Connecting WinHelp to C++/MFC Programs

by Don Lammers

Last updated 2001-02-22

Presented with author's permission by

Shadow Mountain Tech

Contents

Scope 1
Acknowledgements 1
Connecting Context Sensitive WinHelp to C++/MFC Programs 2
Calling WinHelp Directly from Your C++ Code 11
Calling WinHelp from a Command Line 15
Training Cards from C++ 16
Resources 18

Scope

MFC provides several hooks for easily connecting to WinHelp. In addition, you can call the WinHelp API from anywhere in your program for additional access to help. The *Connecting Context Sensitive WinHelp to C++/MFC Programs* section shows how to connect to the built in hooks that MFC provides, and is therefore specific to programming with MFC. The remainder of the document should be easily adaptable to any version of C++, as it sets forth the Windows calls and command line interface for WinHelp.

Acknowledgements

The following information is from my own experimentation and from people who have walked this path before me. Thanks to all of the following:

Microsoft Help WorkShop Help, Microsoft Knowledge Base, *The Developer's Guide to WINHELP.EXE* (Jim Mischel), *Building Windows 95 Help* (Nancy Hickman), Paul O'Rear, *Developing Online Help for Windows 95* (Boggan, Farkas, and Welinske), Gordon F. MacLeod, Burt Abreu.

Connecting Context Sensitive WinHelp to C++/MFC Programs

MFC provides for calling WinHelp topics using the WinHelp function of the CWnd class, from which the Main Frame window of your program is derived if you use the App Wizard. In addition you can override dialog box routines to get control level F1 or right click help if needed.

This section shows the steps necessary to hook context sensitive WinHelp into a program created using MFC.

Create a Program with Built-In Help

If you use the MFC App Wizard to create your project you can include WinHelp support for the main window as part of the automatically generated files.

To create a new project with context sensitive help support:

1. From the Visual C++ **File** menu, choose **New**.
2. Choose **MFC AppWizard (.exe)**.
3. When given the option, make sure that the **Context Sensitive Help** box is checked.
4. Continue through the process to set up the program the way you want.

This creates the following:

- An application with a menu and (if you requested it) a toolbar.
- A help file with topics for each of the automatically generated menu items and toolbar buttons. The help author can use this framework as a starting point.
- A .hm file with the topic ID to number mapping. This file can be used directly by WinHelp.
- An operational What's This? Help button  on the toolbar. Clicking this button puts the program into What's This? Help mode. Clicking on a menu or control then opens its help.
- **Shift-F1** hooked up to put the program into What's this? Help mode.
- **F1** hooked up to open general help for the child window with focus or the control (but not menu) on which the user is currently clicking.
- When you create a dialog box, **F1** hooked up to open a general help topic for the whole dialog box.
- All help topics opened in the main WinHelp window.

Note:

- Since no dialog boxes are created in this process, you will need to add context sensitive help support for each dialog box as you create it.
- Once you have the initial help file and .hm file, it is usually easier to have the help author add new topics using their preferred help authoring tool. You can still have the program generate the framework help each time you build, because that will give you appropriate additional topic mapping entries.

Set the Help File Name

The default help file name for an MFC generated program is the name of the executable with a .hlp extension. If your help file is named anything else, you need to set the **m_pszHelpFilePath** variable before using the built in help features of MFC so that the program knows the help file to call.

To set the help file name for MFC calls:

- Set the **m_pszHelpFilePath** variable member of the application class in the **InitInstance** function. **m_pszHelpFilePath** is a public variable of type **const char***. According to MS, you should first free the memory associated with the default string (the app name with a .hlp extension).

```
//First free the string allocated by MFC at CWinApp startup.
//The string is allocated before InitInstance is called.
free((void*)m_pszHelpFilePath);
//Change the name of the .HLP file.
//The CWinApp destructor will free the memory.
m_pszHelpFilePath=_tcsdup(_T("d:\\somedir\\myhelp.hlp"));
```

Tips:

- **m_pszHelpFilePath** can be changed at runtime, so you can reset it in code as needed to call additional help files.
- If you have more than one help file to call from the program you should put the code for changing help file names in a separate subroutine. This makes any changes during the project much easier to implement.
- You can let the help author edit the help file name and window specification by putting them in a separate text file. Setting this file up in INI file format lets you use standard INI file commands to access this "text database."

Make Sure WinHelp Can Find the Help File

Make sure that WinHelp can always find the help file. Since your program can be started from shortcuts created by the user, the installation program should register the help file or you should call help from your program using the full path and file name.

> If the help system contains more than one .hlp file and a master Table of Contents (.cnt file), and not all of these are in the same folder, it will be necessary to register the help file and the master TOC to make the master TOC work, regardless of any other issues.

To ensure that the program and WinHelp can always find the help file:

- Register the help file by creating a string value under HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Help. The name of the string value is the help file name (without path) and the value is the path.
- Always use the full path and file name when setting **m_pszHelpFilePath**, even if the help file is in the same folder as the program executable.

Manually Add the Context Hooks

If you already have a program and need to add help, you can manually edit the files to create the necessary hooks.

To add context sensitive help manually:

1. Add the necessary message map entries in the message map of the Main Frame class. If you created the program with the MFC App Wizard and specified that you wanted context help, these items should already be in the file.

```
ON_COMMAND(ID_HELP_FINDER, CMDIFrameWnd::OnHelpFinder)
ON_COMMAND(ID_HELP, CMDIFrameWnd::OnHelp)
ON_COMMAND(ID_CONTEXT_HELP, CMDIFrameWnd::OnContextHelp)
ON_COMMAND(ID_DEFAULT_HELP, CMDIFrameWnd::OnHelpFinder)
```

2. Copy the What's This? Help icon  into a button on your toolbar. This icon is available as a bitmap in C:\Program Files\Microsoft Visual Studio\Common\Graphics\Bitmaps\OffCtlBr\Small\Color\help.bmp.
3. Assign ID_CONTEXT_HELP to the What's This? Help button.

Note:

- This creates the same basic hooks in the program as the App Wizard, but does not generate a framework help file or the .hm file with the topic ID mapping. You will need to create a list of IDs for the help author to use.

Override the Default Context Help Behavior

The default WinHelp call made from the hooks automatically created by the App Wizard opens each help topic in the main WinHelp window. Unfortunately the current recommended behavior is:

- Control level help is opened in a popup.
- Window level help is opened in a secondary window.

To create these conditions, you must override the default functions.

To create currently recommended behavior:

1. In the Main Frame class, create an **OnHelpInfo** function using the Class Wizard:

```
BOOL CMainFrame::OnHelpInfo(HELPINFO* pHelpInfo)
{
  return TRUE;
}
```

   This effectively disables the function, and is necessary because, in combination with the next override, all topics accessed through the **F1** key will be called twice. Unfortunately you cannot use the **OnHelpInfo** function in the Main Frame since (unlike in dialog boxes) it is triggered only by the **F1** key and not by **Shift-F1** or the  key.

2. In the Main Frame class, create a **WinHelp** function using the Class Wizard:

```
void CMainFrame::WinHelp(DWORD dwData, UINT nCmd)
{
  CWinApp* theApp = AfxGetApp();
  CString helpFilePath = theApp->m_pszHelpFilePath;
  switch ( dwData ) {
  case HIDR_MAINFRAME:
  case HIDR_MFCWIZTYPE:
  case HIDD_ABOUTBOX:
  case HIDD_TESTDLG:
    //Topics that need to go into a secondary window.
    helpFilePath = helpFilePath + ">second";
    ::WinHelp(m_hWnd, helpFilePath, HELP_CONTEXT, dwData);
    break;
  case WHATEVER
    //Topics that need to go into a different secondary window
    ...
    break;
  default:
    //All the rest are popups
    ::WinHelp(m_hWnd, helpFilePath, HELP_CONTEXTPOPUP, dwData);
    break; }
}
```

Tips:

- It is often easier to create a separate function or class to process the help calls and open topics in the correct window. This can be part of the main program or can be created as a separate DLL. The main advantage of a DLL is that it can be called from other programs if necessary.
- If you want the help author to be able to control the help file and window called by this function, class, or DLL, put the help file and window definitions in an INI file (you can use the topic number as the key) and look them up before calling Help. If you include the help type as an INI file entry you can easily call multiple types of user assistance from the same program.

Activate Control Level Help for a Dialog Box

You can add control level help (What's This? Help) to a dialog box by overriding the **OnHelpInfo** function for the dialog box class.

To add What's This? Help to a dialog box:

1. In the dialog box class, create a **OnHelpInfo** function using the Class Wizard:

```
BOOL cTestDlg::OnHelpInfo(HELPINFO* pHelpInfo) {
  //Get the control ID
  DWORD cControlID = pHelpInfo->iCtrlId;
  //Only proceed if it's not a static control
  if (cControlID != 65535)  //65535 is IDC_STATIC
  {
    //Get the context ID for the control
    DWORD cTopic = pHelpInfo->dwContextId;
    //Get the hWnd for the app
    HWND hwndApp = m_hWnd;
    //Create an app object for this app
    CWinApp* theApp = AfxGetApp();
    //Get the help file name
    CString helpFilePath = theApp->m_pszHelpFilePath;
    //Call the WinHelp API. This bypasses all MFC code.
    ::WinHelp(m_hWnd, helpFilePath, HELP_CONTEXTPOPUP, cTopic);
  }
  return TRUE;
}
```

2. In the **Extended Styles** tab of its property sheet, make sure that **Context help** is on.

Note:

- You cannot add the What's This? Help button to any resizeable window or to a tool window. If you want the What's This? Help button in a tool window, make it a Fixed Dialog window instead. The only difference I've ever seen is that the title bar is slightly smaller on a tool window.

The HELPINFO Structure

Information about the control that is calling help is contained in the HELPINFO structure passed to the **OnHelpInfo** function. The structure contains the following elements:

The HELPINFO structure is defined as follows:

```
typedef struct tagHELPINFO {
  UINT     cbSize;
  int      iContextType;
  int      iCtrlId;
  HANDLE   hItemHandle;
  DWORD    dwContextId;
  POINT    MousePos;
}  HELPINFO, FAR *LPHELPINFO;
```

cbSize is the structure size, in bytes.

iContextType is the type of context for which Help is requested. This can be either HELPINFO_MENUITEM (help was requested from a menu item) or HELPINFO_WINDOW (help was requested from a control or window).

iCtrlId is the identifier of the window or control if *iContextType* is HELPINFO_WINDOW, or the menu identifier if *iContextType* is HELPINFO_MENUITEM.

hItemHandle is the identifier of the child window or control if *iContextType* is HELPINFO_WINDOW, or identifier of the associated menu if *iContextType* is HELPINFO_MENUITEM.

dwContextId is the help context identifier of the window or control.

MousePos is a POINT structure containing the screen coordinates of the mouse cursor.

Create a Single Help Calling Function

If you need to handle anything that is not standard MFC functionality it is usually easier to create a single help calling function (I usually call it OpenTopic) somewhere in the program or in a separate DLL. In each of the dialog box **OnHelpInfo** functions, in the Main Frame **WinHelp** function, and anywhere that you would have called the WinHelp API directly, you call this function instead. This puts all of the logic behind determining how to open a topic in one place and makes it easy to change as needed. You can easily treat other elements of the online user assistance (multimedia, Acrobat files, etc.) as topics by assigning them a topic number and handling the actual calls to them in this function. And of course you can mix help types (for instance HTML Help and WinHelp) if this is necessary for some reason.

Most of the time this function needs only two parameters: the main window hWnd and the topic number. However, if you have multiple programs potentially calling into the same help system, you may want to include a program ID as a third parameter so you know who generated the request.

A sample DLL version of this topic is shown below

GetFullFileSpec is a function that extracts the path from the second parameter (in this case the DLL name) and appends the first parameter (in this case the help file name and window) to create the necessary file name and window combination to call WinHelp.

ChangeExtension is a function that replaces the extension on the first parameter with the string in the second parameter.

To initialize and get the DLL name and location:

```
int __declspec(dllexport) WINAPI OpenTopic (HWND hwndApp, DWORD Topic)
{
  //Get the instance handle for the DLL
  CWinApp* theApp = AfxGetApp();
  HINSTANCE dllInstance = theApp->m_hInstance;
  //Get the name of the DLL
  char cDLLName[512];
  GetModuleFileName(dllInstance, cDLLName, 512);
  char cHelpCall[512];
```

To intercept the call and display the topic number for troubleshooting:

This simply shows the topic number called. To activate it you need to put a ShowTopicNumber=1 entry in the [Genera] section of a text file that has the DLL path and name but with a .dat extension. This little bit of code can often save hours of arguing over where the context help calls are broken, since it shows the exact numeric value being sent by the program to help.

```
  //See if we are supposed to display the topic number for debugging
  char iniFile[256];
  ChangeExtension(cDLLName, ".dat", iniFile);
  if (GetPrivateProfileInt("General", "showTopicNumber", 0, \
        iniFile) != 0) {
    char buffer[256];
    wsprintf(buffer, "About to open topic number %d \
        (%X hex)", Topic, Topic);
    MessageBox( NULL, buffer, \
              "Topic Number", MB_OK | MB_ICONINFORMATION );
  }
```

To open the Help Topics dialog box:

See *Calling the Help Finder* below for more detail.

```
if (Topic == 999)
{  //Open the Help Topics dialog box
  GetFullFileSpec("helpfile.hlp", cDLLName, cHelpCall);
  WinHelp(hwndApp, cHelpCall, HELP_FINDER, NULL);
  return 0;
}
```

To open a warning dialog box called as though it is a topic:

This section opens a warning message dialog box with multiple possible returns. The dialog box is modal (which is one reason why we did not use a WinHelp topic to warn the user) and since the DLL is called from the application the user must respond before the program can continue. When the program determines that a warning condition exists, it calls OpenTopic with the appropriate topic number.

In the original version of this warning dialog, all text for the dialog box was stored in a text file so it could be easily edited by the help author.

m_WarningText is the key that the dialog box will use to find the topic text in a text file.

```
else if (Topic > 0 && Topic < 50)
{ //Warnings (warning agent popup topics)
  char cTopic[64];
  wsprintf(cTopic, "topic%lu", Topic);
  int noShow = GetPrivateProfileInt(cTopic,"NoShow", 0, iniFile);
  if (noShow == 0) {
    CWarnDlg cWarningDialog;
    cWarningDialog.m_WarningText = _T(cTopic);
    ......
    //Open the dialog box
    int branch = cWarningDialog.DoModal();
    return branch; }
  else {
    return 1; }
}
```

To ignore certain topic numbers:

```
else if (Topic == 65535)
{  // Generic Control ID--ignore
  return 0; }
else if (Topic > 140000)
{ //Ignore these topic requests.
  //262144 = Separator between info bar and status bar
  //262150 = Horizontal scroll bar
  ......
  return 0; }
```

To open a topic in a secondary window:

```
    else if (Topic > 130000)
    { //Secondary window Help for dialog box overviews.
      GetFullFileSpec ("helpfile.hlp>second", cDLLName, cHelpCall);
      WinHelp(hwndApp, cHelpCall, HELP_CONTEXT, (DWORD)Topic);
      return 0; }
```

To open a topic as a popup:

```
    else
    {  // Popup help for menu items and controls.
      GetFullFileSpec ("helpfile.hlp", cDLLName, cHelpCall);
      WinHelp(hwndApp, cHelpCall, HELP_CONTEXTPOPUP, (DWORD)Topic);
      return 0; }
    return 0;
  }
```

Notes:

- The above code filters out any messages coming from controls assigned an ID of IDC_STATIC. If you want static controls (labels) to also have context help, they must each be assigned unique identifiers.
- Since all controls in a program should have unique IDs, you may be able to use the control ID rather than the context ID for this function. This is a smaller number and easier to deal with in the help mapping (sorry, just being lazy).

Calling the Help Finder

Unfortunately, the ID sent by the close box [X] in the title bar and by the ID_HELP_FINDER command when they end up at the final WinHelp command are both 0 (zero). This is not a problem if you are doing the straight default implementation of help. However, if you are customizing, you may run into problems.

To call the Help Finder directly:

Use the Class Wizard to create an **OnHelpFinder** function for the Main Frame class:

```
  void CMainFrame::OnHelpFinder() {
    //Create an app object for this app
    CWinApp* theApp = AfxGetApp();
    //Get the help file name
    CString helpFilePath = theApp->m_pszHelpFilePath;
    //Call the WinHelp API. This bypasses all MFC code
    //and makes sure that the help Finder is called.
    ::WinHelp(m_hWnd, helpFilePath, HELP_FINDER, 0);
  }
```

Note:

- If you use the HELP_FINDER command, you must make sure that the help contents file (.cnt) contains the line ":Title". There doesn't need to be an associated value--it just needs the keyword. If this line is not present and you open only secondary windows, Winhlp32.exe will crash if you try to call the Finder from the program while the help system is still open.
- If you have a single help processing function, an alternate to this method is to assign a topic number to the Help Finder and process that particular file in the function. This technique is shown in the OpenTopic example above, where the Help Finder was assigned a topic number of 999.

Connect Context Help to a Right Click on a Button

The traditional way of connecting help to a right click on a control is to provide a popup menu with an item named What's This?. When the user clicks this menu item, help opens to the appropriate topic. the basic steps in creating such a link are:

1. When the user right clicks a control, and before opening a popup menu for the control, save the topic number to call.
2. Implement whatever code is needed to open the popup menu with the appropriate items listed.
3. In the Click event for the **What's This?** menu item, call the WinHelp API with the appropriate topic number.

Tip:

- Unless there are only a few controls that do not implement a right click menu, I recommend calling the topic directly if the user right clicks a button that does not otherwise have a right click menu. A popup menu with a single What's This? item looks a bit strange, and this saves the user one click.

Calling WinHelp Directly from Your C++ Code

You can programmatically call help from anywhere in your code. The most common reasons for doing this would be to open the WinHelp Table of Contents from the help menu or to call a general help topic from a Help button on a form, but you can also call help using keywords.

WinHelp API Syntax

The **WinHelp** function starts Windows Help (WINHLP32.EXE) and passes additional data indicating the nature of the help requested by the application. To use the following API calls you must attach WinHelp.bas or Winhelp.cls (see *References* for where to get these).

```
BOOL WinHelp( HWND hWndMain, LPCTSTR lpszHelp,
    UINT uCommand, DWORD dwData )
```

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

hWndMain is the handle of the window requesting help. The **WinHelp** function uses this handle to keep track of which applications have requested help. If the *uCommand* parameter specifies HELP_CONTEXTMENU or HELP_WM_HELP, *hWndMain* identifies the control requesting help.

lpszHelp is a pointer to a null-terminated string containing the path, if necessary, and the name of the help file that WinHelp is to display.

> 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 in the primary window. The name of the secondary window must have been defined in the [WINDOWS] section of the help project (.hpj) file.

uCommand specifies the type of help requested. For a list of possible values and how they affect the value to place in the *dwData* parameter. For a partial list of possible values, see *Help Command Constants* below. For a full list, see the *Microsoft Help Workshop Help*.

dwData specifies additional data. For a partial list of possible values, see *Help Command Constants* below. For a full list, see the *Microsoft Help Workshop Help*.

WinHelp Command Constants

The following table shows the possible values for the *uCommand* parameter, its value, its action, and the corresponding formats of the *dwDat*a command:

HELP_COMMAND (0x0102) executes a WinHelp macro or macro string. *dwData* is the address of a string that specifies the name of the WinHelp macro(s) to run. If the string specifies multiple macro names, the names must be separated by semicolons. You must use the short form of the macro name for some macros because WinHelp does not support the long name.

HELP_CONTENTS (3 or 0x0003) displays the topic specified by the Contents option in the [OPTIONS] section of the .HPJ file. This command is for backward compatibility. New applications should provide a .CNT file and use the **HELP_FINDER** command. *dwData* is ignored. Set it to 0.

HELP_CONTEXT (1 or 0x0001) displays the topic identified by the specified context identifier defined in the [MAP] section of the .HPJ file. *dwData* is an unsigned long integer containing the context identifier for the topic.

HELP_CONTEXTMENU (10 or 0x000A) displays the **Help** menu for the selected window, then displays the topic for the selected control in a pop-up window. *dwData* is the address of an array of double word pairs. The first double word in each pair is a control identifier, and the second is a context number for a topic.

HELP_CONTEXTPOPUP (8 or 0x0008) displays the topic identified by the specified context identifier defined in the [MAP] section of the .HPJ file in a pop-up window. *dwData* is an unsigned long integer containing the context identifier for a topic.

HELP_CONTEXTNOFOCUS (264 or 0x0108) displays the topic identified by the specified context identifier defined in the [MAP] section of the .HPJ file. *dwData* is an unsigned long integer containing the context identifier for the topic. WinHelp does not change the focus to the window displaying the topic. This command constant is not defined anywhere except in Jim Mischel's book, so if you want to use it you have to define the constant.

HELP_TOPIC_ID (259 or 0x103) displays the topic identified by the specified topic ID string in a main or secondary window. *dwData* is a pointer to a null terminated string that contains the topic ID string of the topic to be displayed. To specify the window append ">*windowName*" to the topic ID string in *dwData.* This constant is not listed by MS, so you probably have to define it in your project. **WARNING:** This has been reported broken in Windows 98, so be very careful if using it.

HELP_POPUPID (260 or 0x0104) displays the topic identified by the specified context string in a pop-up window. *dwData* is a pointer to a null terminated string that contains the context string of the topic to be displayed. If the main help window is currently open it becomes the focused window. If the main help window is not open the pop-up topic appears but the main help window is not displayed (from Jim Mischel). This constant is not listed by MS, so you probably have to define it in your project. **WARNING:** This has been reported broken in Windows 98, so be very careful if using it.

HELP_FINDER (11 or 0x000B) displays the **Help Topics** dialog box with the last tab used. *dwData* is ignored. Set it to 0.

HELP_FORCEFILE (9 or 0x0009) ensures that WinHelp is displaying the correct help file. If the incorrect help file is being displayed, WinHelp opens the correct one; otherwise, there is no action. *dwData* is ignored. Set it to 0.

HELP_HELPONHELP (4 or 0x0004) displays help on how to use Windows Help, if the **WINHLP32.HLP** file is available. *dwData* is ignored. Set it to 0.

HELP_INDEX (3 or 0x0003) is the same as **HELP_CONTENTS**.

HELP_KEY (257 or 0x0101) displays the topic in the keyword table that matches the specified keyword, if there is an exact match. If there is more than one match, displays the Index with the topics listed in the **Topics Found** list box. *dwData* is the address of a keyword string. Multiple keywords must be separated by semicolons.

HELP_MULTIKEY (513 or 0x0201) displays the topic specified by a keyword in an alternative keyword table. *dwData* is the address of a **MULTIKEYHELP** structure that specifies a table footnote character and a keyword.

HELP_PARTIALKEY (261 or 0x0105)displays the topic in the keyword table that matches the specified keyword, if there is an exact match. If there is more than one match, displays the **Topics Found** dialog box. To display the Index without passing a keyword, you should use a pointer to an empty string. *dwData* is the address of a keyword string. Multiple keywords must be separated by semicolons.

HELP_QUIT (2 or 0x0002) informs WinHelp that it is no longer needed. If no other applications have asked for help, Windows closes WinHelp. *dwData* is ignored. Set it to 0.

HELP_CLOSEWINDOW (263 or 0x0107) Closes the specified window. *dwData* is ignored. Set it to 0. To close a secondary window append the > character and the name of the secondary window to the file name passed in the *lpszHelp* parameter.

HELP_SETCONTENTS (5 or 0x0005) specifies the Contents topic. WinHelp displays this topic when the user clicks the **Contents** button if the help file does not have an associated .cnt file. *dwData* is an unsigned long integer containing the context identifier for the Contents topic.

HELP_SETINDEX (5 or 0x0005) is the same as HELP_SETCONTENTS.

HELP_SETPOPUP_POS (13 or 0x000D) sets the position of the subsequent pop-up window. *dwData* is the address of a **POINTS** structure. The popup window is positioned as if the mouse cursor were at the specified point when the pop-up window is invoked.

HELP_SETWINPOS (515 or 0x0203) displays the help window, if it is minimized or in memory, and sets its size and position as specified. *dwData* is the address of a **HELPWININFO** structure that specifies the size and position of either a primary or secondary help window.

HELP_TCARD (32768 or 0x8000) indicates that a command is for a training card instance of WinHelp. Combine this command with other commands using the bitwise OR operator. *dwData* depends on the command with which this command is combined.

HELP_TCARD_DATA (16 or 0x0010)

HELP_TCARD_OTHER_CALLER (17 or 0x0011)

HELP_WM_HELP (12 or 0x000C) displays the topic for the control identified by the *hWndMain* parameter in a pop-up window. *dwData* is the address of an array of double word pairs. The first double word in each pair is a control identifier, and the second is a context identifier for a topic.

The following values may also be used, but they are not in Windows.h so you need to add them to your own header file.

HELP_FORCE_GID (14 or 0x000E) Verifies that the .gid file for the help file specified in *lpszHelpFile* is being used. If not, the .hlp file switches to using it. *dwData* is ignored. Set it to 0 (zero).

HELP_TAB (15 or 0x000F) opens the **Help Topics** dialog box to the specified tab. *dwData* is the tab number. Tab numbering starts at 0 (zero).

The following are for use with custom tabs in the **Help Topics** dialog box

Constant	**Definition**
MSG_TAB_CONTEXT	WM_USER + 38
MSG_TAB_MACRO	WM_USER + 39

WinHelp API Examples

The purpose of calling the WinHelp API outside the constraints of MFC is to call help from a button or other location not handled by MFC, or to make sure that MFC has no chance to intercept the call and do something unexpected. The examples below call the WinHelp API function directly from within MFC.

In the examples below:

- "m_hWnd" is the window handle for the main form in your program.
- Its assumed that you have set the full path and file name for the help file in **m_pszHelpFilePath**.
- "TopicID" is the ID for the desired topic.

To display the help file Contents topic:

```
::WinHelp(m_hWnd, m_pszHelpFilePath, HELP_CONTENTS, NULL)
```

To display the last tab used in the Help Topics dialog box:

```
::WinHelp(m_hWnd, m_pszHelpFilePath, HELP_FINDER, NULL)
```

To display a specific topic in a standard window:

```
::WinHelp(m_hWnd, m_pszHelpFilePath, HELP_CONTEXT, TopicID)
```

To display a specific topic in a popup:

```
::Winhelp(m_hWnd, m_pszHelpFilePath, HELP_CONTEXTPOPUP, TopicID)
```

To display a topic or topics using a K Keyword:

```
//keyWord is a string containing the keyword to look for.
::Winhelp(m_hWnd, m_pszHelpFilePath, HELP_KEY, keyWord)
```

To display a topic or topics using an A Keyword after testing to see if the keyword exists:

```
//keyWord is a string containing the keyword to look for.
Cstring helpCommand = "IF(KL(`"
helpCommand = helpCommand + keyWord
helpCommand = helpCommand + "', 4, `', `'),`KL(`"
helpCommand = helpCommand + keyWord
helpCommand = helpCommand + "', 1, `', `')')"
::WinHelp(m_hWnd, m_pszHelpFilePath, HELP_COMMAND, helpCommand)
```

To force the help file closed:

```
::Winhelp(m_hWnd, m_pszHelpFilePath, HELP_QUIT, NULL)
```

Calling WinHelp from a Command Line

You can use the **CreateProcess**, **ShellExecute**, **ShellExecuteEx**, or **WinExec** API functions to call WinHelp. Each time you use one of these command a new instance of WinHelp is opened. The **ShellExecute** command looks something like:

```
ShellExecute(hWndApp, "winhlp32.exe", Parameters, defFolder,
winStyle)
```

hWndApp is the handle to a parent window. This window receives any message boxes that an application produces.

The options for *Parameters* are shown below.

defFolder is the default folder for running WinHelp. This should be the folder with the help file.

winStyle is optional and specifies how to open the window. See Visual Basic documentation for values.

The file name and parameters for calling WinHlp32.exe are as follows:

```
winhlp32.exe [[-H] [-G[n]] [-W window] [-K keyword] [-P pop-up]
  [-N contextNum] [-I topicID] helpFile]
```

-H displays the Winhlp32.hlp help file.

-G[n] creates a configuration (.gid) file and quits. If a number is specified, it determines which extensible tab to display by default the first time the help file is opened. A value of 1 would be the first tab beyond the **Find** tab. This command cannot be used with -S.

-S creates a configuration (.gid) file without showing an animated icon. Cannot be used with -G.

-W *window* specifies the window for displaying the topic. This command cannot be used with -P.

-P specifies that the topic will be shown in a pop-up window. This command cannot be used with -W. You must use the -P switch in combination with the -N (context number) or -I (topic ID) switch, depending on whether you want to specify the context number (from the [MAP] section of the HPJ file) or the topic ID string (from the # footnote of the topic).

-K *keyword* specifies the topic to open using a keyword. This command cannot be used with -I or -N.

-N *contextNum* specifies the topic to open using a topic number, which must be defined in the [MAP] section of the HPJ file. This command cannot be used with -I or -K.

-I *topicID* specifies the topic to open using a topic ID string (# footnote in the topic). This command cannot be used with -N or -K.

helpFile specifies the help file to open. Unless you are calling WinHelp from the directory with the help file, this must be a full path and file name. If you don't specify a help file, the **File Open** dialog box appears.

Tip:

- If you use long file names with spaces, you must enclose the entire path and file name in quotes.

Training Cards from C++

Training cards allow the help system to communicate directly with the program. This section describes the general procedures for setting up training cards using C++.

The TCard Macro

When the user initiates a TCard macro in a WinHelp file, the WM_TCARD message is sent to the program that initiated the training card. The program must intercept the WM_TCARD message and determine the required action.

The TCard macro syntax is as follows:

```
TCard(command)
```

Command can be a predefined command, a numeric value, or a topic ID string.

- If a predefined command is used, its numeric equivalent is sent as the *wParam* value of the WM_TCARD message.
- If a numeric value is used, HELP_TCARD_DATA (16) is sent as the value for the *wParam* and the numeric value is passed as the *lParam* for the WM_TCARD message.
- If a topic ID is used, the ID is mapped to the numeric value found in the project file and passed to the program.

The predefined TCard commands are:

IDABORT (3) The user clicked an authorable **Abort** button.
IDCANCEL (2) The user clicked an authorable **Cancel** button.
IDCLOSE (8) The user closed the training card.
IDHELP (9) The user clicked an authorable **Help** button.
IDIGNORE (5) The user clicked an authorable **Ignore** button.
IDOK (1) The user clicked an authorable **OK** button.
IDNO (7) The user clicked an authorable **No** button.
IDRETRY (4) The user clicked an authorable **Retry** button.
IDYES (6) The user clicked an authorable **Yes** button.

The WM_TCARD Message

The program must intercept the WM_TCARD message from the help file and determine the correct action to take.

The *wParam* value for the WM_TCARD message is one of the following:

IDABORT The user clicked an authorable **Abort** button.
IDCANCEL The user clicked an authorable **Cancel** button.
IDCLOSE The user closed the training card.
IDHELP The user clicked an authorable **Help** button.
IDIGNORE The user clicked an authorable **Ignore** button.
IDOK The user clicked an authorable **OK** button.
IDNO The user clicked an authorable **No** button.
IDRETRY The user clicked an authorable **Retry** button.
IDYES The user clicked an authorable **Yes** button.
HELP_TCARD_DATA The user clicked an authorable button. The *lParam* parameter contains a long integer specified by the help author.
HELP_TCARD_NEXT The user clicked an authorable Next button.
HELP_TCARD_OTHER_CALLER Another program has requested training cards.

If *wParam* is **HELP_TCARD_DATA**, *lParam* is the number indicated by the TCARD macro. Otherwise *lParam* is 0 (zero).

Processing TCard Commands in the Program

The basic procedure for setting up TCard processing follows. Please see the Windows SDK documentation for details on how to use the **SetWindowsHookEx** and **UnhookWindowsHookEx** functions, and on the required hook processing within your hook procedure.

To intercept and process the WM_TCARD message:

1. With the help author, decide on the codes that will be used for any function that is not predefined.
2. From the program, open WinHelp using the WinHelp command. Call the WinHelp API directly rather than going through MFC to ensure that MFC does not intercept and process the command somewhere.
```
::WinHelp(GetCompatibleHwnd(), m_pszHelpFilePath, \
        HELP_CONTEXT || HELP_TCARD, TopicID)
```
3. Call the **SetWindowsHookEx** function with a **WH_CALLWNDPROC** hook type to set up a hook procedure to monitor WM_TCARD messages sent from help.
4. When the hook procedure detects the WM_TCARD message, process the message to determine the necessary actions.
5. Make sure you release the hook procedure using **UnhookWindowsHookEx** before closing the program.

Resources

All of the following plus links to other reference sites about online user assistance are available through the Shadow Mountain Tech Web site (www.smountain.com).

On the Help/Connecting page:

- Latest update of this document and other documents about connecting online help to your program
- *Programmer's Reference to WinHelp* (by Don Lammers and Paul O'Rear)
- Sample code, including API definition modules.
- David Liske's help subclassing tutorial and modules

On the Help/Bugs & FAQ page:

- *WinHelp 4.0 Unofficial Bug and Anomaly List* (currently maintained by Don Lammers)
- *WinHelp FAQ File* (by Charlie Munro)