# EXHIBIT J

# # Extending the Chicago Shell

Kyle Marsh
Microsoft Developer Network Technology Group

Created: May 10, 1994

## Abstract

Applications can extend the shell for the next version of the Microsoft® Windows™ operating system (code-named Chicago) in a number of ways. Shell extensions enhance the basic functionality of the shell—they give users choices for manipulating file objects or provide additional information. Extending the shell allows applications to simplify the task of browsing through the Chicago file system and networks. Shell extensions also give users easier access to tools for manipulating objects in the file system. This article explains how an application creates shell extensions and how Chicago interacts with these extensions.

## Introduction

In the next version of the Microsoft® Windows™ operating system (code-named Chicago), applications can extend the shell in a number of ways. Extending the shell allows applications to simplify the task of browsing through the file system and networks. Shell extensions also give users easier access to tools that manipulate objects they find in the file system. For example, a shell extension can assign an icon to each file or add specific commands for the file. These commands are added to the context menu the shell displays for the file (when the user clicks the object with the right mouse button) and to the File menu (when the user selects the object with the left mouse button and opens the File menu).

Another type of shell extension is a name-space browser, which allows users to browse the contents of objects using the shell's familiar explorer view. I will be discussing name-space browsers in detail in a future article.

This article explains how an application creates shell extensions and how Chicago interacts with these extensions. The design of Chicago's shell extensions is based on the Component Object Model in object linking and embedding (OLE) version 2.0. The shell accesses objects via interfaces, and applications implement those interfaces as *shell extension dynamic-link libraries (DLLs),* which are similar to the In-Proc Server DLLs in OLE 2.0. Because the new 32-bit OLE to be included in Chicago was not available to the Chicago shell team early enough to use in the first beta, the beta shell does not use OLE to load shell extensions. However, since it uses the same mechanism that OLE uses when it loads In-Proc servers, shell extensions will continue to work when the shell starts using OLE to load shell extensions.

Please note that this article is based on preliminary information that is subject to change before the final version of Chicago.

## Definitions

# msdn_shellext

**File Object**

An item within the shell. The most familiar file objects are files and directories. However, a file object may not actually be a part of the file system—it may only appear that way. For example, printers, Control Panel applets, network workgroups, servers, and shares are all considered file objects.

**File Class**

The file object type. Each file object is a member of a file class. The file class also refers to the code that "owns" the manipulation of files of that type. For example, text files and Microsoft Word documents are examples of file classes. Each file class has specific shell extensions. The shell loads these extensions based on the file class of the object it is acting on.

**Handler**

The code that implements a shell extension.

## Shell Extensions

Shell extensions enhance the basic functionality of the shell by providing additional choices for manipulating file objects or additional information. There are five shell extensions:

- **Context-menu handlers:** These handlers add items to the context menu for a particular file object. (The context menu is displayed when the user clicks a file object with the right mouse button.)
- **Drag-drop handlers:** These are context-menu handlers that are accessed when a user drops an object after dragging it to a new location.
- **Icon handlers:** These handlers usually add instance-specific icons for file objects. They can also be used to add icons for file classes.
- **Property-sheet handlers:** These handlers add pages (specific to a file class or file object) to the property sheet dialog box the shell displays for a file object.
- **Copy-hook handlers:** An application can use these handlers to prevent a folder or printer object from being copied, moved, deleted, or renamed.

## Registering a Shell Extension

All shell extensions are registered in the Windows registry. Each handler must register its class ID under the HKEY_CLASSES_ROOT\Clsid key in the registry. The Clsid key is a DCE RPC globally unique identifier (GUID) such as {00020810-0000-0000-C000-000000000046} and is generated with the UUIDGEN tool. Within this key, the handler adds an InProcServer32 key that gives the location of the handler's DLL. It is best to give the complete path for the handler; using the complete path keeps the handler independent of the current path and speeds up load times.

Applications that create and maintain files such as spreadsheets, word-processing applications, and databases usually register two additional entries in the registry: a file association entry and a key name.

The file association entry maps a file extension to a program identifier. For example, a word-processing application might register the following key under HKEY_CLASSES_ROOT:

```
HKEY_CLASSES_ROOT
   .doc=AWordProcessor
```

The key name ( *.doc* above ) specifies the file extension, and the value of the key (*AWordProcessor*) denotes the key name that contains the information about the application that handles that file extension. The value of the key name is the second registry entry made by an application that handles files. For example:

```
HKEY_CLASSES_ROOT
   AWordProcessor = A Word Processor
    shell = open print preview
        open
            command = c:\aword\aword.exe %1
        print
            command = c:\aword\aword.exe %1
        printTo
            command = c:\aword\aword.exe %1 %2
        preview = Pre&view
            command = c:\aword\aword.exe /r %1
    shellex
        ContextMenuHandlers = ExtraMenu
            ExtraMenu = {00000000-1111-2222-3333-00000000000001}
        PropertySheetHandlers = SummaryInfo
            SummaryInfo = {00000000-1111-2222-3333-00000000000002}
        IconHandler = {00000000-1111-2222-3333-00000000000003}
    DefaultIcon = %1
```

The commands in the shell section of the registry are added to the context menus of the documents that are associated with this type. The **printTo** entry is also used when the user drops a document on a specific printer. As in Windows 3.1, you can use dynamic data exchange (DDE) strings in these definitions. To avoid conflicts with other classes, you must use real GUIDs, not the phony strings I used above.

The new **shellex** key contains the information the shell uses to associate a shell extension handler with a file type.

The shell also uses several other special keys—*, Folder, Drives, Printers, and keys for network providers—under HKEY_CLASSES_ROOT to look for shell extensions.

- The * key can be used to register handlers that the shell will call when it creates a context menu or property sheet for any and every file object. Thus, if the registry contains the following:

  ```
  HKEY_CLASSES_ROOT
     * = *
        shellex
          ContextMenuHandlers = ExtraMenu
              ExtraMenu = {00000000-1111-2222-3333-00000000000001}
          PropertySheetHandlers = SummaryInfo
              SummaryInfo = {00000000-1111-2222-3333-00000000000002}
  ```

  the shell uses instances of the **ExtraMenu** and **SummaryInfo** handlers to add to the context menus and property sheets for every file object.

- The shell uses the Folder key to allow applications to register shell extensions for directories in the file system. An application can register context-menu handlers, copy-hook handlers, and property-sheet handlers in the same way it registers these handlers for the * key. An additional handler, the drag-drop handler, applies only to the Folder and Printers keys. For example:

```
Folder = Folder
   shellex
       DragDropHandlers = ADDHandler
           ADDHandler = {00000000-1111-2222-3333-00000000000004}
       CopyHookHandlers = ACopyHandler
           ACopyHandler = {00000000-1111-2222-3333-00000000000005}
```

- The Drives key allows the same registrations as the Folder key, but is called only for root paths, for example: C:\.
- The Printers key allows the same registrations as the Folder key, but uses additional handlers for printer events, deletion or removal of printers (via the copy-hook handler), and printer properties (with property-sheet handlers and context-menu handlers).
- The shell recognizes context-menu handlers and property-sheet handlers for only one network provider name: the "Microsoft Network" key.

## How the Shell Accesses Shell Extension Handlers

The shell uses two interfaces to initialize instances (objects created by IClassFactory::CreateInstance) of shell extensions: **IShellExtInit** and **IPersistFile**. The shell uses the **IShellExtInit** interface to initialize context-menu handlers, drag-drop handlers, and property-sheet handlers. The shell uses **IPersistFile** to initialize instances of icon handlers. This interface is same as the **IPersistFile** interface in OLE 2.0.

### IShellExtInit

The shell uses this interface to initialize instances of context-menu handlers, drag-drop handlers, and property-sheet handlers. It adds one method, **Initialize**, to the standard **IUnknown** interface.

#### Initialize

**Syntax:**

```
STDMETHOD(Initialize(LPCITEMIDLIST pidlFolder,
              LPDATAOBJECT lpdobj, HKEY hkeyProgID)
```

**Parameters:**

- *pidlFolder*: A pointer to an ID list that points to either the parent folder of selected objects (in the case of the context-menu handlers) or the target folder (in the case of drag-drop handlers).
- *lpdobj*: A pointer to the data object for the file object(s). Normally, this is the selected object in the shell. Handlers should call the object's **AddRef** method in the **IShellExtInit Initialize** method and the object's **Release** method in the **IShellExtInit Release** method.

- *hkeyProgID*: Handle to the registry key for the primary object in the shell (usually the object with the focus). This parameter may be NULL. The handler can call **RegOpenKey** with *hkeyProgID* as the *hkey* parameter to open the key.

The handler should keep a copy of these parameters if it needs them later. For example:

```
//-----------------------------------------------------------------------------
//
//    Shell Extension Sample's IShellExtInit Interface
//
//-----------------------------------------------------------------------------
STDMETHODIMP SHE_ShellExtInit_Initialize(LPSHELLEXTINIT psxi,
      LPCITEMIDLIST pidlFolder,
      LPDATAOBJECT pdtobj, HKEY hkeyProgID)
{
   PSHELLEXTSAMPLE this = PSXI2PSMX(psxi);

    // Initialize can be called more than once.
    if (this->_pdtobj) {
      this->_pdtobj->lpVtbl->Release(this->_pdtobj);
    }

    if (this->_hkeyProgID) {
      RegCloseKey(this->_hkeyProgID);
    }

    // Duplicate the pdtobj pointer, then update the usage count.
    if (pdtobj) {
      this->_pdtobj = pdtobj;
      pdtobj->lpVtbl->AddRef(pdtobj);
    }

    // Duplicate the registry handle.
    if (hkeyProgID) {
      RegOpenKey(hkeyProgID, NULL, &this->_hkeyProgID);
    }

    return NOERROR;
}
```

Each shell extension must implement three routines: an entry point (often called **DllMain** or **LibMain**), **DllCanUnloadNow**, and **DllGetClassObject**.

The entry point is standard for any 32-bit DLL; it usually needs to record the handle to the DLL for future use. The handle must be stored in a per-instance variable. For example:

```
//-------------------------------------------------------------
// LibMain
//-------------------------------------------------------------
BOOL APIENTRY LibMain(HANDLE hDll, DWORD dwReason, LPVOID lpReserved)
{
    switch(dwReason)
    {
    case DLL_PROCESS_ATTACH:
      g_hmodThisDll = hDll;      // g_hmodThisDll must be per-instance.
      break;
    case DLL_PROCESS_DETACH:
     break;
```

```
        case DLL_THREAD_DETACH:
            break;

        case DLL_THREAD_ATTACH:
        default:
          break;
        } // end switch()

        return TRUE;
}
```

The **DllCanUnloadNow** and **DllGetClassObject** functions are essentially the same as they would be for any In-Proc Server DLL in OLE 2.0. **DllCanUnloadNow** is straightforward:

```
//---------------------------------------------------------------------------
// DllCanUnloadNow
//---------------------------------------------------------------------------

STDAPI DllCanUnloadNow(void)
{
   // g_cRefThisDll must be placed in the per-instance data section.

   return ResultFromScode((g_cRefThisDll==0) ? S_OK : S_FALSE);
}
```

**DllGetClassObject** needs to expose the class factory for the object in the DLL. For more information on exposing the class factory, please see the *OLE 2.0 Programmer's Reference*, Vol. 1, Chapter 5, "IClassFactory Interface" (Product Documentation, SDKs, OLE 2) or *Inside OLE 2*, Chapter 4, "Implementing a Component Object and a Server" (Books and Periodicals) in the Development Library.

```
//==========================================================================
// CDefClassFactory class
//==========================================================================
extern CDefClassFactory * NEAR PASCAL CDefClassFactory_Create(
                LPFNCREATEINSTANCE lpfnCI, UINT FAR * pcRefDll, REFIID
riidInst);

//---------------------------------------------------------------------------
//
// DllGetClassObject
//
// This is the entry of this DLL, which all the In-Proc Server DLLs should
// export. See the description of "DllGetClassObject" in the OLE 2.0
// Programmer's Reference manual for details.
//
//---------------------------------------------------------------------------

STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID FAR* ppvOut)
{
        *ppvOut = NULL; // Assume failure

        if (IsEqualIID(rclsid, &CLSID_ShellExtSample))
        {
           if (IsEqualIID(riid, &IID_IClassFactory))
                || IsEqualIID(riid, &IID_IUnknown))
           {
                  CDefClassFactory * pacf = CDefClassFactory_Create(
         ShellExtSample_CreateInstance,
                                                                     &g_cRefThisDll,
                                                                     NULL);
                  if (pacf)
                  {
                          (IClassFactory FAR *)*ppvOut = &pacf->cf;
                          return NOERROR;
                  }
                  return ResultFromScode(E_OUTOFMEMORY);
           }
           return ResultFromScode(E_NOINTERFACE);
        }
        else
        {
                     return ResultFromScode(CLASS_E_CLASSNOTAVAILABLE);
        }
}
```

## IDLists

OLE 2.0 introduced objects call *monikers*, which were used to identify and bind references to link sources to the code for reconnection. The Chicago shell provides a similar object called an *ItemID*. ItemIDs are variable-length byte streams that contain information for identifying a file object within a folder. An ItemID does not contain non-persistent values such as pointers to data structures, window handles, or atoms. This is because the shell may store ItemIDs in persistent storage (that is, on disk) and use them later.

An ItemID is defined as follows:

```
typedef struct _SHITEMID
{
    USHORT  cb;        // Size of the ItemID
    BYTE abID[1];      // The ItemID (variable length)
} SHITEMID, FAR* LPSHITEMID;

typedef const SHITEMID FAR * LPCSHITEMID;
```

ItemIDs may also contain information that helps improve the efficiency with which you can manipulate the file object, for example, the file object's display name or sorting information.

The shell does not need to know the actual content of an ItemID. The only part of an ItemID the shell uses is the first two bytes, which contain the size of the ItemID. The shell does not look at the rest of the ItemID directly; this information is usable only by the handler that created the ItemID.

The shell often concatenates ItemIDs and adds a NULL terminator at the end, creating an *IDList*. An IDList that contains only one ItemID is called a *simple IDList*, and an IDList that contains multiple ItemIDs is called a *complex IDList*. For consistency, the shell always passes a pointer to an IDList even when the receiving handler can only use a single ItemID. IDLists are defined as follows:

```
typedef struct _ITEMIDLIST   // idl
{
    SHITEMID    mkid;
} ITEMIDLIST, FAR* LPITEMIDLIST;

typedef const ITEMIDLIST FAR* LPCITEMIDLIST;
```

## Context-Menu Handlers

An application implements a context-menu handler interface, **IContextMenu**, to add menu items to the drop-down menu the shell displays when the user clicks a file object with the right mouse button. This has the effect of dynamically adding verbs for this file type. The additional menu items can be either class-specific (that is, applicable to all files of a particular type) or instance-specific (that is, applicable to an individual file).

The purpose of a context-menu handler is to add menu items to a menu—not to delete or change items that are already there. Although context-menu handlers can change or remove existing items (because they are passed a handle to a menu that contains the items), they should not be used for that purpose because other handlers may add items before or after a particular handler gets the context-menu handle, and the shell adds items to the menu after all context-menu handlers have been called.

Context-menu handlers are entered in the registry under the *shellex* key within an application's information area. The *ContextMenuHandlers* key lists the names of subkeys that contain the Clsid of each context-menu handler, for example:

```
ContextMenuHandlers = ExtraMenu
           ExtraMenu = {00000000-1111-2222-3333-00000000000001}
```

You can register multiple context-menu handlers for a file type. In this case, the order of the subkey names in the *ContextMenuHandlers* key determines the order of the context menu's items.

In addition to the usual **IUnknown** methods, the context-menu handler interface uses three methods:

- **QueryContextMenu**
- **InvokeCommand**
- **GetCommandString**

When the user selects one of these dynamic verbs, the shell calls the **IContextMenu::InvokeCommand** member to let it process the command. If you register multiple context-menu handlers for a file type, the value of the *ContextMenuHandlers* key will determine the order of the commands.

### QueryContextMenu

Windows calls the **QueryContextMenu** method when it is about to display a context menu for a file. Context-menu handlers insert menu items by position (MF_POSITION) directly into the drop-down menu by calling **InsertMenu**. Menu items must be string items (MF_STRING). As a result, the *fuFlags* parameter to **InsertMenu** must be MF_POSITION | MF_STRING for each menu item the context-menu handler inserts.

**Syntax:**

```
QueryContextMenu (HMENU hMenu,
                  UINT indexMenu,
                  UINT idCmdFirst,
                  UINT idCmdLast,
                   INT uFlags)
```

**Parameters:**

- *hMenu*: The handle to the drop-down menu. This value should be passed as the *hmenu* parameter to **InsertMenu**.
- *indexMenu*: The index to the menu item before which the first menu item should be inserted. Normally, the context-menu handler passes this value to **InsertMenu** as the *idItem* parameter and increments this value each time it calls **InsertMenu**. This parameter will never be –1.
- *idCmdFirst*: The first menu-item identifier that the context-menu handler should use by passing the value to **InsertMenu** as the *idNewItem* parameter. For each subsequent menu item, the context-menu handler should increment the value before passing it to **InsertMenu**.
- *idCmdLast*: The last menu-item identifier that can be used in this menu. Context-menu handlers must make sure they do not use a menu-item identifier with a value higher than this value.
- *uFlags*:
    - CMF_DEFAULTONLY: Chicago sends this flag if the users double-clicks a file with the right mouse button. Context-menu handlers should avoid adding non-default menu items to the context menu when this flag is present.
    - CMF_VERBSONLY: Context-menu handlers should ignore this flag.

**Example:**

```
STDMETHODIMP SHE_ContextMenu_QueryContextMenu(LPCONTEXTMENU pctm,
                                              HMENU hmenu,
                                              UINT indexMenu,
                                              UINT idCmdFirst,
                                              UINT idCmdLast,
                                              UINT uFlags)
{
    UINT idCmd = idCmdFirst;

    if (idCmdFirst+2 > idCmdLast)
      return ResultFromScode(E_FAIL);

    InsertMenu(hmenu, indexMenu++, MF_STRING|MF_BYPOSITION,
               idCmd++, "Check H&DROP (menuext)");
    InsertMenu(hmenu, indexMenu++, MF_STRING|MF_BYPOSITION,
               idCmd++, "Check H&NRES (menuext)");
    return ResultFromScode(MAKE_SCODE(SEVERITY_SUCCESS,
                           FACILITY_NULL, (USHORT)2));
}
```

## InvokeCommand

Windows calls the **InvokeCommand** method when the user selects a menu item that the context-menu handler added to the context menu.

**Syntax:**

```
InvokeCommand(HWND   hwndParent,
       LPCSTR pszWorkingDir,
       LPCSTR pszCmd,
       LPCSTR pszParam,
       int    iShowCmd);
```

**Parameters:**

- *hwndParent*: The window that owned the context menu. This can be the desktop, the file cabinet, or the tray. The context menu handler can use this handle as the owner window of dialog boxes or message boxes that the handler may display from within this method.
- *pszWorkingDir*: This parameter is NULL for menu items inserted by a context-menu handler. Context-menu handlers should ignore this parameter.
- *pszCmd*: This is a pointer to the command that the user selected. If the HIWORD of *pszCmd* is 0, the LOWORD contains the offset from the *idCmdFirst* parameter sent previously to **QueryGetContextMenu**. (Thus, *pszCmd* would be 0 for the first menu item the handler added, 1 for the next menu item, and so on.) If the HIWORD of *pszCmd* is not 0, *pszCmd* points to a language-independent command string that could be used to execute the command. Currently, the shell does not use the command strings.
- *pszParam*: This parameter is NULL for menu items inserted by a context-menu handler. Context-menu handlers should ignore this parameter.
- *iShowCmd*: This parameter is 0 for menu items inserted by a context-menu handler. Context-menu handlers should ignore this parameter.

**Example:**

```
STDMETHODIMP SHE_ContextMenu_InvokeCommand(LPCONTEXTMENU pctm,
                                            HWND hwnd,
                                            LPCSTR pszWorkingDir,
                                            LPCSTR pszCmd,
                                            LPCSTR pszParam,
                                            int    iShowCmd)
{
    PSHELLEXTSAMPLE this = PCTM2PSMX(pctm);
    HRESULT hres = ResultFromScode(E_INVALIDARG);   // assume error
    //
    // No need to support string-based command.
    //
    if (!HIWORD(pszCmd))
    {
      UINT idCmd = LOWORD(pszCmd);

      switch(idCmd)
      {
         case 0:
         hres = DoHDROPCommand(hwnd, pszWorkingDir, pszCmd, pszParam,
                                  iShowCmd);
         break;

         case 1:
```

NOV-B03687526
CONFIDENTIAL

```
            hres = DoHNRESCommand(hwnd, pszWorkingDir, pszCmd, pszParam,
                                 iShowCmd);
            break;
        }
    }
    return hres;
}
```

## GetCommandString

Windows calls the **GetCommandString** method to get a language-independent command string or the help text for a context menu item.

**Syntax:**

```
GetCommandString(UINT        idCmd,
                            UINT         uFlags,
                            UINT FAR *   pwReserved,
                            LPSTR        pszName,
                            UINT         cchMax)
```

**Parameters:**

- *idCmd*: The item identifier of the context menu item.
- *uFlags*: If this parameter is zero, the handler should return the language-independent command string for the menu item. If the parameter is GCS_HELPTEXT, the handler should return the menu item's help string, which the shell will display in its status bar.
- *pwReserved*: This parameter is reserved.
- *pszName*: A pointer to a string to which the method should copy the command string.
- *cchMax*: The maximum number of characters that *pszName* can contain.

## Drag-Drop Handlers

Drag-drop handlers implement the **IContextMenu** interface. In fact, a drag-drop handler is simply a context-menu handler that affects the menu the shell displays when a user drags and drops a file object with the right mouse button. Since this menu is called the drag-drop menu, shell extensions that add items to this menu are called drag-drop handlers. Drag-drop handlers work in the same way as context-menu handlers.

## Icon Handlers

The Chicago shell allows an application to customize the icon that the shell displays for the application's file types. Figure 1 shows the shell's standard icons.

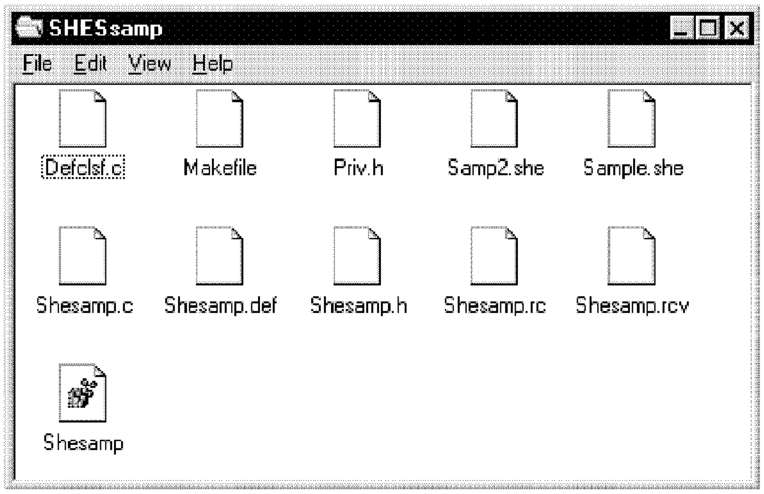


**Figure 1. Standard icons**

Figure 2 shows the customized icons supplied by an icon handler.

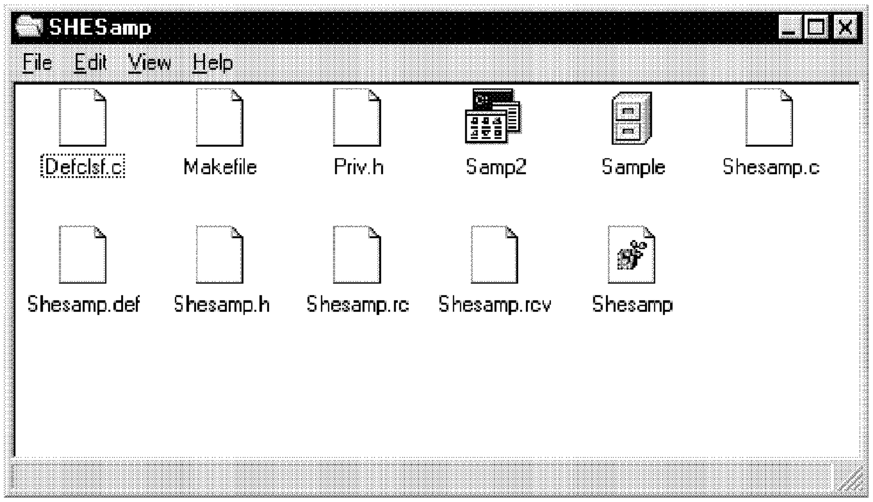


**Figure 2. Icons supplied by an icon handler**

The shell also uses the icon interface to allow applications to specify icons for folders and subfolders within an application's file structure. (This subject will be covered in a separate article.)

An application can specify icons for its file types in two ways. The first, and simplest, way is to specify a class icon to be used for all files of that file type. To specify a class icon, the application adds a *DefaultIcon* key to the registry under the program information. The value of this key specifies the executable (or DLL) that contains the icon, and the index of the icon with the file. For example:

```
DefaultIcon = c:\Mydir\Myapp.exe,1
```

This is identical to the way Windows 3.1 handles default icons. One of the advantages of using a class icon is that it requires no programming; the shell handles displaying the icon for the class.

Chicago adds a new value, %1, for the *DefaultIcon* key. This value denotes that each file instance of this type can have a different icon. The application must supply an icon handler for the file type and add another entry, *IconHandler*, to the *shellex* key for the application. An application can have only one *IconHandler* entry. The value of the *IconHandler* key denotes the Clsid of the icon handler, for example:

```
shellex
   IconHandler = {00000000-1111-2222-3333-00000000000003}
DefaultIcon = %1
```

To have customized icons, an application must supply an extract icon handler that implements the **IExtractIcon** interface. When Chicago is about to display an icon for a file type that has instance-specific icons, it does the following:

1. Gets the Clsid of the handler.
2. Creates an **IClassFactory** object by calling the **DllGetClasObject** entry of the specified DLL.
3. Calls **IClassFactory::CreateInstance** with **IID_IPersistFile** to create its instance.
4. Initializes the instance by calling the **IPersistFile:Load** method.
5. Uses the **QueryInterface** method to get to the **IExtractIcon** interface.
6. Calls the interface's **GetIconLocation ExtractIcon** method.

The **IExtractIcon** interface has two methods in addition to the usual **IUnknown** methods:

- **GetIconLocation**
- **ExtractIcon**

## GetIconLocation

Chicago calls the **GetIconLocation** method to get the location of an icon to display. Normally, the icon location is an executable or DLL filename, but it can be any file.

**Syntax:**

```
GetIconLocation(UINT   uFlags,
                LPSTR  szIconFile,
                UINT   cchMax,
                int  FAR * piIndex,
                UINT FAR * pwFlags)
```

**Parameters:**

- *uFlags*: If the object is a folder, Chicago sends the GIL_OPENICON flag to specify an open folder icon that appears in the left pane of the explorer. If GIL_OPENICON isn't specified, the icon handler should return the closed version of a folder icon by default. An icon handler for non-folder objects should ignore this parameter.

- *szIconFile*: A pointer to a string that contains the fully qualified name of the file that contains the icon. Icon handlers copy the filename to this string.
- *cchMax*: The maximum number of characters available in *szIconFile*. Icon handlers must make sure not to put more than *cchMax* characters in *szIconFile*.
- *piIndex*: If the value is positive, this parameter points to an integer that contains the index to the icon in the file. If the value is negative, the parameter points to the resource ID for an icon. Icon handlers must put the index for the icon in the integer that this parameter points to.
- *pwFlags*:
  - GIL_SIMULATEDOC: Use the document icon for this file type.
  - GIL_PERINSTANCE: Icons for this file type are per instance (that is, different files of this type have different icons).
  - GIL_PERCLASS: Icons for this file type are per class (that is, icons are the same for all files of this type).

**Example:**

```
// First store the filename obtained in IPersistFile's Load method.
//
//
STDMETHODIMP SHE_PersistFile_Load(LPPERSISTFILE pPersistFile,
                                  LPCOLESTR lpszFileName,
                                  DWORD grfMode)
{
   // Get a pointer to my class
   PSHELLEXTSAMPLE this = PPSF2PSMX(pPersistFile);

   int iRet = WideCharToMultiByte(
         CP_ACP,                  // CodePage
              WC_SEPCHARS,        // dwFlags
         lpszFileName,            // lpWideCharStr
            -1,                   // cchWideChar
         this->_szFile,           // lpMultiByteStr
         sizeof(this->_szFile),   // cchMultiByte,
         NULL,                    // lpDefaultChar,
         NULL                     // lpUsedDefaultChar
         );

   // Copy the filename to my holder.
    if (iRet==0)
    {
      LPSTR psz=this->_szFile;
      while(*psz++ = (char)*lpszFileName++);
    }

    return NOERROR;
}


// Now tell the shell where to get the icon.
//
// This sample reads the file to get the location.
//
STDMETHODIMP SHE_ExtractIcon_GetIconLocation(LPEXTRACTICON pexic,
            UINT   uFlags,
            LPSTR  szIconFile,
            UINT   cchMax,
            int  FAR * piIndex,
            UINT FAR * pwFlags)
{
   PSHELLEXTSAMPLE this = PEXI2PSMX(pexic);
    if (this->_szFile[0])
    {
      GetPrivateProfileString("IconImage", "FileName",
            "shell32.dll",szIconFile, cchMax, this->_szFile);
      *piIndex = (int)GetPrivateProfileInt("IconImage", "Index",
                        0, this->_szFile);
    }
    else
    {
      lstrcpy(szIconFile, "shell32.dll");
      *piIndex = -10;
    }
    *pwFlags = 0;

    return NOERROR;
}
```

## ExtractIcon

Chicago calls the **ExtractIcon** method when it needs to display an icon for a file that does not reside in an executable or DLL. Applications usually have the file icons in their executables or DLLs, so icon handlers can simply implement this method as a return-only function that returns E_FAIL. When the icon for a file is in a separate .ICO file (or any other type of file), the icon handler must extract the icon for the shell and return it in this method.

**Syntax:**

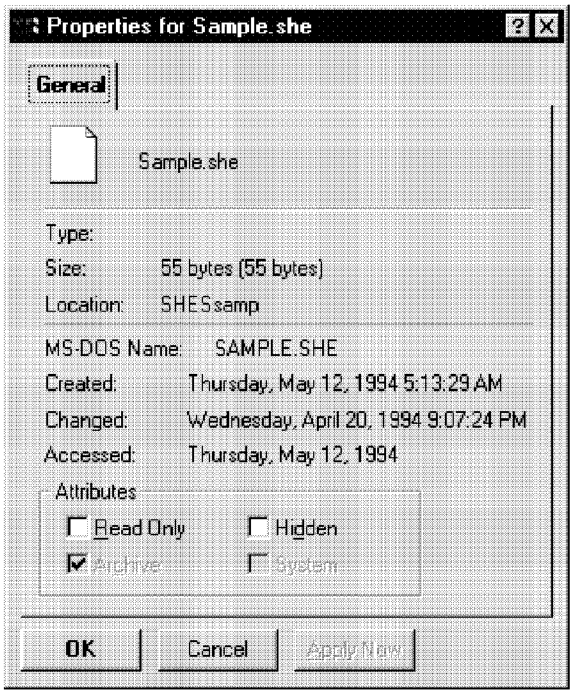
```
ExtractIcon(LPCSTR pszFile,
      UINT  nIconIndex,
      HICON  FAR *phiconLarge,
      HICON  FAR *phiconSmall,
      UINT   nIcons)
```

**Parameters:**

- *pszFile*: The filename that contains the icon. This value was set in the **GetIconLocation** method.
- *nIconIndex*: The index to the icon in the file. This value was set in the **GetIconLocation** method.
- *phiconLarge*: A pointer to an icon handle. The icon handler sets the handle to the large icon for this item.
- *phiconSmall*: A pointer to an icon handle. The icon handler sets the handle to the small icon for this item.
- *nIcons*: The number of icons (large or small) the shell is requesting. This value is always 1 for current implementations of Chicago.

## Property-Sheet Handlers

Another way the shell can be extended is via custom property sheets. When the user selects the properties for a file, the shell displays a standard property sheet (currently, the General property sheet illustrated in Figure 3).



Sample.she

Type:
Size: 55 bytes (55 bytes)
Location: SHESsamp

MS-DOS Name: SAMPLE.SHE
Created: Thursday, May 12, 1994 5:13:29 AM
Changed: Wednesday, April 20, 1994 9:07:24 PM
Accessed: Thursday, May 12, 1994

Attributes
[ ] Read Only [ ] Hidden
[✓] Archive [ ] System

**Figure 3. The standard property sheet**

If the registered file type has a property-sheet handler, the shell will allow the user to access the additional sheets the handler provides (Figure 4). Property-sheet handlers implement the **IShellPropSheetExt** interface.

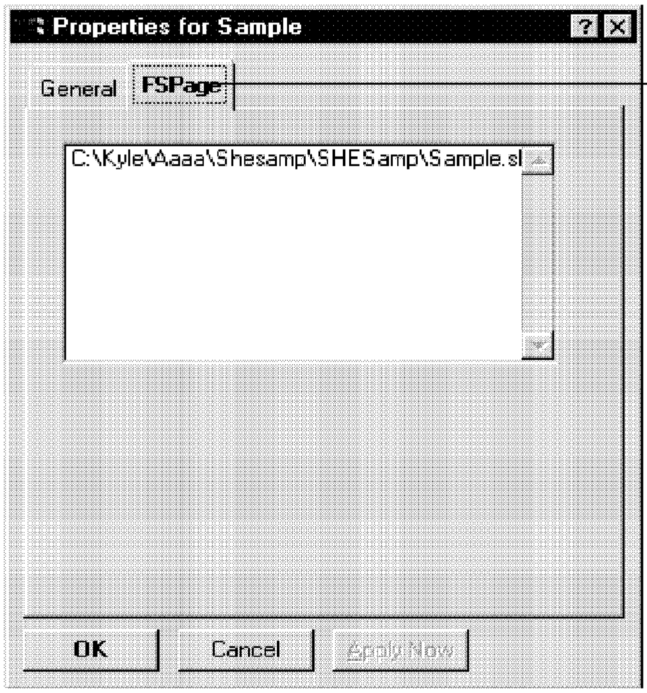

Properties for Sample
General
FSPage
C:\Kyle\Aaaa\Shesamp\SHESamp\Sample.sl
OK
Cancel
Apply Now
Added by a shell extension

**Figure 4. A property sheet page added by a shell extension**

Property-sheet handlers are entered in the registry under the *shellex* key, within an application's information area. The *PropertySheetHandlers* key lists the names of subkeys that contain the Clsid of each context-menu handler. For example:

```
PropertySheetHandlers = SummaryInfo
    SummaryInfo = {00000000-1111-2222-3333-00000000000002}
```

You can register multiple property-sheet handlers for a file type. In this case, the order of the subkey names in the *PropertrySheetHandlers* key determines the order of the additional property sheets. You can use a maximum of 24 (MAXPROPPAGES) property-sheet pages.

The property-sheet handler uses the **AddPages** method in addition to the usual **IUnknown** methods.

## AddPages

Chicago calls the AddPages method when it is about to display a property sheet. Chicago calls each property-sheet handler registered to the file type to allow the handlers to add pages to the property sheets.

**Syntax:**

```
AddPages(LPFNADDPROPSHEETPAGE lpfnAddPage, LPARAM lParam)

typedef BOOL (CALLBACK FAR * LPFNADDPROPSHEETPAGE)(HPROPSHEETPAGE, LPARAM);
```

**Parameters:**

- *pfnAddPage:* Points to the function the property-sheet handler calls to add a page to the property sheet. This function takes a property-sheet handle and an *lParam* as parameters.
- *lParam*: The property-sheet handler passes this argument to the **lpfnAddPage** function as its *lParam* parameter.

**Example:**

For each page it wants to add to the property sheet, a property-sheet handler does the following:

- Fills in a **PROPSHEETPAGE** structure.
- Calls **CreatePropSheetPage**.
- Calls **lpfnAddPage** with the handle returned from **CreatePropSheetPage** and the *lParam* passed in from the shell.

```
STDMETHODIMP CSamplePageExt::AddPages(LPFNADDPROPSHEETPAGE lpfnAddPage,
               LPARAM lParam)
{
       PROPSHEETPAGE psp;
       HPROPSHEETPAGE hpage;

       psp.dwSize       = sizeof(psp);    // no extra data.
       psp.dwFlags      = PSP_USEREFPARENT | PSP_USERELEASEFUNC;
       psp.hInstance    = (HINSTANCE)g_hmodThisDll;
       psp.pszTemplate = MAKEINTRESOURCE(DLG_FSPAGE);
       psp.pfnDlgProc   = FSPage_DlgProc;
       psp.pcRefParent = &g_cRefThisDll;
       psp.pfnRelease   = FSPage_ReleasePage;
       psp.lParam       = (LPARAM)hdrop;

       hpage = CreatePropertySheetPage(&psp);
       if (hpage) {
         if (!lpfnAddPage(hpage, lParam))
            DestroyPropertySheetPage(hpage);
       }

       return NOERROR;
}
```

## Copy-Hook Handlers

An application can register a copy-hook handler that the shell will call before the shell moves, copies, deletes, or renames a folder object. The copy-hook handler does not perform the task itself, but provides approval for the task. When the shell receives approval from the copy-hook handler, it performs the actual file system operation (move, copy, delete, or rename). Copy-hook handlers are not informed about the success of the operation, so they cannot monitor actions that occur to folder objects.

The shell initializes the copy-hook handler interface directly, that is, without using an **IShellExtInit** or **IPersistFile** interface first. A folder object can have multiple copy-hook handlers. The copy-hook handler interface has one method, **CopyCallBack**, in addition to the standard **IUnknown** methods.

### CopyCallBack

The shell calls the **CopyCallBack** method before it copies, moves, renames, or deletes a folder object. The method returns an integer value that indicates whether the shell should perform the operation. The shell will call each copy-hook handler registered for a folder object until either all the handlers have been called, or any handler returns IDCANCEL. The handler can also return IDYES to specify that the operation should be carried out, or IDNO to specify that the operation should not be performed.

**Syntax:**

```
CopyCallback(HWND hwnd, WORD wFunc, WORD wFlags, LPCSTR pszSrcFile, DWORD
dwSrcAttribs, LPCSTR pszDestFile, DWORD dwDestAttribs);
```

**Parameters:**

- *hwnd*: Either the dialog to use as the progress dialog, or the parent from which to create the progress dialog, if FOF_CREATEPROGRESSDLG is set.
- *wFunc*: Operation to be performed:
  - FO_DELETE: Delete files in *pszSrcFile*.
  - FO_RENAME: Rename files in *pszSrcFile*.
  - FO_MOVE: Move files in *pszSrcFile* to *pszDestFile*.
  - FO_COPY: Copy files in *pszSrcFile* to *pszDestFile*.
- *wFlags*: Flags that control the operation:
  - FOF_RENAMEONCOLLISION: In a move/copy/rename operation, if a file of the target name already exists, give the file being operated on a new name (such as "Copy #1 of ...").
  - FOF_NOCONFIRMATION: Respond "yes to all" for any dialog box that might be displayed.
- *pszSrcFile*: Pointer to a string containing the source filename.
- *dwSrcAttribs*: Attributes of the source file.
- *pszDestFile*: Pointer to a string containing the destination filename.
- *dwDestAttribs*: Attributes of the destination file.

## Summary

The shell extensions described in this article allow applications to facilitate the task of navigating within a system or network. In some cases, however, applications may want to extend the shell further with a name-space browser. Name-space browsers allow applications to expose the hierarchical structure of objects through the Chicago shell. A good example of a name-space browser is a file that displays the hierarchy of the user's mail folder. In my next article, I will be discussing name-space browsers in detail. Stay tuned.