High DPI Desktop Application Development on Windows

05/31/2018 16 minutes to read

In this article

This content is targeted at developers who are looking to update desktop applications to handle dynamic display scale factor (a.k.a. DPI) changes so that these applications will be crisp on any display that they re rendered on.

To start off, if you re creating a new Windows app from scratch, it is highly recommended that you create a Universal Windows Platform (UWP) application. UWP applications automatically (and dynamically) scale for each display that they re running on.

Desktop applications that use older Windows programming technologies (raw Win32 programming, Windows Forms, Windows Presentation Framework (WPF), etc.) do not automatically handle DPI scaling and, as a result, will render blurry or be sized incorrectly in many common usage scenarios unless work is done by the developer to handle these scenarios. This document provides context and information about what is involved in updating a desktop application to render correctly.

Display Scale Factor & DPI

As display technology has progressed, display panel manufacturers have packed an increasing number of pixels into each unit of physical space on their panels. This has resulted in the dots-per-inch (DPI) of modern display panels being much higher than they have historically been. In the past, most displays had 96 pixels per linear inch of physical space (96 DPI) but now (as of 2017) there are displays with nearly 300 DPI (or higher) on the market. Most legacy desktop UI frameworks have built-in assumptions that the display DPI is a constant value during the lifetime of the process. When applications that use these frameworks experience a DPI change, they most likely will not update their content to reflect this change.

Today, however, it is very common for applications to have the DPI value of the display that they re running on change multiple times throughout the lifetime of their processes. Some common scenarios where the display scale factor/DPI changes are:

- Multiple-monitor setups where each display has a different scale factor and the application is moved from one display to another (such as a 4K and a 1080p display)
- Docking and undocking a high DPI laptop with a low-DPI external display (or vice versa)
- Connecting via Remote Desktop from a high DPI laptop/tablet to a low-DPI device (or vice versa)
- Making a display-scale-factor settings change while applications are running

In these scenarios, UWP applications pick up on the DPI change and redraw themselves for the new DPI without the developer doing any work although desktop applications will not (unless they ve been updated). Because desktop applications often don t respond to the change in DPI, they show up blurry or sized incorrectly after the DPI change.

DPI Awareness Mode

One of the first concepts to be aware of when updating a desktop application to properly DPI scale is that desktop applications must tell Windows what level of DPI scaling they support. Desktop applications can run under multiple DPI awareness modes (by default, desktop applications are completely DPI unaware and are bitmap-stretched by Windows). By running under these modes, applications tell Windows how they do or not handle DPI scaling. When the display scale factor of the display that a desktop application is rendering on changes, the behavior that the application exhibits depends on the DPI awareness mode that the application is running under.

Below is a list of the different DPI awareness modes that Windows supports:

DPI Unaware

DPI unaware applications render as if the screen that they are on has a DPI value of 96. Whenever these applications are run on a screen with a display scale greater than 100% (> 96 DPI), Windows will stretch the application bitmap to the expected physical size, although this results in the application being blurry.

System DPI Awareness

Desktop applications that are system DPI aware typically detect the DPI of the primary connected monitor on startup. During initialization, they layout their UI appropriately (sizing controls, choosing font sizes, loading assets, etc.) for that single DPI. System DPI-aware applications are not DPI scaled by Windows (bitmap stretched) on the primary display (unless the display scale factor changes while the application is running). When the application is moved to a display with a different scale factor (or the display scale factor otherwise changes), Windows will bitmap stretch the application bitmap, which can result in it being blurry. Effectively, System-DPI-aware desktop applications only render correctly at a single display scale factor and become blurry whenever the DPI changes.

Per-Monitor and Per-Monitor (V2) DPI Awareness

It is recommended that desktop applications are updated to use per-monitor DPI awareness mode in order to render correctly whenever the DPI of the display that they re running on changes. When an application reports to Windows that it wants to run in this mode, Windows will step out of the way and not bitmap stretch the application when the DPI changes. It is completely the application s responsibility to handle resizing itself for the new DPI. The reason that work is required here, by the application, is that most UI frameworks that desktop application use (Windows common controls (comctl32), Windows Forms, Windows Presentation Framework, etc.) do not support automatic DPI scaling by default.

There are two versions of Per-Monitor awareness that an application can register itself as: version 1 and version 2 (PMv2). Registering a process as running in PMv2 awareness mode results in:

1. The application being notified when the DPI changes (both the top-level and child HWNDs)
2. The application seeing the raw pixels of each display
3. The application never being DPI scaled by Windows
4. Non-client area (caption bar, scroll bars, etc.) automatically being DPI scaled by Windows
5. Win32 dialogs (from CreateDialog) automatically DPI scaled by Windows
6. Theme-drawn bitmap assets in common controls (checkboxes, button backgrounds, etc.) being automatically rendered at the appropriate DPI scale factor

When running in Per-Monitor V2 Awareness mode, applications are notified when their DPI has changed. If an application does not resize itself for the new DPI, the application UI will appear too small or too large (depending on the difference in the previous and new DPI values).

Note

Per-Monitor V1 (PMv1) awareness is very limited. It is recommended that applications use PMv2.

The table below shows how applications will render under different scenarios:

DPI Awareness Mode	Windows Version Introduced	Application's view of DPI	Behavior on DPI change
Unaware	N/A	All displays are 96 DPI	Bitmap-stretching (blurry)
System	Vista	All displays have the same DPI (the DPI of the primary display at the time the Windows session was started)	Bitmap-stretching (blurry)

DPI Awareness Mode	Windows Version Introduced	Application's view of DPI	Behavior on DPI change
Per-Monitor	8.1	The DPI of the display that the application window is primarily located on	• Top-level HWND is notified of DPI change • No DPI scaling of any UI elements.
Per-Monitor V2	Windows 10 Creators Update (1703)	The DPI of the display that the application window is primarily located on	• Top-level and child HWNDs are notified of DPI change Automatic DPI scaling of: • Non-client area • Theme-drawn bitmaps in common controls (comctl32 V6) • Dialogs ([CreateDialog] (https://msdn.microsoft.com/library/windows/desktop/ms6

Per Monitor (V1) DPI Awareness

Per-Monitor V1 DPI awareness mode (PMv1) was introduced with Windows 8.1. This DPI awareness mode is very limited and only offers the functionality listed below. It is recommended that desktop applications use Per-Monitor V2 awareness mode.

The initial support for per-monitor awareness only offered applications the following:

1. Top-level HWNDs are notified of a DPI change and provided a new suggested size
2. Windows will not bitmap stretch the application UI
3. The application sees all displays in physical pixels (see virtualization)

Per Monitor DPI Scaling Support by UI Framework / Technology

The table below shows the level of per-monitor DPI awareness support offered by various Windows UI frameworks (as of 2017):

Framework / Technology	**Support**	**OS Version**	**DPI Scaling handled by**	**Further Reading**
Universal Windows Platform (UWP)	Full	1604	UI framework	[Universal Windows Platform (UWP)](https://docs.microsoft.com/windows/uwp/started/whats-a-uwp)
Raw Win32/Common Controls V6 (comctl32.dll)	• DPI change notification messages sent to all HWNDs • Theme-drawn assets render correctly in common controls • Automatic DPI scaling for dialogs	1703	Application	GitHub Sample
Windows Forms	Limited automatic per-monitor DPI scaling for some controls	1703	UI framework	High DPI Support in Windows Forms
Windows Presentation Framework (WPF)	Native WPF applications will DPI scale WPF hosted in other frameworks and other frameworks hosted in WPF do not automatically scale	1604	UI framework	GitHub Sample
GDI	None	N/A	Application	See GDI High-DPI Scaling
GDI+	None	N/A	Application	See GDI High-DPI Scaling

Framework / Technology	Support	OS Version	DPI Scaling handled by	Further Reading
MFC	None	N/A	Application	N/A

Updating Existing Applications

In order to update an existing desktop application to handle DPI scaling properly, it needs to be updated such that, at a minimum, the important parts of its UI are updated to respond to DPI changes.

Most desktop applications run under system DPI awareness mode. System-DPI-aware applications typically scale to the DPI of the primary display (the display that the system tray was located on at the time the Windows session was started). When the DPI changes, Windows will bitmap stretch the UI of these applications, which often results in them being blurry. When updating a system-DPI-aware application to become per-monitor-DPI aware, the code which handles UI layout needs to be updated such that it is not performed only once during application initialization but instead is processed whenever a DPI change notification (handing the WM_DPICHANGED message in the case of Win32) is received. This typically involves revisiting any assumptions in the code that the UI only needs to be scaled once.

Also, in the case of Win32 programming, many Win32 APIs do not have any DPI or display context so they will only return values relative to the primary display s DPI. It can be useful to grep through your code to look for some of these APIs and replace them with DPI-aware variants. Some of the common APIs that have DPI-aware variants are:

Single DPI version	Per-Monitor version
GetSystemMetrics	GetSystemMetricsForDpi
AdjustWindowRectEx	AdjustWindowRectExForDpi
SystemParametersInfo	SystemParametersInfoForDpi
GetDpiForMonitor	GetDpiForWindow

It is also a good idea to search for magic numbers , which are hard-coded sizes that assume a constant DPI, and replace these with code that handles DPI scaling. Below is an example that incorporates all of these suggestions:

Example:

The example below shows a simplified Win32 case of creating a child HWND which assumes that the application is running at 96 DPI

Copy

```
case WM_CREATE:
{
    // Add a button
    HWND hWndChild = CreateWindow(L"BUTTON", L"Click Me",
        WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
        50,
        50,
        100,
        50,
        hWnd, (HMENU)NULL, NULL, NULL);
}
```

The modified code below shows:

1. The window-creation code DPI scaling the position and size of the child HWND for the DPI of its parent window
2. Responding to DPI change by repositioning and resizing the child HWND
3. Hard-coded (magic numbers) removed and replaced with code that responds to DPI changes

Copy

```
#define INITIALX_96DPI 50
#define INITIALY_96DPI 50
#define INITIALWIDTH_96DPI 100
#define INITIALHEIGHT_96DPI 50


// DPI scale the position and size of the button control
void UpdateButtonLayoutForDpi(HWND hWnd)
{
    int iDpi = GetDpiForWindow(hWnd);
    int dpiScaledX = MulDiv(INITIALX_96DPI, iDpi, 96);
    int dpiScaledY = MulDiv(INITIALY_96DPI, iDpi, 96);
    int dpiScaledWidth = MulDiv(INITIALWIDTH_96DPI, iDpi, 96);
    int dpiScaledHeight = MulDiv(INITIALHEIGHT_96DPI, iDpi, 96);
    SetWindowPos(hWnd, hWnd, dpiScaledX, dpiScaledY, dpiScaledWidth, dpiScaledY,
SWP_NOZORDER | SWP_NOACTIVATE);
}

...

case WM_CREATE:
{
    // Add a button
    HWND hWndChild = CreateWindow(L"BUTTON", L"Click Me",
        WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
        0,
        0,
        0,
        0,
        hWnd, (HMENU)NULL, NULL, NULL);
    if (hWndChild != NULL)
    {
        UpdateButtonLayoutForDpi(hWndChild);
    }
}
break;

case WM_DPICHANGED:
{
    // Find the button and resize it
    HWND hWndButton = FindWindowEx(hWnd, NULL, NULL, NULL);
    if (hWndButton != NULL)
    {
        UpdateButtonLayoutForDpi(hWndButton);
    }
}
break;
```

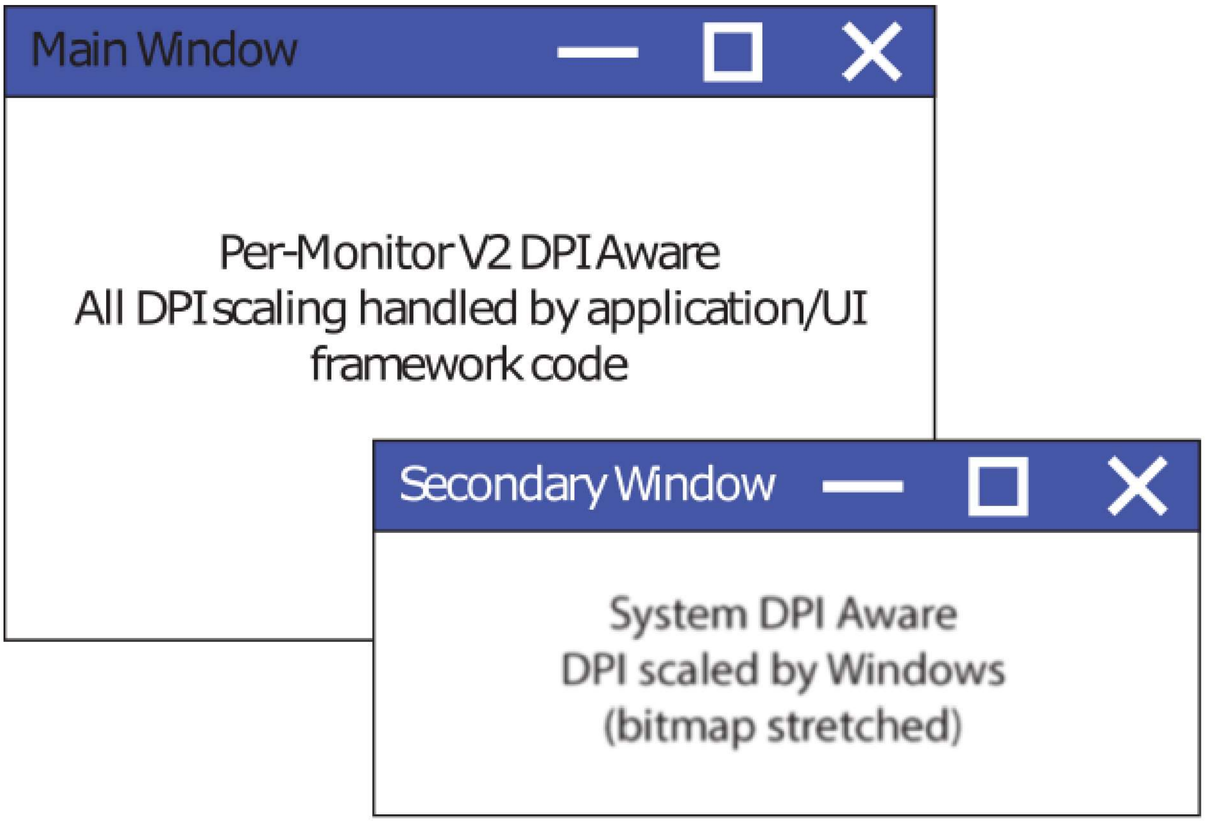
When updating a system-DPI aware application, some common steps to follow are:

1. Mark the process as per-monitor DPI aware (V2) using an application manifest (or other method, depending on the UI framework(s) used).
2. Make UI layout logic reusable and move it out of application-initialization code such that it can be reused when a DPI change occurs (WM_DPICHANGED in the case of Windows (Win32) programming).
3. Invalidate any code that assumes that DPI-sensitive data (DPI/fonts/sizes/etc.) never need to be updated. It is a very common practice to cache font sizes and DPI values at process initialization. When updating an application to become per-monitor DPI aware, DPI-sensitive data must be reevaluated whenever a new DPI is encountered.
4. When a DPI change occurs, reload (or re-rasterize) any bitmap assets for the new DPI or, optionally, bitmap stretch the currently loaded assets to the correct size.
5. Grep for APIs that are not DPI sensitive and replace them with DPI-sensitive APIs (where applicable). Example: replace GetSystemMetrics with GetSystemMetricsForDpi.
6. Test your application on a multiple-display/multi-DPI system.
7. For any top-level windows in your application that you are unable to update to properly DPI scale, use mixed-mode DPI scaling (described below) to have these top-level windows be bitmap stretched by Windows.

Mixed-Mode DPI Scaling (Sub-Process DPI Scaling)

When updating an application to support per-monitor DPI awareness, it can become impractical or impossible (in some cases) to update every window in the application. This can simply be due to the time and effort required to update and test all UI or because you do not own all of the UI code that you need to run (if your application perhaps loads third-party UI). In these situations, Windows offers a way to ease in to the world of per-monitor awareness by letting you run some of your application windows (top-level only) in their original DPI-awareness mode while you focus your time and energy updating the more important parts of your UI.

Below is an illustration of what this could look like: you update your main application UI (Main Window in the illustration) to run with per-monitor DPI awareness while you run other windows in their existing mode (Secondary Window).



Prior to the Windows 10 Anniversary Update (1604), the DPI awareness mode of a process was a process-wide property. In the Windows 10 Anniversary Update sub-process DPI awareness support was introduced. Sub-process DPI awareness enables you to have a different DPI awareness mode for each top-level window (having different DPI scaling modes for child HWNDs is not supported, as of the Windows 10 Creators Update (1703)) in your application. A top-level window is defined as a window with no parent. This is typically a regular window with minimize, maximize, and close buttons. The scenario that sub-process DPI awareness is intended for is to have secondary UI scaled by Windows (bitmap stretched) while you focus your time and resources on updating your primary UI.

To enable sub-process DPI awareness, simply call SetThreadDpiAwarenessContext before and after any window creation calls. The window that is created will be associated with the DPI awareness that you set via SetThreadDpiAwarenessContext. Use the second call to restore the current thread s DPI awareness.

While using sub-process DPI scaling enables you to rely on Windows to do some of the DPI scaling for your application, it can increase the complexity of your application. It is important that you understand the drawbacks of this approach and nature of the complexities that it introduces. For more information about sub-process DPI awareness, see Mixed-Mode DPI Scaling and DPI-aware APIs.

Testing Your Changes

After you have updated your application to become per-monitor DPI aware it is important that you validate your changes in a mixed-DPI environment. It is important to validate that your UI code responds properly to DPI changes when your application windows are moved from one display to another that have different DPI values. Some best-practices are:

1. Move application windows back and forth between displays with different DPI values
2. Start your application on displays with different DPI values
3. Change the scale factor for your monitor while the application is running
4. Change the display that you use as the primary display, log out and back in to Windows, and re-test your application. This test is particularly useful in finding code that uses hard-coded sizes/dimensions.

Common Pitfalls (Win32)

Not using the suggested rectangle that is provided in WM_DPICHANGED

When Windows sends your application window a WM_DPICHANGED message, this message includes a suggested rectangle that you should resize your application to. It is critical that your application use this rectangle to resize itself because using this rectangle will:

1. Ensure that the mouse cursor will stay in the same relative position on the Window when dragging between displays
2. Prevent the application window from getting into a recursive dpi-change cycle where one DPI change triggers a subsequent DPI change, which triggers yet another DPI change.

If you have application-specific requirements that prevent you from using the suggested rectangle that Windows provides in the WM_DPICHANGED message, see WM_GETDPISCALEDSIZE. This message can be used to give Windows a desired size that you d like the window to use once the DPI change has occurred, while still avoiding the issues described above.

Lack of documentation about virtualization

When an HWND or process is running as either DPI unaware or system DPI aware it can be bitmap stretched by Windows. When this happens, Windows returns DPI-sensitive information from some API calls scaled to the coordinate space of the calling thread. For example, if a process is DPI unaware (and running on a high-DPI display) and queries the screen size, Windows will virtualize the answer given to the application such that it sees the screen size in 96 DPI units. Alternatively, when a system-DPI-aware HWND is running on a display with a different DPI than was in use when the system was started, Windows will DPI-scale some API calls into the coordinate space that the HWND would be using if it were running at its original DPI scale factor.

When you update your desktop application to DPI scale properly, it is difficult to know which API calls are virtualized and which are not, and this information is not sufficiently documented by Microsoft. Be aware that if you call any system API from a DPI-unaware or system-DPI-aware thread context, the return value might be virtualized.

Many Windows APIs do not have an DPI context

Another challenge that you ll face while updating an existing desktop application to become per-monitor DPI aware is that many legacy Windows APIs do not have DPI or HWND context as part of their interface. Because of this, developers often have to do additional work to handle DPI scaling any DPI-sensitive information (such as sizes, points, or icons). One example of where this lack of DPI context can become difficult is LoadIcon. In the case of LoadIcon, developers will have to bitmap stretch icon bitmaps to the appropriate DPI or find an alternative way to load icons for the appropriate DPI.

Forced reset of process-wide DPI awareness

In general, the DPI awareness mode of your process cannot be changed after process initialization. Windows can, however, forcibly change the DPI awareness mode of your process if you attempt to break the requirement that all HWNDs in a window tree have the same DPI awareness mode. As for Windows 10 1703, it is not possible to have different HWNDs in an HWND tree run in different DPI awareness modes. If you attempt to create a child-parent relationship that breaks this rule, the DPI awareness of the entire process can be reset. This can be triggered by:

1. A CreateWindow call where the passed in parent window is of a different DPI awareness mode than the calling thread.
2. A SetParent call where the two windows are associated with different DPI awareness modes.

The table below shows what happens if you attempt to violate this rule:

Operation	Windows 8.1	Windows 10 (1607 and earlier)	Windows 10 (1703 and later)
CreateWindow (In-Proc)	N/A	**Child inherits** (mixed mode)	**Child inherits** (mixed mode)
CreateWindow (Cross-Proc)	**Forced reset** (of caller's process)	**Child inherits** (mixed mode)	**Forced reset** (of caller's process)
SetParent (In-Proc)	N/A	**Forced reset** (of current process)	**Fail** (ERROR_INVALID_STATE)
SetParent (Cross-Proc)	**Forced reset** (of child window's process)	**Forced reset** (of child window's process)	**Forced reset** (of child window's process)

Related topics

High DPI API Reference

Mixed-Mode DPI Scaling and DPI-aware APIs.