.tmMaxCharWidth/2,
              (tm.tmHeight + tm.tmExternalLeading)/2 +
              irgch*(tm.tmHeight + tm.tmExternalLeading),
              rgrgchPrinters[irgch], lstrlen(rgrgchPrinters[irgch]) );
    EndPaint( hwnd, &ps );
    return 0L;
```

Obtaining Rendering Information (EWM_RENDER)

PrinterListProc also processes the EWM_RENDER message sent by Windows Help. The **wParam** parameter determines the type of rendering information that **PrinterListProc** returns. This parameter has the value CF_BITMAP or CF_TEXT.xe "EWM_RENDER window message:in example"§

CF_BITMAP

If **wParam** is CF_BITMAP, **PrinterListProc** creates a bitmap listing the system printers and returns a handle to this bitmap (**hbm**). **PrinterListProc** gets the information it needs to create the bitmap in two ways:

- Copying the values passed in the **lParam** parameter to the EWM_RENDER message.
- Sending an EWM_QUERYSIZE message to the embedded window and obtaining its size.

The **lParam** parameter points to a **RENDERINFO** structure. Type **RENDERINFO** is defined (in DLL.H), as follows:

```
typedef struct
 {
 RECT  rc;
 HDC   hdc;
 } RENDERINFO, FAR *QRI;
```

To create the bitmap, **PrinterListProc** copies the device context from **lParam**. It then obtains the size required for the bitmap by sending an EWM_QUERYSIZE message, as follows:

```
switch( wParam )
      {
      case CF_BITMAP:
        /*-------------------------------------------------------------*\
        | Prepare a bitmap image of the printer list.  This will
        | appear in a similar manner as the layout, but we will need
        | to draw the border explicitly here, as well as making sure
        | the background is filled in correctly.  We must use the
        | default colors for this hdc.
        \*-------------------------------------------------------------*/

qri = (QRI)lParam;
     hdc = CreateCompatibleDC( NULL );

     if (hdc)
     {
        hfont = 0;

        /*-------------------------------------------------------------*\
        | Create a monochrome bitmap for the DC, sized for the screen.
        \*-------------------------------------------------------------*/
        SendMessage( hwnd, EWM_QUERYSIZE, hdc, (long)(LPPOINT)&pt );
        rc.left = 0;
        rc.top = 0;
        rc.right = pt.x;
        rc.bottom = pt.y;
```

PrinterListProc creates the bitmap using the default foreground and background colors, but it draws the window border explicitly, as follows:

```
hbm = CreateCompatibleBitmap( qri->hdc, pt.x, pt.y );
        if (hbm)
        {
           hbmDefault = SelectObject( hdc, hbm );
           /*----------------------------------------------------------*\
           | Clear out the bitmap.
           \*----------------------------------------------------------*/
           hbrush = CreateSolidBrush( GetBkColor( hdc ) );
           if (hbrush)
           {
              FillRect( hdc, &rc, hbrush );
              DeleteObject( hbrush );
```

```
            }

            Rectangle( hdc, rc.left, rc.top, rc.right, rc.bottom );

            GetTextMetrics( hdc, &tm );
            for (irgch = 0; irgch  MAX_PRINTERS &&
               rgrgchPrinters[irgch][0]; irgch++)
            TextOut( hdc, tm.tmMaxCharWidth/2,
                  (tm.tmHeight + tm.tmExternalLeading)/2 +
                  irgch*(tm.tmHeight + tm.tmExternalLeading),
                  rgrgchPrinters[irgch],
                  lstrlen( rgrgchPrinters[irgch] ) );
         /*----------------------------------------------------------*\
         | At this point, hbm is the desired bitmap, sized for a
         | screen display.  This will be stretched as needed for
         | the target display.
         \*----------------------------------------------------------*/
         hbm = SelectObject( hdc, hbmDefault );
      }
      else
      {
         /*----------------------------------------------------------*\
         | Not enough memory.  Clean up and return NULL.
         \*----------------------------------------------------------*/
         hbm = NULL;
      }

      DeleteDC( hdc );
       }

    lReturn = (long)hbm;
    break;
```

PrinterListProc creates the bitmap using the default foreground and background colors for the device context, but it draws the window border explicitly, as follows:

```
hdc = CreateCompatibleDC( qri->hdc );
    /*----------------------------------------------------------*\
    | Make hdc really compatible with the source hdc.
    \*----------------------------------------------------------*/
    if (hdc)
    {
       SetTextColor( hdc, GetTextColor( qri->hdc ) );
       SetBkColor( hdc, GetBkColor( qri->hdc ) );
       hbrushSource = SelectObject( qri->hdc,
                           GetStockObject( NULL_BRUSH ) );
       if (hbrushSource)
          SelectObject( hdc, hbrushSource );
       hpenSource = SelectObject(qri->hdc, GetStockObject(NULL_PEN) );
       if (hpenSource)
          SelectObject( hdc, hpenSource );
    }

    hbm = CreateCompatibleBitmap( qri->hdc, rc.right, rc.bottom );
    if (hdc && hbm && (hbmDefault = SelectObject( hdc, hbm )))
    {
       GetTextMetrics( hdc, &tm );
```

```
    for (irgch = 0; irgch  MAX_PRINTERS && rgrgchPrinters[irgch][0];
        irgch++)
        TextOut( hdc, tm.tmMaxCharWidth/2,
                (tm.tmHeight + tm.tmExternalLeading)/2 +
                irgch*(tm.tmHeight + tm.tmExternalLeading),
                rgrgchPrinters[irgch],
                lstrlen( rgrgchPrinters[irgch] ) );
    hbm = SelectObject( hdc, hbmDefault );
}
.
.
.
lReturn = (long)hbm;
break;
```

CF_TEXT

If **wParam** is CF_TEXT, **PrinterListProc** simply creates an ASCII list of the system printers in a Clipboard compatible format, as follows:

```
case CF_TEXT:
    /*-------------------------------------------------------------*\
    | List out the printers in a format suitable for the Clipboard.
    | Since this list will be embedded in the text of the topic,
    | use blank lines for separators.
    \*-------------------------------------------------------------*/
    gh = GlobalAlloc( GMEM_MOVEABLE | GMEM_NOT_BANKED,
                sizeof(rgrgchPrinters) );
    lReturn = (long)gh;
    if (gh)
    {
        sz = GlobalLock( gh );
        lstrcpy( sz, "\r\n" );
        for (irgch = 0; irgch  MAX_PRINTERS && rgrgchPrinters[irgch][0];
            irgch++)
        {
            lstrcat( sz, rgrgchPrinters[irgch] );
            lstrcat( sz, "\r\n" );
        }
        GlobalUnlock( gh );
    }
    break;
```

Obtaining the Window Size (EWM_QUERYSIZE)

The **PrinterListProc** function is also responsible for processing EWM_QUERYSIZE messages. These messages can come from Windows Help, or they can be sent by the code in **PrinterListProc** that processes the EWM_RENDER message.xe "EWM_QUERYSIZE window message:in example"§

As described earlier, the EWM_QUERYSIZE message returns the dimensions of

the embedded window. The wParam parameter to EWM_QUERYSIZE is the device context for the window display, and the lParam parameter points to a **POINT** structure that returns the length and height of the window display, as shown below:

```
case EWM_QUERYSIZE:
    /*-----------------------------------------------------------------*\
    *   Size query message from Windows Help
    *   wParam is the target hdc.
    *   wLong is the address of the point to return size in.
    *   Return non-zero to indicate that we did something.
    \*-----------------------------------------------------------------*/
    hfont = SelectObject( (HDC)wParam, GetStockObject( SYSTEM_FONT ) );
    GetTextMetrics( (HDC)wParam, &tm );
    ((LPPOINT)lParam)->x = ((LPPOINT)lParam)->y = 0;
    for (irgch = 0; irgch < MAX_PRINTERS && rgrgchPrinters[irgch][0];
         irgch++)
       {
       DWORD dwExt = GetTextExtent( (HDC)wParam, rgrgchPrinters[irgch],
         lstrlen( rgrgchPrinters[irgch] ) );
       ((LPPOINT)lParam)->x = max( (WORD)((LPPOINT)lParam)->x, LOWORD(dwExt) );
       ((LPPOINT)lParam)->y += HIWORD(dwExt);
       }
    ((LPPOINT)lParam)->x += tm.tmMaxCharWidth;
    ((LPPOINT)lParam)->y += tm.tmHeight + tm.tmExternalLeading;
    if (hfont)
      SelectObject( (HDC)wParam, hfont );
    return 1;
   }
```

xe "EWM_QUERYSIZE window message:in example"§xe "DLLs:embedded-window, example of"§xe "Embedded window:controlling with DLLs:example of"§