Writing DPI-Aware Desktop and Win32 Applications

This practical guide offers insight into writing DPI–aware Microsoft Win32 applications. Writing a DPI–aware application is the key to making a UI look consistently good across a wide variety of high-DPI display settings. Applications that are not DPI–aware but are running on a high-DPI display setting can suffer from many visual artifacts, including incorrect scaling of UI elements, clipped text, and blurry images. By adding support in your application for DPI awareness, you ensure that the presentation of your application's UI is more predictable, making it more visually appealing to users. This gives the user the best possible experience on any display.

Since Windows XP, dots per inch (DPI) settings have been a component of the Windows development platform. For many years, high density displays did not consume a large percentage of the market. Now, due to the clear and perceivable user benefits, such as text legibility and image presentation, high density displays are becoming increasingly popular, particularly in mobile form factors. With the growing relevance of high DPI devices, it is important to enable each monitor of a system to run with optimal DPI settings to take full advantage of all display hardware. Windows 8.1 adds developer support that enables desktop applications to not only become aware of different monitor DPI settings, but to also respond to any dynamic DPI changes. This gives the user with the best possible experience on any display.

Writing a DPI–aware application is the key to making a UI look consistent across a wide variety of DPI display settings. An application that is not DPI–aware but is running on a high DPI display or across monitors of different DPIs will be scaled by the system to the appropriate size so that it is still usable, but can suffer from visual artifacts including incorrect scaling of UI elements, clipped text, and blurriness. By adding support in your application for DPI awareness level, you can present your application’s UI in a predictable manner. By updating your app to respond to dynamic changes in DPI, you create an application that is crisp, making it more visually appealing to users.

Because manufacturers now ship greater numbers of high-resolution displays, you should not write applications that assume the default 96 DPI setting. For a list of optimal DPI configuration examples, see Appendix B: Optimal Configuration Examples. In Windows 8.1, each monitor in a configuration can run at a different DPI. This means that you should not write applications that assume that DPI is a static value that applies to all displays during a single session. To provide the best user experience, make sure that your applications are DPI–aware and can respond to changes in DPI.

This guide explores DPI features and issues in Windows XP, Windows Vista, and later versions of operating systems, with extra emphasis on Windows 8.1. It provides a set of guidelines for assessing DPI awareness level and a set of solutions that can help you address DPI awareness level issues.

High DPI Features in Windows

This section provides an overview of the high DPI features supported in Windows XP through Windows 8.1. The following table shows a list of high DPI-related features that are supported by each platform.

Feature	Windows XP	Windows Vista	Windows 7	Windows 8	Windows 8.1

Control Panel setting of DPI	Yes	Yes	Yes	Yes	Yes
DPI virtualization of not DPI–aware applications	No	Yes	Yes	Yes	Yes
DPI virtualization of system-DPI aware applications	No	No	No	No	Yes
API to declare DPI awareness level	No	Yes	Yes	Yes	Yes*
APIs to retrieve system metrics and DPI	Yes	Yes	Yes	Yes	Yes
Window notification	No	No	No	No	Yes
APIs to retrieve monitor DPI	No	No	No	No	Yes
Requires a reboot/log off for monitor DPI change	N/A	N/A	N/A	N/A	No
Requires a reboot/log off for system DPI change	Reboot	Reboot	Log off	Log off	Log off
Per user DPI setting	No	No	Yes	Yes	Yes
Auto configuration of DPI at first logon	No	No	Yes	Yes	Yes
Per monitor-DPI aware	No	No	No	No	Yes
Viewing distance incorporated in default DPI calculation	No	No	No	No	Yes

*These features have been updated and expanded to account for per monitor-DPI aware in Windows 8.1.

When you install Windows 7 onto a computer that supports extended display identification data (EDID), Windows 7 automatically configures the computer with an optimal high-DPI setting to make best use of the display's physical DPI, except in the cases where the display's effective resolution is less than 1024 x 768.

To change the user DPI setting, follow the instructions in the section Setting High DPI in Windows 7.

Setting DPI by Using Control Panel

Windows XP, Windows Vista, and later versions support the ability to change high-DPI display settings. For information about setting high-DPI on Windows XP, Windows Vista, and later versions, refer to Appendix A: Setting High DPI in Windows.

When the DPI settings are changed, the system fonts and system UI elements change in size. This is the primary reason that applications need to consider the system DPI setting in their rendering and layout code. Applications that are not DPI–aware can potentially exhibit some visual artifacts such as mismatched font sizes, clipped text, or clipped UI elements.

System DPI vs. Per-Monitor DPI

From Windows XP to Windows 8, Windows operated on a system-wide DPI, referred to as system DPI. Windows calculated the system DPI value on first logon with the goal of selecting the DPI that provides the best experience for the given hardware. Users can override the default system DPI in Control Panel to make the UI larger or smaller than the default. This override is explained in Appendix A: Setting High DPI in Windows.

Windows 8.1 introduces per-monitor DPI. On logon, Windows selects the optimal DPI for each monitor of the system. Users can still override these DPI values in Control Panel, as explained in Windows 8.1 Control Panel Overview. In Windows 8.1, system DPI is maintained for backward-compatibility. Many existing applications use system DPI to select which size to render. The system DPI also determines the size of system assets that an application uses. Additionally, APIs that do not have a monitor-specific context reflect values that are based on system DPI when not virtualized.

Categories of Applications

In Windows 8.1, desktop applications fall into three categories with respect to DPI:

- Not DPI–aware applications
- System–DPI aware applications
- Per monitor–DPI aware applications

Not DPI–aware Applications

Not DPI–aware applications are applications that always render at 96 DPI, the lowest desktop DPI plateau. This class of applications are unaware of different system DPIs. The Desktop Window Manager (DWM)virtualizes and scales these applications to account for high DPI.

System–DPI Aware Applications

System–DPI aware applications are considered DPI–aware in Windows Vista through Windows 8. These applications render at the system DPI to avoid being scaled. These applications address the issues enumerated in Appendix C: Common High DPI Issues, but they do so on a system DPI level rather than per-monitor DPI because they cannot respond to dynamic changes in DPI during a single session. System–DPI aware applications render optimally on the primary display, and DWM does not scale and virtualize them. However, if the user moves the application to a display with a higher or lower DPI, DWM scales it up or down. The effect is that the window and content size are appropriate for every display, but the scaling introduces blurriness.

Per Monitor–DPI Aware Applications

Per monitor–DPI aware applications are a new class of applications in Windows 8.1. These applications dynamically scale up or down when a user changes the DPI or moves the application between monitors that have different DPIs. These applications always render crisply and at the correct size for a given display. DWM does not scale and virtualize this class of applications.

Note that the non-client area of a per monitor–DPI aware application is not scaled by Windows, and will appear proportionately smaller on a high DPI display.

DPI and the Desktop Scaling Factor

DPI is the physical measurement of the number of pixels in a linear inch of a display. For desktop monitors, this is typically 96 DPI or lower. However, newer displays for tablets and laptops may be 192 DPI or higher. In Windows 8.1, DPI values for the desktop are divided into four groups: 96, 120, 144, and 192. Each group contains a range of DPI values.

The desktop scaling factor is a percentage that indicates how Windows scales the UI of a desktop application when its DPI awareness level allows scaling. In general, the desktop scaling factor is based on the DPI:

- 96 DPI = 100% scaling
- 120 DPI = 125% scaling
- 144 DPI = 150% scaling
- 192 DPI = 200% scaling

In practice, other things can affect the desktop scaling factor, such as viewing distance or minimum required lines of vertical resolution. For example:

- A projector that is 96 DPI may end up with a scaling factor of 125% or 150% because such devices have a higher optimal viewing distance than desktop monitors.
- The Surface Pro is 212 DPI but has a 150% default scaling factor. This is because its native resolution of 1920x1080 would shrink to a logical resolution of 960x640 if a 200% scaling factor were used. That resolution is below the minimum resolution of 1024x720, so the scaling factor steps down to the next lower plateau of 150%.

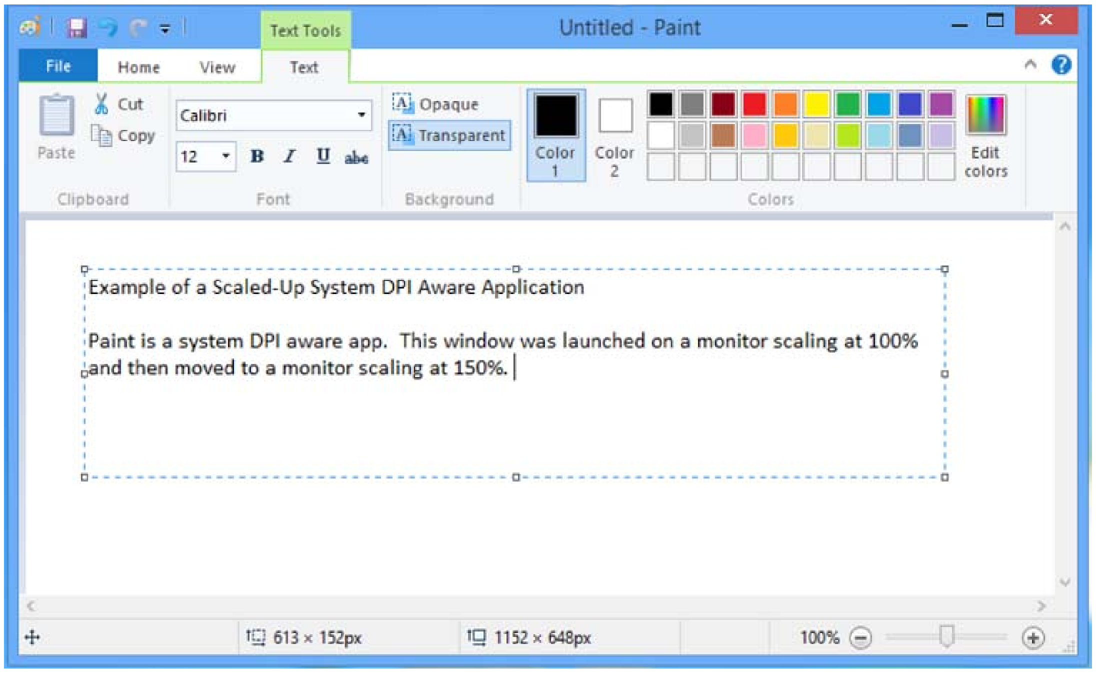
DPI Virtualization and Scaling

Windows Vista introduced a feature called DPI *virtualization*, which provides a level of automatic scaling support to applications that are not DPI–aware. With this feature, Windows scales the size of the text and UI elements of applications that are not DPI-aware so that they are appropriately sized on high DPI settings without changes in the application. This prevents potential usability and readability issues that occur when applications render too small on high DPI screens.

In Windows Vista through Windows 8, this feature provides "virtualized" system metrics and UI elements to not DPI–aware applications, as if they are running at 96 DPI. The application then renders to a 96 DPI off-screen surface and DWM scales the resulting application window to match the DPI setting. For example, if the DPI display setting is 144, DWM scales the application's window by 150%, or 144/96.

In Windows 8.1, this feature is extended to provide “virtualized” monitor metrics and UI elements to system DPI–aware applications, as if they are running at the system DPI. Similar to the virtualization of not DPI–aware applications, this virtualization means the application renders at system DPI to an off-screen surface and DWM scales the resulting application window to match the monitor DPI. For example, if the system DPI is 144 and an application is dragged to a secondary monitor of 96 DPI, DWM scales the application window 67%, or 96/144.

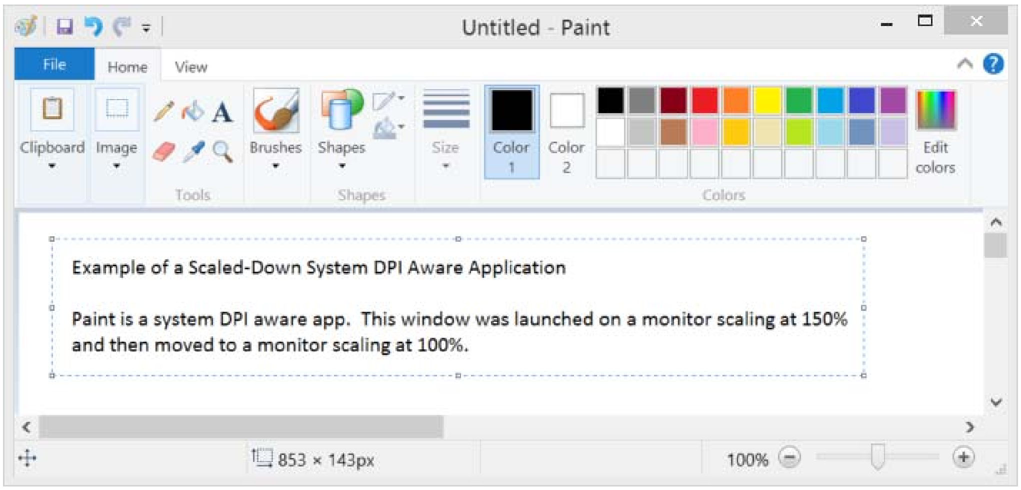
The type of scaling that DPI virtualization uses when scaling up in size is based on pixel stretching. While this enables applications to be appropriately sized, blurriness occurs due to the stretched pixels. The next screen shot shows the type of visual artifact caused by stretched pixels due to DPI virtualization.



Example of a Scaled-Up System DPI Aware Application

Paint is a system DPI aware app. This window was launched on a monitor scaling at 100% and then moved to a monitor scaling at 150%.

The type of scaling that DPI virtualization uses when scaling down in size is based on pixel sampling. While this enables applications to be appropriately sized, some wash-out or loss of information occurs due to the sampling of pixels. The next screen shot shows the type of visual artifact caused by sampled pixels due to DPI virtualization.



Example of a Scaled-Down System DPI Aware Application

Paint is a system DPI aware app. This window was launched on a monitor scaling at 150% and then moved to a monitor scaling at 100%.

The goal of the DPI virtualization feature is to ensure that the size of the text and UI in applications that are not per monitor-DPI aware is appropriate for the DPI of the monitor on which they are being displayed.

Virtualization is not able to solve all application issues due to the differences in application behavior throughout the entire ecosystem.

If your application has issues on Windows Vista or later versions that are caused by DPI virtualization, users can turn off DPI virtualization and scaling for your application without affecting other applications. To disable DPI virtualization for a single application, right-click the name of the application executable file, click **Properties**, click the **Compatibility** tab, and then select the box labeled **Disable display scaling on high DPI settings**.

For more information about compatibility settings for Windows, see the Make older programs run in this version of Windows topic in Windows Help and How-to.

Comparison of DPI Awareness Levels

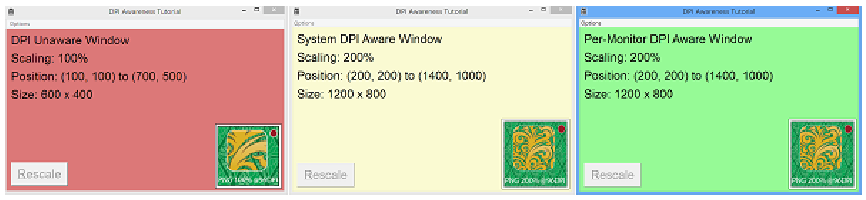
Here are scenarios that outline when and how each class of application is handled in different monitor configurations. In addition, the High DPI Tutorial shows a simple application at each awareness level.

Scenario 1: Primary display = 200% scaling; secondary display = 100% scaling

Table A. User launches the application on the 200% display.

DPI awareness	**Description**

level	
Not DPI–aware	Not DPI–aware applications are always virtualized to 96 DPI (100%). On the high DPI (200%) display, the system scales up the UI. Note that the Not DPI–aware application thinks that it is 600x400 on a 100% scaling display. In reality, on the 200% display it is twice that size in terms of pixels. This makes its actual size on the high DPI display appear similar to its size when unscaled on the low DPI display.
System-DPI aware	The application determines the DPI when the user launches it and renders itself appropriately. Its window size and other UI account for the high DPI. Its bitmap and fonts are appropriate for the high DPI display.
Per monitor-DPI aware	The application renders itself for the DPI when the user launches it, with window size and UI elements appropriate for the high DPI display.



DPI Unaware Window
Scaling: 100%
Position: (100, 100) to (700, 500)
Size: 600 x 400

System DPI Aware Window
Scaling: 200%
Position: (200, 200) to (1400, 1000)
Size: 1200 x 800

Per-Monitor DPI Aware Window
Scaling: 200%
Position: (200, 200) to (1400, 1000)
Size: 1200 x 800

Next is a close-up view of the text that shows how pixel duplication affects not DPI–aware applications the system scales up. By comparison, the per monitor-DPI aware application selects a font that is appropriate for that DPI level. The DPI–aware application provides better output.

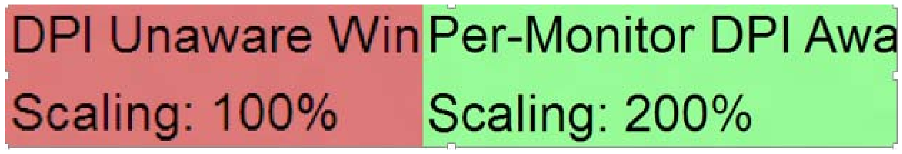
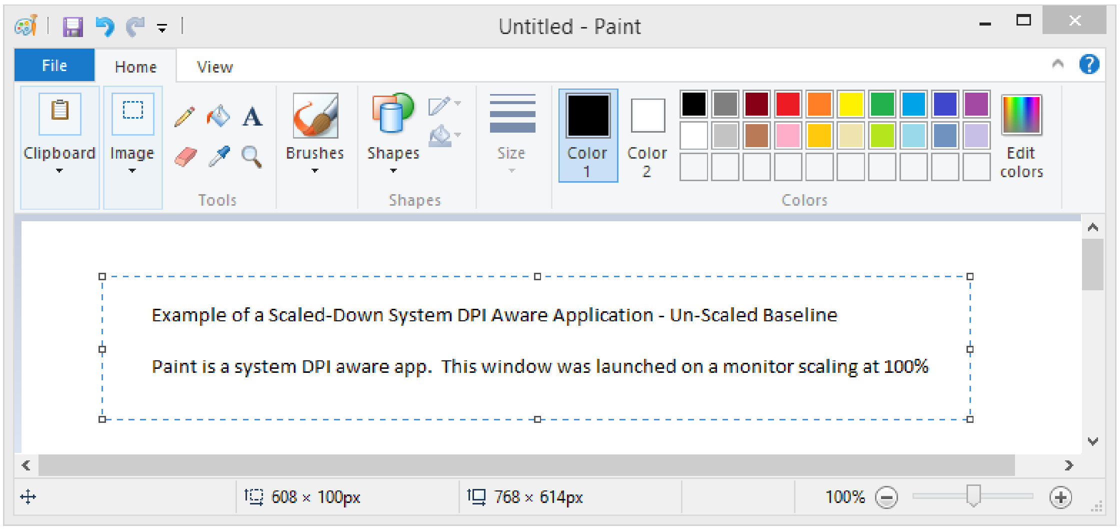


Table B. User launches the application on the 200% display and moves it to the 100% display.

DPI awareness level	**Description**
Not DPI–aware	Not DPI–aware applications are always virtualized to 96 DPI (100%). On a 100% display, the UI is unscaled.
System-DPI aware	When the user launches the application on the 200% display, the application adjusts for the high DPI. The application does not account for the possibility of other DPI levels. In this case, the application was initially rendered for high DPI. When the user moves the application, the system scales it down so that its UI remains consistent in size on the low DPI display. Without this scaling, the application would appear disproportionately large on the 96 DPI monitor. Note that the system DPI–aware application still thinks that it is 1200x800 on a 200% scaling display and still has the bitmap selected for 200%.
Per monitor-DPI aware	The application resizes its window and the elements of its UI, including its child window (button) and fonts. It renders a different bitmap that is tailored for 100 DPI.

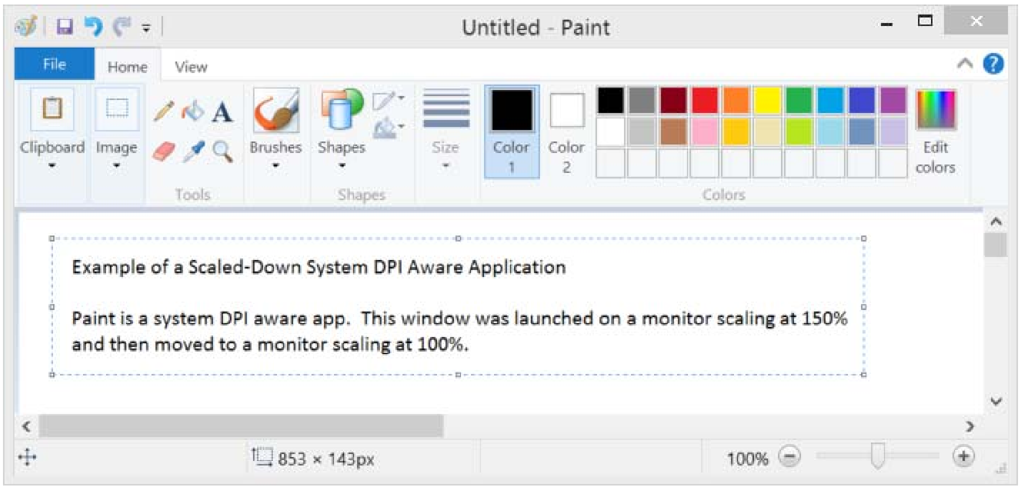
Applications may appear blurry when a system-DPI aware app is scaled down. The blurriness is most pronounced when the scaling is non-integral. The following example compares a Paint window that a user launched on a 100% display with a Paint window that a user launched on a 150% display and then moved to a 100% display.

The top image is the unscaled Paint window. The bottom image is the scaled down Paint window. Note the scaled-down effect on the text and graphical elements, such as the dotted line icon of the Image button.



Example of a Scaled-Down System DPI Aware Application - Un-Scaled Baseline

Paint is a system DPI aware app. This window was launched on a monitor scaling at 100%



Example of a Scaled-Down System DPI Aware Application

Paint is a system DPI aware app. This window was launched on a monitor scaling at 150% and then moved to a monitor scaling at 100%.

Scenario 2: Primary display = 100% scaling; secondary display = 200% scaling

Table A. User launches the application on the 100% display.

DPI awareness level	Description
Not DPI–aware	In this case, the application's UI is equivalent for all three DPI awareness levels. The system-DPI aware and per monitor-DPI aware applications selected the bitmap for the 100% display.
System-DPI aware	
Per monitor-DPI aware	

Table B. User launches the application on the 100% display and moves it to the 200% display.

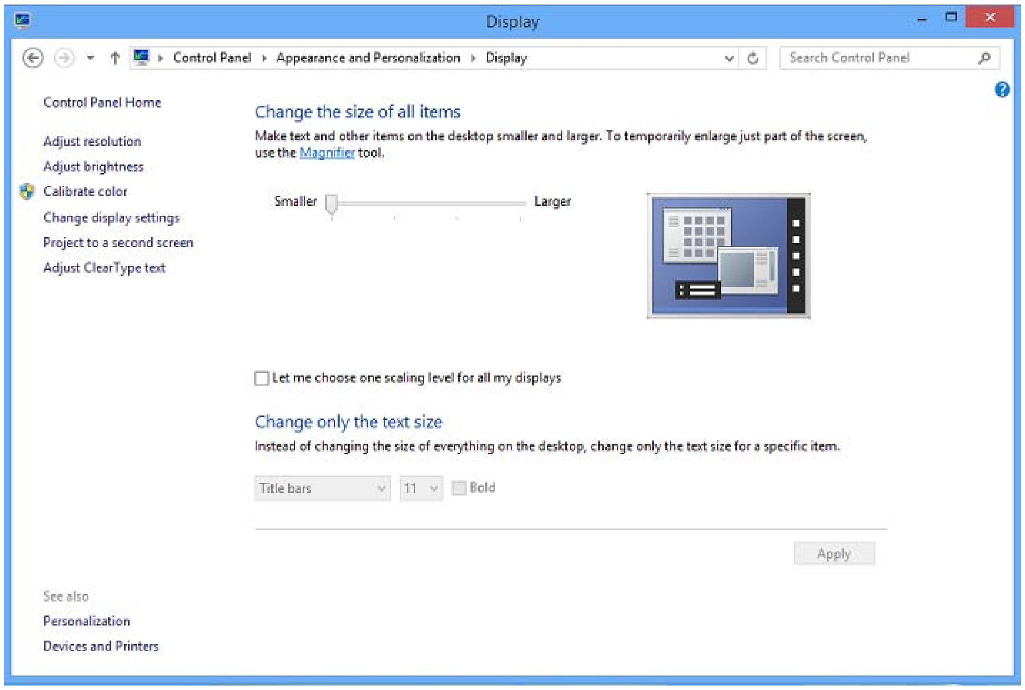
DPI awareness level	Description
Not DPI–aware	The system scales up the application's UI to 200%. The fonts and bitmap are magnified, which results in pixilation on the high DPI display.
System-DPI aware	The system scales up the application's UI to 200%. The system-DPI aware application rendered itself expecting that 96 DPI (100%) is the only DPI level on the system.
Per monitor-DPI aware	The application resizes its window and the elements of its UI, including its child window (button) and fonts. It renders a different bitmap that is tailored for 200 DPI.

Setting DPI Using Control Panel

The system calculates the optimal DPI for each monitor at first logon. In addition, users can change DPI settings in Control Panel. In Windows 8.1, Control Panel accounts for per monitor-DPI awareness, and also allows users to revert back to a setting in which all monitors have the same DPI.

Windows 8.1 Control Panel Overview

The **Set one scaling level for all my displays** checkbox determines whether Windows 8 (checked) or Windows 8.1 (unchecked) scaling behavior is applied:



Change the size of all items

Make text and other items on the desktop smaller and larger. To temporarily enlarge just part of the screen, use the Magnifier tool.

☐ Let me choose one scaling level for all my displays

Change only the text size

Instead of changing the size of everything on the desktop, change only the text size for a specific item.

- Windows 8 behavior: Not DPI–aware desktop applications are scaled according to the DPI of the primary monitor. Users can change this system-wide scaling factor by selecting a different radio button or entering a custom value.
- Windows 8.1 behavior: Not DPI–aware desktop applications are scaled according to the DPI of each monitor individually. Users can further adjust by using the desktop scaling override.

If the user checks the **Set one scaling level for all my displays** checkbox, the Windows 8 functionality is active and Control Panel displays the Windows 8 radio buttons for applying a system-wide scale factor. In other words, per-monitor scaling is disabled.

If the user does not check the **Set one scaling level for all my displays** checkbox, the Windows 8.1 per-monitor scaling functionality is active, and Control Panel replaces the radio buttons with a **Smaller/Larger** slider control. This control sets a desktop scaling override. The slider can have several settings depending on the range of DPI plateaus in the current machine configuration.

If the user applies a new desktop scaling override value, the scaling of each monitor is adjusted up or down accordingly, if possible. If a monitor is already at the minimum (96 DPI = 100% scaling) or maximum (192 DPI = 200% scaling) the override has no effect for that monitor. If scaling up for a monitor would result in its virtual size being less than the minimum resolution of 1024x720, the override has no effect. When the user applies a change to the **Smaller/Larger** slider control (via the **Apply** button) the rescaling takes effect immediately. This bitmap rescaling affects all applications. DPI–aware applications may have some content blurriness until the next sign-in when they re-read the DPI and render themselves accordingly.

For information about setting DPI in Control Panel on Windows XP through Windows 8.1, see Appendix A: Setting High DPI in Windows.

System DPI Change

When the system DPI settings change, for example, when a user logs off and on, the system fonts and system UI elements change in size. This is the primary reason that applications have needed to consider the system DPI setting in their rendering and layout code since Windows XP. Applications that are not DPI–aware can potentially exhibit some visual artifacts such as mismatched font sizes, clipped text, or clipped UI elements.

Dynamic DPI Change

With the introduction of per monitor-DPI awareness in Windows 8.1, DPI changes occur without user logoff. If you update your applications to be per monitor-DPI aware, they dynamically respond to DPI changes. Applications that are not per monitor-DPI aware do not respond dynamically. Instead, the system virtualizes and scales them to the new appropriate size until the user logs off. Verify that your application responds to dynamic DPI change appropriately.

Legacy Mode

In Windows 8.1 Control Panel, users can turn off the per monitor-DPI awareness feature. To do this, a user selects the **Let me choose one scaling level for**

all my displays check box. This change applies the system DPI setting to all monitors, which means the system is in the same state as in Windows 8. Verify that your application works in this mode.

Supporting Dynamic DPI Changes

You must perform the following steps to make your application per monitor-DPI aware:

- Set the DPI awareness level
- Get the DPI for the current monitor
- Listen for DPI changes
- Respond to changes

For a detailed walkthrough, see the Tutorial: Writing High-DPI Win32 Applications topic.

Step 1: Set the DPI Awareness Level

Mark the application as per monitor-DPI aware by modifying the application manifest or by calling the **SetProcessDpiAwareness**API. We recommend that you use the application manifest because that sets the DPI awareness level when the application is launched. Use the API only in the following cases:

- Your code is in a dll that runs via rundll32.exe. This is a launch mechanism that does not support the application manifest.
- You need to make complex run-time decisions based on OS version or other considerations. For example, if you need the application to be system-DPI aware on Windows 7 and dynamically aware on Windows 8.1, use the True/PM manifest setting.

If you use the **SetProcessDpiAwareness** method to set the DPI awareness level, you must call **SetProcessDpiAwareness** prior to any Win32API call that forces the system to begin virtualization.

DPI awareness manifest value	Description
False	Sets the application to not DPI-aware.
True	Sets the application to system DPI–aware.
Per-monitor	On Windows 8.1, sets the application to per monitor-DPI aware. On Windows Vista through Windows 8, sets the application to not DPI–aware.
True/PM	On Windows 8.1, sets the application to per monitor-DPI aware. On Windows Vista through Windows 8, sets the application to system-DPI aware.

For information about how to make this change in either the manifest or by calling the API, see the Tutorial: Writing High-DPI Win32 Applications.

Step 2: Get the DPI for the current monitor

Query for the DPI of the monitor that the application is currently on by calling **GetDPIForMonitor**. This returns the DPI value that the application should use in determining the appropriate assets and layout.

Step 3: Listen for DPI changes

Dynamically DPI–aware applications must first draw based on monitor DPI. Then they must respond to DPI changes. By listening for the **WM_DPICHANGED** notification, the application receives notification when a DPI change occurs. The application can query again for the monitor DPI. This notification fires when the user moves the application between monitors of different DPI and when the user changes the DPI in Control Panel. When this notification fires, a pointer to the suggested new window size and location is sent. The application should use these to avoid constantly rescaling between the two monitors while the user moves the application. If you do not want your application to use the suggested size, you can ignore the suggested size and location information.

Step 4: Respond to DPI changes

When an application receives notification that DPI has changed, it scales its content, selects new assets, redraws, and adjust the layout based on the new DPI. For more information, see the Tutorial: Writing High-DPI Win32 Applications topic.

Assessing DPI Compatibility

To assess DPI compatibility, do the following:

- Address all common High DPI issues. For more information, see Appendix C: Common High DPI Issues.
- Fix any additional DPI concerns in your application.
- Test your application at the recommended high DPI and per-monitor DPI display settings and resolutions. For more information, see the Testing DPI Compatibility section in the Tutorial: Writing High-DPI Win32 Applications topic.
- Address DPI issues by using the techniques described in the topic Addressing High DPI Issues.

Additional DPI Concerns

Custom Gestures

To handle gestures across multiple monitors with different DPIs, you must scale himetrics in addition to applications. If your application uses custom gestures, you must calculate the scale factor that is used for himetrics so that you can unscale before you process and respond to gestures. If your application uses only gestures that are provided by the Windows platform, you do not need to make changes to account for gestures.

To get the scale factor that the system uses, complete the following calculation:

$$ScaleFactor = \frac{(logical\ monitor\ width)/(logical\ desktop\ width)}{(physical\ monitor\ width)/(physical\ desktop\ width)}$$

To get the logical monitor width, use one of the following:

```
MONITORINFOEX LogicalMonitorInfo;
LogicalMonitorInfo.cbSize = sizeof(MONITORINFOEX);
GetMonitorInfo(hMonitor, &LogicalMonitorInfo);
LogicalMonitorWidth = LogicalMonitorInfo.rcMonitor.right – LogicalMonitorInfo.rcMonitor.left;
```

To get the logical desktop width, use:

```
LogicalDesktopWidth = GetSystemMetrics(SM_CXVIRTUALSCREEN);
```

To get the physical desktop width and physical monitor width, call **QueryDisplayConfig**. For each **DISPLAYCONFIG_MODE_INFO** in the returned *pModeInfoArray*, check the **pModeInfoArray[i].sourceMode.width** (and **height** and **position**) to reconstruct the bounding rect of all the monitors.

Transforming from Physical to Logical Space

In Windows 8, system DPI–aware applications translate between physical and logical space using **PhysicalToLogicalPoint** and **LogicalToPhysicalPoint**. In Windows 8.1, the additional virtualization of the system and inter-process communications means that for the majority of applications, you do not need these APIs. As a result, in Windows 8.1, these APIs no longer transform points. The system returns all points to an application in its own coordinate space.

This behavior preserves functionality for the majority of applications, but there are some exceptions in which you must make changes to ensure that the application works as expected.

Accessing Window Trees while System DPI–aware

Applications for accessibility or developer tools often need to walk the entire window tree of another process. In this case, the application needs actual physical coordinates rather than virtualized physical coordinates and use **PhysicalToLogicalPointForPerMonitorDPI** and **LogicalToPhysicalPointForPerMonitorDPI**. These APIs return actual physical coordinates regardless of the process awareness level. If applications updated to be per-monitor DPI–aware, they always receive actual physical coordinates, so this change is not needed.

Using Absolute Pixel Values before Transformation

If an application queries for a point and performs operations on that point with absolute pixel values before it transforms between logical and physical coordinate spaces, the application will face offsets in this point because of system virtualization. Because applications receive all information in their own coordinate space, these transformations happen on the point after transformation between physical/logical spaces rather than before. Any operations, such as multiplying or dividing a point by a given value, still work as expected, but if an application was adding x pixels to a point before calling the transform functions in Windows 8, it will experience a slight offset from this operation. If an application makes any change after the transformation between physical and logical coordinate spaces, the change is unaffected.

Testing DPI Compatibility

To assess your application's DPI compatibility, test your application at a variety of resolutions with different high DPI settings.

The following table provides a recommended set of DPI settings and minimum resolutions to consider when testing DPI awareness level.

DPI Setting	Minimum resolution
96 (100%)	1024x720
120 (125%)	1280x960

144 (150%)	1536x1080
192 (200%)	2048x1440

The following table provides a recommended set of DPI configurations, with the expected result from each. This table is designed such that the table reflects what happens to an application when it is moved from primary to secondary. In this case, primary and secondary could also mean DPI before user override and DPI after user override, since the system scales until logoff in this case.

Secondary (Primary)	**96 (100%)**	**120 (125%)**	**144 (150%)**	**192 (200%)**
96 (100%)	No Change	Application scaled 125%	Application scaled 150%	Application scaled 200%
120 (125%)	Application scaled 80%	No Change	Application scaled 120%	Application scaled 160%
144 (150%)	Application scaled 66.7%	Application scaled 83.3%	No Change	Application scaled 133.3%
192 (200%)	Application scaled 50%	Application scaled 62.5%	Application scaled 75%	No Change

When you log any DPI issues into your bug tracking system, it is often useful to include the following data points:

- Screen shot of the visual artifact
- DPI configuration, including whether DPI virtualization is enabled
- Screen resolution used to reproduce the issue

Resources

Creating a DPI–aware Application
Pixel Density
How to Write High DPI Applications
DPI–aware applications in Windows Vista
Windows Vista DPI scaling: my Vista is bigger than your Vista
What are SYSTEM_FONT and DEFAULT_GUI_FONT
Windows User Experience Interaction Guidelines
Make older programs run in this version of Windows
SetProcessDpiAwareness
GetProcessDpiAwareness
GetDPIForMonitor
WM_DPICHANGED
QueryDisplayConfig
PhysicalToLogicalPoint
LogicalToPhysicalPoint
DPI Tutorial sample

Addressing High-DPI Issues

There are several techniques you can use to resolve high-DPI issues in your application. These techniques include:

- Declaring DPI awareness
- Using system metrics to calculate layout
- Determining the DPI scale factor
- Scaling text
- Scaling graphics
- Scaling layout
- Handling minimum effective resolution

Declaring DPI Awareness

When an application declares itself to be DPI-aware, it is a statement specifying that the application behaves well at DPI settings up to 200 percent DPI. In Windows XP, DPI awareness has no impact on the application or the operating system, but it has meaning on both Windows Vista and later versions. In Windows Vista and later versions, when DPI virtualization is enabled, applications that are not DPI-aware are scaled, and applications receive virtualized data from the system APIs, such as the **GetSystemMetric** function.

Note By default, the DPI virtualization feature is enabled only when the DPI display setting is greater than 120 (125 percent).

Although the Win32 API provides a function declaring an application as DPI-aware, its use is discouraged, except in very specific circumstances. For more information, see the SetProcessDPIAware function. In general, using an application manifest is the recommended process for declaring an application to be DPI-aware.

Using an Application Manifest

To declare your application to be DPI-aware, add <dpiAware> to the application manifest. Here is an example of how to use the <dpiAware> element in an application manifest.

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0" xmlns:asmv3="urn:schemas-microsoft-com:asm.v3"
  <asmv3:application>
    <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">
      <dpiAware>true</dpiAware>
    </asmv3:windowsSettings>
  </asmv3:application>
</assembly>
```

Note If the <dpiAware> element appears in the assembly manifest of a DLL component, the setting is ignored. Only the assembly manifest for the application can enable DPI awareness.

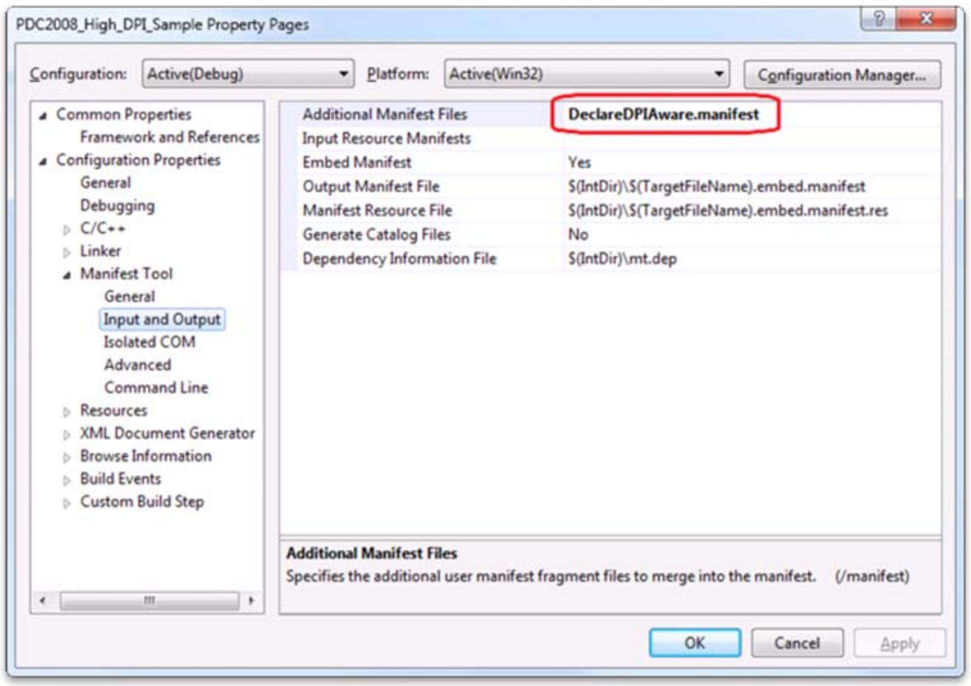
To merge with the existing assembly manifest for your application, you can use Mt.exe, which is available in the latest Windows SDK, or use Visual Studio.

Merge using Mt.exe

1. Save the manifest information in the preceding example to a file named DeclareDPIAware.manifest.
2. Download the latest Windows SDK.
3. Merge with the existing manifest "DeclareDPIAware.manifest" by running the following commands:
   - mt.exe -inputresource:application_name.exe;#1 -out:extracted.manifest
   - mt.exe -manifest extracted.manifest DeclareDPIAware.manifest -out:merged.manifest
   - mt.exe -outputresource:application_name.exe;#1 -manifest merged.manifest

Merge using Visual Studio

1. Save the manifest information in the preceding example to a file named DeclareDPIAware.manifest.
2. Open your application solution. On the **Project** menu, click **Property**. The **Property Pages** dialog box appears.
3. In the left pane, expand **Configuration Properties**, expand **Manifest Tool**, and then click **Input and Output**.
4. In the **Additional Manifest Files** text box, type **DeclareDPIAware.manifest**, and then click **OK**, as shown in the following screen shot from Microsoft Visual Studio 2008.



Additional Manifest Files	DeclareDPIAware.manifest
Input Resource Manifests	
Embed Manifest	Yes
Output Manifest File	$(IntDir)\$(TargetFileName).embed.manifest
Manifest Resource File	$(IntDir)\$(TargetFileName).embed.manifest.res
Generate Catalog Files	No
Dependency Information File	$(IntDir)\mt.dep

Additional Manifest Files
Specifies the additional user manifest fragment files to merge into the manifest. (/manifest)

For more information about the use of manifests, see the MSDN Library topic, Manifest Generation in Visual Studio. For more information about using assembly manifests, see the MSDN Library topic, Manifests [Side-by-Side Assemblies].

Note You might receive the warning: manifest authoring warning 81010002: Unrecognized Element "application" in namespace "urn chemas-microsoft-com:asm.v3". This is due to a known bug in the manifest compiler and you can safely ignore it.

Note To avoid the above warning, install Windows 7 SDK or Visual Studio 2010.

After you have added the manifest to your application, you can test your application by running it at 144 DPI with DPI virtualization enabled. If you have successfully declared your application as DPI-aware, you should not see the blurring of your application UI due to scaling because of DPI virtualization.

SetProcessDPIAware Function

The **SetProcessDPIAware** function in Windows Vista and later versions sets the current process as DPI-aware. However, the use of the **SetProcessDPIAware** function is discouraged. For example, if a DLL caches DPI settings during initialization, invoking **SetProcessDPIAware** in your application might generate a possible race condition. For this reason, we recommend that an application enable DPI awareness by using the application's assembly manifest rather than by calling **SetProcessDPIAware**.

By adding the <dpiAware> element to your application's assembly manifest, you mark your application as being DPI-aware. The user32.dll module, which provides Windows user interface functionality, checks the application's DPI awareness setting. If an application is determined to be DPI-aware, the user32.dll module calls **SetProcessDPIAware** on behalf of the application.

Note A DLL component should respect the DPI-aware setting of an application and not call **SetProcessDPIAware**.

The most common case for requiring the **SetProcessDPIAware** function is generic hosts that load a DLL and execute from a specified entry point. Three examples of generic hosts are the command-line utility program Rundll32, the DllHost process, and the Microsoft Management Console (MMC).

When a DLL is loaded from a generic host, it is essentially the entry point of the application. Any DLLs implicitly linked to by your DLL will be initialized before the application. Therefore, **SetProcessDPIAware** should always be called before any initialization to prevent any of those DLLs from caching DPI-sensitive metrics.

However, you should avoid generic hosts whenever possible when writing new code. Instead, you should write a small executable containing the proper manifest entries. Many Control Panel applets in Windows Vista and later versions use this technique.

Getting System Information

The following Win32 API functions are useful for retrieving information about the current display setting:

- **GetDeviceCaps** Retrieves device-specific information for the specified device.
- **GetSystemMetrics** Retrieves the specified metric or system configuration setting.
- **SystemParametersInfo** Retrieves or sets the value of one of the system-wide parameters.

GetDeviceCaps Function

The **GetDeviceCaps** function enables you to retrieve the number of pixels per logical inch along the screen width and height. In a system with multiple display monitors, this value is the same for all monitors. The following code example shows how to retrieve the horizontal and vertical DPI for the current display setting.

```
// From CDPI::_Init()
HDC hdc = GetDC(NULL);
if (hdc)
{
    _dpiX = GetDeviceCaps(hdc, LOGPIXELSX);
    _dpiY = GetDeviceCaps(hdc, LOGPIXELSY);
    ReleaseDC(NULL, hdc);
}

// Using CDPI example class
CDPI g_metrics;

int dpiX = g_metrics.GetDPIX();
int dpiY = g_metrics.GetDPIY();
```

GetSystemMetrics Function

The **GetSystemMetrics** function enables you to retrieve the specified system metric or system configuration setting. Note that all dimensions retrieved by

GetSystemMetrics are in pixels. The following code example shows how to retrieve the horizontal and vertical resolution for the current display setting.

```
// Retrieve the horizontal and vertical resolution of the current display setting.
int cxScreen = GetSystemMetrics(SM_CXSCREEN);
int cyScreen = GetSystemMetrics(SM_CYSCREEN);
```

The CDPI example class listed in Appendix B (and shown here) offers methods to get the screen dimensions scaled based on DPI (known as relative pixels).

```
CDPI g_metrics;
int cxScreen = g_metrics.ScaledScreenWidth();
int cyScreen = g_metrics.ScaledScreenHeight();
```

SystemParametersInfo Function

The **SystemParametersInfo** function enables you to retrieve or set the value of one of the system-wide parameters. This function can also update the user profile while setting a parameter. The following code example shows how to retrieve the number of lines to scroll when the vertical mouse wheel is moved.

```
int g_ucScrollLines;

// Retrieves the number of lines to scroll when the vertical mouse wheel is moved.
SystemParametersInfo(SPI_GETWHEELSCROLLLINES, 0, &g_ucScrollLines, 0);
```

Determining the DPI Scale Factor

If you define your application as a DPI-aware application, you will need to scale your application appropriately at high-DPI settings. To scale correctly, you must determine the relative DPI scale factor. The DPI scale factor uses 96 DPI as the baseline setting for determining the value. The following code example shows how to determine the DPI scale factor.

```
int ScaleX(int x) { _Init(); return MulDiv(x, _dpiX, 96); }
int ScaleY(int y) { _Init(); return MulDiv(y, _dpiY, 96); }

// Example using CDPI class.
int cxScaledWidth = g_metrics.ScaleX(100);
```

After you have determined the relative DPI scale factor, you should apply that scale factor when selecting fonts, loading images, and laying out UI elements in your application.

Scaling Text

Scaling text is critical to making your application DPI-aware. To ensure properly scaled text, here are a few recommendations:

- Do not use the default Windows or DC fonts, because these are bitmap fonts and do not scale well. Instead, use a TrueType or OpenType font.
- Custom fonts based on pixel size should apply the DPI scale factor.
- Verify that UI layout and text will scale well at various high-DPI settings.

Note If you are not sure whether your application uses a TrueType or OpenType font (both with suffix .ttf), look at the font definition file in Control Panel. Right-click the file, and check whether it is a .ttf file.

CreateFont Functions

To create a font, use either the **CreateFont**, **CreateFontIndirect**, or **CreateFontIndirectEx** function. You should apply the DPI scale factor when defining the characteristics of the font. The following example shows how to apply the DPI scale factor to the **lfHeight** member of the LOGFONT structure when using the **CreateFontIndirect** function.

```
// From CDPI: convert a point size (1/72 of an inch) to raw pixels.
int PointsToPixels(int pt) { return MulDiv(pt, _dpiY, 72); }

LOGFONT lf;
lf.lfHeight = -g_metrics.PointsToPixels(12);
// Fill in the rest of the structure.
HFONT hfont = CreateFontIndirect(&lf);
```

GetTextExtent Functions

To determine the pixel dimensions of your scaled text string, use the GetTextExtent family of functions, because the physical pixel size of the font is different depending on the DPI display setting. The following code example shows how to determine the width and height of the specified text string by using the **GetTextExtentPoint32** function before drawing a rectangle around the string.

```
static const TCHAR szString[] = TEXT("TEST");

// Retrieve width and height extents of specified string.
SIZE size;
GetTextExtentPoint32(hdc, szString, lstrlen(szString), &size);

// Calculate scaled rect.
RECT rcText;
SetRect(&rcText, 10, 10, 10 + size.cx, 10 + size.cy);
g_metrics.ScaleRect(&rcText);

// Calculate border based on text rect.
RECT rcBorder = rcText;
InflateRect(&rcBorder, g_metrics.ScaleX(4), g_metrics.ScaleY(4));

// Draw rectangle (adjusted for DPI scaling factor) around string.
Rectangle(hdc, rcBorder.left, rcBorder.top, rcBorder.right, rcBorder.bottom);

// Draw string inside rectangle.
TextOut(hdc, rcText.left, rcText.top, szString, lstrlen(szString));
```

Selecting Fonts

The **ChooseFont** function enables you to display a **Fonts** dialog box so that a user can choose the attributes of a font. If you use this function, ensure that you specify TrueType or OpenType fonts as part of the **Flags** member of the **CHOOSEFONT** structure. The following example shows how to initialize the **ChooseFont** function so that only TrueType or OpenType fonts are enumerated in the **Fonts** dialog box.

```
#include "commdlg.h"

CHOOSEFONT data = { sizeof(data) };
data.hwndOwner = hWnd;
// List only TrueType or OpenType screen fonts in the Fonts dialog box.
data.Flags = CF_TTONLY | CF_SCREENFONTS;
ChooseFont(&data);
```

The Windows User Experience Interaction Guidelines recommend that applications use standard visual styles where possible and that applications use a scalable TrueType or OpenType font instead of a DC font.

The following code example uses the **TEXT_BODYTEXT** style for the main text. Note that you need to provide alternate rendering code if visual styles are not enabled, that is, if you use the Windows Classic theme.

```
HTHEME hTheme = OpenThemeData(hWnd, VSCLASS_TEXTSTYLE);
if (hTheme)
{
    DrawThemeText(hTheme, hdc, TEXT_BODYTEXT, 0, szText, -1, DT_SINGLELINE,0,&rcText);
    CloseThemeData(hTheme);
}
else
{
    // Visual styles are not enabled.
    DrawText(hdc, szText, -1, &rcText, DT_SINGLELINE);
}
```

Scaling Graphics

Applications whose graphics scale poorly are less visually appealing to users. For a better UI experience at high-DPI display settings, consider providing custom scaling for all key graphics content, such as images, bitmaps, icons, and toolbars. Although there are many techniques for scaling graphics in applications, there are two primary techniques that work well for scaling graphics across a range of high-DPI display settings. These two techniques are multiple resolution support and the closest fit technique.

Multiple Resolution Support

The multiple resolution support technique requires that you provide your graphics at multiple resolutions so that you have a version that renders well for each targeted DPI setting, such as 96, 120, 144, and 192. In this case, these values are equivalent to 100 percent, 125 percent, 150 percent, and 200 percent of the baseline DPI setting of 96. At run time, your application's logic first determines the correct DPI scaling factor, and then uses it to determine which resolution of the graphics to use.

Closest Fit

This technique is a refinement of multiple resolution support. In addition to providing multiple versions of graphics that render well at various targeted DPI display settings, it also enables you to target your graphics for any specific custom DPI display setting. The key to this technique is to load the "closest fit" graphic (the one that's slightly larger) among the targeted DPI versions, and then scale the graphic down to the current DPI display setting.

For example, if the current custom DPI setting is 132 and you have provided multiple resolution graphics for 96, 120, and 144 DPI, you would first load the graphic that has slightly greater resolution to the current DPI. In this case, you would first load the 144 DPI version of the graphic. You would then scale it down to render well at 132 DPI. The following example shows how to use the closest fit technique to provide support for any custom DPI display setting.

```
int destRectHeight = 20;
int destRectWidth = 20;
int xStart = 40;
int yStart = 85;
// In this example there are 4 versions of the bitmap: 96, 120, 144, and 192.
// The sizes of these are 20x20, 25x25, 30x30, and 40x40.
// These correspond to 96DPI, 120Dpi, etc.

// Assume there is a global gDPI already set via GetDeviceCaps(hDC, LOGPIXELSX).
// The first thing we do is figure out which source image to load.
int iSourceImageDPIToUse = 96; // We will assume 96 by default.

if (gDPI > 144)
iSourceImageDPIToUse = 192;
else if (gDPI > 120)
iSourceImageDPIToUse = 144;
else if (gDPI > 96)
iSourceImageDPIToUse = 120;

LPCTSTR pBitmapResourceName = NULL;
```

```
// Now select the right resource to load.
switch(iSourceImageDPIToUse)
{
    case 120:
        pBitmapResourceName = MAKEINTRESOURCE(MY_RESOURCE_120DPI);
        break;
    case 144:
        pBitmapResourceName = MAKEINTRESOURCE(MY_RESOURCE_144DPI);
        break;
    case 192:
        pBitmapResourceName = MAKEINTRESOURCE(MY_RESOURCE_192DPI);
        break;
    default: // default to 96 DPI
        pBitmapResourceName = MAKEINTRESOURCE(MY_RESOURCE_96DPI);
        break;
}// Now load the right resource.
HBITMAP hbmImage = (HBITMAP)LoadImage(hinst,
pBitmapResourceName, IMAGE_BITMAP, 0, 0, LR_DEFAULTCOLOR);

// Now set the stretch mode.  Assume hdcDest is the DC handle to the destination.
SetStretchBltMode(hdcDest, HALFTONE);

// It is assumed that hdcTargetWindow is your destination window for the bitmap.
HDC hdcBitmap = CreateCompatibleDC(hDCDest);
HBITMAP hbmOld = (HBITMAP)SelectObject(hdcBitmap, hbmImage);

BITMAP bitmap;
GetObject(hbmImage, sizeof(bitmap), &bitmap);

// StretchBlt the image scaled up or down to match the scaled destination size.
// You may want to adjust x & y if you need to change the layout based on DPI.
// The destRectWidth & destRectHeight are the 96-dpi (default) sizes of the layout.
// NOTE: For better quality use GDI+ or WIC with a high quality scale filter.

StretchBlt(hdcDest, g_metrics.ScaleX(xStart), g_metrics.ScaleY(yStart), g_metrics.ScaleX(destRectWidth), g_metrics.Scale



// Don't forget to delete the handles when you are finished with them.
SelectObject(hdcBitmap, hbmOld);
DeleteDC(hdcBitmap);
DeleteObject(hbmImage);
```

In the previous example, we use GDI to load the bitmap. If your application currently loads images using GDI+, you should use the **SetInterpolationMode** method with the *interpolationMode* parameter set to **InterpolationModeHighQualityBicubic**. If you are using Windows Imaging Component (WIC) objects, you should set the interpolation mode to **WICBitmapInterpolationModeFant**.

One of the convenient features of WIC is its native support for DPI -- images can be tagged with DPI attributes, which can be retrieved using the **WICBitmapSource::GetResolution** method. This enables designers to include multiple DPI versions of the source graphic in the image file. The image file can be retrieved programmatically by specifying the DPI attribute, rather than having to resort to resource-naming conventions.

For more information on GDI, GDI+, and WIC image scaling see:

- GDI Image Scaling
- GDI+ Image Scaling
- WIC Image Scaling

Icons

Be sure that your icon (.ico) files have an additional 256 x 256 resolution version to make them look attractive at high-DPI settings in list views with large icons. For more information on creating .ico files and the suggested sizes and scaling factors at high-DPI settings, see Design Concept Guidelines for Icons.

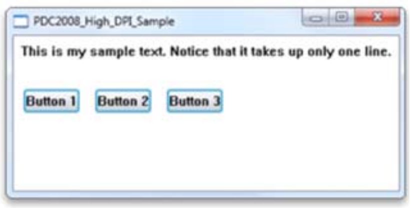
Scaling Layout

Many application windows and dialog boxes are created based on a layout from a resource file, which specifies the layout in logical units that are independent of DPI. These windows, in general, should require no special effort to make them DPI-aware.

However, there are scenarios where custom UI elements must be programmatically created and laid out. In these cases, it is important to take DPI into account both when selecting the size of the overall window, and when positioning the layout of each UI element in the window. In the following example, the application creates a main window and three buttons. In the WM_PAINT message handler, the application also draws text to the screen.

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   static TCHAR szButton1Text[] = TEXT("Button 1");
   static TCHAR szButton2Text[] = TEXT("Button 2");
   static TCHAR szButton3Text[] = TEXT("Button 3");

   HWND hwndMainWindow;

   hInst = hInstance; // Store instance handle in our global variable.

   hwndMainWindow = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, 420, 200, NULL, NULL, hInstance, NULL);

   if (!hwndMainWindow) return FALSE;


   if (!CreateMyButton(10, 55, 60, 26, szButton1Text, hwndMainWindow, hInstance)) return FALSE;
   if (!CreateMyButton(85, 55, 60, 26, szButton2Text, hwndMainWindow, hInstance)) return FALSE;
   if (!CreateMyButton(160, 55, 60, 26, szButton3Text, hwndMainWindow, hInstance)) return FALSE;

   ShowWindow(hwndMainWindow, nCmdShow);
   UpdateWindow(hwndMainWindow);

   return TRUE;
}


// Helper function to create a button.
HWND CreateMyButton(int x, int y, int nWidth, int nHeight, LPCTSTR      szButtonText, HWND hWndParent, HINSTANCE hInstar
{
   DWORD dwFlags = WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON;
   HWND hwndButton = CreateWindow(L"BUTTON",szButtonText,
dwFlags, x, y, nWidth, nHeight,hWndParent,NULL, hInstance, NULL);
   return hwndButton ;
}

// Now here is a snippet from the WM_PAINT handler.
case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        TextOut(hdc,8,8,szText,lstrlen(szText));
        EndPaint(hWnd, &ps);
        break;
```

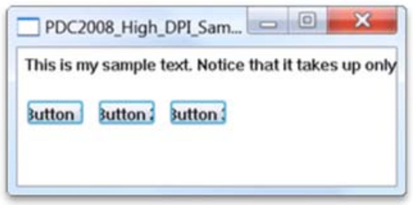
The following screen shot represents the application in the previous code example being displayed at a DPI display setting of 96 in Windows Vista.



This is my sample text. Notice that it takes up only one line.

In this example, the application's UI displays no visual artifacts. This is because the layout coordinates are calculated to display correctly at 96 DPI. All of the UI elements are constructed using coordinates relative to the upper-left corner of the application client area.

But what happens when an application lays out content relative to other content? For example, Button 2 is x number of pixels to the right of Button 1. When you run this example at a 144-DPI display setting with DPI virtualization disabled (which simulates the application being declared DPI-aware), a number of visual artifacts appear, as shown in the following screen shot.



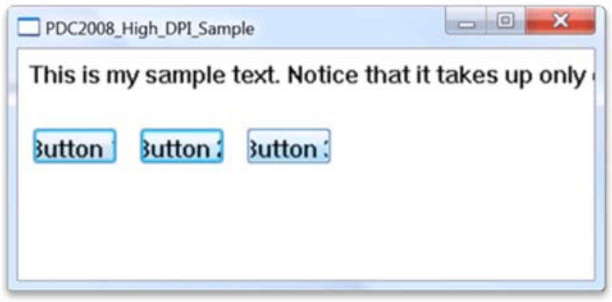
The preceding screen shot shows several UI issues:

- The window frame is not resized, because the application did not scale to the current DPI setting.
- The text is scaled correctly, but gets clipped by the unscaled frame.
- The text on the buttons is scaled correctly, but the button size is not, which causes clipped text.

Each one of these issues reveals incorrect scaling of the application layout. In this case, the overall window size and the padding between controls must be scaled.

Layout and DPI Virtualization Enabled

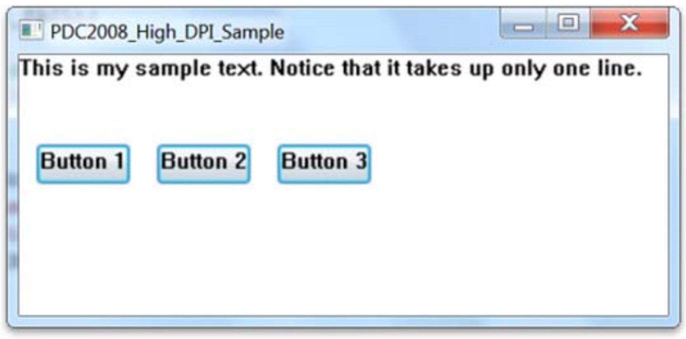
The following screen shot shows the same example again, still running at 144 DPI, but this time with DPI virtualization enabled in Windows Vista.



Notice that the overall size of the window has been scaled correctly, and the text is larger. However, the resized text string in the window is clipped. The buttons have been scaled, but the text inside the buttons is not scaled properly and is clipped.

The clipped text string occurs because in Windows Vista the TextOut function is not supported by DPI virtualization and so the text is essentially double scaled. In fact, only a portion of the Win32 APIs support DPI virtualization in Windows Vista. This is why relying solely on the DPI virtualization feature to provide high-DPI support for your application is not a solution; it is merely a stopgap.

The following screen shot shows the same example again, still running at 144 DPI, but this time with DPI virtualization enabled in Windows 7.


PDC2008_High_DPI_Sample
This is my sample text. Notice that it takes up only one line.
Button 1
Button 2
Button 3

Notice that the overall size of the window has been scaled correctly, and the text is larger. The buttons have been scaled, and the text inside the buttons is scaled properly. This is the result of the extended Win32 API support for DPI virtualization in Windows 7.

Regardless of the different visual results from Windows Vista and Windows 7, you declare an application DPI-aware in the same way.

To make this example truly DPI-aware, you must have the following additional layout support:

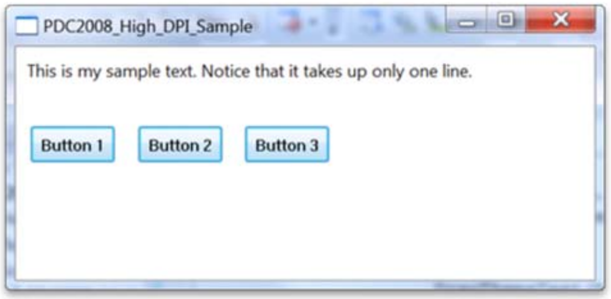
- Specify a scaled size for the window when it is created.

- Specify a scaled size and layout for the buttons when they are created.
- Use Visual Style APIs to render text and other UI elements based on the system-supplied visual styles.

The following code example shows an update to the previous code example that provides additional scaling of the application's layout.

```
// These are the scaling helper functions described earlier.
CDPI g_metrics;

// This is the modified InitInstance code; changes are in bold type.
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   static TCHAR szButton1Text[] = TEXT("Button 1");
   static TCHAR szButton2Text[] = TEXT("Button 2");
   static TCHAR szButton3Text[] = TEXT("Button 3");

   HWND hwndMainWindow;

   hInst = hInstance; // Store instance handle in our global variable.

hwndMainWindow = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, g_metrics.ScaleX(420), g_metrics.ScaleY(200), NULL, NULL, hInstance, NULL);

   if (!hwndMainWindow) return FALSE;

   if (!CreateMyButton(10, 55, 60, 26, szButton1Text, hwndMainWindow, hInstance)) return FALSE;
   if (!CreateMyButton(85, 55, 60, 26, szButton2Text, hwndMainWindow, hInstance)) return FALSE;
   if (!CreateMyButton(160, 55, 60, 26, szButton3Text, hwndMainWindow, hInstance)) return FALSE;

   ShowWindow(hwndMainWindow, nCmdShow);
   UpdateWindow(hwndMainWindow);

   return TRUE;
}

HWND CreateMyButton(int x, int y, int nWidth, int nHeight, LPCTSTR szButtonText, HWND hWndParent, HINSTANCE hInstance)
{
   HWND hwndButton = NULL;
   DWORD dwFlags = WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON;
   hwndButton = CreateWindow(L"BUTTON",szButtonText,dwFlags,
       g_metrics.ScaleX(x), g_metrics.ScaleY(y),
       g_metrics.ScaleX(nWidth), g_metrics.ScaleY(nHeight)
       ,hWndParent,NULL, hInstance, NULL);
   return hwndButton ;
}

#include <uxtheme.h>
#include <vsstyle.h>
#pragma comment(lib, "uxtheme.lib")
…
case WM_PAINT:
    {
        hdc = BeginPaint(hWnd, &ps);
        if (hdc)
        {
            RECT rcText;
            GetClientRect(hWnd, &rcText);
            OffsetRect(&rcText, g_metrics.ScaleX(8), g_metrics.ScaleY(8));

            HTHEME hTheme = OpenThemeData(hWnd, VSCLASS_TEXTSTYLE);
            if (hTheme)
            {
                DrawThemeText(hTheme, hdc, TEXT_BODYTEXT, 0, szText, -1, DT_SINGLELINE,
                    0, &rcText);
                CloseThemeData(hTheme);
            }
            else
            {
                // Visual styles are not enabled.
                DrawText(hdc, szText, -1, &rcText, DT_SINGLELINE);
            }
            EndPaint(hWnd, &ps);
        }
    }
    break;
```

Now when we run the application at 144 DPI, we can see that the buttons and the text are displayed correctly based on scaling the layout from the original application values. Keep in mind that if you are running with DPI virtualization enabled, and if you have not yet added the manifest entry declaring your application to be DPI-aware, you see a different result because your application will be using virtualized system metrics. The following screen shot shows everything displayed correctly.


PDC2008_High_DPI_Sample

This is my sample text. Notice that it takes up only one line.

Tip It is useful to test your application initially with DPI virtualization disabled. The easiest way to do this is to either add the tag to the application manifest, or disable DPI virtualization from Control Panel.

Handling Minimum Effective Resolution

Some applications require a specific minimum resolution, such as 800 × 600 or 1024 × 768. At high-DPI settings, text and UI elements are rendered by using more pixels. This has the effect of lowering the screen real estate available to the application. Therefore, applications should detect the effective resolution instead of the physical resolution.

For example, an application that is running at 120 DPI at a physical screen resolution of 1280 × 1024 has an effective resolution of 1024 × 819. In this case, the effective resolution is 20 percent smaller than the physical resolution.

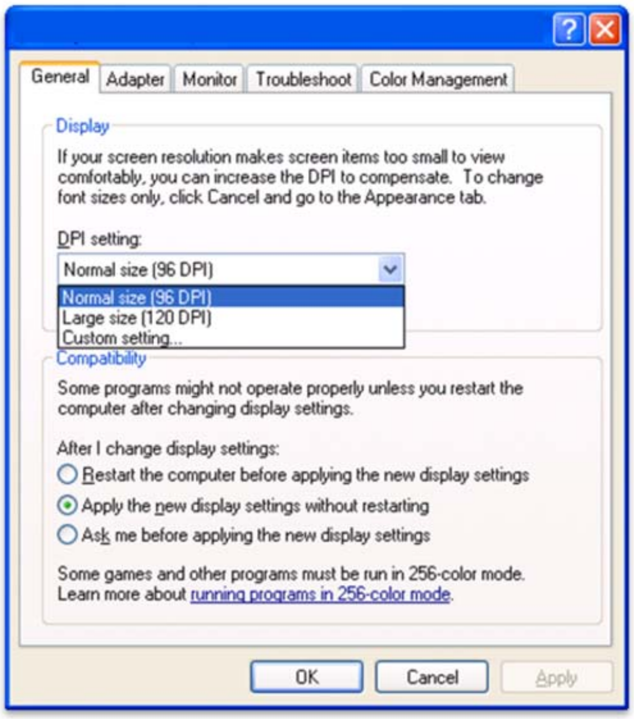
When displaying warning or error messages, your application should mention both screen resolution and DPI settings to the user as possible remedies to get more screen real estate. The following code example shows how to detect the effective screen resolution at run time.

```
CDPI g_metrics;
// Make sure the effective resolution is high enough before continuing.
if (!g_metrics.IsResolutionAtLeast(800, 600))
{
   if (MessageBox(NULL,
   L"Effective screen resolution must be at least 800x600. It is recommended that you either increase your screen resolu
   L"Warning",
   MB_YESNO | MB_ICONWARNING)
   == IDNO)
   {
       return FALSE;
   }
}
```

Appendix A: Setting High DPI in Windows

To change the DPI Setting in Windows XP

1. Right-click the Windows desktop, and then click **Properties**.
2. Click the **Settings** tab and then click **Advanced**.
3. On the **General** tab, in the **DPI setting** drop down list, choose **Large size (120 DPI)**, and then click **OK**.



If your screen resolution makes screen items too small to view comfortably, you can increase the DPI to compensate. To change font sizes only, click Cancel and go to the Appearance tab.

DPI setting: Normal size (96 DPI)

Normal size (96 DPI)
Large size (120 DPI)
Custom setting...

Some programs might not operate properly unless you restart the computer after changing display settings.

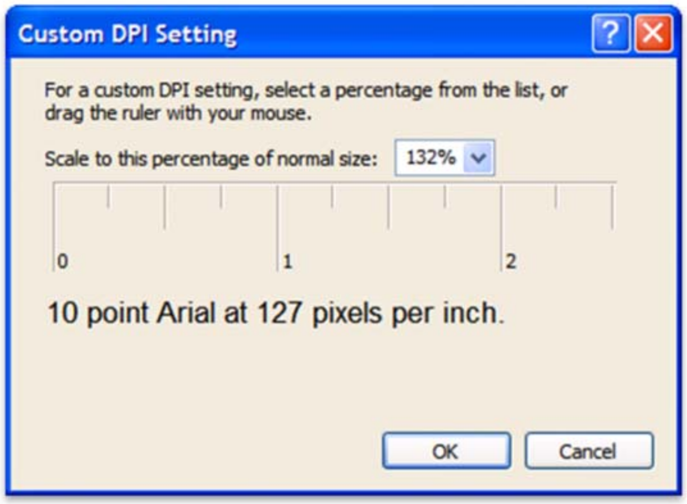
After I change display settings:
- Restart the computer before applying the new display settings
- [✓ Apply the new display settings without restarting]
- Ask me before applying the new display settings

Some games and other programs must be run in 256-color mode. Learn more about running programs in 256-color mode.

4. To see the changes, close all of your programs, and then restart Windows.

To set a custom DPI setting in Windows XP

1. Right-click the Windows desktop, and then click Properties.
2. Click the Settings tab and then click Advanced.
3. On the **General** tab, in the **DPI settings** drop down list, choose **Custom settings**.
4. In the **Custom DPI Setting** dialog box, in the **Scale to this percentage of normal size** list, enter the percentage you want, and then click **OK**.
5. To see the changes, close all of your programs, and then restart Windows.



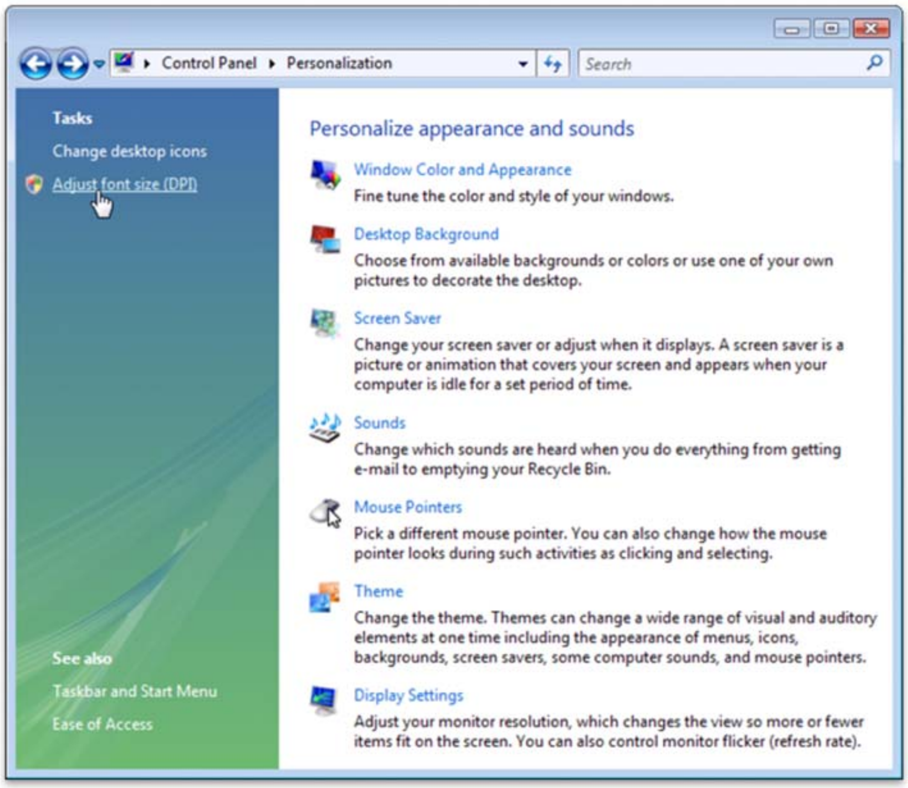
For a custom DPI setting, select a percentage from the list, or drag the ruler with your mouse.

Scale to this percentage of normal size: 132%

10 point Arial at 127 pixels per inch.

To change the DPI setting in Windows Vista

1. On the **Start** menu, click **Control Panel**, click **Appearance and Personalization**, and then click **Personalization**.



Personalize appearance and sounds

Window Color and Appearance
Fine tune the color and style of your windows.

Desktop Background
Choose from available backgrounds or colors or use one of your own pictures to decorate the desktop.

Screen Saver
Change your screen saver or adjust when it displays. A screen saver is a picture or animation that covers your screen and appears when your computer is idle for a set period of time.

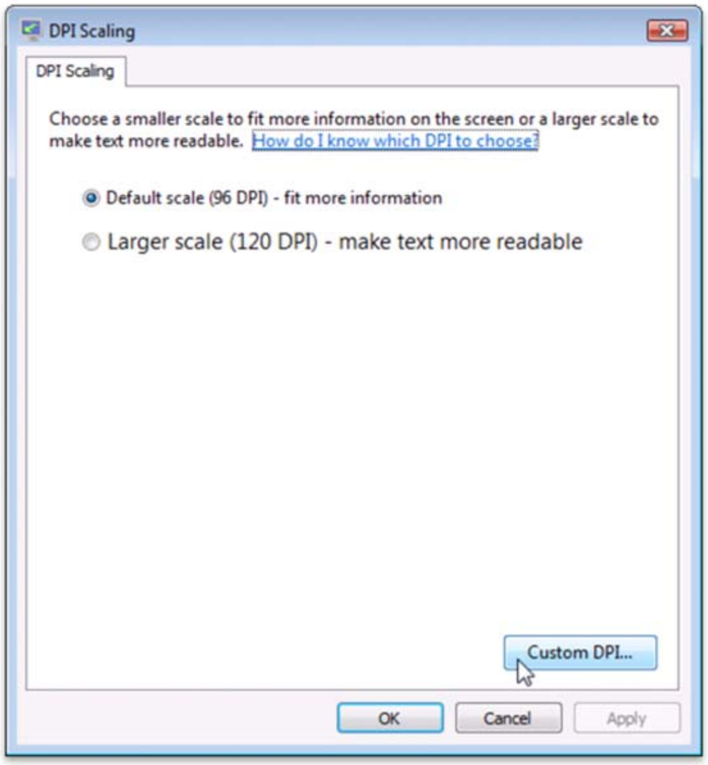
Sounds
Change which sounds are heard when you do everything from getting e-mail to emptying your Recycle Bin.

Mouse Pointers
Pick a different mouse pointer. You can also change how the mouse pointer looks during such activities as clicking and selecting.

Theme
Change the theme. Themes can change a wide range of visual and auditory elements at one time including the appearance of menus, icons, backgrounds, screen savers, some computer sounds, and mouse pointers.

Display Settings
Adjust your monitor resolution, which changes the view so more or fewer items fit on the screen. You can also control monitor flicker (refresh rate).

2. In the left pane, click **Adjust font size (DPI)**. If you are prompted for an administrator password or confirmation, type the password or provide confirmation.
3. In the **DPI Scaling** dialog box, do one of the following:
   - To increase the size of text and other items on the screen, click **Larger scale (120 DPI) - make text more readable**, and then click **OK**.
   - To decrease the size of text and other items on the screen, click **Default scale (96 DPI) - fit more information**, and then click **OK**.



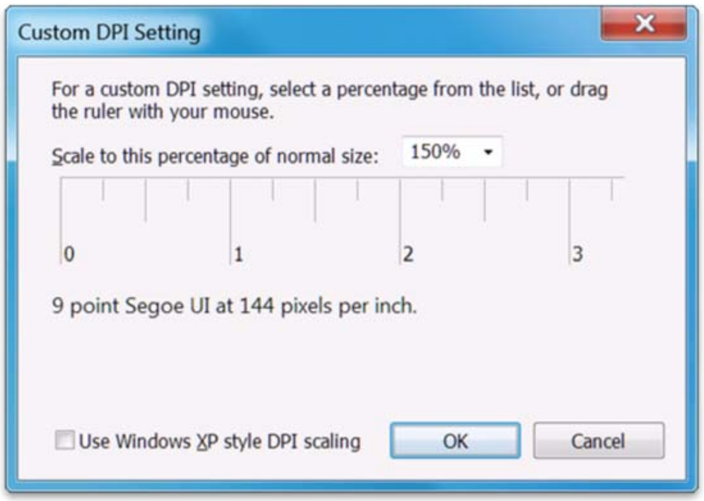
Choose a smaller scale to fit more information on the screen or a larger scale to make text more readable.

[✓ Default scale (96 DPI) - fit more information]
Larger scale (120 DPI) - make text more readable

4. To see the changes, close all of your programs, and then restart Windows.

To set a custom DPI setting in Windows Vista

1. On the **Start** menu, click **Control Panel**, click **Appearance and Personalization**, and then click **Personalization**.
2. In the left pane, click **Adjust font size (DPI)**. If you are prompted for an administrator password or confirmation, type the password or provide confirmation.



Custom DPI Setting

For a custom DPI setting, select a percentage from the list, or drag the ruler with your mouse.

Scale to this percentage of normal size: 150%

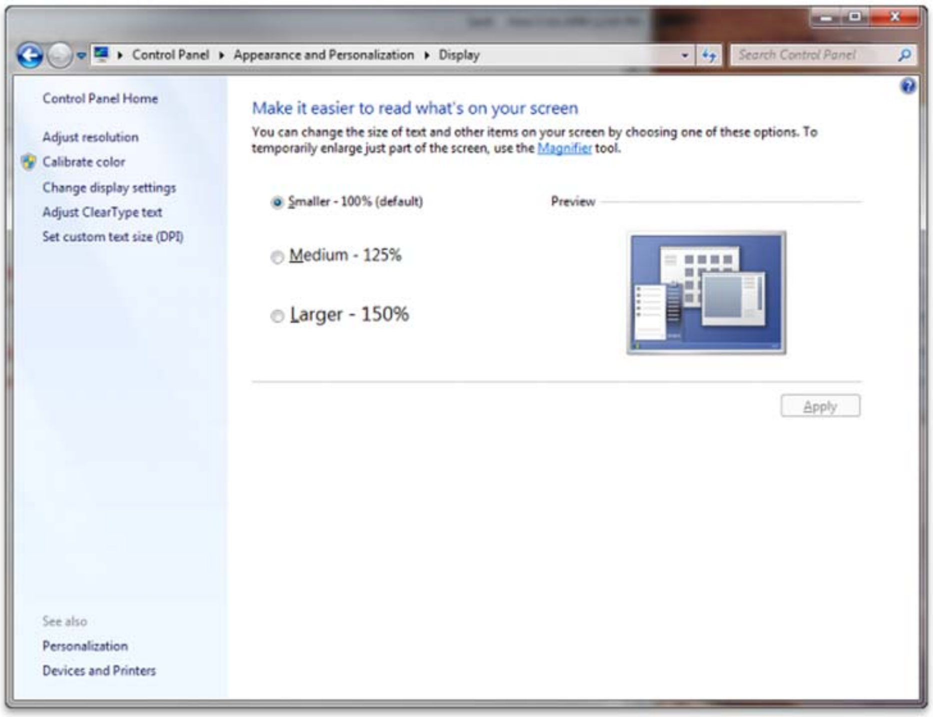
9 point Segoe UI at 144 pixels per inch.

Use Windows XP style DPI scaling

3. In the **DPI Scaling** dialog box, click the **Custom DPI** button.
4. In the **Custom DPI Setting** dialog box, in the **Scale to this percentage of normal size** list, enter the percentage you want, and click **OK**. In this case, the percentage value of 150% of the default value of 96 DPI is equal to 144 DPI. Notice the check box **Use Windows XP Style DPI Scaling**. Selecting this check box disables DPI virtualization.
5. To see the changes, close all of your programs, and then restart Windows.

To change the DPI setting in Windows 7 and Windows 8

1. Open **Control Panel**, click **Appearance and Personalization**, and then click **Display**.



Make it easier to read what's on your screen

You can change the size of text and other items on your screen by choosing one of these options. To temporarily enlarge just part of the screen, use the Magnifier tool.

[● Smaller - 100% (default)]

Medium - 125%

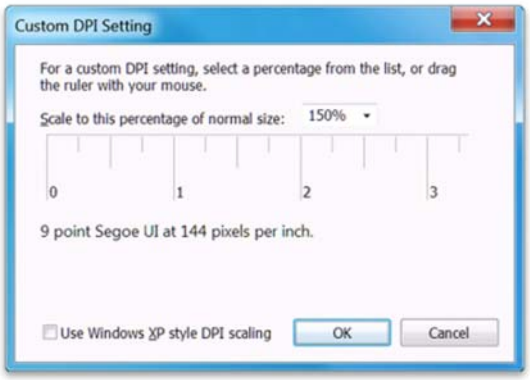
Larger - 150%

2. On the **Display** screen, do one of the following:
   - To increase the size of text and other items on the screen, click **Medium – 125%** or **Larger – 150%**, and then click **OK**.
   - To decrease the size of text and other items on the screen, click **Smaller – 100% (default)**, and then click **OK**.

3. To see the changes, close all of your programs, and log off.

To set a custom DPI setting in Windows 7 and Windows 8

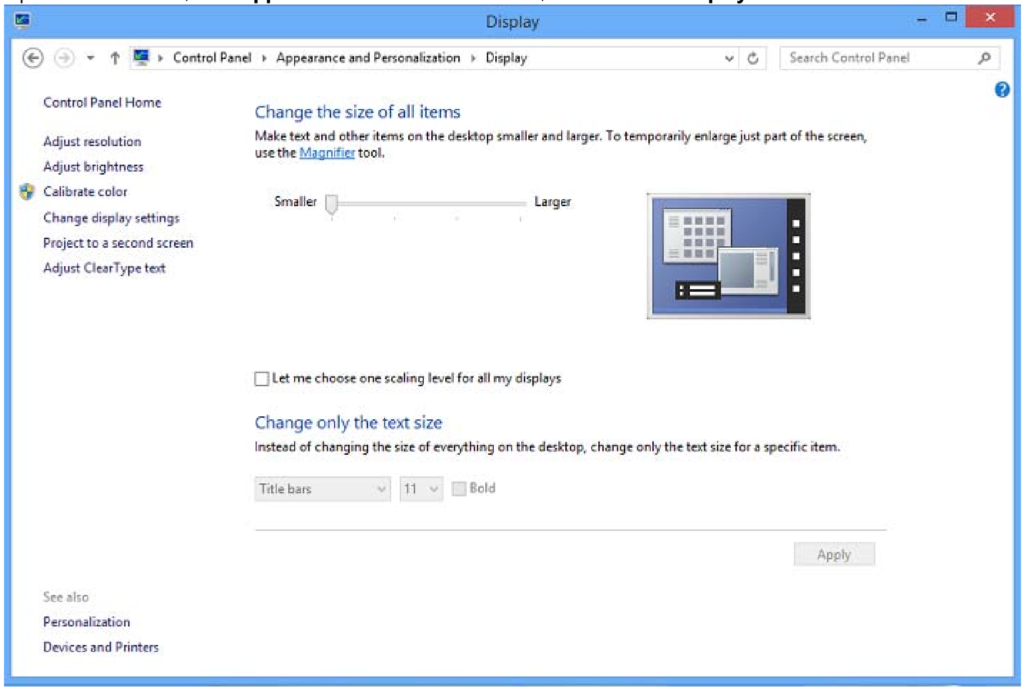
1. Open **Control Panel**, click **Appearance and Personalization**, and then click **Display**.


Custom DPI Setting
For a custom DPI setting, select a percentage from the list, or drag the ruler with your mouse.
Scale to this percentage of normal size: 150%
0
1
2
3
9 point Segoe UI at 144 pixels per inch.
Use Windows XP style DPI scaling
OK
Cancel

2. On the **Display** screen, click **Set Custom text size (DPI)**.
3. In the **Custom DPI Setting** dialog box, in the **Scale to this percentage of normal size** list, enter the percentage you want, and click OK. In this case, the percentage value of 150% of the default value of 96 DPI is equal to 144 DPI. Notice the check box **Use Windows XP Style DPI Scaling**. Selecting this check box disables DPI virtualization.
4. To see the changes, close all of your programs, and then log off and log on again.

To set a custom DPI setting in Windows 8.1 with per-monitor scaling

1. Open **Control Panel**, click **Appearance and Personalization**, and then click **Display**.


Display
Control Panel › Appearance and Personalization › Display
Search Control Panel
Control Panel Home
Adjust resolution
Adjust brightness
Calibrate color
Change display settings
Project to a second screen
Adjust ClearType text
Change the size of all items
Make text and other items on the desktop smaller and larger. To temporarily enlarge just part of the screen, use the Magnifier tool.
Smaller
Larger
Let me choose one scaling level for all my displays
Change only the text size
Instead of changing the size of everything on the desktop, change only the text size for a specific item.
Title bars
11
Bold
Apply
See also
Personalization
Devices and Printers

2. On the **Display** screen, do one of the following:
   - To increase the size of text and other items on the screen, move the slider toward **Larger** and click **Apply**.
   - To decrease the size of text and other items on the screen, move the slider toward **Smaller** and click **Apply**.

3. These changes are applied immediately. Applications are scaled and virtualized until logoff/on.

To set a custom DPI setting in Windows 8.1 without per-monitor scaling

1. On the **Start** menu, click **Control Panel**, click **Appearance and Personalization**, and then click **Display**.
2. Check the **Let me choose one scaling level for all my displays** box. The Windows 7 and Windows 8 screen is displayed. The familiar radio buttons allow you to change the DPI.
3. See Setting High DPI in Windows for step by step instructions on how to change DPI from this UI.

Appendix B: Optimal Configuration Examples

The following table lists the DPI features for several common displays. Panel DPI indicates the native pixel density of the panel. OS DPI indicates the closest match OS DPI to calibrate the display to a standard setting.

Display size	Display resolution	Horizontal (pixels)	Vertical (pixels)	Panel DPI	OS DPI	Scale level
10.6"	FHD	1920	1080	208	144	150%
10.6"	HD	1366	768	148	96	100%
11.6"	WUXGA	1920	1200	195	144	150%
11.6"	HD	1366	768	135	96	100%
13.3"	WUXGA		1200	170	144	150%
13.3"	QHD	2560	1440	221	192	200%
13.3"	HD	1366	768	118	96	100%
15.4"	FHD	1920	1080	143	120	125%
15.6"	QHD+	3200	1800	235	192	200%
17"	FHD	1920	1080	130	120	125%
23"	QFHD (4K)	3840	2160	192	192	200%
24"	QHD	2560	1440	122	120	125%

Appendix C: Common High DPI Issues

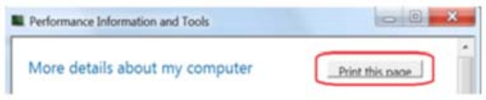
Applications that do not account for DPI and do not adjust for the larger font and UI sizes can cause various classes of issues. This section describes the most common categories of issues and shows examples that illustrate them. The categories of high DPI issues include:

- Clipped UI Elements or Text
- Incorrect Font Size
- Incorrect Layout
- Blurred UI Elements
- Pixelated Text
- Input Issues
- Partial Rendering of a Full-screen Application
- Incorrect use of Effective Resolution

In the Assessing DPI Compatibility section, you can find a summary of the high DPI issues, the potential root cause of each issue, and possible techniques for resolving each issue.

Clipped UI elements or text

Applications that do not handle layout properly sometimes exhibit clipped UI elements or text. In the following screen shot, notice that the text "Print this page" is clipped at the bottom by the button.


Performance Information and Tools
More details about my computer
Print this page

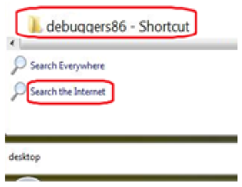
This is a result of text being resized while other UI elements containing the text, such as list box items and buttons, are not resized. In this case the application fails to scale the size of the UI layout in proportion to the larger text.

Here are other scenarios that can result in clipped UI elements or text:

- UI elements no longer fit into the application window due to increased sizes of text and controls.
- The application resizes partially in response to system metrics, but fails to resize completely. For example, the application may resize buttons to fit the new text font, but fails to increase the size of the child window that contains the buttons.

Incorrect Font Size

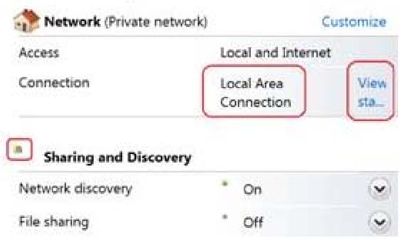
Applications that are not DPI–aware often display incorrect font sizes for text. In the following screen shot, notice the inconsistent font sizes in the application's UI.



This type of artifact typically occurs when an application incorrectly creates and uses fonts. There are many ways to create and use fonts in Windows applications. Some font APIs enable an application to resize text dynamically in response to the change in DPI setting, while others do not. To avoid the inconsistent display of font sizes in your application, make sure that you create and use fonts in a DPI-independent way. For more information about creating and using fonts, see Scaling Text.

Incorrect Layout

Applications that are not DPI–aware can display incorrect layout, which results in a visual artifacts and makes it difficult for users to interact with the application. In the following screen shot, notice that the UI layout does not scale, which results in a clipped UI element. Also notice that some of the text word-wraps event though there is adequate space available.

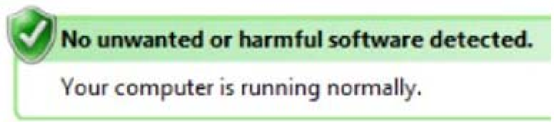


Miscalculations of system metrics, such as screen size, can cause incorrect layout. In some cases, system metrics are calculated correctly for some UI elements, but not all UI elements.

Running at a low effective resolution can also cause incorrect layout. The term "effective resolution" refers to the resulting resolution that takes into account both the physical resolution and the DPI setting of the display. You must use the correct effective screen resolution when you calculate layout. For more information about effective resolution, see Handling Minimum Effective Resolution.

Blurred UI Elements

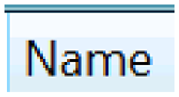
When an entire application appears blurred on Windows Vista or later versions, it is typically the result of DPI virtualization. In the following screen shot, the pixels have been stretched, which results in the blurred effect.

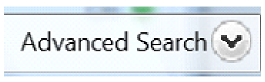

No unwanted or harmful software detected.
Your computer is running normally.

In Windows 8.1, if the user changes DPI without logging off or if the user moves the application to a monitor that has a different DPI, both system DPI–aware applications and applications that are not DPI-aware are scaled based on the change. You should update applications that are not per-monitor DPI–aware, including applications that are system DPI–aware, so that they use the new DPI APIs and window notification available in Windows 8.1 to respond to dynamic changes in DPI and avoid being virtualized and scaled. To override the effects of DPI virtualization, follow the guidance in the Assessing DPI Compatibility section. In addition, you should follow the testing guidelines in the Testing DPI Compatibility section to test that your application behaves as expected at the most common DPI configurations by using the testing guidelines in the Assessing DPI Compatibility section.

Pixelated Text

In certain instances, text appears pixelated but the application exhibits no other visual artifacts. Pixelated text is most noticeable in the diagonal strokes of characters. In the following screen shots, the text "Name" is pixelated. The text "Advanced Search" is not pixelated.





An application that uses bitmapped fonts can cause pixelated text. If you scale the text of a bitmapped font, the text is stretched rather than redrawn with a higher point-size font. Stretched text results in pixelation. To avoid pixelated text, use TrueType or OpenType fonts in your DPI–aware application. Both TrueType and OpenType fonts are vector based and scale appropriately as larger font sizes in response to changes in DPI display settings.

For more information about fonts, see About Fonts. For information about Win32APIs for text and fonts, see Using the Font and Text Output Functions of GDI.

Input Issues

DPI virtualization can cause input issues. In versions of Windows from Windows XP through Windows 8, drag-and-drop operations no longer work during virtualization due to the misalignment of an application's coordinate space when it is remapped to the system's coordinate space during scaling.

In Windows 8.1, both input and inter-process APIs are to eliminate as many of these issues as possible; however, translating coordinates does not always work because there is no way of knowing how the application is going to use the offset system coordinates.

Virtualization was introduced with the goal of ensuring a reasonably good experience for applications that are not fully DPI–aware; however, this virtualization is not able to solve all application issues due to the differences in application behavior throughout the entire ecosystem.

Inter-process communication can continue to be a problem when there are processes running at different DPI awareness levels. Although many of the Win32 APIs have been virtualized, private inter-process communication is not. Additionally, running **SetParent** across processes can cause the DPI awareness level to be inherited from the new parent.

Partial Rendering of a Full-screen Application

DPI virtualization sometimes causes partial rendering of a full screen application. In most cases, this side effect of DPI virtualization occurs in full-screen gaming applications because they commonly do programmatic screen resolution mode changes. This issue affects only Windows Vista or later applications that do not declare themselves as DPI–aware. Because virtualized applications get scaled system metrics, an application that changes the display mode based on these metrics may change the display to an incorrect mode while virtualized. Since full screen applications like games are natively resolution-independent, in most cases this means they are also natively DPI independent. The recommended solution for this class of applications is to declare themselves as per monitor-DPI aware. Note that some applications may have windowed-mode installers and uninstallers, so it is important to do an application test pass according to the DPI testing guidelines to ensure that the application behaves properly at high DPI.

Incorrect Use of Effective Resolution

Applications often require a minimum display resolution to run. For example, suppose an application has a minimum required resolution of 1024x720. At run time, the application queries the system to determine the current screen resolution. If the application determines that the screen resolution is less than the required minimum, it prompts the user with a warning.

Because high DPI settings cause the system to use larger fonts and larger UI elements, it also means that applications take up more pixels when they draw. The result is a lower effective resolution because more pixels are required to render the same UI. The formula to calculate the effective resolution is: Effective Resolution = Physical Resolution / (DPI/96).

For example, suppose the user has a display of 1200x900 with 144 DPI. The effective resolution of this device is only 800x600 because 144/96 = 150%, which reduces the effective screen real-estate by 50%. This calculation means that the size of the UI and text at this setting is roughly the same as if the user were running at 96 DPI with an 800x600 screen resolution.

Community Additions ADD