

From the November 2001 issue of MSDN Magazine.



New Graphical Interface: Enhance Your Programs with New Windows XP Shell Features

Dino Esposito

This article assumes you're familiar with GDI

Level of Difficulty 1 2 3

SUMMARY The Windows XP shell introduces many new features that both users and developers are sure to welcome. The interface supports a number of styles that will be new to users, and it also supports customization of those styles through a new concept called themes. There are more shell registry settings available to the user and developer, a facility for customizing infotips, and infotip shell extensions. In addition, folder views can be customized.

This article covers these shell changes and includes a discussion of a number of other Windows XP additions. These include fast user switching, which lets users log on and off quickly, and AutoPlay support for a variety of devices and file types not previously supported.

Like it or not, the new Windows® XP user interface (formerly codenamed "Luna") is something that most people have strong feelings about when they first see it. I confess that when I first completed the Windows XP installation and I was confronted with the new Windows XP visual style, I was tempted to drop it and return to the more classic and familiar Windows 2000 interface (which is an option with Windows XP). I wanted my old Windows back!

The Windows XP shell introduces a rationalization of the user interface services and features found in previous versions of Windows. As a result, a few things you could do with Windows 2000 are either unsupported under Windows XP or are supported in a more restrictive way. To make up for this, new common controls and, above all, new listview capabilities, make the Windows shell easier to use with richer programming support.

In this article, I'll delve into the Windows XP shell to cover shell settings, shell extensions, new registry entries, and folder customization. Finally, I'll summarize the key things you need to know about supporting Windows XP visual styles in your own apps through the new family of common controls. This article is based on the first Release Candidate of Windows XP, so it's remotely possible that some of this information might change when the product ships. However, if you follow along you'll find the kind of information you need to get ready for Windows XP now.

What's New in the Windows XP Shell?

Windows XP does not support any new shell extensions that aren't also supported in Windows 2000, with one exception I'll cover shortly. However, folder customization through hypertext templates (.htt files) has been deemphasized in Windows XP and important restrictions have been applied.

There are places in the Windows XP shell where the user interface is drawn in a completely new way. The underlying technology, though, is not documented or exposed at present.

There are now more shell registry settings, letting you better personalize more aspects of the shell. For example, you can exercise significant control over infotips on a per-file-class basis without writing a

shell extension. Additional shell enhancements include common controls, fast user switching (the ability to have the system work with multiple users simultaneously connected), and Autoplay. The one new type of shell extension Windows XP *does* have that allows you to register new handlers for music, video, and pictures delivered through CD, DVD and other media. (See the "Automatically Detect and React to New Devices on a System" article in this issue.)

The contents of my Windows 2000 article "[Enhance Your User's Experience with New Infotip and Icon Overlay Shell Extensions](#)" (*MSDN Magazine*, March 2000) is still largely valid. However, I'm going to revisit some of the information in that article in light of Windows XP.

After playing with Windows XP for a couple of days, I got the distinct impression that the system tends to control what the user can and cannot do much better than earlier versions of the Windows operating system did. I've written at least three articles for *MSDN® Magazine* about the Windows 2000 shell and registry. In addition to the aforementioned piece about shell extensions, I covered advanced folder customization in "Extending Explorer Views by Customizing HyperText Template Files" in the June 2000 issue and the registry in "Latest Features and APIs Provide the Power to Customize and Extend Your Apps" that appeared in the November 2000 issue. All these articles (whose URLs are listed in this article summary) illustrate hidden gems in the Windows 2000 shell architecture and shed light on poorly documented, yet important topics and features that satisfy diverse needs and make developers more productive. For example, how do you change a folder's icon? Based on some early MSDN documentation, if you create a file called desktop.ini in a folder, and add the following content

```
[.ShellClassInfo]
IconFile=shell32.dll
IconIndex=41
```

then setting the readonly attribute for the folder forces Explorer to pay special attention to the desktop.ini file. This makes Explorer read the .ini file and display the specified icon when folders are listed. While this is not a particularly difficult task for developers and power users, it required knowledge of implementation details and familiarity with the system.

In Windows XP, any user who wants to mark a particular folder with a custom icon need only right-click on the folder, select the Customize tab and follow the instructions shown in **Figure 1**. What happens under the covers is similar to the process I just described, but the details are hidden from the user.

 Figure 1 Choose Icon

Figure 1 Choose Icon

Another interesting feature that was once the exclusive domain of power users, but which is now available to any Windows XP user, is the ability to associate actions with classes of files. For example, suppose you want to add the Register and Unregister items on the context menu of DLL files to register and unregister the file as a COM component. Before Windows XP, this required a Windows Script Host (WSH) script to directly manipulate the registry or some compiled code using the registry API. While this remains the only way to accomplish it programmatically, users can now access a sequence of dialogs to associate commands with classes of files. Just click on the Folder Options item from the Explorer Tools menu, open the File Types tab, select or add the particular file type, and click the Advanced button. What happens next is shown in **Figure 2**.

 Figure 2 Advanced Options

Figure 2 Advanced Options

Customizing the layout of a Windows folder view is yet another user interface feature that's changed significantly in Windows XP. No longer will a wizard guide you through various steps. Instead, there is a simple dropdown list in the Customize tab of the folder's properties. From there, you can choose the style of the folder from a number of predefined options. Apparently, there is no longer a way to completely customize the structure of a folder view. (I'll have more to say about this later on.)

In general, Windows XP comes with a significant number of user interface improvements, especially for users and power users. In prior versions of Windows, some of these improvements were only useful to programmers. In Windows XP, many more people can take advantage of these UI improvements. From the programmer's perspective things have been left more or less unchanged, but the tasks are now more thoroughly documented in the latest Platform SDK.

Shell Registry Settings

Let's start this tour of the Windows XP shell by looking at some little-known yet useful registry tweaks that can help you to better configure an application and fuse it to the system shell.

A large number of the shell features are governed by settings stored in the system registry. By changing values and creating entries, you can tune up the overall behavior of the shell and its constituent elements. For example, you can control the way in which a file is processed (AlwaysShowExt, NoOpen, NoOpenWith, EditFlags settings), the way a system object presents itself through Explorer, and the way Infotips are formatted. A majority of the settings mentioned here also work with Windows 2000.

The AlwaysShowExt setting serves the purpose of overriding one of the folder options that hides the extension of known file types. A known file type is any file class whose extension is registered with the system, along with handlers for actions like open, print, and edit. Technically speaking, you know that a file class is properly registered with the operating system when you find the following registry key exists:

```
HKEY_CLASSES_ROOT  
\.ext
```

Here, .ext represents the file extension. If the AlwaysShowExt value is present in the registry, the file extension is always displayed in Explorer, regardless of the setting of the "Hide extensions for known file types" folder option. [Figure 3](#) and **Figure 4** show the various settings for typical DLL files.



Figure 4 Folder Options

Figure 4 Folder Options

Another rather interesting attribute is NoOpen. Through not frequently used, it helps the system notify users that they're about to do something dangerous. If your program creates files that users should not modify, you can register the corresponding file type with a NoOpen attribute under the file type identifier key. A file type identifier key takes the form of extfile, so the identifier for DLLs would be dllfile.

If the user attempts to open any file from a class that has been marked with the NoOpen attribute, then the system displays a warning message. If you assign some text as the value of the NoOpen attribute, the system displays that text in the message box. However, bear in mind that despite the attribute's name, signing your files this way does not prevent users from opening the files; they only get a warning. In addition, notice that NoOpen does not work if any command action has been specified for the class. For example, by default a DLL file has no custom command registered. (Except on my system I added a custom command through the dialogs shown in **Figure 2**.) In [Figure 3](#), you also see that DLL files are marked as NoOpen. Try to double-click on a DLL file under these conditions and you see a message that's similar to those in **Figure 5**. The top window shows the default message, whereas the bottom window displays a custom message that is the text of the NoOpen registry entry.



Figure 5 Default and Custom Messages

Figure 5 Default and Custom Messages

Assigning attributes to a file class allows you to control some aspects of its behavior. It also allows you to limit the extent to which the user can modify various aspects of the class, such as its icon or

verbs, through the File Types tab of the Folder Options dialog. The attributes are defined as binary flags and are stored in the EditFlags entry. EditFlags must be a REG_BINARY entry. For DLL files, its registry path evaluates to

```
HKEY_CLASSES_ROOT  
\dllfile
```

To assign the EditFlags entry of a file class, take the sum of any combination of the attributes listed in [Figure 6](#). For example, setting EditFlags to 0x00100100 combines the ability to preserve the description of the file class with the restriction that files of that type cannot be added to the Recent Documents folder after being opened.

Reading about this feature validated all of my conjectures and hunches a few years ago when I realized that not all files that I opened were listed in the Recent Documents menu. This is the perfect example of system features that work with Windows XP and are documented for Windows XP, but are also present in earlier versions of Windows.

Customized Infotips

Each file type can be associated with an infotip by creating a registry entry called Infotip under the following path:

```
HKEY_CLASSES_ROOT  
\textfile
```

If you register infotips in this way, you cannot apply them on a per-file basis. The string that you specify will be displayed for every file of that file type.

To customize infotips, you can write a shell extension, as I'll show you shortly. As an alternative, Windows XP supports a huge number of property names, each of which points to a special piece of information about files. Some of the supported property names, which also apply to Windows 2000, are listed in [Figure 7](#). The Infotip registry entry is assigned with a semicolon-separated sequence of these properties. Then, when Explorer is called to display the infotip for that file, it retrieves the information specific to that file. For example, if you look at [Figure 3](#) you'll notice that DLLs have the following infotip structure:

```
prop:FileDescription;Company;FileVersion;Create;Size
```

Any infotip text that includes property names must begin with prop: as this is the prefix that tells Explorer to apply a special treatment to the text that follows. Some of the property names in the list are specific to Windows XP, namely FileDescription, Company, and FileVersion. As you may have guessed already, they refer to information that can be stored in the VersionInfo resource of Windows executables. The list of property names includes information that is typical of a certain category of files, but not all. When a certain property name does not have a corresponding value, then it is simply ignored. In **Figure 8** you can see the infotips for two different DLLs. Guess what? Only the bottom one has version information.

 Figure 8 Infotips

Figure 8 Infotips

Speaking of DLLs, another particular shell registry setting you may have noticed in [Figure 3](#) is TileInfo. This refers to a new, Windows XP-specific view mode called Tiles. Tiles mode works side by side with other popular views in Windows such as thumbnails, details, and large. The TileInfo registry entry defines the format that displayed information should have when the file name is rendered in tiling mode in a folder view. TileInfo follows the same syntax rules as InfoTip.

 Figure 9

Figure 9

Figure 9 shows how the menu looks. Notice that, unlike the thumbnail view which is implemented internally by the system, tiles and groups (another new feature for arranging icons) are new functionalities of the Windows XP listview common control.

AutoPlay

AutoPlay is a feature that detects and properly handles special content such as pictures, music, or video files stored on removable media and devices. AutoPlay should not be confused with AutoRun. AutoRun, introduced with Windows 95, enables removable media like a CD to automatically launch an application when placed in the drive. Autorun is accomplished through the use of a file called `autorun.inf`, located in the root directory of the CD.

AutoPlay takes in the essence of AutoRun, but applies specifically to certain categories of files and provides a way for you to register handlers for those kinds of content. Put another way, AutoRun makes a CD self-installing, or at least self-running. It does not rely on files or information stored on the machine. AutoPlay addresses a different requirement. When a user connects a specialized peripheral device such as an MP3 player or digital photo reader, AutoPlay allows machine-specific handling of the device. For instance, when you attach a digital camera, you might be asked whether you want to display the camera as a drive, browse the photos, print them, and so on. For more information on Autoplay see the article on page 77 in this issue.

Revamped Shell Extensions

With the exception of `IHWEHandler`, which is involved with Autoplay, you will not find any new shell extensions in Windows XP that weren't in Windows 2000. However, the documentation now uses slightly different terminology for a few of the better-known extensions. Context menu shell extensions are often referred to as shortcut menu extensions. The column handler shell extension is frequently called the details shell extension.

I selected three types of shell extensions that you should always use as companion code for your custom documents. They are the `infotip` shell extension – a must-have if you have very structured and complex documents – the column handler – now the only way you have to display custom information about the selected file in the folder view, and finally the thumbnail viewer. A thumbnail viewer shell extension is a piece of software that provides a graphical preview of the content of a given file. In Windows 2000, the output of such a module was displayed in the standard template upon which the folder view was based. In Windows XP there is no longer the notion of a folder template, at least not one that is used for all the files throughout the system. For performance reasons, the Windows XP UI design team opted for a binary mechanism (a namespace extension) that allows very little customization or "reverse-engineering." I'll have more to say about this by the end of the article. When the folder view turns to the thumbnail mode (see [Figure 10](#)), then all the folder items are displayed in a larger size and, whenever possible, use a dynamically generated thumbnail. To do this, you must write a shell extension. As you may notice in [Figure 10](#), not all the documents of the same type provide a thumbnail. This has nothing to do with the shell extension itself, but relates to the way in which the Office 2000 thumbnail extractor really works. It relies on the preview image dynamically generated and stored in the document itself. This holds true for Word, Excel and PowerPoint® documents. If the document does not include such a preview image it will not be displayed by the thumbnail extractor. You can check the preview image for an Office 2000 document by popping up the Properties dialog. Let's take a quick tour of how to write these shell extensions.

Infotip Shell Extensions

A shell extension is a (relatively) small, simple COM object that implements one or two interfaces. Which interfaces are implemented and how depends on the specific tasks the component you're writing will perform. The `infotip` extension in particular implements the `IPersistFile` and the `IQueryInfo` interfaces. `IPersistFile` is necessary to let the component know about the name of the file it has to

process. IQueryInfo provides the methods through which Explorer retrieves the string to be displayed. When implementing the IPersistFile interface, the shell extension should use the following method of IPersistFile to read and store the name of the file to work with.

```
STDMETHOD (Load) (LPCOLESTR wszFile, DWORD dwMode)
```

The IQueryInfo interface provides two methods, one of which is not used. The key method is GetInfoTip.

```
// IQueryInfo::GetInfoTip
STDMETHOD (GetInfoTip) (DWORD dwFlags, LPWSTR *ppwszTip)
{
    // return some useful text
    wcscpy(*ppwszTip, L"Text");
    return S_OK;
}
```

In the body of the GetInfoTip method you retrieve the name of the file that you saved, possibly in a class data member, open it, and extract any information you want to display. The following code shows how to return the width and the height of a BMP file from an infotip shell extension.

```
// Opens the BMP file and returns size (and colors)
CComBSTR bstrInfo;
GetBitmapInfo( (CComBSTR *) &bstrInfo);

*ppwszTip = (WCHAR*) m_pAlloc->Alloc (
    (bstrInfo.Length() + 1) * sizeof(WCHAR) );
if (*ppwszTip)
    wcscpy( *ppwszTip, (WCHAR*) (BSTR) bstrInfo );
```

There is not much more to say about infotip shell extensions. When writing this code, you might want to use the following trick to prevent possible memory leaks. Allowing memory leaks in shell- level code is extremely dangerous because shell extensions can be invoked quite frequently. This is not true of all shell extensions, but it is certainly true for infotips. Given the structure of the shell extension, and especially the signature of the GetInfoTip method, the extension ends up passing one piece of its own memory to Explorer. The second argument of GetInfoTip is a pointer to a Unicode string. Sooner or later, Explorer will finish with it, but then there will be no way for your code to get back and deallocate that piece of memory. To circumvent such problems, the Windows shell defines a COM object whose only purpose is to allocate global memory that Explorer can destroy when needed. You obtain a reference to this memory allocator by calling the API SHGetMalloc. The allocator is rendered with the IMalloc interface from which you normally use the Alloc method and the Free method.

The infotip shell extension follows a non-standard schema for registration. You must create the following registry hierarchy

```
HKCR
  \.ext
    \shellex
      \{00021500-0000-0000-C000-000000000046}
```

The default entry for this key must evaluate to the CLSID of the shell extension.

Details Shell Extensions

When Explorer works in Details mode, it normally displays several standard columns. Each column lists information, such as the file size or type, for each file in the current folder. By implementing and registering a column handler, you can make custom columns available for display. I was reminded of

the importance of paying attention to details when writing shell extensions a couple of months ago when a client asked me how he could provide additional information for some particular XML files through the normal Explorer shell. Basically, his goal was to have the Type column display for certain files extra information taken from the file itself rather than the usual description for all XML files. In Windows 2000 and Windows XP there is no way to hook into the creation of the Type column. As an alternative solution, I suggested he use a slightly modified folder template where some ad-hoc script code could call into a safe COM object and retrieve all the information. In Windows XP there is no longer any way to do this. In Windows XP, you can't mix your script code (HTML hypertext templates) with the namespace extension in use to render the content of a folder. For this reason, using a column handler is the only way you have to interfere with the process that determines the content of the folder view. To make sure that his problem was solved successfully under Windows XP, a column handler that adds all the extra information for those XML files to the view must be written. Writing a column shell extension requires that a COM object be written that implements the sole IColumnProvider interface. The following two methods form the programming toolset of IColumnProvider.

```
STDMETHOD (GetColumnInfo) (DWORD, SHCOLUMNINFO *);
STDMETHOD (GetItemData) (LPCSHCOLUMNID, LPCSHCOLUMNDATA, VARIANT *);
```

The GetColumnInfo method provides the shell with structural information about the column. In particular, it informs Explorer about how many columns the extension supports, the expected data type (mostly strings), the default width in pixels, the description, and the alignment of the column. The shell calls the method GetItemData for each file displayed in the folder where this column has been selected. The SHCOLUMNID structure precisely identifies the column in case you're handling more than one. A column is uniquely identified by a pair of values, the format ID and the property ID. You can use these values to assign a special meaning to your column. For example, if you are writing this extension to provide the title of HTML documents you could use an existing column ID so that in the same Title column both the title of HTML documents and the title of Word documents appear. A format ID is just a 128-bit value that can be your object's CLSID if you want to provide a unique column. Column handler must be registered under the following path:

```
HKEY_CLASSES_ROOT
  \Folder
    \ShellEx
      \ColumnHandlers
        \CLSID
```

where CLSID is the CLSID of your column shell extension. A registered column does not show up in the Explorer view unless it is explicitly selected using the dialog box shown in **Figure 11**. You get that dialog box by right-clicking on the Explorer's column header bar and selecting the More button at the bottom of the context menu. The new column can be added on a per-folder basis.

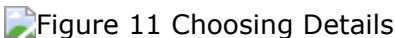


Figure 11 Choosing Details

Thumbnail Viewer Extensions

The thumbnail extractor is a COM object that implements two interfaces: IPersistFile and IExtractImage. IPersistFile plays the same role I reviewed earlier for the infotip shell extension. It lets you save the name of the file as a class data member. IExtractImage has two methods called GetLocation and Extract.

```
STDMETHOD (GetLocation) (
  LPWSTR pszPathBuffer,
  DWORD cchMax,
  DWORD *pdwPriority,
```

```
const SIZE *prgSize,
DWORD dwRecClrDepth,
DWORD *pdwFlags);
STDMETHOD (Extract) (HBITMAP *);
```

GetLocation is used to request the path of an image and specify how the image should be rendered. You should return images that fit within the boundaries defined by prgSize parameter. You do not need to fill in the unused part of the rectangle. Depending on how much the aspect ratio of the image differs from what is specified, the image may be distorted.

The following code shows a possible way to implement the Extract method for an extremely simple, single-threaded component. The Extract method should return to the shell a bitmap object that fits in the thumbnail area.

```
HRESULT CMyExtractor::Extract(HBITMAP* phThumbnail)
{
    HICON hi = ExtractIcon(NULL, m_szFile, 0);
    ICONINFO ii;
    GetIconInfo(hi, &ii);

    *phThumbnail = ii.hbmColor;

    DestroyIcon(hi);
    return NOERROR;
}
```

Explorer's Folder View

As I mentioned before, you cannot mix the standard view of Explorer with your own code. In other words, you cannot modify the folder.htt file in the c:\winnt\web folder to provide a customized view of all folders. This was possible in Windows 2000, but not in Windows XP. Generally speaking, in Windows XP the customization of folders became restricted. When you click on the "Customize this folder" item of the folder context menu, you don't get a wizard; a simpler dialog box pops up. It allows you to choose the icon for the folder and the picture to display on the big folder icon when in Thumbnail mode. Using the topmost panel however, you can choose the template for the current folder and all of its subfolders (see **Figure 12**). However, the templates available are very specific and designed for special user activities such as managing pictures, music or videos. There's really nothing there for the programmer! So, you may think there is no way to customize a folder in Windows XP, but that's not really an accurate statement.



Figure 12 Source Properties

Figure 12 Source Properties

When you set the new icon for the folder, Windows XP more or less does the same things that Windows 2000 does. It creates a desktop.ini file in the folder and stores in it the same information you would have in Windows 2000. This is the content generated for the scenario depicted in **Figure 12**.

```
[.ShellClassInfo]
IconFile=%SystemRoot%\system32\SHELL32.dll
IconIndex=24
```

One key thing that differentiates Windows XP from Windows 2000 is that in Windows XP the desktop.ini must be marked as system and hidden. This is the magic combination that lets Explorer know that desktop.ini is a special file. Given this, there is no need to mark the folder as read-only. So from Windows 2000 to Windows XP, the event that assigns a special meaning to desktop.ini has changed. It was the containing folder marked read-only in Windows 2000; it is the same file marked as

"system" in Windows XP. So, at this point, you can play around with desktop.ini. Let's see what happens if you add the following code:

```
[ExtShellFolderViews]
{5984FFE0-28D4-11CF-AE66-08002B2E1262}=
{5984FFE0-28D4-11CF-AE66-08002B2E1262}

[{{5984FFE0-28D4-11CF-AE66-08002B2E1262}}]
PersistMoniker=file://Navigator.htm
```

Next, re-open the Customize tab in the folder's Properties dialog. A new item magically appears stating that the folder is rendered according to a user-defined template. [Figure 13](#) shows a very simple template that switches between two views. The way you write HTT files is exactly the same as in Windows 2000. There is really nothing different except that Windows XP does very little to simplify this task. In any case, folder customization applies to a single folder only and shows up only if users don't select the Windows classic template from the dialog shown in **Figure 14**.

 Figure 14 Template Dialog

Figure 14 Template Dialog

Windows XP Visual Styles

The key feature of the Windows XP user interface is that it is based on themes. A theme is a consistent, system-wide way to draw user interface elements. These elements include controls, custom controls, icons, and HTML pages.

From the developer's standpoint, Windows XP visual styles means interaction with XML manifests, the newest version 6.0 of the ComCtl32 library, side-by-side components, and color-rich icons. (Manifests are files that specify versioning information so that different versions of DLLs don't collide. Doug Boling's article in this issue discusses manifests in detail.)

Without touching the code, any Win32®-based application can get a facelift when running under Windows XP. The differences in the Windows XP visual styles, when compared to plain old Windows styles, can be summarized in three elements: non-rectangular and irregularly shaped controls, hot states, and extended support for alpha transparency. These features are provided by version 6.0 of the ComCtl32 library that now incorporates all Windows and shell controls. This DLL won't ship downlevel, but on older platforms programmers can use version 5.8 to get closer to Windows XP styles. However, version 5.8 doesn't provide the same features and the same set of controls. For example, it doesn't support the new hyperlink control or the bitmapped button control.

How can you write an application that exploits visual styles? The Windows XP visual styles are not something you can hard code. There is no special API to call to enable it. All you do is write a plain old Win32-based application and add an XML manifest.

ComCtl32 version 6.0 ships side-by-side with version 5.8 and to avoid breaking existing applications; its use must be explicitly required and authorized.

In the XML manifest you specify the dependencies of your application. Dependencies include all the DLL files your application uses. This ensures that your application will always work as long as the components it has been tested with are present. It also gives you a way to link to the Windows XP theme manager to have your Win32 controls rendered the Windows XP way. The association is done on a per-application basis through a local XML file named program.exe.manifest.

For themes to work, Win32-based applications should make a call to InitCommonControls before using common controls. While this is normal for C++ applications, Visual Basic® programmers will have to adjust their thinking. So if you are writing applications with Visual Basic for Windows XP, bear in mind that if you don't call InitCommonControls at the very beginning of your app, it won't exploit Windows XP visual styles.

In [Figure 15](#) you can see a typical manifest file. As you can see, manifests are XML files filled with boilerplate code. If your goal is merely to enable visual styles, then you can take the file in the figure

as is and rename it. In addition to standalone files, manifests can be embedded in the application as a resource of type RT_MANIFEST. The DLL loader module knows how to look for and deal with manifests.

 Figure 16 Two Controls

Figure 16 Two Controls

Figure 16 shows the two faces of a simple Visual C++® control-intensive application. The first screen shows what the app looks like when Windows XP can successfully locate a manifest file. The second is how Windows XP defaults back to the old-style appearance when no manifest exists. This happens with any Win32-based application.

The Theme Manager

The key element in the Windows XP visual architecture is the theme manager (uxtheme.dll). In Windows XP all controls have been separated into their constituent client and non-client regions. The client part is made up of the content that the control itself is responsible for providing. The non-client part includes scrollbars, menus, title, background, borders, and so on. The non-client parts for each window are drawn by the theme manager according to the settings of the current theme. The Windows XP SDK defines the parts of each control that can have themes.

The theme manager gets into the game for the common controls, custom controls, and controls drawn within an HTML page viewed with Internet Explorer 6.0. Input tags and other typical HTML controls can now inherit the visual styles of Windows XP. This is an interesting shift, since until Internet Explorer 5.5, the browser contained code to render windowless HTML controls. The new architecture is more modular and consistent with a clean separation between the logic of the control and its design. Version 6.0 of the ComCtl32 DLL does not draw controls anymore, limiting the chance that new versions will break existing applications and making Windows XP inherently more stable than previous versions. For more information on themes and manifests, again take a look at Doug Boling's article in this issue.

Custom Controls

When it comes to developing custom controls for Windows XP, keeping separate states and parts is a must. For example, toolbar parts include the splitter, the background, the button, the separator, the dropdown list. States are normal, hot, checked, hot and checked, disabled, and pushed. A Windows XP guideline recommends that you always provide hover feedback, emphasizing when a control is ready to receive user input.

Custom and composite controls can be rendered, in whole or in part, like standard controls. For example, you can have a portion of a certain control render like a toolbar. To do so, you need three Windows XP-specific APIs: OpenThemeData, DrawThemeBackground, and CloseThemeData.

OpenThemeData returns a handle of type HTHEME and takes the HWND of the control and the name of the control class to mimic as parameters.

```
HTHEME h = OpenThemeData(hwnd, L"Toolbar");
```

You pass this HTHEME to DrawThemeBackground together with a flag that indicates the part of the base control you want to mimic. For example, TP_BUTTON refers to the button. You close the handle with CloseThemeData. The Theme SDK provides several DrawXxx functions you can use to ask the theme manager to draw a certain part of the control. Of course, if the control has a completely custom interface and is not made of other controls then the visual style simply does not make sense.

Icons

Typical icons are 32-bit color, 24 bits for colors and 8 bits for alpha blending. You should provide nine icons per symbol to accommodate all the needs from 4 to 32 bits of color depth. These icons must have different sizes, including 48×48 for thumbnails. Icon sizes 32×32 and 16×16 remain the typical

sizes for the shell while 24×24 and 16×16 are large and small toolbar buttons. There is a tool called gifmoviegear that can help you create these icons. It's available at <http://www.gamani.com/foricons/>. Although the icon format has not changed, support has been added to the imagelist API to display these 32-bit icons.

The icon size of 48×48 pixels is used for thumbnails and should always be supplied, especially if you don't provide an image extractor shell extension. Icons must degrade gracefully on machines with low colors for example, when the machine is working in safe mode or under Terminal Server. If your application is not supposed to be used in safe mode, you can skip this recommendation. If you're writing, say, a backup tool, then this step can be important. To help Windows XP to automatically smooth colors for toolbars and other icon-based controls, remember to set the flag ILC_COLOR32 when calling ImageList_Create. This will ensure an acceptable downgrade to 256 or 16 colors.

The ultimate guide for Windows XP icons is available at <http://msdn.microsoft.com/library/en-us/dnwxp/html/winxpicons.asp>. It provides an in-depth overview of icons, shadows, blending, and colors, and walks you through the design and creation of Windows XP-style icons, AVIs, and bitmaps.

HTML Pages

Themed HTML controls are the HTML-based controls that the browser renders in response to tags like `<INPUT TYPE=button>`. Normally, with Internet Explorer 6.0 and Windows XP those controls are rendered with Windows XP visual styles. However you can control this to some extent through a metatag.

```
<head>  
<meta http_equiv="MSThemeCompatible" Content="yes|no">  
</head>
```

Visual styles don't apply to HTML controls if CSS styles have been set for properties that are in conflict with what the theme manager is supposed to do.

New Common Controls

The listview control has been enhanced to provide two extra views in addition to large and small icons, details, and list. The two new view modes are tiles and groups. Tiling lets you tile the client area of the control with items using 32×32-pixel icons and formattable text automatically rendered with two colors.

Grouping allows you to arrange items into groups that are visually divided on the page using a horizontal line and a title. Groups are created using a new bunch of messages that let you add items to groups, groups to views, and sort and query for information about items. A third innovation on listview controls are insertion marks. Like the QuickLaunch toolbar, insertion marks enable the placement of visual elements during drag and drop for user feedback. The relative messages allow you to detect hits and specify the location and appearance of the insertion mark.

The traditional button control has been improved to allow you to display a little bitmap close to the text. This is achieved by associating an imagelist with the button. The imagelist should include images for the various states of the button in much the same way as it happens for toolbars. To use it you just create the imagelist and send a new message to the button window.

Finally, there's the long-awaited hyperlink control of which all brave Win32 developers have their own custom incarnation. The hyperlink control is implemented through a separate file called hlink.dll that is available only on Windows XP. This DLL is not portable outside Windows XP because it requires side-by-side component support and an explicit link to ComCtl32 version 6.0. The control is a window that renders marked-up text and launches appropriate applications when the user clicks one of the embedded links. You create hyperlink controls by calling CreateWindow specifying the WC_LINK window class name. The lpWindowName parameter of CreateWindow then specifies the text to display. The hyperlink control does not map to a single link. Instead, it is a sort of label with the ability to recognize any anchor tag `<A>` and process it properly. For this reason, a single hyperlink control can

handle more than one link. You should use the ID attribute to distinguish between the various links in the same control. HREF is a valid attribute and the most popular protocols are supported including FTP, HTTP, and mailto. It does not work with VBScript or JScript® because the host application is not expected to support a script engine. You set HREF and other attributes using the LVM_SETITEM message.

Fast User Switching

Last, but not least, Windows XP supports a new feature called fast user switching. As I mentioned earlier, Windows XP supports multiple users logged on at the same time. When one user is switched off, all her applications and even Internet connections are preserved. Fast user switching works together with the Welcome screen as an alternative to the classic user logon. This feature must be explicitly enabled, and it has a limited impact on applications. The most important recommendation is to pay attention to where you save documents they should go into folders specific to the current user. (This guideline was also part of the Windows 2000 Logo requirements.)

Typically, an application does not need to be notified when a desktop switch occurs. However, applications that want to be notified of user switches have to register by calling the WTSRegisterSessionNotification API function. Doug Boling's article, mentioned earlier, discusses fast user switching in more depth.

For related articles see:

[Microsoft Windows XP Fast User Switching: Design Guide for Building Business Applications Using Windows XP Visual Styles](#)

[More Windows 2000 UI Goodies: Extending Explorer Views by Customizing Hypertext Template Files](#)
[Windows 2000 Registry: Latest Features and APIs Provide the Power to Customize and Extend Your Apps](#)

[What's New in the Shell](#)

Dino Esposito is an instructor and consultant based in Rome, Italy. Author of *Building Web Solutions with ASP .NET and ADO .NET* (Microsoft Press), he now spends most of his time teaching classes on ASP .NET and ADO .NET for Wintellect (<http://www.wintellect.com>). Get in touch with Dino at dinoe@wintellect.com.