

# Creating Context Menu Handlers

## Creating Context Menu Handlers

When a user right-clicks a Shell object, the Shell displays its context menu. For file system objects there are a number of standard items, such as Cut and Copy, that are on the menu by default. If the object is a file that is a member of a class, additional items can be specified in the registry. Finally, the Shell checks the registry to see if the file class is associated with any context menu handlers. If it is, the Shell queries the handler for additional context menu items.

Adding entries to Windows Explorer Shell context (right-click) menu easily &ndash; How it may be done ?

Add entries to Windows Explorer Shell context menu easily with Windows Explorer Shell Context Menu. This powerful component for custom items adding to Windows Explorer Shell context menu will add all your custom application entries to Explorer Shell context menu. This .Net component , C++ and Visual Basic .NET support include detailed C# / VB.NET samples, tutorials , user-friendly manuals and support all you may need to add your entries to Explorer context menu :

- Add all your items to Windows Explorer Shell context menu to be shown on any Windows OS (all operating systems are supported &ndash; XP, Vista, x64 of all types , etc.)
- Add any type of items to Windows Explorer Shell context menu to be shown in any way - with your custom caption and icon, as separator or sub-menu
- Add items of any types to Explorer Shell context menu to be shown for all types of files or shown only for computer files of particular type (for example, only for .PDF .TXT , .MP3,.WMA,.AAC , .AVI files)
- Add items to Windows Explorer Shell context menu, sub-menus, sub-sub-menus, sub-menus of unlimited depth and add to Explorer context menu entries of all types

Windows Explorer Shell Context Menu - is a .Net component that support all you may need to insert your items to the Explorer Shell context menu - in a fast and a very easy way. Add all your items to Explorer context menu right now &ndash; add items to context menu fast and exactly as you want :

A context menu handler is a shell extension handler that adds commands to an existing context menu. Context menu handlers are associated with a particular file class and are called any time a context menu is displayed for a member of the class. While you can add items to a file class context menu with the registry, the items will be the same for all members of the class. By implementing and registering such a handler, you can dynamically add items to an object's context menu, customized for the particular object.

A closely related Shell extension handler is the drag-and-drop handler. It is a context menu handler that adds items to the context menu that is displayed when a user drags and drops a file with the right mouse button.

The procedures for implementing and registering a Shell extension handler are discussed in Creating Shell Extension Handlers. This document focuses on those aspects of implementation that are specific to context menu handlers.

The following topics are discussed.

- \* How Context Menu Handlers Work
- \* Registering a Context Menu Handler
- \* Implementing IContextMenu
- \* Creating Drag-and-Drop Handlers

## How Context Menu Handlers Work

Context menu handlers are a type of Shell extension handler. Like all such handlers, they are in-process Component Object Model (COM) objects implemented as DLLs. Context menu handlers must export two interfaces in addition to IUnknown: IShellExtInit and IContextMenu. Optionally, a context menu handler can also export IContextMenu2 and

IContextMenu3. These interfaces handle the messaging needed to implement owner-drawn menu items. For more information on owner-drawn menu items, see the [Creating Owner-Drawn Menu Items](#) section under [Using Menus](#).

The IShellExtInit interface is used by the Shell to initialize the handler. When the Shell calls IShellExtInit::Initialize, it passes in a data object with the object's name and the pointer to an item identifier list (PIDL) of the folder that contains the file. The hKeyProgID parameter is not used with context menu handlers. The IShellExtInit::Initialize method must extract the file name from the data object and store the name and the folder's PIDL for later use. For further details, see [Implementing IShellExtInit](#).

The remainder of the operation takes place through the handler's IContextMenu interface. The Shell first calls IContextMenu::QueryContextMenu. It passes in an HMENU handle that the method can use to add items to the context menu. If the user selects one of the commands, IContextMenu::GetCommandString is called to retrieve the Help string that will be displayed on the Microsoft Windows Explorer status bar. If the user clicks one of the handler's items, the Shell calls IContextMenu::InvokeCommand. The handler can then execute the appropriate command.

### Registering a Context Menu Handler

Context menu handlers are associated with either a file class or a folder. For file classes, the handler is registered under the following subkey.

```
* HKEY_CLASSES_ROOT
  o Program ID
    + shellex
      # ContextMenuHandlers
```

Create a subkey under ContextMenuHandlers named for the handler, and set the subkey's default value to the string form of the handler's class identifier (CLSID) GUID.

You can also associate a context menu handler with different kinds of folders. Register the handler the same way you would for a file class, but under the following subkey, where FolderType is the name of the type of folder.

```
* HKEY_CLASSES_ROOT
  o FolderType
    + shellex
      # ContextMenuHandlers
```

For a discussion of how to register Shell extension handlers and more information about which folder types you can register handlers for, see [Registering Shell Extension Handlers](#).

If a file class has a context menu associated with it, double-clicking an object normally launches the default command. The handler's IContextMenu::QueryContextMenu method is not called. To specify that the handler's IContextMenu::QueryContextMenu method should be called when an object is double-clicked, create a shellex\MayChangeDefaultMenu subkey under the handler's CLSID key. When an object associated with the handler is double-clicked, IContextMenu::QueryContextMenu will be called with the CMF\_DEFAULTONLY flag set in the uFlags parameter.

**Note** Setting the MayChangeDefaultMenu key forces the system to load the handler's DLL when an associated item is double-clicked. If your handler does not change the default verb, you should not set MayChangeDefaultMenu. Doing so causes the system to load your DLL unnecessarily. Context menu handlers should set this key only if they might need to change the context menu's default verb.

The following example illustrates registry entries that enable a context menu handler for an example .myp file class. The handler's CLSID key includes a MayChangeDefaultMenu subkey to guarantee that the handler is called when the user double-clicks a related object.

```
* HKEY_CLASSES_ROOT
  o
    o .myp
      (Default) = MyProgram.1
    o CLSID
      + {00000000-1111-2222-3333-444444444444}
        # InProcServer32
          (Default) = C:\MyDir\MyCommand.dll
```

```

        ThreadingModel = Apartment
        # shellex
        * MayChangeDefaultMenu
    o MyProgram.1

        (Default) = MyProgram Application
        + shellex
        # ContextMenuHandler

        MyCommand = {00000000-1111-2222-3333-444444444444}

```

## Implementing IContextMenu

Most of the operation described in [How Context Menu Handlers Work](#) is handled by the IContextMenu interface. This section discusses how to implement its three methods.

### QueryContextMenu Method

The Shell calls IContextMenu::QueryContextMenu to allow the context menu handler to add its menu items to the menu. It passes in the HMENU handle in the hmenu parameter. The indexMenu parameter is set to the index to be used for the first menu item that is to be added.

Any menu items that are added by the handler must have identifiers that fall between the idCmdFirst and idCmdLast parameters. Typically, the first command identifier is set to idCmdFirst, which is incremented by one (1) for each additional command. This practice ensures that you don't exceed idCmdLast and maximizes the number of available identifiers in case the Shell calls more than one handler.

An item identifier's command offset is the difference between the identifier and idCmdFirst. Store the offset of each item that your handler adds to the context menu because the Shell might use it to identify the item if it subsequently calls IContextMenu::GetCommandString or IContextMenu::InvokeCommand.

You should also assign a verb to each command you add. A verb is a language-independent string that can be used instead of the offset to identify the command when IContextMenu::InvokeCommand is called. It is also used by functions such as ShellExecuteEx to execute context menu commands.

Only three of the flags that can be passed in through the uFlags parameter are relevant to context menu handlers.

- CMF\_DEFAULTONLY The user has selected the default command, usually by double-clicking the object. IContextMenu::QueryContextMenu should return control to the Shell without modifying the menu.
- CMF\_NODEFAULT No item in the menu should be the default item. The method should add its commands to the menu.
- CMF\_NORMAL The context menu will be displayed normally. The method should add its commands to the menu.

Use either InsertMenu or InsertMenuItem to add menu items to the list. Then return an HRESULT value with the severity set to SEVERITY\_SUCCESS. Set the code value to the offset of the largest command identifier that was assigned, plus one (1). For example, assume that idCmdFirst is set to 5 and you add three items to the menu with command identifiers of 5, 7, and 8. The return value should be MAKE\_HRESULT(SEVERITY\_SUCCESS, 0, 8 - 5 + 1).

The following example shows a simple implementation of IContextMenu::QueryContextMenu that inserts a single command. The identifier offset for the command is IDM\_DISPLAY, which is set to zero. The m\_pszVerb and m\_pwszVerb variables are private variables used to store the associated language-independent verb string in both Unicode and ANSI formats.

Copy Code

```

#define IDM_DISPLAY 0

STDMETHODIMP CMenuExtension::QueryContextMenu(HMENU hMenu,
        UINT indexMenu,
        UINT idCmdFirst,
        UINT idCmdLast,
        UINT uFlags)
{
    HRESULT hr;

    if(!(CMF_DEFAULTONLY & uFlags))
    {
        InsertMenu(hMenu,
            indexMenu,

```

```
MF_STRING | MF_BYPOSITION,
idCmdFirst + IDM_DISPLAY,
"&Display File Name");
```

```
// TODO: Add error handling to verify HRESULT return values.
```

```
hr = StringCbCopyA(m_pszVerb, sizeof(m_pszVerb), "display");
hr = StringCbCopyW(m_pwszVerb, sizeof(m_pwszVerb), L"display");
```

```
return MAKE_HRESULT(SEVERITY_SUCCESS, 0, USHORT(IDM_DISPLAY + 1));
```

```
}
```

```
return MAKE_HRESULT(SEVERITY_SUCCESS, 0, USHORT(0));
```

```
}
```

## GetCommandString Method

If a user highlights one of the items added by a context menu handler, the handler's `IContextMenu::GetCommandString` method is called to request a Help text string that will be displayed on the Windows Explorer status bar. This method can also be called to request the verb string that is assigned to a command. Either ANSI or Unicode verb strings can be requested.

The `idCmd` parameter holds the identifier offset of the command that was defined when `IContextMenu::QueryContextMenu` was called. If a Help string is requested, `uFlags` will be set to either `GCS_HELPTEXTA` or `GCS_HELPTEXTW`, depending on whether an ANSI or Unicode string is desired. Copy the Help string to the `pszName` buffer, casting it to an `LPWSTR` for the Unicode case. The verb string is requested by setting `uFlags` to either `GCS_VERBA` or `GCS_VERBW`. Copy the appropriate string to `pszName`, just as with the Help string. The `GCS_VALIDATEA` and `GCS_VALIDATEW` flags are not used by context menu handlers.

The following example shows a simple implementation of `IContextMenu::GetCommandString` that corresponds to the `IContextMenu::QueryContextMenu` example given in the `QueryContextMenu Method` section. Since the handler adds only one menu item, there is only one set of strings that can be returned. The method simply tests whether `idCmd` is valid and, if it is, returns the requested ANSI or Unicode string.

The `StringCchCopyN` function is used to copy the requested string to `pszName` to ensure that the copied string doesn't exceed the size of the buffer given by `uMaxNameLen`. The ANSI and Unicode versions of the function are used explicitly, because the format defined when the handler is compiled might not match the requested format.

Copy Code

```
STDMETHODIMP CMenuExtension::GetCommandString(UINT idCommand,
        UINT uFlags,
        LPUINT lpReserved,
        LPSTR pszName,
        UINT uMaxNameLen)
```

```
{
```

```
    HRESULT hr = E_INVALIDARG;
```

```
    if(idCommand != IDM_DISPLAY)
```

```
    {
```

```
        return hr;
```

```
    }
```

```
    switch(uFlags)
```

```
    {
```

```
        case GCS_HELPTEXTA:
```

```
            hr = StringCchCopyNA(pszName,
                lstrlen(pszStr)/sizeof(pszStr(0)),
                "Display File Name",
                uMaxNameLen);
```

```
            break;
```

```
        case GCS_HELPTEXTW:
```

```
            hr = StringCchCopyNW((LPWSTR)pszName,
                lstrlen(pszStr)/sizeof(pszStr(0)),
```

```

        L"Display File Name",
        uMaxNameLen);
    break;

case GCS_VERBA:
    hr = StringCchbCopyNA(pszName,
        lstrlen(pszStr)/sizeof(pszStr(0)),
        m_pszVerb,
        uMaxNameLen);
    break;

case GCS_VERBW:
    hr = StringCchCopyNW((LPWSTR)pszName,
        lstrlen(pszStr)/sizeof(pszStr(0)),
        m_pwszVerb,
        uMaxNameLen);
    break;

default:
    hr = S_OK;
    break;
}
return hr;
}

```

### InvokeCommand Method

This method is called when a user clicks a menu item to tell the handler to run the associated command. The `pici` parameter points to a structure that contains the needed information.

Although `pici` is declared in `Shlobj.h` as a `CMINVOKECOMMANDINFO` structure, in practice it often points to a `CMINVOKECOMMANDINFOEX` structure. This structure is an extended version of `CMINVOKECOMMANDINFO` and has several additional members that allow Unicode strings to be passed.

Check the `cbSize` member of `pici` to determine which structure was passed in. If it is a `CMINVOKECOMMANDINFOEX` structure and the `fMask` member has the `CMIC_MASK_UNICODE` flag set, cast `pici` to `CMINVOKECOMMANDINFOEX`. This allows your application to use the Unicode information contained in the last five members of the structure.

The structure's `lpVerb` or `lpVerbW` member is used to identify the command to be executed. There are two ways to identify commands.

- \* The command's verb string
- \* The command's identifier offset

To distinguish between these two cases, check the high-order word of `lpVerb` for the ANSI case or `lpVerbW` for the Unicode case. If the high-order word is nonzero, `lpVerb` or `lpVerbW` holds a verb string. If the high-order word is zero, the command offset is in the low-order word of `lpVerb`.

The following example shows a simple implementation of `IContextMenu::InvokeCommand` that corresponds to the `IContextMenu::QueryContextMenu` and `IContextMenu::GetCommandString` samples given in the previous sections. The method first determines which structure is being passed in. It then determines whether the command is identified by its offset or verb. If `lpVerb` or `lpVerbW` holds a valid verb or offset, the method displays a message box.

Copy Code

```

STDMETHODIMP CShellExtension::InvokeCommand(LPCMINVOKECOMMANDINFO lpcmi)
{
    BOOL fEx = FALSE;
    BOOL fUnicode = FALSE;

    if(lpcmi->cbSize == sizeof(CMINVOKECOMMANDINFOEX))
    {
        fEx = TRUE;
        if((lpcmi->fMask & CMIC_MASK_UNICODE))
        {

```

```

        fUnicode = TRUE;
    }
}

if( !fUnicode && HIWORD(lpcmi->lpVerb))
{
    if(StrCmpIA(lpcmi->lpVerb, m_pszVerb))
    {
        return E_FAIL;
    }
}

else if( fUnicode && HIWORD(((CMINVOKECOMMANDINFOEX *) lpcmi)->lpVerbW))
{
    if(StrCmpIW(((CMINVOKECOMMANDINFOEX *)lpcmi)->lpVerbW, m_pwszVerb))
    {
        return E_FAIL;
    }
}

else if(LOWORD(lpcmi->lpVerb) != IDM_DISPLAY)
{
    return E_FAIL;
}

else
{
    MessageBox(lpcmi->hwnd,
        "The File Name",
        "File Name",
        MB_OK|MB_ICONINFORMATION);
}

return S_OK;
}

```

## Creating Drag-and-Drop Handlers

When a user right-clicks a Shell object to drag an object, a context menu is displayed when the user attempts to drop the object. The following illustration shows a typical drag-and-drop context menu.

### Drag-and-drop context menu

A drag-and-drop handler is a context menu handler that can add items to this context menu. Drag-and-drop handlers are typically registered under the following subkey.

```

* HKEY_CLASSES_ROOT
  o Directory
    + shellex
      # DragDropHandlers

```

Add a subkey under the DragDropHandlers subkey named for the drag-and-drop handler, and set the subkey's default value to the string form of the handler's CLSID GUID. The following example enables the MyDD drag-and-drop handler.

```

* HKEY_CLASSES_ROOT
  o Directory
    + shellex
      # DragDropHandlers
        * MyDD

        (Default) = {MyDD CLSID GUID}

```

The basic procedure for implementing a drag-and-drop handler is the same as for conventional context menu handlers.

However, context menu handlers normally use only the IDataObject pointer passed to the handler's IShellExtInit::Initialize method to extract the object's name. A drag-and-drop handler could implement a more sophisticated data handler to modify the behavior of the dragged object.