The forgotten common controls: The MenuHelp function

devblogs.microsoft.com/oldnewthing/20060608-09

June 8, 2006

Raymond Chen

The `MenuHelp` function is one of the more confusing ones in the common controls library. Fortunately, you will almost certainly never had need to use it, and once you learn the history of the `MenuHelp` function, you won't **want** to use it anyway.

Our story begins with 16-bit Windows. The `WM_MENUSELECT` message is sent to notify a window of changes in the selection state of a menu that has been associated with the window, either by virtue of being the window's menu bar or by having been passed as the owner window to a function like `TrackPopupMenu`. The parameters to the `WM_MENUSELECT` message in 16-bit Windows were as follows:

wParam	=	menu item ID	if selection is a plain menu item
	=	pop-up menu handle	if selection is a pop-up menu
lParam	=	MAKELPARAM(flags, parent menu handle)	

The `MenuHelp` function parsed the parameters of the `WM_MENUSELECT` message in conjunction with a table describing the mapping between menu items and help strings, displaying the selected string in the status bar. The information was provided in the confusing format of an array of `UINT`s that took the following format (expressed in pseudo-C):

```
struct MENUHELPPOPUPUINTS {
 UINT uiPopupStringID;
 HMENU hmenuPopup;
};
struct MENUHELPUINTS {
 UINT uiMenuItemIDStringOffset;
 UINT uiMenuIndexStringOffset;
 MENUHELPPOPUPUINTS rgwPopups[];
};
```

The `uiMenuItemIDStringOffset` specifies the value to add to the menu ID to convert it to a string ID that is to be displayed in the status bar. For example, if you had

```
    MENUITEM "&New\tCtrl+N"    ,200
```

in your menu template, and you specified an offset of `1000`, then the `MenuHelp` function will look for the help string as string identifier `200 + 1000 = 1200`:

```
STRINGTABLE BEGIN
1200 "Opens a new blank document."
END
```

The `uiMenuIndexStringOffset` does the same thing for pop-up menus that were direct children of the main menu, but since pop-up menus in 16-bit Windows didn't have IDs, the zero-based menu index was used instead. For example, if your menu had the top-level structure

```
BEGIN
  POPUP "&File"
  BEGIN
  ...
  END
  POPUP "&View"
  BEGIN
  ...
  END
END
```

and you specified a `uiMenuIndexStringOffset` of `800`, then the string for the File menu was expected to be at `0 + 800 = 800` and the string for the View menu was expected at `1 + 800 = 801`.

```
STRINGTABLE BEGIN
800 "Contains commands for working with the current document."
801 "Contains edit commands."
END
```

The last case is a pop-up menu that is a grandchild (or deeper descendant) of the main menu. As we saw above, the `WM_MENUSELECT` message encoded the handle of the pop-up menu rather than its ID. This handle was looked up in the variable-length array of `MENUHELPPOPUPUINTS` elements (terminated by a `{0, 0}` entry). Notice that the second member of the `MENUHELPPOPUPUINTS` structure is an `HMENU` rather than a `UINT`. But in 16-bit Windows, `sizeof(HMENU) == sizeof(UINT) == 2`, and 16-bit code (such as the `WM_MENUSELECT` message) relied heavily on coincidences like this.

If a pop-up window had the handle, say, `(HMENU)0x1234`, the `MenuHelp` function would look for a `MENUHELPPOPUPUINTS` entry which had a `hMenuPopup` equal to `(HMENU)0x1234`, at which point it would use the corresponding `uiPopupStringID` as the help string.

Let's take a look at one of these in practice. Here's a menu and a corresponding string table:

```
1 MENU
BEGIN
  POPUP "&File"
  BEGIN
    MENUITEM "&New\tCtrl+N"    ,200
    MENUITEM "&Open\tCtrl+O"   ,201
    MENUITEM "&Save\tCtrl+S"   ,202
    MENUITEM "Save &As"        ,203
    MENUITEM ""                ,-1
    MENUITEM "E&xit"           ,204
  END
  POPUP "&View"
  BEGIN
    MENUITEM "&Status bar"     ,240
    MENUITEM "&Full screen"    ,230
    POPUP "Te&xt Size"
    BEGIN
      MENUITEM "&Large"        ,225
      MENUITEM "&Normal"       ,226
      MENUITEM "&Small"        ,227
    END
  END
END
STRINGTABLE BEGIN
 800 "Contains commands for loading and saving files."
 801 "Contains commands for manipulating the view."
1200 "Opens a new blank document."
1201 "Opens an existing document."
1202 "Saves the current document."
1203 "Saves the current document with a new name."
1225 "Selects large font size."
1226 "Selects normal font size."
1227 "Selects small font size."
1230 "Maximizes the window to full screen."
1240 "Shows or hides the status bar."
2006 "Specifies the relative size of text."
END
```

Notice that there was no requirement that the menu item identifiers be consecutive. All that the `MenuHelp` function cared about is that the relationship between the menu item identifiers and the help strings was in the form of a simple offset.

The table that connects the menu to the string table goes like this:

```
UINT rguiHelp[] = {
  1000,    // uiMenuItemIDStringOffset
   800,    // uiMenuIndexStringOffset
  2006, 0, // uiPopupStringID, placeholder
     0, 0  // end of MENUHELPPOPUPUINTS
};
```

Since there is a grandchild pop-up menu, we created a placeholder entry that will be filled in with the menu handle at run time:

```
BOOL
OnCreate(HWND hwnd, LPCREATESTRUCT lpcs)
{
  HMENU hmenuMain = GetMenu(hwnd);
  HMENU hmenuView = GetSubMenu(hmenuMain, 1);
  HMENU hmenuText = GetSubMenu(hmenuView, 2);
  rguiHelp[3] = (UINT)hmenuText;
  g_hwndStatus = CreateWindow(STATUSCLASSNAME, NULL,
        WS_CHILD | CCS_BOTTOM | SBARS_SIZEGRIP | WS_VISIBLE,
        0, 0, 0, 0, hwnd, (HMENU)100, g_hinst, 0);
  return g_hwndStatus != NULL;
}
```

We locate the "Text Size" menu and put its menu handle into the `rguiHelp` array so that the `MenuHelp` command can find it. The window procedure would then include the line:

```
...
    case WM_MENUSELECT:
      MenuHelp(uiMsg, wParam, lParam, GetMenu(hwnd),
               g_hinst, g_hwndStatus, rguiHelp);
      break;
...
```

That last step finally connects all the pieces. When the `WM_MENUSELECT` message arrives, the `MenuHelp` function looks at the item that was selected, uses it to look up the appropriate string resource, loads the resource from the provided `HINSTANCE` and displays it in the status bar.

To make the sample complete, we need to do a little extra bookkeeping:

```
HWND g_hwndStatus;
void
OnSize(HWND hwnd, UINT state, int cx, int cy)
{
  MoveWindow(g_hwndStatus, 0, 0, cx, cy, TRUE);
}
// change to InitApp
    wc.lpszMenuName = MAKEINTRESOURCE(1);
```

(I'd invite you to code up this sample 16-bit program and run it, but I doubt anybody would be able to take me up on the invitation since very few people have access to a 16-bit compiler for Windows any more.)

This method works great for 16-bit code. But look at what happened during the transition to 32-bit Windows: The parameters to the `WM_MENUSELECT` message had to change since menu handles are 32-bit values. There was no room in two 32-bit window message parameters to

shove 48 bits of data (two window handles and 16 bits of flags). Something had to give, and what gave was the pop-up menu handle. Instead of passing the handle, the index of the pop-up menu was passed in the message parameters. This did not result in any loss of data since the menu handle could be recovered by passing the parent menu handle and the pop-up menu index to the `GetSubMenu` function. The repacking of the parameters thus goes like this:

LOWORD(wParam)	=	menu item ID	if selection is a plain menu item
	=	pop-up menu index	if selection is a pop-up menu
HIWORD(wParam)	=	flags	
lParam	=	parent menu handle	

The array of `UINT`s therefore changed its meaning to match the new message packing:

```
struct MENUHELPPOPUPUINTS {
 UINT uiPopupStringID;
 UINT uiPopupIndex;
};
struct MENUHELPUINTS {
 UINT uiMenuItemIDStringOffset;
 UINT uiMenuIndexStringOffset;
 MENUHELPPOPUPUINTS rgwPopups[];
};
```

The advantage of changing the value from an `HMENU` to a `UINT` index is that the array does not need to be modified at run time. Okay, let's actually try this. Start with the scratch program, attach the resources I gave above, and use the following help array and code:

```
UINT rguiHelp[] = {
  1000,    // uiMenuItemIDStringOffset
   800,    // uiMenuIndexStringOffset
  2006, 2, // uiPopupStringID, uiPopupMenuIndex
  0,0      // end of MENUHELPPOPUPUINTS
};
HWND g_hwndStatus;
void
OnSize(HWND hwnd, UINT state, int cx, int cy)
{
  MoveWindow(g_hwndStatus, 0, 0, cx, cy, TRUE);
}
BOOL
OnCreate(HWND hwnd, LPCREATESTRUCT lpcs)
{
  g_hwndStatus = CreateWindow(STATUSCLASSNAME, NULL,
        WS_CHILD | CCS_BOTTOM | SBARS_SIZEGRIP | WS_VISIBLE,
        0, 0, 0, 0, hwnd, (HMENU)100, g_hinst, 0);
  return g_hwndStatus != NULL;
}
// add to WndProc
    case WM_MENUSELECT:
      MenuHelp(uiMsg, wParam, lParam, GetMenu(hwnd),
               g_hinst, g_hwndStatus, rguiHelp);
      break;
// change to InitApp
    wc.lpszMenuName = MAKEINTRESOURCE(1);
```

Notice that this is identical to the code needed for the 16-bit `MenuHelp` function, except that we didn't initialize the `UINT` array with the pop-up menu handle.

Run this program and see how the help text in the status bar changes based on which menu item you have seleced. The `MenuHelp` function also knows about the commands on the System menu and provides appropriate help text for those as well.

Wow, this sounds like a neat function. Why then did I say that you probably will decide that you don't want to use it? Let's look at the limitations of the `MenuHelp` function.

First, notice that all the help strings for the menu must come from the same `HINSTANCE`. Furthermore, the offset from the menu item identifier to the help string must remain constant across all menu items. These two points mean that you cannot build a menu out of pieces from multiple DLLs since you can pass only one `HINSTANCE` and offset.

Second, the fixed offset means that you cannot have menus whose content expands dynamically, because you won't have help strings for the dynamic content. What's worse, if the dynamically-added menu item identifiers happen to, when added to the fixed offset, coincide with some other string resource, that other string resource will be used as the help string! For example, in our example above, if we dynamically added a menu item whose

identifier is 1000, then the `MenuHelp` function would look for the string whose resource identifier is `1000 + 1000 = 2000`. And if you happened to have some other string at position 2000 for some totally unrelated reason, that string will end up as the menu help.

But hopefully you've spotted the fatal flaw in the `MenuHelp` function by now: That pop-up menu index. I carefully designed this example to avoid the flaw. The index of the "Text Size" pop-up menu is 2, and it is the only pop-up menu whose index is 2. (The "File" menu is index 0 and the "View" menu is index 1.) In real life, of course, you do not have the luxury of fiddling the menus to ensure that no two pop-up menus have the same index. And when they end up with the same index, the help strings get all confused since the `MenuHelp` function can't tell which of the multiple "second pop-up menu" you wanted to use string 2006 for.

Could this be fixed? If you tried to return to the old `HMENU`-based way of identifying pop-ups, you'd run into some new problems: First, the introduction of 64-bit Windows means that you cannot just cast an `HMENU` to a `UINT` because an `HMENU` is a 64-bit value and `UINT` is only 32 bits. You could work around this by expanding the parameter to the `MenuHelp` function to be an array of `UINT_PTR` values instead of an array of `UINT`s, but that's not the only problem.

The `HMENU`-base mechanism supports only one window at a time since the global array needs to be edited for each client. To make it support multiple windows, you would have to make a copy of the global array and edit the private copy. To avoid making a private copy, you would have to come up with some other way of specifying the pop-up window.

Now, you could spend even more time trying to come up with a solution to the `HMENU` problem, but that still leaves the other problems we discussed earlier. Trying to salvage a `MenuHelp`-like solution to those problems leads to even more complicated mechanisms for expressing the relationship between a menu item identifier and the corresponding help string. Eventually, you come to the point where the general solution is too complicated for its own good and you're better off just coming up with an ad-hoc solution for your particular situation, like we did when we added menu help to our hosted shell context menus.

(The only people I see using the `MenuHelp` function ignore dealing with pop-up menus and use only the first two `UINT`s, thereby avoiding the whole `HMENU` problem.)

Raymond Chen

Follow