The WinHelp API

If Windows Help is used as an online Help system with context sensitivity, the application must be programmed so that the user can access the Help application and the appropriate Help file. The **WinHelp** API supports both context-sensitive and topical searches of the Help file.

This chapter explains the **WinHelp** function and describes different ways to call

The WinHelp Function

Help from a Windows application.

An application makes a Help system available to the user by calling the **WinHelp**

Syntax

function. The **WinHelp** function uses the following C-language syntax.

BOOL WinHelp (*hWnd*, *lpszHelpFile*, *wCommand*, *dwData*)

```
HWND hwnd;              /* handle of window requesting help     */
LPCSTR lpszHelpFile;    /* address of directory-path string     */
UINT wCommand;          /* type of help                         */
DWORD dwData;           /* additional data                      */
```

The **WinHelp** function starts Windows Help (WINHELP.EXE) and passes optional data indicating the nature of the help requested by the application. The application specifies the name and, where required, the path of the Help file that

Parameters

the Help application is to display.

hWnd

Identifies the window requesting Help. The **WinHelp** function uses this handle to keep track of which applications have requested Help.

lpszHelpFile

Points to a null-terminated string containing the path, if necessary, and the name and extension of the Help file that the Help application is to display. The Help file extension (usually .HLP) is required, and a directory path to the Help file is recommended.

The filename may be followed by an angle bracket (>) and the name of a secondary window if the topic is to be displayed in a secondary window rather than the main Help window. The name of the secondary window must have been defined in the [WINDOWS] section of the Help project file for the Help file being called. For a description of the possible effects on the main and secondary Help windows, see the following "Comments" section.

wCommand

Specifies the type of help requested. For a list of possible values and how they affect the value to place in the *dwData* parameter, see the following "Comments" section.

dwData

Specifies additional data. The value used depends on the value of the *wCommand* parameter. For a list of possible values, see the following "Comments" section.

Return Value

WinHelp returns a nonzero value if the function is successful. Otherwise, the return value is zero.

Comments

Before closing the window that requested Help, the application must call **WinHelp** with *wCommand* set to HELP_QUIT. Until all applications have done this, Windows Help does not terminate.

The following table describes the possible effects on the main and secondary Help windows when using the *>WindowName* parameter:

State at time of call		Effect on window after API call	
Main	**Secondary**	**Main**	**Secondary**
closed	closed	Not opened	Opens the window and displays the Help file and topic specified in the call.
open	open	Unaffected	Displays the specified Help file and topic in the open window if the call specifies the same window name as the secondary window that is already open. Otherwise, the call closes the secondary window, opens the new secondary window, and displays the specified Help file and topic.

The following table shows the possible values for the *wCommand* parameter and the corresponding formats of the *dwData* parameter.

wCommand value dwData format	Action
HELP_CONTEXT An unsigned long integer containing the context number for the topic.	Displays the topic identified by a context number that has been defined in the [MAP] section of the Help project file.
HELP_CONTEXTNOFOCUS An unsigned long integer containing the context number for the topic.	Displays the topic identified by a context number that has been defined in the [MAP] section of the Help project file. Help does not change the focus to the window displaying the topic.

HELP_CONTEXTPOPUP An unsigned long integer containing the context number for the topic.	Displays in a pop-up window the topic identified by a context number that has been defined in the [MAP] section of the Help project file. The main Help window is not displayed.
HELP_CONTENTS Ignored; applications should set to 0L	Displays the topic defined by the **CONTENTS** option in [OPTIONS] section of the Help project file.
HELP_SETCONTENTS An unsigned long integer containing the context number for the topic the application wants to designate as the Contents topic.	Determines which Contents topic Help should display when a user presses F1 or chooses the Contents button in Help. This call should never be used with HELP_CONTENTS. If a Help file has two or more Contents topics, the application must assign one as the default. To ensure that the correct Contents topic remains set, the application should call **WinHelp** with *wCommand* set to HELP_SETCONTENTS and *dwData* specifying the corresponding context identifier. Each call to **WinHelp** should be followed with a command set to HELP_CONTEXT.
HELP_POPUPID A long pointer to a string that contains the context string of the topic to be displayed.	Displays in a pop-up window the topic identified by a specific context string. The main Help window is not displayed.
HELP_KEY A long pointer to a string that contains a keyword for the requested topic.	Displays the topic in the keyword list that matches the keyword passed in the *dwData* parameter if there is one exact match. If there is more than one match, it displays the first topic found. If there is no

match, it displays an error message.

HELP_PARTIALKEY A long pointer to a string that contains a keyword for the requested topic.	Displays the topic in the keyword list that matches the keyword passed in the *dwData* parameter if there is one exact match. If there is more than one match, it displays the Search dialog box with the topics found listed in the Go To box. If there is no match, it displays the Search dialog box. If you just want to bring up the Search dialog box without passing a keyword (the third result), you should use a long pointer to an empty string.
HELP_MULTIKEY A long pointer to the **MULTIKEYHELP** structure, as defined in WINDOWS.H. This structure specifies the table footnote character and the keyword.	Displays the topic identified by a keyword in an alternate keyword table.
HELP_COMMAND A long pointer to a string that contains a Help macro to be executed.	Executes the Help macro string specified in the *dwData* parameter. Help must be running and the Help file must be open when Help receives this API message; otherwise, Help may ignore this message.
HELP_SETWINPOS A far pointer to the **HELPWININFO** structure, as defined in WINDOWS.H. This structure specifies the size and position of the main Help window or a secondary window.	Positions the Help window according to the data passed. If the Help window is minimized, it is opened first and then positioned.

HELP_CLOSEWINDOW Ignored; applications should set to 0L.	Closes the main Help window, or a secondary window if specified in the *lpszHelpFile* argument.
HELP_FORCEFILE Ignored; applications should set to 0L.	Ensures that the correct Help file is displayed. If the correct Help file is currently displayed, there is no action. If the correct Help file is *not* displayed, **WinHelp** opens the correct file and displays the topic defined by the **CONTENTS** option in the [OPTIONS] section of the Help project file.
HELP_HELPONHELP Ignored; applications should set to 0L.	Displays the Contents topic of the designated How To Use Help file.
HELP_QUIT Ignored; applications should set to 0L.	Informs the Help application that Help is no longer needed. If no other applications have requested Help, Windows closes the Help application.

The following table shows the complete list of **#defines** for **WinHelp** commands.

wCommand	**Hexadecimal value**
#define HELP_CONTEXT	0x0001
#define HELP_QUIT	0x0002

```
#define HELP_INDEX              0x0003 (Windows Help version 3.0)

#define HELP_CONTENTS           0x0003

#define HELP_HELPONHELP         0x0004

#define HELP_SETINDEX           0x0005 (Windows Help version 3.0)

#define HELP_SETCONTENTS        0x0005

#define HELP_CONTEXTPOPUP       0x0008

#define HELP_FORCEFILE          0x0009

#define HELP_KEY                0x0101

#define HELP_COMMAND            0x0102

#define HELP_POPUPID            0x0104
```

```
#define HELP_PARTIALKEY         0x0105

#define HELP_CLOSEWINDOW        0x0107

#define HELP_CONTEXTNOFOCUS     0x0108

#define HELP_MULTIKEY           0x0201

#define HELP_SETWINPOS          0x0203
```

Using Help in a Windows Application

Windows applications can offer help to their users by using the **WinHelp** function to start Windows Help and display topics in the application's Help file. The **WinHelp** function gives a Windows application complete access to the Help file, as well as to the menus and commands of Windows Help. Many applications use **WinHelp** to implement context-sensitive Help. Context-sensitive Help enables users to view topics about specific windows, menus, menu items, and control windows by selecting the item with the keyboard or the mouse. For example, a user can learn about the Open command on the File menu by selecting the command (using the direction keys) and pressing the F1 key.

Choosing Help from the Help Menu

Every application should provide a Help menu to allow the user to open the Help file with either the mouse or the keyboard. The Help menu should contain at least one Contents menu item that, when chosen, displays the Contents or the main topic in the Help file. To support the Help menu, the application's main window procedure should check for the Contents menu item and call the **WinHelp** function, as in the following example:

```
case WM_COMMAND:
    switch (wParam) {
    case IDM_HELP_CONTENTS:
        WinHelp(hwnd, "myhelp.hlp", HELP_CONTENTS, 0L);
        return 0L;
        .
        .
        .


    }
    break;
```

You can add other menu items to the Help menu for topics containing general information about the application. For example, if your Help file contains a topic that describes how to use the keyboard, you can place a Keyboard menu item on the Help menu. To support additional menu items, your application must specify either the context string or the context identifier for the corresponding topic when it calls the **WinHelp** function. The following example uses a Help macro to specify the context string IDM_HELP_KEYBOARD for the Keyboard topic:

```
case IDM_HELP_KEYBOARD:
    WinHelp(hwnd, "myhelp.hlp", HELP_COMMAND,
            (DWORD)(LPSTR)"JumpID(\"myhelp.hlp\",\"IDM_HELP_KEYBOARD\")");
    return 0L;
```

A better way to display a topic is to use a context identifier. To do this, the Help file must assign a unique number to the corresponding context string, in the [MAP] section of the Help project file. For example, the following section assigns the number 101 to the context string IDM_HELP_KEYBOARD:

```
[MAP]
IDM_HELP_KEYBOARD    101
```

An application can display the Keyboard topic by specifying the context identifier in the call to the **WinHelp** function, as in the following example:

```
#define IDM_HELP_KEYBOARD   101

WinHelp(hwnd, "myhelp.hlp", HELP_CONTEXT, (DWORD)IDM_HELP_KEYBOARD);
```

To make maintenance of an application easier, most programmers place their

defined constants (such as IDM_HELP_KEYBOARD in the previous example) in a single header file. As long as the names of the defined constants in the header file are identical to the context strings in the Help file, you can include the header file in the [MAP] section to assign context identifiers, as shown in the following example:

```
[MAP]
#include <myhelp.h>
```

If the defined constants in the header file are different from the context strings in the Help file, you can use Help Author to perform the context mapping.

Defining More Than One Help Contents

Some applications may require more than one Help Contents topic, depending on the state of the application. For example, the interface and options in PIF Editor are different for Standard mode and Enhanced mode. Therefore, the Help offered by PIF Editor depends on which mode the user is running. When running in Standard mode, the user sees the Help Contents tailored to that mode, and the user see a different Help Contents when running in Enhanced mode.

If a Help file contains two or more Contents topics, the application can assign one as the default by using the context identifier and the HELP_SETCONTENTS value in a call to the **WinHelp** function.

The sample application Helpex applies a somewhat different model by defining a function that the application can use instead of **WinHelp**. This function sends the HELP_SETCONTENTS value and sets the Contents topic without opening Windows Help.

```
BOOL MyWinHelp(hwnd, lpHelpfile, wCommand, dwData)
HWND     hwnd;
LPSTR    lpHelpfile;
WORD     wCommand;
DWORD    dwData;
{
  static DWORD   ctxContents = (DWORD)-1L;

  if (wCommand == HELP_SETCONTENTS) {
    ctxContents = dwData;
    return (TRUE);
  }

  if (wCommand == HELP_CONTENTS && ctxContents != (DWORD)-1L) {
    WinHelp(hwnd, lpHelpfile, HELP_CONTEXT, ctxContents);
  }
  else {
    WinHelp(hwnd, lpHelpfile, wCommand, dwData);
  }
```

```
  if (wCommand != HELP_QUIT && ctxContents != (DWORD)-1L) {
    WinHelp (hwnd, lpHelpfile, HELP_SETCONTENTS, dwData);
  }
}
```

After the Contents topic is set, the application can use any **WinHelp** API, including HELP_CONTENTS.

Choosing Help with the Keyboard

An application can enable the user to choose a help topic with the keyboard by intercepting the F1 key. Intercepting this key lets the user select a menu, menu item, dialog box, message box, or control window and view Help for it with a single keystroke.

Note

The sample code in this section assumes that resource ID numbers are all unique and correspond directly to context ID numbers used for Help. Although this assumption makes things more restrictive than they have to be, it is a worthwhile option to consider using because it simplifies the code.

To intercept the F1 key, the application must install a message filter procedure by using the **SetWindowsHook** function. This allows the application to examine all keystrokes for the application, regardless of which window has the input focus. If the filter procedure detects the F1 key, it posts a WM_F1DOWN message (application-defined) to the application's main window procedure. The procedure then determines which Help topic to display.

The filter procedure should have the following form:

```
int FAR PASCAL FilterFunc(nCode, wParam, lParam)
int nCode;
WORD wParam;
DWORD lParam;
{
  LPMSG lpmsg = (LPMSG)lParam;

  if ((nCode == MSGF_DIALOGBOX || nCode == MSGF_MENU) &&
      lpmsg->message == WM_KEYDOWN && lpmsg->wParam == VK_F1) {
    PostMessage(hWnd, WM_F1DOWN, nCode, 0L);
```

```
    }

    DefHookProc(nCode, wParam, lParam, &lpFilterFunc);
    return 0;
}
```

The application should install the filter procedure after creating the main window, as shown in the following example:

```
lpProcInstance = MakeProcInstance(FilterFunc, hInstance);
if (lpProcInstance == NULL)
   return FALSE;
lpFilterFunc = SetWindowsHook(WH_MSGFILTER, lpProcInstance);
```

Note

Be sure that **FilterFunc** is exported in the .DEF file.

Like all callback functions, the filter procedure must be exported by the application.

The filter procedure sends a WM_F1DOWN message only when the F1 key is pressed in a dialog box, message box, or menu. Many applications also display the Contents topic if no menu, dialog box, or message box is selected when the user presses the F1 key. In this case, the application should define the F1 key as an accelerator key that starts Help.

To create an accelerator key, the application's resource definition file must define an accelerator table, as follows:

```
1 ACCELERATORS
BEGIN
    VK_F1, IDM_HELP_CONTENTS, VIRTKEY
END
```

To support the accelerator key, the application must load the accelerator table by using the **LoadAccelerators** function and translate the accelerator keys in the main message loop by using the **TranslateAccelerator** function.

In addition to installing the filter procedure, the application must keep track of which menu, menu item, dialog box, or message box is currently selected. In other words, when the user selects an item, the application must set a global variable indicating the current context. For dialog and message boxes, the application should set the *dwCurrentHelpId* global variable immediately before calling the **DialogBox** or **MessageBox** function. For menus and menu items, the application should set the variable whenever it receives a WM_MENUSELECT

message. As long as identifiers for all menu items and controls in an application are unique, an application can use code similar to the following example to monitor menu selections:

```
case WM_MENUSELECT:
   /*
    * Set dwCurrentHelpId to the Help ID of the menu item that is
    * currently selected.
    */

   if (HIWORD(lParam) == 0)            /* no menu selected */
      dwCurrentHelpId = ID_NONE;

   else if (lParam & MF_POPUP) {            /* pop-up selected */
      if ((HMENU)wParam == hMenuFile)
         dwCurrentHelpId = ID_FILE;
      else if ((HMENU)wParam == hMenuEdit)
         dwCurrentHelpId = ID_EDIT;
      else if ((HMENU)wParam == hMenuHelp)
         dwCurrentHelpId = ID_HELP;
      else
         dwCurrentHelpId = ID_SYSTEM;
   }
   else           /* menu item selected */
      dwCurrentHelpId = wParam;

   break;
```

In this example, the *hMenuFile*, *hMenuEdit*, and *hMenuHelp* parameters must previously have been set to specify the corresponding menu handles. An application can use the **GetMenu** and **GetSubMenu** functions to retrieve these handles.

When the main window procedure finally receives a WM_F1DOWN message, it should use the current value of the global variable to display a Help topic. The application can also provide Help for individual controls in a dialog box by determining which control has the focus at this point, as shown in the following example:

```
case WM_F1DOWN:
   /*
    * If there is a current Help context, display it.
    */

   if (dwCurrentHelpId != ID_NONE) {

      DWORD dwHelp = dwCurrentHelpId;

      /*
       * Check for context-sensitive Help for individual dialog box
       * controls.
       */

      if (wParam == MSGF_DIALOGBOX) {
```

```
        WORD wID = GetWindowWord(GetFocus(), GWW_ID);
        if (wID != IDOK && wID != IDCANCEL)
            dwHelp = (DWORD) wID;
    }

    WinHelp(hWnd, szHelpFileName, HELP_CONTEXT, dwHelp);

    /*
     * This call is used to remove the highlighting from the system
     * menu, if necessary.
     */

    DrawMenuBar(hWnd);
  }

  break;
```

When the application ends, it must remove the filter procedure by using the **UnhookWindowsHook** function and free the procedure instance for the function

Choosing Help with the Mouse

by using the **FreeProcInstance** function.

An application can let the user choose a Help topic with the mouse by intercepting mouse input messages and calling the **WinHelp** function. To distinguish requests to view Help from regular mouse input, the user must press the SHIFT+F1 key combination. In such cases, the application sets a global variable when the user presses the key combination and changes the cursor shape to a question-mark pointer to indicate that the mouse can be used to choose a Help topic.

To detect the SHIFT+F1 key combination, an application checks for the VK_F1 virtual-key value in each WM_KEYDOWN message sent to its main window procedure. It also checks for the VK_ESCAPE virtual-key code. The user presses the ESC key to quit Help and restore the mouse to its regular function. The following example checks for these keys:

```
case WM_KEYDOWN:
  if (wParam == VK_F1) {

    /* If Shift+F1, turn on Help mode and set Help cursor. */

    if (GetKeyState(VK_SHIFT)) {
        bHelp = TRUE;
        SetCursor(hHelpCursor);
        return (DefWindowProc(hWnd, message, wParam, lParam));
    }
```

```
      /* If F1 without shift, call Help main index topic. */

      else {
         WinHelp(hWnd,szHelpFileName,HELP_INDEX,0L);
      }
   }

   else if (wParam == VK_ESCAPE && bHelp) {

      /* To escape during Help mode, turn off Help mode. */

      bHelp = FALSE;
      SetCursor((HCURSOR) GetClassWord(hWnd, GCW_HCURSOR));
   }

   break;
```

Until the user clicks the mouse or presses the ESC key, the application responds to WM_SETCURSOR messages by resetting the cursor to the arrow and question-mark combination.

```
case WM_SETCURSOR:
   /*
    * In Help mode it is necessary to reset the cursor in response to
    * every WM_SETCURSOR message. Otherwise, by default, Windows resets
    * the cursor to that of the window class.
    */

   if (bHelp) {
      SetCursor(hHelpCursor);
      break;
   }
   return (DefWindowProc(hWnd, message, wParam, lParam));

case WM_INITMENU:
   if (bHelp) {
      SetCursor(hHelpCursor);
   }
   return (TRUE);
```

If the user clicks the mouse button in a nonclient area of the application window while in Help mode, the application receives a WM_NCLBUTTONDOWN message. By examining the *wParam* value of this message, the application can determine which context identifier to pass to **WinHelp**.

```
case WM_NCLBUTTONDOWN:
   /*
    * If we are in Help mode (Shift+F1), display context-sensitive
    * Help for nonclient area.
    */

   if (bHelp) {
      dwHelpContextId =
         (wParam == HTCAPTION) ?(DWORD) HELPID_TITLE_BAR:
         (wParam == HTSIZE) ? (DWORD) HELPID_SIZE_BOX:
         (wParam == HTREDUCE) ? (DWORD) HELPID_MINIMIZE_ICON:
         (wParam == HTZOOM) ? (DWORD) HELPID_MAXIMIZE_ICON:
```

```
        (wParam == HTSYSMENU) ?(DWORD) HELPID_SYSTEM_MENU:
        (wParam == HTBOTTOM) ? (DWORD) HELPID_SIZING_BORDER:
        (wParam == HTBOTTOMLEFT) ? (DWORD) HELPID_SIZING_BORDER:
        (wParam == HTBOTTOMRIGHT) ?(DWORD) HELPID_SIZING_BORDER:
        (wParam == HTTOP) ?(DWORD) HELPID_SIZING_BORDER:
        (wParam == HTLEFT) ?(DWORD) HELPID_SIZING_BORDER:
        (wParam == HTRIGHT) ?(DWORD) HELPID_SIZING_BORDER:
        (wParam == HTTOPLEFT) ?(DWORD) HELPID_SIZING_BORDER:
        (wParam == HTTOPRIGHT) ? (DWORD) HELPID_SIZING_BORDER:
        (DWORD) 0L;

      if (!((BOOL) dwHelpContextId))
        return (DefWindowProc(hWnd, message, wParam, lParam));
      bHelp = FALSE;
      WinHelp(hWnd, szHelpFileName, HELP_CONTEXT, dwHelpContextId);
      break;
    }

    return (DefWindowProc(hWnd, message, wParam, lParam))
```

If the user clicks a menu item while in Help mode, the application intercepts the WM_COMMAND message and sends the Help request:

```
case WM_COMMAND:

    /* Are we in Help mode (Shift+F1)? */

    if (bHelp) {
      bHelp = FALSE;
      WinHelp(hWnd,szHelpFileName,HELP_CONTEXT, (DWORD)wParam);
      return NULL;
    }
```

Searching for Help with Keywords

An application can enable the user to search for Help topics based on full or partial keywords. This method is similar to employing the Search dialog box in Windows Help to find useful topics. The following example searches for the keyword “Keyboard” and displays the corresponding topic, if found:

```
WinHelp (hWnd, "myhelp.hlp", HELP_KEY, "Keyboard");
```

If the topic is not found, Windows Help displays an error message. If more than one topic has the same keyword, Windows Help displays only the first topic.

An application can give the user more options in a search by specifying partial keywords. When a partial keyword is given, Windows Help usually displays the Search dialog box to allow the user to continue the search or return to the application. However, if there is an exact match and no other topic exists with the

given keyword, Windows Help displays the topic. The following example opens the Search dialog box and selects the first keyword in the list starting with the letters Ke:

```
WinHelp(hwnd, "myhelp.hlp", HELP_PARTIALKEY, "Ke");
```

When the HELP_KEY and HELP_PARTIALKEY values are specified in the **WinHelp** function, Windows Help searches the *K* keyword table. This table contains keywords generated by using the letter *K* with **\footnote** statements in the topic file. However, your application may have commands or terms that correspond to terms in a similar, but different, application.

An application can search alternative keyword tables by specifying the HELP_MULTIKEY value in the **WinHelp** function. In this case, the application must specify the footnote character for the alternate keyword table and the full keyword in a **MULTIKEYHELP** structure. The **MULTIKEYHELP** structure specifies a keyword table and an associated keyword to be used by the Windows Help application.

The **MULTIKEYHELP** structure has the following form:

```
typedef struct tagMULTIKEYHELP {    /* mkh */
   WORD  mkSize;
   BYTE  mkKeyList;
   BYTE  szKeyPhrase[1];
} MULTIKEYHELP;
```

where

mkSize
Specifies the length, in bytes, of the **MULTIKEYHELP** structure, including the keyword (or phrase) and the associated keyword-table letter.

mkKeyList
Contains a single character that identifies the keyword table to be searched.

szKeyPhrase
Contains a null-terminated text string that specifies the keyword to be located in the alternate keyword table.

The following example illustrates a keyword search for the word “frame” in the alternate keyword table designated with the footnote character L:

```
HANDLE hqmk;
MULTIKEYHELP far *qmk;
```

```
char szKeyword[] = "frame";

case MULTIKEY:
  hqmk = GlobalAlloc(GHND, (sizeof(MULTIKEYHELP) + lstrlen(szKeyword)));
  if (hqmk == NULL)
     break;
  qmk = (MULTIKEYHELP far *) GlobalLock(hqmk);
  qmk -> mkSize = sizeof(MULTIKEYHELP) + strlen(szKeyword);
  qmk -> mkKeylist = 'L';
  lstrcpy(qmk->szKeyphrase, szKeyword);

  WinHelp(hWnd, szHelpFileName, HELP_MULTIKEY, (DWORD) (LPSTR) qmk);

  GlobalUnlock(hqmk);
  GlobalFree(hqmk);
```

Displaying Help in a Secondary Window

```
  break;
```

An application can display Help topics in secondary windows instead of in Windows Help's main window. Secondary windows are useful whenever the user does not need the full capabilities of Windows Help. The Windows Help menus and buttons are not available in secondary windows.

To display Help in a secondary window, the application specifies the name of the secondary window along with the name of the Help file. The following example displays the Help topic having the context identifier IDM_FILE_SAVE in the secondary window named "wnd_menu":

```
WinHelp(hwnd, "myhelp.hlp>wnd_menu", HELP_CONTEXT, IDM_FILE_SAVE);
```

The name and characteristics of the secondary window must be defined in the [WINDOWS] section of the Help project file, as in the following example:

```
[WINDOWS]
wnd_menu = "Menus", (128, 128, 256, 256), 0
```

Windows Help displays the secondary window with the initial size and position specified in the [WINDOWS] section. However, an application can set a new size and position by specifying the HELP_SETWINPOS value in the **WinHelp** function. In this case, the application sets the members in a **HELPWININFO** structure to specify the window size and position. The following example sets the "wnd_menu" secondary window to a new size and position:

```
HANDLE hhwi;
LPHELPWININFO lphwi;
WORD wSize;
```

```
char *szWndName = "wnd_menu";

wSize = sizeof(HELPWININFO) + lstrlen(szWndName);
hhwi = GlobalAlloc(GHND, wSize);
lphwi = (LPHELPWININFO)GlobalLock(hhwi);

lphwi->wStructSize = wSize;
lphwi->x          = 256;
lphwi->y          = 256;
lphwi->dx         = 767;
lphwi->dy         = 512;
lphwi->wMax       = 0;
lstrcpy(lphwi->rgchMember, szWndName);

WinHelp(hwnd, "myhelp.hlp", HELP_SETWINPOS, lphwi);

GlobalUnlock(hhwi);
```

Canceling Help

```
GlobalFree(hhwi);
```

Windows Help requires an application to explicitly cancel Help so that Windows Help can free any resources it used to keep track of the application and its Help files. The application can do this at any time.

An application cancels Windows Help by calling the **WinHelp** function and specifying the HELP_QUIT value, as shown in the following example:

```
case WM_DESTROY:
   WinHelp (hwnd, "myhelp.hlp", HELP_QUIT, NULL);
```

If the application has made any calls to the **WinHelp** function, it must cancel Help before it closes its main window (for example, in response to the WM_DESTROY message in the main window procedure). If the application has opened more than one Help file, it must call WinHelp to cancel Help for each file. Windows Help remains running until all applications or dynamic-link libraries that have called **WinHelp** function have canceled Help.

We strongly recommend that an application use the HELP_QUIT parameter in its exit routine even if it has not actually sent any other **WinHelp** calls.