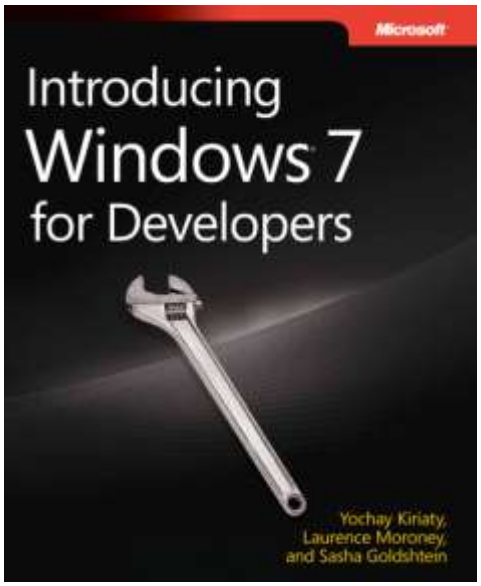


Introducing Windows 7 for Developers

Yochay Kiriaty,

Laurence Moroney,

and Sasha Goldshtein

To learn more about this book, visit Microsoft Learning at http://www.microsoft.com/MSPress/books/

9780735626829

Microsoft Press

© Microsoft Corporation

DRAFT

© Microsoft Corporation

Table of Contents

DRAFT

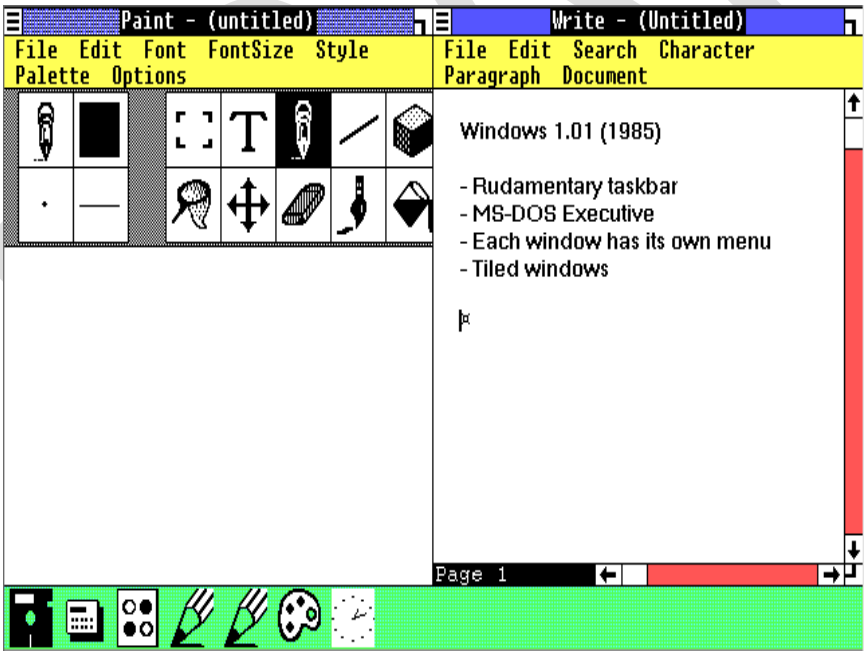
Chapter 2

Integrate with the Windows 7 Taskbar, Part 1

Excitement and anticipation followed me everywhere at the Microsoft Professional Developers Conference (PDC) in October 2008. New technologies were being announced at every corner; you couldn't find your way from one session to another without seeing another lab, another ad, another brochure on yet another Microsoft technology. And yet the most exciting of all, for me, was the unveiling of Windows 7, the new operating system from Microsoft. With eyes glued to the screen, thousands of attendees waited as Steven Sinofsky showed us around the M3 (6801) build of Windows 7.

The first feature that had everyone nodding with approval was the new Windows 7 taskbar. It was like a breath of fresh air in the conditioned air of the keynote hall, and it contributes to the sleek, light look of Windows 7.

Why is the Windows 7 taskbar so different from previous versions of Windows? What happened to the slow, methodical evolution of features being added with every release? Previous versions of Windows gave us the Quick Launch bar, desktop icons, the system tray, the Start Menu, the Search text box, the Run dialog, and many other launch surfaces—consolidating them all into square taskbar buttons seemed like a bold move. Some might say it was returning to the roots of the Windows user interface, the taskbar of Windows 1.0:



If I were to encounter the Windows Vista desktop for the first time and were asked how to open Outlook, I would be confused—I see multiple Outlook icons!

Apparently, as telemetry information collected at Microsoft over several years and as usability studies conducted with thousands of users indicate, quantity does not always translate to quality. When user interface design is concerned, it is often advisable to have only one way of accomplishing a given task.

© Microsoft Corporation

Again, a multitude of ways to do something can be more confusing than liberating. When most users work with fewer than 10 open windows during their session and most "advanced" features offered by the existing taskbar are not used, there is no choice but to radically redesign the state of affairs. This redesign gives us the new Windows 7 taskbar. It is a revolution of launch surfaces.

> Note An analysis of user sessions, open windows, and taskbar customizations performed by Microsoft and discussed at the PDC 2008 reveals that 90 percent of user sessions involve fewer than 15 open windows, 70 percent of user sessions involve fewer than 10 windows, and non-default taskbar options (such as auto-hide, docking the taskbar at the top of the screen, etc.) are used by fewer than 10 percent of users. Some other options are used by fewer than 1 percent of users.
>
> A good bit of the Windows UI was designed and implemented for complicated scenarios and affected the user's ability to easily switch between windows and to launch applications. This was not acceptable and led to the revolution in the Windows 7 taskbar.

Running applications, multiple instances of running applications, pinned programs—these concepts are all consolidated into the new taskbar. The Quick Launch toolbar is deprecated, the notification area (system tray) is considered out of bounds for applications, and large taskbar buttons dominate the user experience after the first logon to Windows 7.

This chapter will take you through the design goals of the Windows 7 taskbar and on a whirlwind tour of its new features. We will explore in depth the governing principle of the application ID and see how to light up applications with taskbar overlay icons and progress bars. In the next chapter, we'll take a look at the taskbar's more advanced features.

Design Goals of the Windows 7 Taskbar

The Windows 7 taskbar was engineered with several design goals in mind, resulting from a series of usability studies and requirement processing. These design goals are as follows:

- Single launch surface for frequent programs and destinations Applications and data that you use all the time should be at your fingertips; no more scouring through the Start Menu to find your favorite photo album application. Recent documents, frequently visited Web sites, and favorite photos should all be a click away via the new taskbar.
- Easily controllable Windows and running applications should be easily controllable and reachable for the user. Switching between applications, controlling the activity of another window, and obtaining a live preview of another window should all be performed without loss of productivity.
- Clean, noise-free, simple Plagued with features creeping into new releases of Windows, the "old" taskbar was crippled with desk bands, toolbars, notification icons, context menus, each customized for yet another application. The Windows 7 taskbar should have a sleek, clean appearance because it is the first UI element to greet the user after the logon screen.
- Revolution The Windows 7 taskbar offers new extensibility opportunities and differentiating opportunities for applications willing to take advantage of the new user experience design guidelines. Adhering to the design guidelines of the Windows 7 taskbar is almost guaranteed to improve your application's usability and your users' productivity.

DRAFT

A Feature Tour of the Windows 7 Taskbar

Large, animated taskbar buttons greet you at the gate of the Windows 7 desktop. Highlighted as you hover over them with your mouse and slightly morphing when you click them or touch them with your finger, taskbar buttons are a vivid representation of running programs and of shortcuts for launching inactive applications. Running applications are always visible in the taskbar; inactive applications can be pinned to the taskbar by the user (and by the user only) for quick-access launching.

Four types of effects distinguish taskbar button states, all shown here:

The Internet Explorer icon (the first one after the Start button from the left) is enclosed in a rectangle, meaning that it is currently running. The stacked taskbar buttons that appear behind it indicate that multiple windows of Internet Explorer are currently running on the system. Clicking the stack will bring up an array of thumbnails that represent the various windows of Internet Explorer and will allow us to effortlessly choose a specific window.

Next from the left, the new Windows Explorer icon is enclosed in a rectangle without any stacked buttons behind it, meaning that only one Explorer window is currently open. It is followed by the Windows Media Player and the Microsoft® Visual Studio buttons, neither of which is enclosed by a rectangle, meaning that both are currently inactive. Clicking one of these buttons will launch the associated application.

The second button from the right is the Office Outlook taskbar button, which has a light background, indicating that it is the currently active application. Finally, the icon on the far right is a Command Prompt, and it is highlighted because the mouse pointer is currently hovering above it. This feature is called Color Hot-Tracking, and it uses a blend of colors from the application's icon to determine the dynamic ambience of the button.

Taskbar buttons are the clean façade of the Windows 7 taskbar, and behind them is an abundance of new functionality for us to explore.

Jump Lists

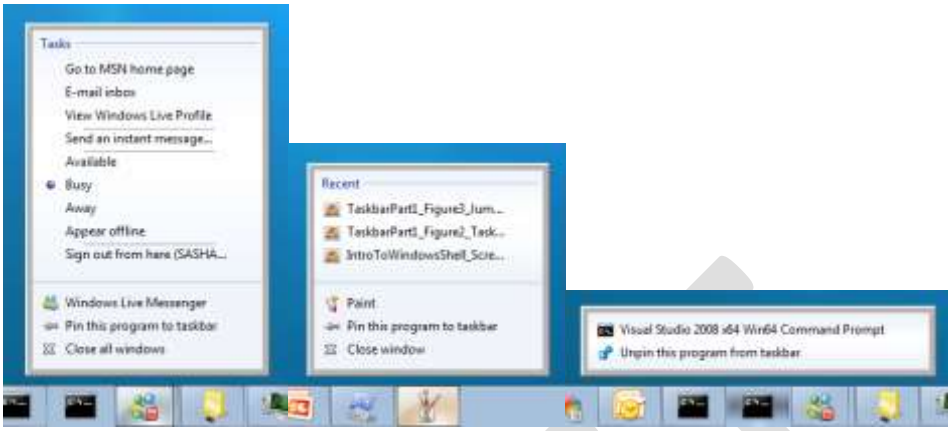
Right-click any taskbar button, and there's a menu of choices sliding from the bottom of the screen (or from any other side, if you repositioned your taskbar), giving you access to frequent tasks and destinations for your application.

The Start Menu has been the canonical location for application tasks and documents for all applications installed on the system. To launch an application, you would wearily navigate the Start Menu until you reach its program group. To take a quick look at your recent Excel spreadsheets, you would have to manually filter out recent documents from all other programs – which you are not interested in seeing right now.

The jump list is your application's opportunity for its very own "mini" Start Menu – an area where you can group popular task and destinations to enhance your users' productivity. Two types of items can be placed in a jump list – destinations, which are essentially files your application can open and handle, which can be grouped into categories; and tasks, which are launchers for common functionality your users frequently need. It's easy to envision destinations as your recent Word documents or your Internet Explorer browsing history; it takes a little more imagination to come up with useful tasks. The Windows Live Messenger jump list goes to great lengths to provide useful tasks, accessible without

DRAFT

even opening the application window – you can change your online presence status or go to your Live Mail inbox from the taskbar jump list.



Even if an application does nothing to enhance its jump list (which is precisely what Windows Paint does), Windows automatically populates the jump list with recent documents of the application's registered file type, as well as a set of three predefined system tasks. However, it is expected of well-behaved applications to use this opportunity and provide a convenient, one-click-away mechanism for launching and interacting with programs and data. Showing absolutely nothing in the jump list (such as the Visual Studio Command Prompt button above) will be frowned upon as users are more and more accustomed to using the jump list for interacting with programs and data.

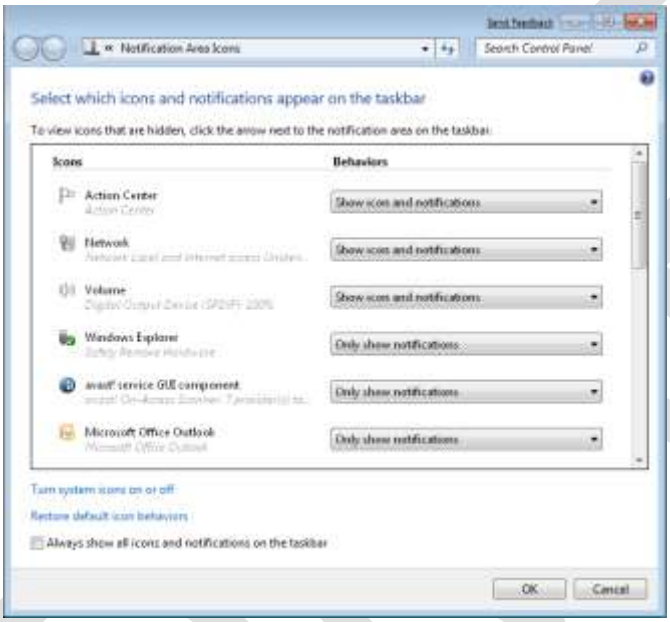
Even though the most natural place to find the jump list is right next to the taskbar button, the jump list can also appear within the Start Menu, the same Start Menu that it is destined to replace. If an application is currently visible in the Start Menu frequent area, a small arrow exposes the presence of the same jump list you would see if you clicked the application's taskbar button:

Properly interacting with jump lists, as well as designing and implementing applications which use them, is the subject of the subsequent chapter. For now, it will suffice to say that almost any application has a justification to customize its jump list for the benefit of its users. If you're writing a document-oriented application, you will immediately reap the benefits of the Recent and Frequent categories.

Even if it seems that you have no well-defined file type, and you need to consider which useful tasks and destinations will be beneficial for your users, you will quickly become addicted to the ease of use and the productivity gains that the jump list introduces to your application.

Taskbar Overlay Icons and Progress Bars

The typical way to convey status information in previous version of Windows was through the system notification area (affectionately known as the system tray). Ranging from the relatively unobtrusive balloon tips, through flashing the taskbar button multiple times, and all the way to focus-stealing system-wide-modal pop-up dialogs, there hasn't been a single consolidated mechanism for conveying status information from an application that is not in the foreground. Windows 7 ensures that the notification area stays uncluttered by letting only the user decide which applications are allowed to show notifications and be visible in the system notification area.



We have already seen how a traditional approach to launching applications and accessing frequently-used data is abstracted away and tied to the consolidated launch surface – the taskbar buttons. In a similar way, status information can be exposed from a Windows 7 taskbar button by using overlay icons – small icons which appear on the lower right of the taskbar button and provide immediate feedback to the user without switching to the application or even previewing its state.

Windows Live Messenger is a great demonstration of this functionality's usefulness. Your online status (Away, Busy, etc.) is always present in the application's taskbar button:

However, overlay icons are not always enough, especially if the status information is very dynamic. This is often the case when you need to expose progress information from your application. Fortunately, it is also possible to light-up the taskbar button with progress information, making it a mini-progress bar for your application. In fact, if you use some of the default Windows APIs for manipulating files (specifically, the SHFileOperation function or the IFileOperation interface which superseded it in Windows Vista) you get the file operation's progress automatically reflected in your taskbar button.

© Microsoft Corporation

Otherwise, you have to work a little harder but the result is very pleasing aesthetically – consider the following transitions of the Internet Explorer taskbar button when downloading a modestly large file:

The biggest benefit of taskbar overlay icons and progress bars is that the user can focus on one task at a time, without being distracted by status information interrupting his work flow. To immediately know the status of the most recent download or file copy operation, all the user has to do is take a look at the relevant taskbar button. It might take only a fraction of a second less than switching to or previewing the target window; but when these fractions accumulate, without even realizing it, we're up to an immense productivity gain.

Thumbnail Toolbars

Much as the jump list is a "mini" Start Menu for your application, regardless of whether it's running or not, a thumbnail toolbar provides a "remote control" for an individual window in an active application. Thumbnail toolbars let you control the state of another application without switching to its window – maximizing performance (because you do not perform a full switch to the application, which requires drawing code to run and possibly paging) and productivity (because you do not lose focus of your currently active work).

The classic example of a thumbnail toolbar is the Windows Media Player – like every media player, it offers the user the ability to switch to the next and previous items, as well as to pause and resume media playback. Considering that Media Player used to install a "taskbar toolbar" (also known as a desk-band) for the same purpose, consuming valuable screen estate and confusing users, the simplicity and elegance of the new Media Player's thumbnail toolbar are highly attractive.

Note that there is a significant difference between jump list tasks and items which belong on a thumbnail toolbar. Jump list tasks are fairly static, and do not depend on having an active instance of the application running – example Media Player tasks include "Resume last playlist" and "Play all music shuffled". On the other hand, thumbnail toolbars are only present and visible when the application is running, and each individual window can have its own thumbnail toolbar buttons to control the state of that window.

Live Window Thumbnails

The productivity features discussed earlier in this section minimize the need to switch to another application's windows. However, this is still a very acute need, and the success at delivering a live, vivid preview of another window is crucial for the following two key scenarios:

DRAFT

- Working in one application and quickly previewing the status of another application or the data presented by another application.
- Determining which window to switch to by examining the previews of multiple windows (prior to Windows 7, this was usually accomplished by using Flip-3D, a three-dimensional stack of windows previews that was relatively difficult to navigate).

Flip-3D is still available in Windows 7, but window switching and preview is another subset of the functionality consolidated into the new taskbar. The Windows Vista thumbnails were the first live window representations at the taskbar; the Windows 7 taskbar introduces several significant enhancements to these thumbnails, including:

- Multiple thumbnails for tabbed-document interface (TDI) applications, such as Internet Explorer, including switching support to a specific tab
- Live preview of the window (in full size) when hovering over the thumbnail
- A quick-access Close button on the thumbnail itself

This makes window navigation and switching so aesthetically pleasing that you might find yourself or your users playing with the live preview features (also known as "Aero Peek") without any intent to switch between windows!

TDI applications or applications willing to expose a custom window preview can take advantage of these new features to plug into the Desktop Window Manager's (DWM) pipeline for thumbnail and preview generation.

Backwards Compatibility

Although the Windows 7 taskbar is fully compatible with Windows XP and Windows Vista applications, considering compatibility up-front in your porting process will ensure that your applications seamlessly integrate with the experience expected by Windows 7 users. Among the topics you should be considering are:

- The quick launch area of the taskbar is deprecated and not shown by default; although it is possible to enable it, most users are likely to never do so. Installation programs should refrain from asking the user whether they want to install a quick launch icon.
- The system notification area ("system tray") belongs to the user and should be kept clean; applications should not attempt to pop-up messages or otherwise escape the notification area boundaries. Overlay icons and taskbar progress bars should replace the need for notification area icons.

DRAFT

- Proper file associations are required for integration with taskbar jump-lists (the subject of the subsequent chapter). The Recent and Frequent document categories cannot be populated by the system if your application does not have a properly registered file type.
- Users will expect destinations and tasks surfaced in the jump list, as a replacement for Start Menu navigation. Presenting an empty or default jump list is likely to leave users confused.
- Child windows which represent an important part of an application (for example, MDI child windows or web browser tabs) should be represented as separate thumbnails, similarly to Internet Explorer.

The Windows API Code Pack

Most of the Windows 7 APIs have no equivalent in managed (.NET) code. Some of them are easy to use from managed code, requiring only mild interoperability efforts, such as adding a reference to a COM type library and using it directly. The Windows 7 taskbar APIs can be used in this way as well – the vast majority of them are exposed through the ITaskbarList3 COM interface.

Nonetheless, Microsoft has collaboratively developed a project called the Windows API Code Pack (originally called the Windows Vista Bridge Sample Library), which provides managed wrappers for Windows features that are otherwise inaccessible directly from .NET applications. Like any managed library, this project can be used from any .NET language.

The Windows API Code Pack is available in open-source form on the Microsoft CodePlex Web site, under the MS-PL license. If you're writing managed applications, then to reproduce the samples in this book and to compile your own you will need to download the Windows API Code Pack from http://code.msdn.microsoft.com/WindowsAPICodePack.

Integrating with the Windows 7 Taskbar

We hope you've enjoyed the whirlwind tour of the Windows 7 taskbar features. It's time to focus in depth on the various features in order to light-up your application on Windows 7.

When designing your application for the Windows 7 taskbar, your first and foremost concern should be your taskbar button. A beautiful taskbar button with a clear, properly sized icon with a reasonable color balance will whet the users' appetite for interacting with your application. Remember that only users can pin applications to the taskbar, and if you've missed your chance by providing an ugly taskbar button, there's no way for you to force your application on the user.

This might sound trivial, but you must test your taskbar button with various Windows color themes and glass colors – some of the billions of Windows users are using the pink theme with almost transparent glass, others use the Windows Basic theme without Aero support, and yet others require high-contrast themes for accessibility reasons. The same applies for high DPI, which is discussed in depth later in this book, and might mutate your icon beyond comprehension if you do not ship it in various sizes.

If you've got your taskbar button right, you're already half-way through to a stunning Windows 7 application.

Application ID

During our discussion of the Windows 7 taskbar buttons, one thing might have struck you as odd: How does the shell determine which windows are associated with a specific taskbar button? Somehow, as if

by magic, all Internet Explorer tabs are grouped under the Internet Explorer button, all Word documents are stacked behind the Word taskbar button, and so on. It might appear as if the only control your can exercise over this grouping is to launch another window or another process, which will be associated with the same taskbar button as all other windows or processes belonging to the same application.

However, the truth is slightly more subtle than that. Some applications might require a behavior that is more sophisticated than just associating all windows with the same taskbar button as the process. For example, assume that you have a host process that runs all kinds of office productivity applications (a word processor, a spreadsheet, a finance manager). The host process in this case does not want a taskbar button at all, while the plug-ins require separate taskbar buttons for each application type.

Various Combinations of Windows and Taskbar Buttons

To better demonstrate the various combinations of windows and taskbar buttons, which complicate the requirements analysis for associating processes, windows, and their taskbar representation, in this sidebar you will find a collection of screenshots showing the different possibilities.

The following screenshot is an example of a single process that creates multiple windows, all grouped together to the same taskbar button.

The following screenshot is an example of multiple processes, each creating a single window, all grouped together to the same taskbar button. Note that the last two screenshots are virtually indistinguishable.

Next, the following screenshot shows that multiple windows belonging to the same process can have different taskbar buttons – using the same icon for all taskbar buttons or different icons for some of them.

DRAFT

It would be impossible for the system to automatically come up with a heuristic that would join windows from different processes together, and separate windows from the same process to different taskbar buttons. This is why application IDs were born.

This variety of scenarios is addressed by assigning each window an identifier called the application ID, which determines the taskbar button to which the window belongs. The default application ID for a window is a default application ID generated for the process to which the window belongs, which is in turn a default application ID generated for the executable file that the process runs. These defaults explain very well the default behavior for windows within multiple processes of the same executable (try running Notepad several times). However, customizing these defaults to suit our productivity application host involves setting an explicit application ID for the process (which affects all windows within that process) or even for an individual window to a different value, so that each plug-in application type can get its very own taskbar button.

The various scenarios for multiplexing processes, windows and application IDs are summarized in Table 2-1.

Table 2-1 XXXX

Scenario	Number of Processes	Number of Windows	Number of App IDs
Application host	One	Multiple	Multiple
Simple application	One	One	One
Document application	One	Multiple	One
Multi-instance application	Multiple	Multiple	One

Figure 2-1 demonstrates exactly how the application ID is determined for a specific window. The dashed arrows represent fallback scopes – if the window does not have an explicit app ID, then the process app ID is checked; if the process does not have an explicit app ID, then the shortcut is checked; finally, the executable itself is used to compute the application ID.

DRAFT

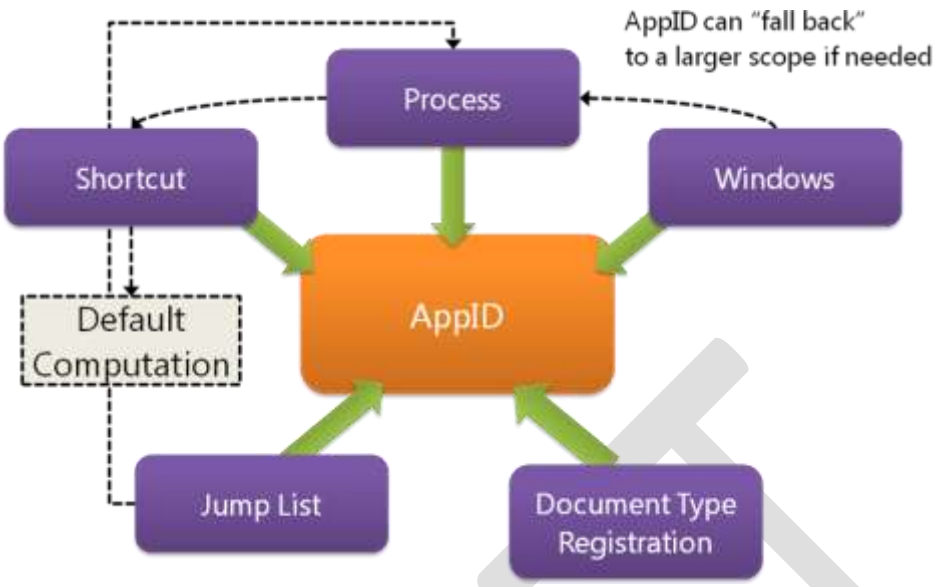


Figure 2-1 XXXX

Setting the explicit application ID for a process involves a single call to the SetCurrentProcessExplicitAppUserModelID function from shell32.dll. In the managed wrapper, this is exposed by the Taskbar.AppId static property which can be set to a string, and sets the application ID.

Setting the application ID for a window is slightly less straightforward – it requires calling the SHGetPropertyStoreForWindow function and then manipulating the resulting IPropertyStore object to retrieve the requested property. While this isn't exactly rocket science, the managed wrapper is more user-friendly as it exposes the Windows7Taskbar.SetWindowAppId static method and an extension method that accepts a System.Windows.Forms.Form object.

The very nature of these APIs makes it possible to change the application ID of a window or of a process dynamically, at run-time. The following code demonstrates how by pressing a button, an application toggles its main window to a different application ID, which makes it "jump" across the taskbar to a different taskbar button.

```
string currentAppID = "AppID0";

void ToggleAppID_Clicked(object sender, EventArgs e)
{
    if (currentAppID == "AppID0")
    {
        currentAppID = "AppID1";
    }
    else
    {
        currentAppID = "AppID0";
    }
    Taskbar.AppId = currentAppID;
}
```

This has interesting side effects, because the taskbar button is associated with additional resources such as the application jump list (which is also dependent on the application ID) and the taskbar overlay icon. Switching application IDs at run-time, however, has a great potential to confuse users, so the

recommended best practice is to set the application ID for your process or window during startup (or before startup, when configuring an IShellLink object or a shell association for your file type) and not change it at run-time.

> Note If terms like "IShellLink" and "IShellItem" seem confusing to you, make sure that you've read the section titled "Introduction to the Windows Shell" in the first chapter of this book. The following sections assume that you are familiar with fundamental shell concepts and interfaces, and they use the ITaskbarList3 interface extensively for interacting with the Windows 7 taskbar.

Because it is less straightforward to work with the unmanaged interface for changing a specific window's application ID, the following partial code listing should make it easier for you:

```
PROPVARIANT pv;
InitPropVariantFromString(L"MyAppID", &pv);

IPropertyStore *pps;
HRESULT hr = SHGetPropertyStoreForWindow(hwnd, IID_PPV_ARGS(&pps));
pps->SetValue(PKEY_AppUserModel_ID, pv);
pps->Commit();
```

It's important to remember that once you begin using explicit application IDs (overriding the default shell algorithm for determining the application ID for your windows) you must stick to the same consistent and deterministic approach. For example, if you forget to specify the application ID explicitly when populating a jump list, you will likely encounter an exception or undefined behavior if your window is associated with an explicit application ID.

One last thing to note is that only top-level windows can be associated with an application ID. While it might seem useful to associate a child window, an individual MDI document or a TDI tab with a taskbar button, this can accomplished through the use of custom window switchers, without using the explicit application ID mechanism. Custom window switchers are the subject of the subsequent chapter.

Taskbar Progress Bars and Overlay Icons

Earlier in this chapter, we have seen how taskbar overlay icons and progress bars give your application a stylish way to convey status information even if the application's window is not currently in the foreground or is not even shown. In this section, we will see the Win32 and managed APIs which your application must call to take advantage of this feature.

Setting a taskbar overlay icon is an extremely simple process. The ITaskbarList3 shell interface provides the SetOverlayIcon method, which you call passing two parameters – an icon handle (HICON) and an accessibility description string. The managed equivalent is the Windows7Taskbar.SetTaskbarOverlayIcon static method or an extension method that accepts a System.Windows.Forms.Form object and sets the icon. Setting the icon to null removes the overlay altogether, giving the taskbar button its original appearance.

Unfortunately, there is no simple process without a trick. If you're tempted to toggle the overlay icon during your application's startup process, there is a potential subtlety you must be aware of. Attempting to set an overlay icon before the taskbar button is created may result in an exception (when creating the ITaskbarList3 object). Therefore, you must register for the notification that the taskbar button has been created, which is delivered to your window procedure as a window message.

This message does not have a predefined code, so you must use the RegisterWindowMessage function (passing the TaskbarButtonCreated string as a parameter) to obtain the message number.

When using the managed wrapper, the application must first set the window handle for its main window using the Taskbar.ApplicationWindowHandle property. When using the taskbar APIs directly, the window handle is provided as a parameter to the ITaskbarList3::SetOverlayIcon method.

> Note To obtain a window handle in a Windows Forms application, use the Handle property of the Form class inherited by your form. In a Windows Presentation Foundation (WPF) application, instantiate a WindowInteropHelper instance with your WPF Window instance, and use the resulting object's Handle property to obtain the window handle.

The following code snippet shows how an instant messaging application (such as Windows Live Messenger) could change the overlay icon on its taskbar button as a result of changing the online presence status of the user. Providing an icon description for accessibility reasons is highly encouraged.

```
//C#
void OnlinePresenceChanged(PresenceStatus newStatus)
{
    Icon theIcon = _overlayIcons[(int)newStatus];
    Taskbar.OverlayImage = new OverlayImage(theIcon, newStatus.ToString());
}

//C++
ITaskbarList3* ptl;
CoCreateInstance(CLSID_TaskbarList, NULL, CLSCTX_ALL, IID_ITaskbarList3, (LPVOID*)&ptl);
ptl->SetOverlayIcon(hwnd, hicon, L"Accessible Description");
```

Controlling the taskbar progress bar is also the responsibility of the ITaskbarList3 interface, this time through its SetProgressState and SetProgressValue methods. The former method accepts an enumeration value which can have any of four values (discussed later), and the latter method accepts a current and a maximum value, expressed as unsigned long parameters.

The managed wrapper contains the same functionality in the Taskbar.ProgressBar.State and Taskbar.ProgressBar.CurrentValue static methods, but also requires setting the maximum value using the Taskbar.ProgressBar.MaxValue property before proceeding to use the progress bar.

> Note It's also fairly easy to write a progress bar control which automatically exposes progress information to the window's taskbar button, if present.

Taskbar-Integrated Progress Bar Control

A taskbar-integrated progress bar control reports to the taskbar any changes of the progress control value. This is a highly useful pattern to for an application which has a single progress bar displayed at a time – the progress bar's status is automatically reflected to the taskbar.

This is a classic example of the decorator design pattern – the TaskbarProgressBarControl class contains a ProgressBar and acts as one, but also reports progress to the Windows 7 taskbar when its value is updated.

```
public sealed class TaskbarProgressBarControl : UserControl
{
    ProgressBar progressBar;
```

```
    public TaskbarProgressBarControl()
    {
        progressBar = new ProgressBar();
    }

    public int Value
    {
        get
        {
            return progressBar.Value;
        }
        set
        {
            progressBar.Value = value;
            Taskbar.ProgressBar.CurrentValue = (ulong)value;
        }
    }

    public int MaxValue
    {
        get
        {
            return progressBar.Maximum;
        }
        set
        {
            progressBar.Maximum = value;
            Taskbar.ProgressBar.MaxValue = (ulong)value;
        }
    }

    public int MinValue
    {
        get
        {
            return progressBar.Minimum;
        }
        set
        {
            progressBar.Minimum = value;
        }
    }
}
```

This is a Windows Forms implementation of the control; it's quite straightforward to produce a similar implementation for Windows Presentation Foundation (WPF).

The taskbar progress bar is actually quite sophisticated, and besides displaying progress values it can also be set to four different states (indeterminate, normal, paused, and error). Here are the appearances of those taskbar progress states:

Indeterminate	Normal	Paused	Error
A green marquee 	A green bar 	A yellow bar 	A red bar 

The following code shows how to change the progress bar's value and state when performing work in an application using the Background Worker pattern of Windows Forms:

```
BackgroundWorker worker = new BackgroundWorker();

private void startButton_Click(object sender, EventArgs e)
{
    Taskbar.ProgressBar.State = TaskbarButtonProgressState.Normal;
    Taskbar.ProgressBar.MaxValue = (ulong)100;
    worker.DoWork += new DoWorkEventHandler(worker_DoWork);
    worker.RunWorkerAsync();
}

void worker_DoWork(object sender, DoWorkEventArgs e)
{
    for (int i = 0; i < 100; ++i)
    {
        //Do some work

        Taskbar.ProgressBar.CurrentValue = (ulong)i;
        worker.ReportProgress(i);

        if (worker.CancellationPending)
        {
            Taskbar.ProgressBar.State = TaskbarButtonProgressState.Error;
            return;
        }
    }
}

private void cancelButton_Click(object sender, EventArgs e)
{
    worker.CancelAsync();
}
```

Note that the creation time limitation applies to this API as well – you can modify the taskbar progress state or value only after the taskbar button has been created. See above for how to intercept this notification and ensure that you do not perform undefined operations.

As mentioned before, applications taking advantage of the Windows shell built-in file operations functionality (the SHFileOperation function and IFileOperation interface) get the taskbar progress behavior by default. Interacting with these APIs is quite simple, and provides a cancelable user interface for file operations exactly similar to that used by Windows Explorer. Using these APIs ensures that your file operations behave consistently to what the user expects, and as a bonus you get the taskbar progress for free while the operation takes place. Even a console application can take advantage of this API.

Using IFileOperation from managed code can be a bit tricky. Aside from the fact that you have to interact with COM interfaces, the advanced capabilities of this feature require that you register a COM sink (the COM event-handler equivalent), which is not quite straightforward in managed code. Fortunately, Stephen Toub's MSDN Magazine article ".NET Matters: IFileOperation in Windows Vista," published in December 2007, provides an elegant framework for performing shell I/O operations using the IFileOperation interface.

To design and develop your own applications taking advantage of IFileOperation, download the code for Stephen's article from http://msdn.microsoft.com/en-us/magazine/cc163304.aspx, make the FileOperation class public, and you're ready to go. First you feed the FileOperation object a list of

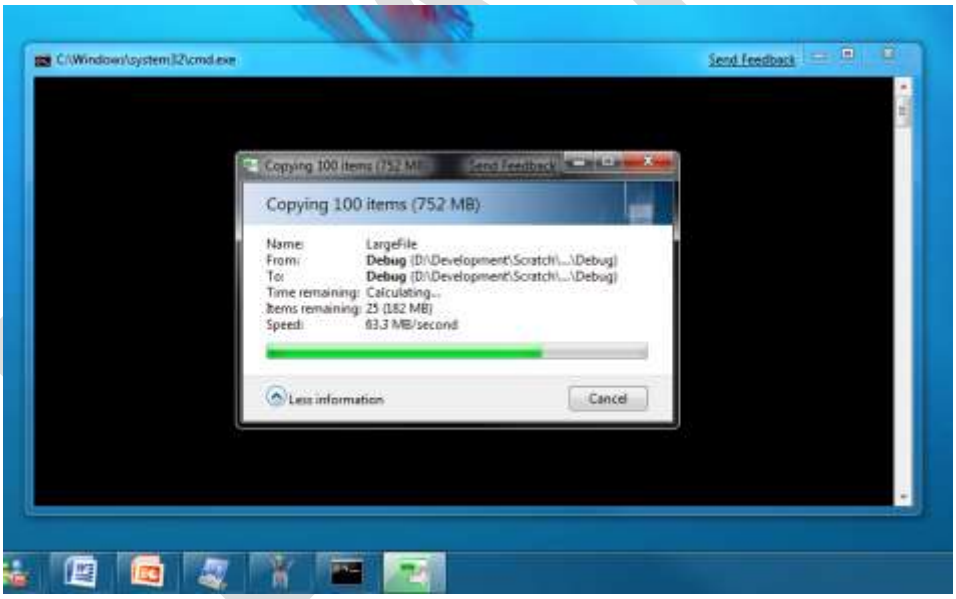
operations that you want to perform (copy, new item, delete, rename), and then you call the PerformOperations method to batch all operations together.

> Note Despite its appearance, the IFileOperation interface does not provide transactional semantics to file operations; the Transactional File System introduced in Windows Vista is accessible through a completely different set of APIs. A good place to start would be the documentation for the Win32 CreateFileTransacted function.

The following code shows how to use the file operation APIs to copy files from one location to another:

```
static void Main(string[] args)
{
    string file = CreateLargeFile();
    FileOperation operation = new FileOperation();
    for (int i = 0; i < 100; ++i)
    {
        operation.CopyItem(file, Path.GetDirectoryName(file),
            Path.ChangeExtension(Path.GetFileName(file), ".bak" + i));
    }
    operation.PerformOperations();
}
```

The progress dialog and progress in the taskbar icon are depicted here:



Summary

In this chapter, you've seen why the Windows 7 taskbar has been designed as a revolution of launch surfaces, a consolidated area of the Windows shell that contains facilities for application launching, switching, and obtaining status information, which improves user productivity.

The Windows 7 taskbar is a collection of new features for application developers, giving you a differentiating opportunity to shine on the Windows 7 platform and giving your users an unforgettable experience from their very first logon. These features include jump lists, thumbnail toolbars, progress bars, overlay icons, live previews, and tab thumbnails.

We also explored the details of assigning an application ID to a process or a window to exercise fine-grained control over the allocation of taskbar buttons to active windows. You saw how to use the Win32 and the managed APIs to set an overlay icon for a taskbar button and to modify a taskbar button's progress state and value.

In Chapter 3, "Integrate with the Windows 7 Taskbar, Part 2," we'll continue to experiment with the new Windows 7 taskbar APIs, and we'll take a deep look at the implementation of jump lists, thumbnail toolbars, custom window switchers, and thumbnail customization.

DRAFT