Roadmap for Windows Runtime apps using C# or Visual Basic

[This article is for Windows 8.x and Windows Phone 8.x developers writing Windows Runtime apps. If you're developing for Windows 10, see the latest documentation]

Here are key resources to help you get started developing Windows Runtime apps using C# or Visual Basic. This is not a comprehensive list of all the features or available resources. The fundamentals are listed first, and it's a good idea to start there. But this topic is also designed so that you can skip around and learn about features you're interested in. Bookmark this page and come back to it again when you need to learn how to add another feature to your app.

If you'd rather use another programming language, see:

- Roadmap for Windows Runtime apps using JavaScript
- Roadmap for Windows Runtime apps using C++

In this article

Get started

Essential downloads	Download Windows evaluation copies and Microsoft Visual Studio.
Category ideas	Here are examples of good design for a few categories of apps (e.g. games, productivity apps, news apps, etc.). Naturally, this is a tiny subset of what's possible; nevertheless, these articles can provide you with a flavor of Windows Store app look and behavior.
Defining vision	What kind of app should you make? How do you plan for different devices? How should you monetize your app? Make the right decisions during the planning phase to simplify deployment and maximize your app's potential.
Migrating Silverlight or WPF XAML/code to a Windows Store app	This migration topic is useful if you have experience with other XAML-based UI frameworks like Windows Presentation Foundation (WPF) or Microsoft Silverlight, particularly if you have an app to migrate.
Create your first Windows Store app using C# or Visual Basic	Get started with the tools and create your first Windows Store app.
Create your first Windows Phone Store app using C#	Get started with the tools and create your first Windows Phone Store app.
C#, VB, and C++	You need to select a template when you start developing your Windows

project templates for apps	Runtime app. Use this topic to learn what templates to use and what comes with them.
Reversi board game app in XAML, C#, and C++	This is a simple casual game (Reversi) sample. If you prefer to dive into an end-to-end sample to see how everything is put together, this sample is a great resource. It's extensively commented and has real-world patterns and practices guidance built-in to how the code is written and presented.
Windows Store app samples	Browse a variety of Windows Store app samples and filter by language.
Windows Phone Store app samples	Browse a variety of Windows Phone Store app samples and filter by language.

Basic features and concepts for Windows Runtime apps

Quickstart: Adding controls and handling events	Create controls and connect them to code.
Controls list	See what controls are available.
Controls by function	See what controls are available in various functional categories.
Quickstart: Control templates	In the XAML framework for Windows Store apps, you create a control template when you want to customize a control's visual structure and visual behavior.
Quickstart: Adding app bars	Add an app bar (needed by mostWindows Runtime apps).
Quickstart: Styling controls	Use styles to customize the appearance of your app, and reuse your appearance settings across your app.
Quickstart: Adding text input and editing controls	Display text and provide controls for entering and editing text.
Quickstart: Defining layouts	Position controls and text where you want them.
Quickstart: Touch input	Make your app work with touch.
Responding to keyboard interactions	Make your app work with the keyboard.
Responding to mouse interactions	Make your app work with the mouse.
Quickstart: Pointers	Work with pointer capture and pointer events.
App capability declarations	Enable app capabilities like Internet access or document-library access for running in the security sandbox.
Quickstart: Navigating between pages	Navigate between pages and pass data between them.
Launching, resuming, and multitasking	This section explains how you can activate, suspend, and resume your Windows Runtime app in response to the normal app lifecycle events, file and protocol associations, and AutoPlay events. This is a must for most apps.
Working with tiles, badges, and toast notifications	At the very least you need a tile to allow users to open your Windows Runtime app. In addition, you can increase the utility and visibility of your app by using notifications and creating "live tiles".
Quickstart: Printing from your app	Print from your app.

Accessibility for Windows Runtime apps using C#/VB/C++ and XAML	Make your app accessible. A Windows Runtime app with a XAML UI can provide app-specific information that is reported to any Microsoft UI Automation client. This includes common assistive technologies such as screen readers.
Globalizing your app	Windows is used worldwide and so it is important for you to design your Windows Runtime app to appeal to an international audience in order to get maximum distribution.
Adding a splash screen	Add a splash screen to provide your users with immediate feedback as your app loads its resources.
Publish your app in the Windows Store	The Windows Store lets you reach the millions of customers who use Windows.

The programming model for Windows Runtime apps using C#/VB/C++

XAML overview	This topic provides a full overview on the XAML markup language as it is used by the Windows Runtime, and links to related reference and conceptual material such as how to use each XAML markup extension, and how to use XAML names.
Quickstart: Calling asynchronous APIs in C# or Visual Basic	The Windows Runtime includes many asynchronous APIs, for example methods of **MediaCapture** and **StorageFile**, so that your app remains responsive when it accesses functionality that might take an extended amount of time. Your app can remain responsive because large operations can complete asynchronously while the main thread execution continues. Most of the asynchronous APIs don't have synchronous counterparts, so you need to be sure to understand how to use the asynchronous APIs with C# or Microsoft Visual Basic in your Windows Runtime app.
Dependency properties overview	Your Windows Runtime app uses dependency properties. Dependency properties are a Windows Runtime concept that supports other common features such as animation, styles and data binding.
Events and routed events overview	Wire your app's event handlers in XAML. Learn about the routed event concept, which is relevant to many UI-related events of the **UIElement** class.
ResourceDictionary and XAML resource references	A resource dictionary is a way you can declare a resource item in XAML markup, which you can then access as a shared value for other XAML-defined properties by using a StaticResource markup extension.
Custom dependency properties	Define your own custom property that can participate in data binding, styles, animations, and callbacks for real-time value changes.
Storyboarded animations	Storyboarded animations are custom animations that target dependency property values and change them over time. This isn't just for traditional visually oriented animations, it's also a way to implement app states and add run-time behavior.
Key-frame animations and easing function animations	Key-frame animations are type of storyboarded animation that can set intermediate values along a timeline using a key-frame metaphor. Easing functions are a way to change the interpolation of values while the animation runs. These are both useful for defining a more advanced style of animation than is possible with either a simple storyboarded animation or the animation library.
Storyboarded animations for visual states	Visual states are a technique for applying sets of property changes that are in response to a known state of a control, page, or other part of your app. You use storyboarded animations to define visual states, and there are best practices you should follow when you use storyboarded animations for a visual state.

Rich visuals and media

Animating your UI	An introduction to how animation works in XAML.
Quickstart: Animating your UI using library	Animations are built into many of the controls you use; however, you can add the same library of animations that the controls use and apply the transition animations and theme animations to other components of your

animations	UI.
Quickstart: Video and audio	Integrate media into your app.
Quickstart: Drawing shapes	Draw scalable vector graphics shapes, such as ellipses, rectangles, polygons, and paths.
Quickstart: Using brushes	Draw to a UI surface with colors, gradients, and image sources.
3-D perspective effects for XAML UI	You can apply 3-D effects to content in your Windows Runtime app using perspective transforms. For example, you can create the illusion that an object is rotated toward or away from you.
How to create custom media transport controls	Create a media player app by using the **MediaElement** API and defining your own transport control UI in XAML.
How to use the system media transport controls	Create a basic media player app by using the **MediaElement** control and setting **AreTransportControlsEnabled** to **true**.
Quickstart: Image and ImageBrush	Learn how to include images into your Windows Runtime app UI.

Working with data

Quickstart: Data binding to controls	Bind a control to a single item or bind a list control to a collection of items. This can be used for displaying data, such as stock prices or headlines, in controls.
Quickstart: Reading and writing files	Read from and write to a file.
Quickstart: Accessing files with file pickers	Use the file picker to let the user open or save a file.
How to continue your Windows Phone app after calling a file picker	Use the file picker in a Windows Phone Store app.
Data binding overview	Use data binding features in a XAML UI, including features such as change notification, binding to collections, incremental loading, grouping, and per-binding data conversions.
App data	Learn how Windows Runtime apps can store data and about the scenarios where the various app data techniques work best.

Sensors

Responding to motion and orientation sensors	Use motion and orientation sensors.
Quickstart: Responding to changes in lighting	Use an ambient light sensor.
Quickstart: Detecting a user's location	Use location services.
Maps and directions	Provide maps and directions in Windows Phone Store apps.

Searching, sharing, and connecting

Quickstart: Integrating	You can help users pick files from one app directly within another app. Users gain freedom and flexibility. Apps

with file picker contracts	increase their popularity by supporting the File Open Picker contract.
Adding Share	Great apps make it easy for users to share what they are doing with their friends and family. Apps that support the Share contract can automatically share content to and from any other app that also supports the Share contract.
Auto-launching with file and URI associations	You can use the association launching API to launch the user's default app for a file type or protocol. You can also enable your app to be the default app for a file type or protocol.
Proximity and tapping	Use proximity to connect computers with a simple *tap* gesture. If two computers are near each other, or are tapped together, the operating system becomes aware of the nearby computer.
Streaming media to devices using Play To	Use the Play To contract to let users stream audio, video, or images from their computer to devices in their home network.
Auto-launching with AutoPlay	Use the AutoPlay events to make your app do the right thing automatically when a device is connected to the PC, or a camera memory card, thumb drive, or DVD is inserted into the PC.
Adding support for networking	Learn how to how to set network capabilities required for network access, how to handle network connections as background tasks, and how to secure and troubleshoot network connections for a Windows Runtime app.

Guidelines and best practices

Index of UX guidelines for Windows Runtime apps	Use this resource to find best practices for a variety of specific design implementations and features like file pickers, **SemanticZoom**, cross-slide, and so on.
Input and feedback patterns	Windows provides a concise set of touch interactions that are used throughout the system. Applying this touch language consistently makes your app feel familiar to what users already know.
Performance best practices for Windows Store apps	Here are some concepts and guidelines to consider to ensure that your app performs well.
Guidelines and checklist for accessibility	Describes the guidelines that you should follow if you want to declare that your app is accessible, as part of the Windows Store submission process.

API reference

Here are the key APIs supported in your Windows Runtime apps using C# or Visual Basic.

Windows API reference for Windows Runtime apps	If you are familiar with UI frameworks like Silverlight, many of these APIs will look familiar (they have "XAML" in the namespace name). These APIs provide access to all core platform features.
.NET for Windows Store apps — supported APIs	The subset of the Microsoft .NET API that you can use in a Windows Store app using C# or Visual Basic.

Related topics

App architecture
.NET for Windows Store apps — supported APIs
Windows API reference for Windows Runtime apps

C#, VB, and C++ programming concepts for Windows Runtime apps

[This article is for Windows 8.x and Windows Phone 8.x developers writing Windows Runtime apps. If you're developing for Windows 10, see the latest documentation]

Purpose

This section includes topics that explain programming concepts that are generally applicable to any app that you write, if you are using C#, Microsoft Visual Basic or Visual C++ component extensions (C++/CX) as your programming language and XAML for your UI definition. This includes basic programming concepts such as using properties and events, and how these apply to Windows Runtime app programming. The Windows Runtime extends C#, Visual Basic and C++ concepts of properties and their values by adding the dependency property system. Topics in this section also document the XAML language as it is used by the Windows Runtime, and cover basic scenarios and advanced topics explaining how to use XAML to define the UI for your Windows Runtime app.

In this section

Topic	Description
How to continue your Windows Phone app after calling an AndContinue method	Certain memory-intensive API on Windows Phone 8.1 contain **AndContinue** methods. When you call an **AndContinue** method, your app is deactivated while the operation completes in order to conserve memory. On low-memory devices, your app might even be terminated. Because of this possibility, you have to call different methods in a Windows Phone Store app than you call in a Windows Store app to continue your app after these operations. The following table shows these methods.
Dependency properties overview	This topic explains the dependency property system that is available when you write a Windows Runtime app using C++, C#, or Visual Basic along with XAML definitions for UI.
Custom dependency properties	Explains how to define and implement custom dependency properties for a Windows Runtime app using C++, C#, or Visual Basic.
Attached properties overview	Explains the concept of an attached property in XAML, and provides some examples.
Custom attached properties	Explains how to implement a XAML attached property as a dependency property and how to define the accessor convention that is necessary for your attached property to be usable in XAML.
Events and routed events overview	We describe the programming concept of events in a Windows Runtime app, when using C#, Visual Basic or C++/CX as your programming language, and XAML for your UI definition. You can assign handlers for events as part of the declarations for UI elements in XAML, or you can add the handlers in code. Windows Runtime supports *routed events*: certain input events and data events can be handled by objects beyond the object that fired the event. Routed events are useful when you define control templates, or use pages or layout containers.
Exception handling for	Learn how to handle exceptions (or errors) in Windows Runtime apps using C# or Visual Basic. You can handle exceptions as Microsoft .NET exceptions with **try**-**catch** blocks in your app code, and you can process app-level exceptions by

Windows Runtime apps in C# or Visual Basic	handling the **Application.UnhandledException** event.
Cross-reference: Standard exceptions and error codes	Cross-references Windows Runtime app error codes and symbolic **HRESULTs** to the .NET standard exceptions.
XAML overview	We introduce the XAML language and XAML concepts to the Windows Runtime app developer audience, and describe the different ways to declare objects and set attributes in XAML as it is used for creating a Windows Runtime app.
ResourceDictionary and XAML resource references	Explains how to define a **ResourceDictionary** element and keyed resources, and how XAML resources relate to other resources that you define as part of your app or app package.
XAML theme resources reference	Windows Runtime XAML provides a set of resources that apply different values depending on which system theme is active. Here we list the keys/names of all the theme-specific XAML resources that specify colors, brushes, and other related UI resources, and the values that each of these resources has using the themes **Default** (**Light**), **Dark**, and **HighContrast**. Also, we describe additional named XAML styles that are defined in themeresources.xaml. These styles are for text elements and text-related controls, and for view items from data.

Developer audience

This topic is for use by any Windows Runtime app developer that uses C#, Visual Basic, or C++/CX.

Related topics

Roadmap for Windows Runtime apps using C# or Visual Basic
Quickstart: Calling asynchronous APIs in C# or Visual Basic
.NET for Windows Store apps overview
Using the thread pool in Windows Store apps
Resources for other platform developers
Language reference for Windows Store apps

Develop UWP apps

How-to articles for UWP apps on Windows 10

Instructions and code examples for all kinds of tasks, such as using geolocation services, transferring data over a network, and porting apps to Windows 10. (For individual classes, methods, properties and other APIs, see the API reference.)

Accessibility

Apps for education

API reference

App settings and data

App-to-app communication

Audio, video, and camera

Contacts and calendar

Controls and patterns

Data access

Data binding

Debugging, testing, and performance

Devices, sensors, and power

Enterprise

Files, folders, and libraries

Games and DirectX

Globalization and localization

Graphics and animation

Input and devices

Launching, resuming, and background tasks

Layout

Lumia SDK

Maps and location

Monetization, customer engagement, and Store services

Networking and web services

Packaging apps

Porting apps to Windows 10

Security

Services

Speech

Threading and async programming

Tiles, badges and notifications

UWP on Xbox

Windows Runtime components

XAML platform

HTML and JavaScript (WinJS) development

WinJS interactive site

Get the latest version of WinJS, try out demos, and read the WinJS tutorial to learn how to use HTML, JavaScript, and the Windows Library for JavaScript to

WinJS reference

Detailed reference info for WinJS APIs. Use these APIs to create UWP apps, Windows Runtime 8.x apps, and apps for other platforms.

Windows 8 How-to articles for WinJS

Can't find what you're looking for? Our Windows 8 archive contains hundreds of How-to articles for WinJS.

create UWP apps and apps for other platforms.

Other versions and related app types

Windows Runtime 8.x apps (Windows 8)

How-to articles for creating Windows Runtime apps that run on Windows 8.x.

Windows Phone Silverlight 8 apps

How-to articles and API reference docs for Windows Phone Silverlight development on Windows 8.x.

Windows desktop applications

Docs for building applications using Win32 and COM APIs.

Guide to Universal Windows Platform (UWP) apps

Tyler Whitney | Last Updated: 8/3/2016 | 5 Contributors

IN THIS ARTICLE +

[Updated for UWP apps on Windows 10. For Windows 8.x articles, see the archive]

In this guide, you'll learn about:

- What a *device family* is, and how to decide which one to target.
- New UI controls and panels that allow you to adapt your UI to different device form factors.
- How to understand and control the API surface that is available to your app.

Windows 8 introduced the Windows Runtime (WinRT), which was an evolution of the Windows app model. It was intended to be a common application architecture.

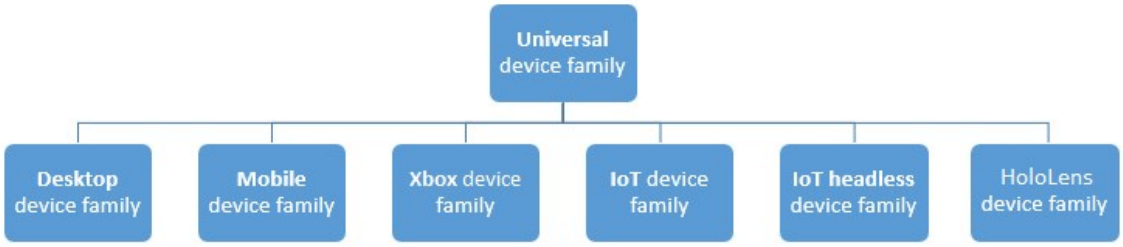
When Windows Phone 8.1 became available, the Windows Runtime was aligned between Windows Phone 8.1 and Windows. This enabled developers to create *Universal Windows 8 apps* that target both Windows and Windows Phone using a shared codebase.

Windows 10 introduces the Universal Windows Platform (UWP), which further evolves the Windows Runtime model and brings it into the Windows 10 unified core. As part of the core, the UWP now provides a common app platform available on every device that runs Windows 10. With this evolution, apps that target the UWP can call not only the WinRT APIs that are common to all devices, but also APIs (including Win32 and .NET APIs) that are specific to the device family the app is running on. The UWP provides a guaranteed core API layer across devices. This means you can create a single app package that can be installed onto a wide range of devices. And, with that single app package, the Windows Store provides a unified distribution channel to reach all the device types your app can run on.

Because your UWP app runs on a wide variety of devices with different form factors and input modalities, you want it to be tailored to each device and be able to unlock the unique capabilities of each device. Devices add their own unique APIs to the guaranteed API layer. You can write code to access those unique APIs conditionally so that your app lights up features specific to one type of device while presenting a different experience on other devices. Adaptive UI controls and new layout panels help you to tailor your UI across a broad range of screen resolutions.

Device families

Windows 8.1 and Windows Phone 8.1 apps target an operating system (OS): either Windows, or Windows Phone. With Windows 10 you no longer target an operating system but you instead target your app to one or more device families. A device family identifies the APIs, system characteristics, and behaviors that you can expect across devices within the device family. It also determines the set of devices on which your app can be installed from the Store. Here is the device family hierarchy.



A device family is a set of APIs collected together and given a name and a version number. A device family is the foundation of an OS. PCs run the desktop OS, which is based on the desktop device family. Phones and tablets, etc., run the mobile OS, which is based on the mobile device family. And so on.

The universal device family is special. It is not, directly, the foundation of any OS. Instead, the set of APIs in the universal device family is inherited by child device families. The universal device family APIs are thus guaranteed to be present in every OS and on every device.

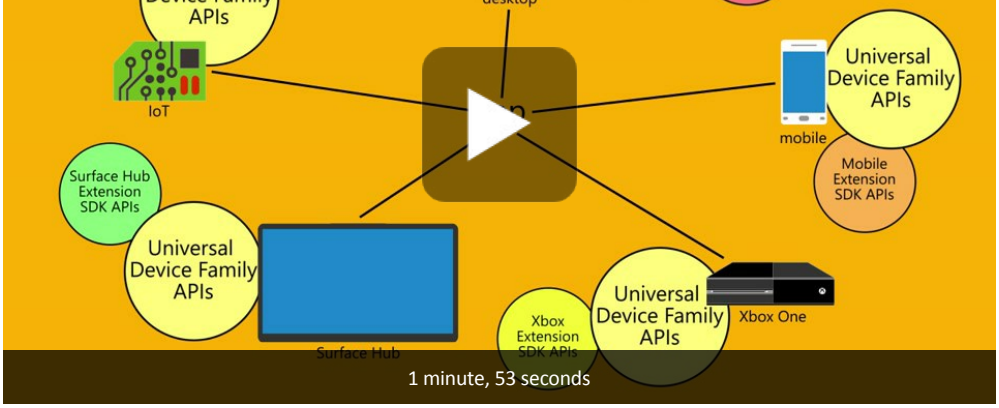
Each child device family adds its own APIs to the ones it inherits. The resulting union of APIs in a child device family is guaranteed to be present in the OS based on that device family, and on every device running that OS.

One benefit of device families is that your app can run on any, or even all, of a variety of devices from phones, tablets, desktop computers, Surface Hubs, Xbox consoles, and HoloLens. Your app can also use adaptive code to dynamically detect and use features of a device that are outside of the universal device family.

The decision about which device family (or families) your app will target is yours to make. And that decision impacts your app in these important ways. It determines:

- The set of APIs that your app can assume to be present when it runs (and can therefore call freely).
- The set of API calls that are safe only inside conditional statements.
- The set of devices on which your app can be installed from the Store (and consequently the form factors that you need to consider).

There are two main consequences of making a device family choice: the API surface that can be called unconditionally by the app, and the number of devices the app can reach. These two factors involve tradeoffs and are inversely related. For example, a UWP app is an app that specifically targets the universal device family, and consequently is available to all devices. An app that targets the universal device family can assume the presence of only the APIs in the universal device family (because that's what it targets). Other APIs must be called conditionally. Also, such an app must have a highly adaptive UI and comprehensive input capabilities because it can run on a wide variety of devices. A Windows mobile app is an app that specifically targets the mobile device family, and is available to devices whose OS is based on the mobile device family (which includes phones, tablets, and similar devices). A mobile device family app can assume the presence of all APIs in the mobile device family, and its UI has to be moderately adaptive. An app that targets the IoT device family can be installed only on IoT devices and can assume the presence of all APIs in the IoT device family. That app can be very specialized in its UI and input capabilities because you know that it will run only on a specific type of device.



Here are some considerations to help you decide which device family to target:

Maximizing your app's reach

To reach the maximum range of devices with your app, and to have it run on as many kinds of devices as possible, your app will target the universal device family. By doing so, the app automatically targets every device family that's based on universal (in the diagram, all the children of universal). That means that the app runs on every OS based on those device families, and on all the devices that run those operating systems. The only APIs that are guaranteed to be available on all those devices is the set defined by the particular version of the universal device family that you target. (With this release, that version is always 10.0.x.0.) To find out how an app can call APIs outside of its target device family version, see Writing code later in this topic.

Limiting your app to one kind of device

You may not want your app to run on a wide range of devices; perhaps it's specialized for a desktop PC or for an Xbox console. In that case you can choose to target your app at one of the child device families. For example, if you target the desktop device family, the APIs guaranteed to be available to your app include the APIs inherited from the universal device family plus the APIs that are particular to the desktop device family.

Limiting your app to a subset of all possible devices

Instead of targeting the universal device family, or targeting one of the child device families, you can instead target two (or more) child device families. Targeting desktop and mobile might make sense for your app. Or desktop and HoloLens. Or desktop, Xbox and Surface Hub, and so on.

Excluding support for a particular version of a device family

In rare cases you may want your app to run everywhere except on devices with a particular version of a particular device family. For example, let's say your app targets version 10.0.x.0 of the universal device family. When the operating system version changes in the future, say to 10.0.x.2, at that point you can specify that your app runs everywhere except version 10.0.x.1 of Xbox by targeting your app to 10.0.x.0 of universal and 10.0.x.2 of Xbox. Your app will then be unavailable to the set of device family versions within Xbox 10.0.x.1 (inclusive) and earlier.

By default, Microsoft Visual Studio specifies **Windows.Universal** as the target device family in the app package manifest file. To specify the device family or device families that your app is offered to from within the Store, manually configure the **TargetDeviceFamily** element in your Package.appxmanifest file.

UI and universal input

A UWP app can run on many different kinds of devices that have different forms of input, screen resolutions, DPI density, and other unique characteristics. Windows 10 provides new universal controls, layout panels, and tooling to help you adapt your UI to the devices your app may run on. For example, you can tailor the UI to take advantage of the difference in screen resolution when your app is running on a desktop computer versus on a mobile device.

Some aspects of your app's UI will automatically adapt across devices. Controls such as buttons and sliders automatically adapt across device families and input modes. Your app's user-experience design, however, may need to adapt depending on the device the app is running on. For example, a photos app should adapt the UI when running on a small, hand-held device to ensure that usage is ideal for single-hand use. When the photos app is running on a desktop computer, the UI should adapt to take advantage of the additional screen space.

Windows helps you target your UI to multiple devices with the following features:

- Universal controls and layout panels help you to optimize your UI for the screen resolution of the device
- Common input handling allows you to receive input through touch, a pen, a mouse, or a keyboard, or a controller such as a Microsoft Xbox

controller
- Tooling helps you to design UI that can adapt to different screen resolutions
- Adaptive scaling adjusts to resolution and DPI differences across devices

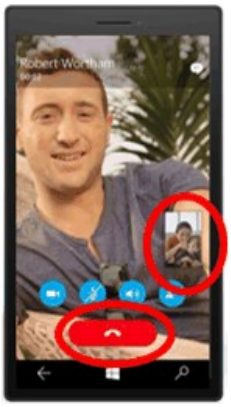
Universal controls and layout panels

Windows 10 includes new controls such as the calendar and split view. The pivot control, which was previously available only for Windows Phone, is also now available for the universal device family.

Controls have been updated to work well on larger screens, adapt themselves based on the number of screen pixels available on the device, and work well with multiple types of input such as keyboard, mouse, touch, pen, and controllers such as the Xbox controller.

You may find that you need to adapt your overall UI layout based on the screen resolution of the device your app will be running on. For example, a communication app running on the desktop may include a picture-in-picture of the caller and controls well suited to mouse input:

However, when the app runs on a phone, because there is less screen real-estate to work with, your app may eliminate the picture-in-picture view and make the call button larger to facilitate one-handed operation:



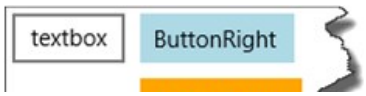
To help you adapt your overall UI layout based on the amount of available screen space, Windows 10 introduces adaptive panels and design states.

Design adaptive UI with adaptive panels

Layout panels give sizes and positions to their children, depending on available space. For example, **StackPanel** orders its children sequentially (horizontally or vertically). **Grid** is like a CSS grid that places its children into cells.

The new **RelativePanel** implements a style of layout that is defined by the relationships between its child elements. It's intended for use in creating app layouts that can adapt to changes in screen resolution. The **RelativePanel** eases the process of rearranging elements by defining relationships between elements, which allows you to build more dynamic UI without using nested layouts.

In the following example, **blueButton** will appear to the right of **textBox1** regardless of changes in orientation or layout, and **orangeButton** will appear immediately below, and aligned with, **blueButton**—even as the width of **textBox1** changes as text is typed into it. It would previously have required rows and columns in a **Grid** to achieve this effect, but now it can be done using far less markup.



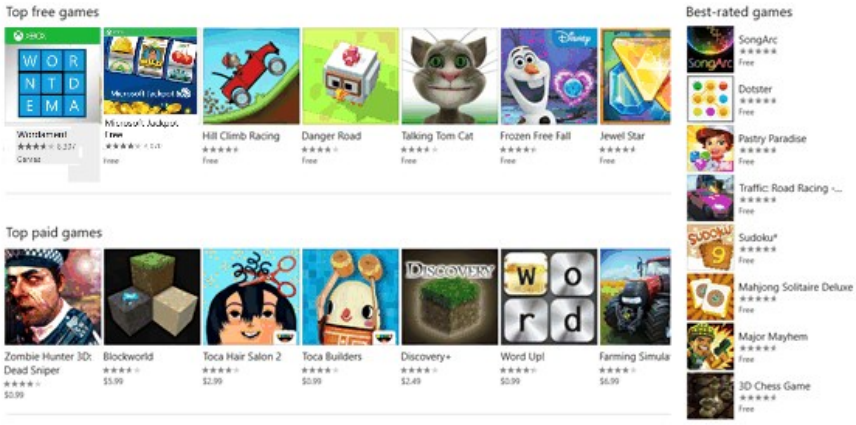

```
<RelativePanel>
    <TextBox x:Name="textBox1" Text="textbox" Margin="5"/>
    <Button x:Name="blueButton" Margin="5" Background="LightBlue" Content="ButtonRight" RelativePanel.RightOf="textBox1"/>
    <Button x:Name="orangeButton" Margin="5" Background="Orange" Content="ButtonBelow" RelativePanel.RightOf="textBox1" RelativePanel.
Below="blueButton"/>
</RelativePanel>
```

Use visual state triggers to build UI that can adapt to available screen space

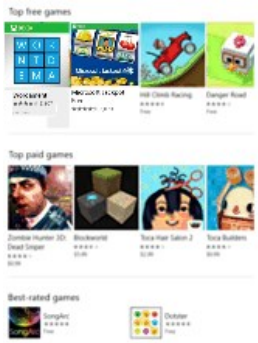
Your UI may need to adapt to changes in window size. Adaptive visual states allows you to change the visual state in response to changes in the size of the window.

StateTriggers define a threshold at which a visual state is activated, which then sets layout properties as appropriate for the window size that triggered the state change.

In the following example, when the window size is 720 pixels or more in width, the visual state named **wideView** is triggered, which then arranges the **Best-rated games** panel to appear to the right of, and aligned with the top of, the **Top free games** panel.



When the window is less than 720 pixels, the **narrowView** visual state is triggered because the **wideView** trigger is no longer satisfied and so no longer in effect. The **narrowView** visual state positions the **Best-rated games** panel below, and aligned with the left of, the **Top paid games** panel:



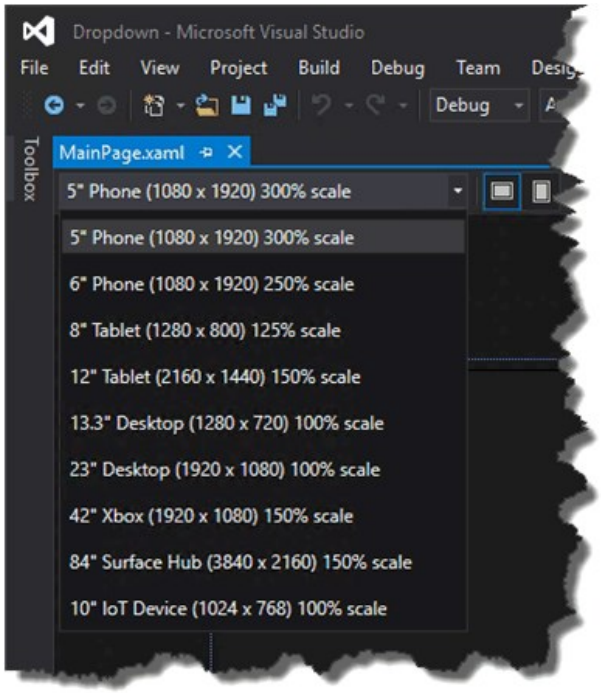
Here is the XAML for the visual state triggers described above. The definition of the panels, alluded to by "`...`" below, has been removed for brevity.

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup>
            <VisualState x:Name="wideView">
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="720" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
```

```
                <Setter Target="best.(RelativePanel.RightOf)" Value="free"/>
                <Setter Target="best.(RelativePanel.AlignTopWidth)" Value="free"/>
```

Tooling

By default, you'll probably want to target the broadest possible device family. When you're ready to see how your app looks and lays out on a particular device, use the device preview toolbar in Visual Studio to preview your UI on a small or medium mobile device, on a PC, or on a large TV screen. That way you can tailor and test your adaptive visual states:



You don’t have to make a decision up front about every device type that you'll support. You can add an additional device size to your project later.

Adaptive scaling

Windows 10 introduces an evolution of the existing scaling model. In addition to scaling vector content, there is a unified set of scale factors that provides a consistent size for UI elements across a variety of screen sizes and display resolutions. The scale factors are also compatible with the scale factors of other operating systems such as iOS and Android. This makes it easier to share assets between these platforms.

The Store picks the assets to download based in part of the DPI of the device. Only the assets that best match the device are downloaded.

Common input handling

You can build a Universal Windows app using universal controls that handle various inputs such as mouse, keyboard, touch, pen, and controller (such as the Xbox controller). Traditionally, inking has been associated only with pen input, but with Windows 10, you can ink with touch on some devices, and with any pointer input. Inking is supported on many devices (including mobile devices) and can easily be incorporated with a just few lines of code.

The following APIs provide access to input:

- **CoreIndependentInputSource** is a new API that allows you to consume raw input on the main thread or a background thread.
- **PointerPoint** unifies raw touch, mouse, and pen data into a single, consistent set of interfaces and events that can be consumed on the main thread or background thread by using **CoreInput.**
- **PointerDevice** is a device API that supports querying device capabilities so that you can determine what input modalities are available on the device.
- The new **InkCanvas** XAML control and **InkPresenter** Windows Runtime APIs allow you to access ink stroke data.

Writing code

Your programming language options for your Windows 10 project in Visual Studio include Visual C++, C#, Visual Basic, and JavaScript. For Visual

C++, C#, and Visual Basic, you can use XAML for a full-fidelity, native UI experience. For Visual C++ you can choose to draw with DirectX either instead of or as well as using XAML. For JavaScript, your presentation layer will be HTML, and HTML is of course a cross-platform web standard. Much of your code and UI will be universal and it will run the same way everywhere. But for code tailored to particular device families, and for UI tailored to particular form factors, you'll have the option to use adaptive code and adaptive UI. Let's look at these different cases.

Calling an API that's implemented by your target device family

Whenever you want to call an API, you'll need to know whether the API is implemented by the device family that your app is targeting. If in doubt, you can look it up in the API reference documentation. If you open the relevant topic and look at the Requirements section, you'll see what the implementing device family is. Let's say that your app is targeting version 10.0.x.0 of the universal device family and you want to call members of the **Windows.UI.Core.SystemNavigationManager** class. In this example, the device family is "Universal". It's a good idea to further confirm that the class members that you want to call are also within your target, and in this case they are. So in this example, you now know that the APIs are guaranteed to be present on every device that your app can be installed on, and you can call the APIs in your code just like you normally would.

```
Windows.UI.Core.SystemNavigationManager.GetForCurrentView().BackRequested += TestView_BackRequested;
```

As another example, imagine that your app is targeting version 10.0.x.0 of the Xbox device family, and the reference topic for an API that you want to call says that the API was introduced in version 10.0.x.0 of the Xbox device family. In that case, again, the API is guaranteed to be present on every device that your app can be installed on. So you would be able to call that API in your code in the normal way.

Note that Visual Studio's IntelliSense will not recognize APIs unless they are implemented by your app's target device family or any extension SDKs that you have referenced. Consequently, if you haven't referenced any extension SDKs, you can be sure that any APIs that appear in IntelliSense must therefore be in your target device family and you can call them freely.

Calling an API that's NOT implemented by your target device family

There will be cases when you want to call an API, but your target device family is not listed in the documentation. In that case you can opt to write adaptive code in order to call that API.

Writing adaptive code with the ApiInformation class

There are two steps to write adaptive code. The first step is to make the APIs that you want to access available to your project. To do that, add a reference to the extension SDK that represents the device family that owns the APIs that you want to conditionally call. See Extension SDKs.

The second step is to use the **Windows.Foundation.Metadata.ApiInformation** class in a condition in your code to test for the presence of the API you want to call. This condition is evaluated wherever your app runs, but it evaluates to true only on devices where the API is present and therefore available to call.

If you want to call just a small number of APIs, you could use the **ApiInformation.IsTypePresent** method like this.

```
// Note: Cache the value instead of querying it more than once.
bool isHardwareButtonsAPIPresent =
    Windows.Foundation.Metadata.ApiInformation.IsTypePresent("Windows.Phone.UI.Input.HardwareButtons");

if (isHardwareButtonsAPIPresent)
{
    Windows.Phone.UI.Input.HardwareButtons.CameraPressed +=
        HardwareButtons_CameraPressed;
}
```

In this case we can be confident that the presence of the **HardwareButtons** class implies the presence of the **CameraPressed** event, because the class and the member have the same requirements info. But in time, new members will be added to already-introduced classes, and those members will have later "introduced in" version numbers. In such cases, instead of using **IsTypePresent**, you can test for the presence of individual members by using **IsEventPresent**, **IsMethodPresent**, **IsPropertyPresent**, and similar methods. Here's an example.

```
bool isHardwareButtons_CameraPressedAPIPresent =
    Windows.Foundation.Metadata.ApiInformation.IsEventPresent
```

```
        ("Windows.Phone.UI.Input.HardwareButtons", "CameraPressed");
```

The set of APIs within a device family is further broken down into subdivisions known as API contracts. You can use the **ApiInformation.IsApiContractPresent** method to test for the presence of an API contract. This is useful if you want to test for the presence of a large number of APIs that all exist in the same version of an API contract.

```
bool isWindows_Devices_Scanners_ScannerDeviceContract_1_0Present =
    Windows.Foundation.Metadata.ApiInformation.IsApiContractPresent
        ("Windows.Devices.Scanners.ScannerDeviceContract", 1, 0);
```

Win32 APIs in the UWP

A UWP app or Windows Runtime Component written in C++/CX has access to the Win32 APIs that are part of the UWP. These Win32 APIs are implemented by all Windows 10 device families. Link your app with Windowsapp.lib. Windowsapp.lib is an "umbrella" lib that provides the exports for the UWP APIs. Linking to Windowsapp.lib will add to your app dependencies on dlls that are present on all Windows 10 device families.

For the full list of Win32 APIs available to UWP apps, see API Sets for UWP apps and Dlls for UWP apps.

User experience

A Universal Windows app allows you to take advantage of the unique capabilities of the device on which it is running. Your app can make use of all of the power of a desktop device, the natural interaction of direct manipulation on a tablet (including touch and pen input), the portability and convenience of mobile devices, the collaborative power of Surface Hub, and other devices that support UWP apps.

Good design is the process of deciding how users will interact with your app, as well as how it will look and function. User experience plays a huge part in determining how happy people will be with your app, so don't skimp on this step. Design basics introduce you to designing a Universal Windows app. See the Introduction to Universal Windows Platform (UWP) apps for designers for information on designing UWP apps that delight your users. Before you start coding, see the device primer to help you think through the interaction experience of using your app on all the different form factors you want to target.

In addition to interaction on different devices, plan your app to embrace the benefits of working across multiple devices. For example:

- Use cloud services to sync across devices. Learn how to connect to web services in support of your app experience.
- Consider how you can support users moving from one device to another, picking up where they left off. Include notifications and in-app purchases in your planning. These features should work across devices.
- Design your workflow using Navigation design basics for UWP apps to accommodate mobile, small-screen, and large-screen devices. Lay out your user interface to respond to different screen sizes and resolutions.
- Consider whether there are features of your app that don't make sense on a small mobile screen. There may also be areas that don't make sense on a stationary desktop machine and require a mobile device to light up. For example, most scenarios around location imply a mobile device.
- Consider how you'll accommodate multiple input modalities. See the Guidelines for interactions to learn how users can interact with your app by using Cortana, Speech, Touch interactions, the Touch keyboard and more.

  See the Guidelines for text and text input for more tradition interaction experiences.

Submit a Universal Windows app through your Dashboard

The new unified Windows Dev Center dashboard lets you manage and submit all of your apps for Windows devices in one place. New features simplify processes while giving you more control. You'll also find detailed analytic reports combined payout details, ways to promote your app and

engage with your customers, and much more.

See Using the unified Windows Dev Center dashboard to learn how to submit your apps for publication in the Windows Store.

Guide to Universal Windows Platform (UWP) apps

Tyler Whitney | Last Updated: 8/3/2016 | 5 Contributors

IN THIS ARTICLE +

[Updated for UWP apps on Windows 10. For Windows 8.x articles, see the archive]

In this guide, you'll learn about:

- What a *device family* is, and how to decide which one to target.
- New UI controls and panels that allow you to adapt your UI to different device form factors.
- How to understand and control the API surface that is available to your app.

Windows 8 introduced the Windows Runtime (WinRT), which was an evolution of the Windows app model. It was intended to be a common application architecture.

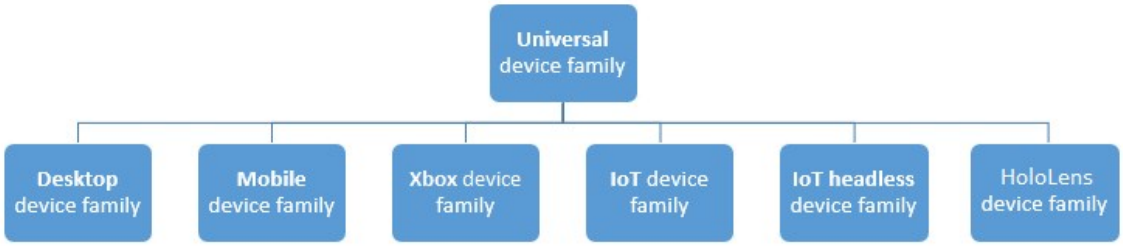
When Windows Phone 8.1 became available, the Windows Runtime was aligned between Windows Phone 8.1 and Windows. This enabled developers to create *Universal Windows 8 apps* that target both Windows and Windows Phone using a shared codebase.

Windows 10 introduces the Universal Windows Platform (UWP), which further evolves the Windows Runtime model and brings it into the Windows 10 unified core. As part of the core, the UWP now provides a common app platform available on every device that runs Windows 10. With this evolution, apps that target the UWP can call not only the WinRT APIs that are common to all devices, but also APIs (including Win32 and .NET APIs) that are specific to the device family the app is running on. The UWP provides a guaranteed core API layer across devices. This means you can create a single app package that can be installed onto a wide range of devices. And, with that single app package, the Windows Store provides a unified distribution channel to reach all the device types your app can run on.

Because your UWP app runs on a wide variety of devices with different form factors and input modalities, you want it to be tailored to each device and be able to unlock the unique capabilities of each device. Devices add their own unique APIs to the guaranteed API layer. You can write code to access those unique APIs conditionally so that your app lights up features specific to one type of device while presenting a different experience on other devices. Adaptive UI controls and new layout panels help you to tailor your UI across a broad range of screen resolutions.

Device families

Windows 8.1 and Windows Phone 8.1 apps target an operating system (OS): either Windows, or Windows Phone. With Windows 10 you no longer target an operating system but you instead target your app to one or more device families. A device family identifies the APIs, system characteristics, and behaviors that you can expect across devices within the device family. It also determines the set of devices on which your app can be installed from the Store. Here is the device family hierarchy.



A device family is a set of APIs collected together and given a name and a version number. A device family is the foundation of an OS. PCs run the desktop OS, which is based on the desktop device family. Phones and tablets, etc., run the mobile OS, which is based on the mobile device family. And so on.

The universal device family is special. It is not, directly, the foundation of any OS. Instead, the set of APIs in the universal device family is inherited by child device families. The universal device family APIs are thus guaranteed to be present in every OS and on every device.

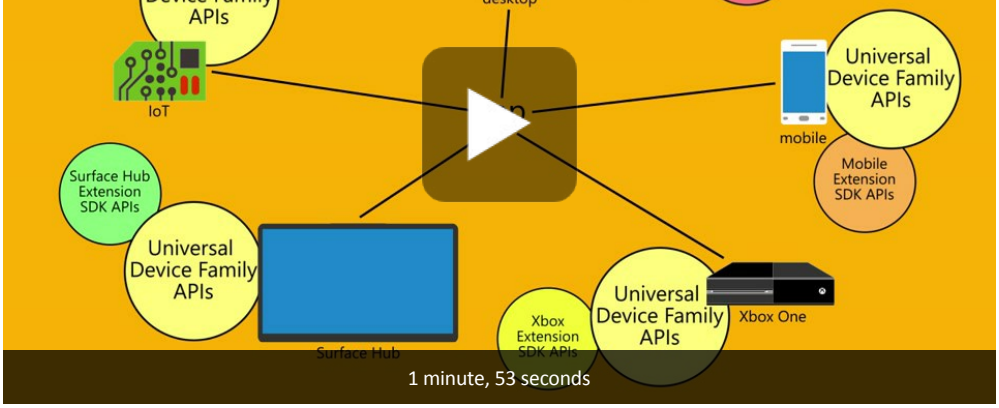
Each child device family adds its own APIs to the ones it inherits. The resulting union of APIs in a child device family is guaranteed to be present in the OS based on that device family, and on every device running that OS.

One benefit of device families is that your app can run on any, or even all, of a variety of devices from phones, tablets, desktop computers, Surface Hubs, Xbox consoles, and HoloLens. Your app can also use adaptive code to dynamically detect and use features of a device that are outside of the universal device family.

The decision about which device family (or families) your app will target is yours to make. And that decision impacts your app in these important ways. It determines:

- The set of APIs that your app can assume to be present when it runs (and can therefore call freely).
- The set of API calls that are safe only inside conditional statements.
- The set of devices on which your app can be installed from the Store (and consequently the form factors that you need to consider).

There are two main consequences of making a device family choice: the API surface that can be called unconditionally by the app, and the number of devices the app can reach. These two factors involve tradeoffs and are inversely related. For example, a UWP app is an app that specifically targets the universal device family, and consequently is available to all devices. An app that targets the universal device family can assume the presence of only the APIs in the universal device family (because that's what it targets). Other APIs must be called conditionally. Also, such an app must have a highly adaptive UI and comprehensive input capabilities because it can run on a wide variety of devices. A Windows mobile app is an app that specifically targets the mobile device family, and is available to devices whose OS is based on the mobile device family (which includes phones, tablets, and similar devices). A mobile device family app can assume the presence of all APIs in the mobile device family, and its UI has to be moderately adaptive. An app that targets the IoT device family can be installed only on IoT devices and can assume the presence of all APIs in the IoT device family. That app can be very specialized in its UI and input capabilities because you know that it will run only on a specific type of device.



Here are some considerations to help you decide which device family to target:

Maximizing your app's reach

To reach the maximum range of devices with your app, and to have it run on as many kinds of devices as possible, your app will target the universal device family. By doing so, the app automatically targets every device family that's based on universal (in the diagram, all the children of universal). That means that the app runs on every OS based on those device families, and on all the devices that run those operating systems. The only APIs that are guaranteed to be available on all those devices is the set defined by the particular version of the universal device family that you target. (With this release, that version is always 10.0.x.0.) To find out how an app can call APIs outside of its target device family version, see Writing code later in this topic.

Limiting your app to one kind of device

You may not want your app to run on a wide range of devices; perhaps it's specialized for a desktop PC or for an Xbox console. In that case you can choose to target your app at one of the child device families. For example, if you target the desktop device family, the APIs guaranteed to be available to your app include the APIs inherited from the universal device family plus the APIs that are particular to the desktop device family.

Limiting your app to a subset of all possible devices

Instead of targeting the universal device family, or targeting one of the child device families, you can instead target two (or more) child device families. Targeting desktop and mobile might make sense for your app. Or desktop and HoloLens. Or desktop, Xbox and Surface Hub, and so on.

Excluding support for a particular version of a device family

In rare cases you may want your app to run everywhere except on devices with a particular version of a particular device family. For example, let's say your app targets version 10.0.x.0 of the universal device family. When the operating system version changes in the future, say to 10.0.x.2, at that point you can specify that your app runs everywhere except version 10.0.x.1 of Xbox by targeting your app to 10.0.x.0 of universal and 10.0.x.2 of Xbox. Your app will then be unavailable to the set of device family versions within Xbox 10.0.x.1 (inclusive) and earlier.

By default, Microsoft Visual Studio specifies **Windows.Universal** as the target device family in the app package manifest file. To specify the device family or device families that your app is offered to from within the Store, manually configure the **TargetDeviceFamily** element in your Package.appxmanifest file.

UI and universal input

A UWP app can run on many different kinds of devices that have different forms of input, screen resolutions, DPI density, and other unique characteristics. Windows 10 provides new universal controls, layout panels, and tooling to help you adapt your UI to the devices your app may run on. For example, you can tailor the UI to take advantage of the difference in screen resolution when your app is running on a desktop computer versus on a mobile device.

Some aspects of your app's UI will automatically adapt across devices. Controls such as buttons and sliders automatically adapt across device families and input modes. Your app's user-experience design, however, may need to adapt depending on the device the app is running on. For example, a photos app should adapt the UI when running on a small, hand-held device to ensure that usage is ideal for single-hand use. When the photos app is running on a desktop computer, the UI should adapt to take advantage of the additional screen space.

Windows helps you target your UI to multiple devices with the following features:

- Universal controls and layout panels help you to optimize your UI for the screen resolution of the device
- Common input handling allows you to receive input through touch, a pen, a mouse, or a keyboard, or a controller such as a Microsoft Xbox

controller
- Tooling helps you to design UI that can adapt to different screen resolutions
- Adaptive scaling adjusts to resolution and DPI differences across devices

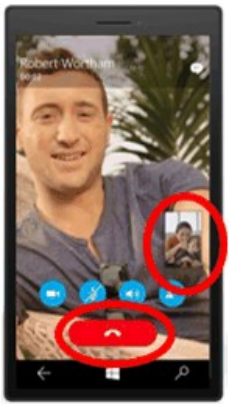
Universal controls and layout panels

Windows 10 includes new controls such as the calendar and split view. The pivot control, which was previously available only for Windows Phone, is also now available for the universal device family.

Controls have been updated to work well on larger screens, adapt themselves based on the number of screen pixels available on the device, and work well with multiple types of input such as keyboard, mouse, touch, pen, and controllers such as the Xbox controller.

You may find that you need to adapt your overall UI layout based on the screen resolution of the device your app will be running on. For example, a communication app running on the desktop may include a picture-in-picture of the caller and controls well suited to mouse input:

However, when the app runs on a phone, because there is less screen real-estate to work with, your app may eliminate the picture-in-picture view and make the call button larger to facilitate one-handed operation:



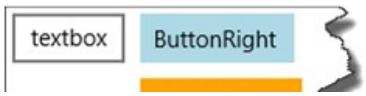
To help you adapt your overall UI layout based on the amount of available screen space, Windows 10 introduces adaptive panels and design states.

Design adaptive UI with adaptive panels

Layout panels give sizes and positions to their children, depending on available space. For example, **StackPanel** orders its children sequentially (horizontally or vertically). **Grid** is like a CSS grid that places its children into cells.

The new **RelativePanel** implements a style of layout that is defined by the relationships between its child elements. It's intended for use in creating app layouts that can adapt to changes in screen resolution. The **RelativePanel** eases the process of rearranging elements by defining relationships between elements, which allows you to build more dynamic UI without using nested layouts.

In the following example, **blueButton** will appear to the right of **textBox1** regardless of changes in orientation or layout, and **orangeButton** will appear immediately below, and aligned with, **blueButton**—even as the width of **textBox1** changes as text is typed into it. It would previously have required rows and columns in a **Grid** to achieve this effect, but now it can be done using far less markup.



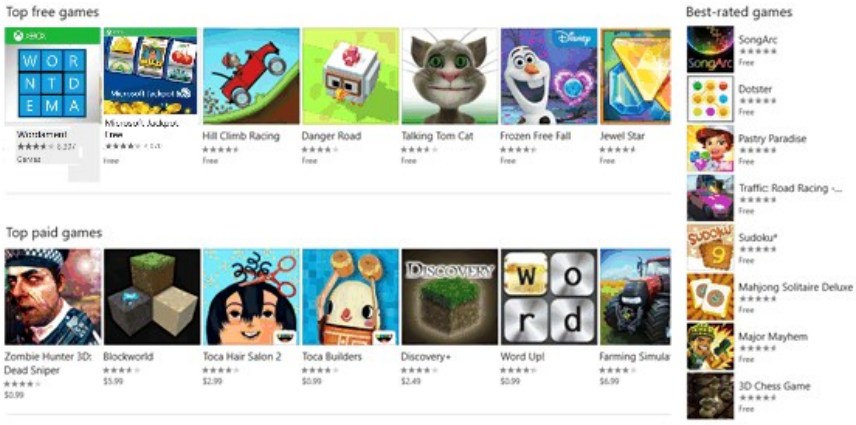

```
<RelativePanel>
    <TextBox x:Name="textBox1" Text="textbox" Margin="5"/>
    <Button x:Name="blueButton" Margin="5" Background="LightBlue" Content="ButtonRight" RelativePanel.RightOf="textBox1"/>
    <Button x:Name="orangeButton" Margin="5" Background="Orange" Content="ButtonBelow" RelativePanel.RightOf="textBox1" RelativePanel.
Below="blueButton"/>
</RelativePanel>
```

Use visual state triggers to build UI that can adapt to available screen space

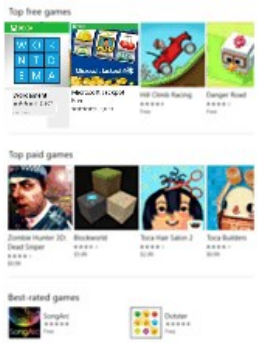
Your UI may need to adapt to changes in window size. Adaptive visual states allows you to change the visual state in response to changes in the size of the window.

StateTriggers define a threshold at which a visual state is activated, which then sets layout properties as appropriate for the window size that triggered the state change.

In the following example, when the window size is 720 pixels or more in width, the visual state named **wideView** is triggered, which then arranges the **Best-rated games** panel to appear to the right of, and aligned with the top of, the **Top free games** panel.



When the window is less than 720 pixels, the **narrowView** visual state is triggered because the **wideView** trigger is no longer satisfied and so no longer in effect. The **narrowView** visual state positions the **Best-rated games** panel below, and aligned with the left of, the **Top paid games** panel:



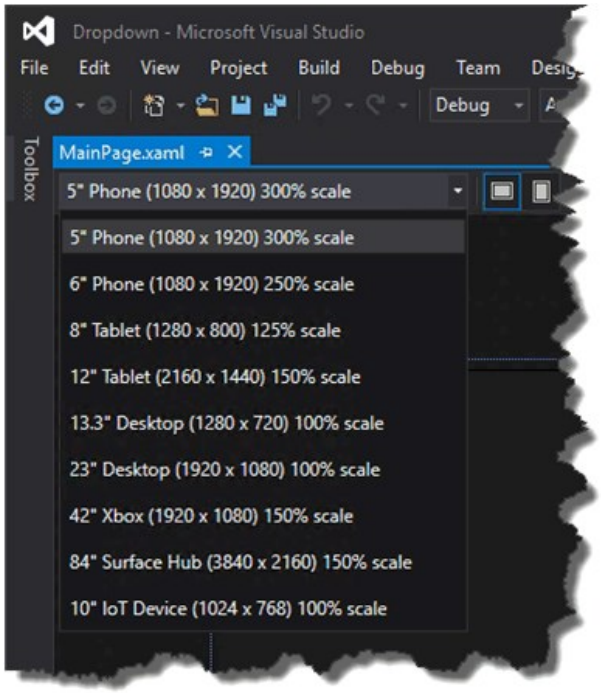
Here is the XAML for the visual state triggers described above. The definition of the panels, alluded to by "`...`" below, has been removed for brevity.

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup>
            <VisualState x:Name="wideView">
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="720" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
```

```
                <Setter Target="best.(RelativePanel.RightOf)" Value="free"/>
                <Setter Target="best.(RelativePanel.AlignTopWidth)" Value="free"/>
```

Tooling

By default, you'll probably want to target the broadest possible device family. When you're ready to see how your app looks and lays out on a particular device, use the device preview toolbar in Visual Studio to preview your UI on a small or medium mobile device, on a PC, or on a large TV screen. That way you can tailor and test your adaptive visual states:



You don’t have to make a decision up front about every device type that you'll support. You can add an additional device size to your project later.

Adaptive scaling

Windows 10 introduces an evolution of the existing scaling model. In addition to scaling vector content, there is a unified set of scale factors that provides a consistent size for UI elements across a variety of screen sizes and display resolutions. The scale factors are also compatible with the scale factors of other operating systems such as iOS and Android. This makes it easier to share assets between these platforms.

The Store picks the assets to download based in part of the DPI of the device. Only the assets that best match the device are downloaded.

Common input handling

You can build a Universal Windows app using universal controls that handle various inputs such as mouse, keyboard, touch, pen, and controller (such as the Xbox controller). Traditionally, inking has been associated only with pen input, but with Windows 10, you can ink with touch on some devices, and with any pointer input. Inking is supported on many devices (including mobile devices) and can easily be incorporated with a just few lines of code.

The following APIs provide access to input:

- **CoreIndependentInputSource** is a new API that allows you to consume raw input on the main thread or a background thread.
- **PointerPoint** unifies raw touch, mouse, and pen data into a single, consistent set of interfaces and events that can be consumed on the main thread or background thread by using **CoreInput.**
- **PointerDevice** is a device API that supports querying device capabilities so that you can determine what input modalities are available on the device.
- The new **InkCanvas** XAML control and **InkPresenter** Windows Runtime APIs allow you to access ink stroke data.

Writing code

Your programming language options for your Windows 10 project in Visual Studio include Visual C++, C#, Visual Basic, and JavaScript. For Visual

C++, C#, and Visual Basic, you can use XAML for a full-fidelity, native UI experience. For Visual C++ you can choose to draw with DirectX either instead of or as well as using XAML. For JavaScript, your presentation layer will be HTML, and HTML is of course a cross-platform web standard. Much of your code and UI will be universal and it will run the same way everywhere. But for code tailored to particular device families, and for UI tailored to particular form factors, you'll have the option to use adaptive code and adaptive UI. Let's look at these different cases.

Calling an API that's implemented by your target device family

Whenever you want to call an API, you'll need to know whether the API is implemented by the device family that your app is targeting. If in doubt, you can look it up in the API reference documentation. If you open the relevant topic and look at the Requirements section, you'll see what the implementing device family is. Let's say that your app is targeting version 10.0.x.0 of the universal device family and you want to call members of the **Windows.UI.Core.SystemNavigationManager** class. In this example, the device family is "Universal". It's a good idea to further confirm that the class members that you want to call are also within your target, and in this case they are. So in this example, you now know that the APIs are guaranteed to be present on every device that your app can be installed on, and you can call the APIs in your code just like you normally would.

```
Windows.UI.Core.SystemNavigationManager.GetForCurrentView().BackRequested += TestView_BackRequested;
```

As another example, imagine that your app is targeting version 10.0.x.0 of the Xbox device family, and the reference topic for an API that you want to call says that the API was introduced in version 10.0.x.0 of the Xbox device family. In that case, again, the API is guaranteed to be present on every device that your app can be installed on. So you would be able to call that API in your code in the normal way.

Note that Visual Studio's IntelliSense will not recognize APIs unless they are implemented by your app's target device family or any extension SDKs that you have referenced. Consequently, if you haven't referenced any extension SDKs, you can be sure that any APIs that appear in IntelliSense must therefore be in your target device family and you can call them freely.

Calling an API that's NOT implemented by your target device family

There will be cases when you want to call an API, but your target device family is not listed in the documentation. In that case you can opt to write adaptive code in order to call that API.

Writing adaptive code with the ApiInformation class

There are two steps to write adaptive code. The first step is to make the APIs that you want to access available to your project. To do that, add a reference to the extension SDK that represents the device family that owns the APIs that you want to conditionally call. See Extension SDKs.

The second step is to use the **Windows.Foundation.Metadata.ApiInformation** class in a condition in your code to test for the presence of the API you want to call. This condition is evaluated wherever your app runs, but it evaluates to true only on devices where the API is present and therefore available to call.

If you want to call just a small number of APIs, you could use the **ApiInformation.IsTypePresent** method like this.

```
// Note: Cache the value instead of querying it more than once.
bool isHardwareButtonsAPIPresent =
    Windows.Foundation.Metadata.ApiInformation.IsTypePresent("Windows.Phone.UI.Input.HardwareButtons");

if (isHardwareButtonsAPIPresent)
{
    Windows.Phone.UI.Input.HardwareButtons.CameraPressed +=
        HardwareButtons_CameraPressed;
}
```

In this case we can be confident that the presence of the **HardwareButtons** class implies the presence of the **CameraPressed** event, because the class and the member have the same requirements info. But in time, new members will be added to already-introduced classes, and those members will have later "introduced in" version numbers. In such cases, instead of using **IsTypePresent**, you can test for the presence of individual members by using **IsEventPresent**, **IsMethodPresent**, **IsPropertyPresent**, and similar methods. Here's an example.

```
bool isHardwareButtons_CameraPressedAPIPresent =
    Windows.Foundation.Metadata.ApiInformation.IsEventPresent
```

```
        ("Windows.Phone.UI.Input.HardwareButtons", "CameraPressed");
```

The set of APIs within a device family is further broken down into subdivisions known as API contracts. You can use the **ApiInformation.IsApiContractPresent** method to test for the presence of an API contract. This is useful if you want to test for the presence of a large number of APIs that all exist in the same version of an API contract.

```
bool isWindows_Devices_Scanners_ScannerDeviceContract_1_0Present =
    Windows.Foundation.Metadata.ApiInformation.IsApiContractPresent
        ("Windows.Devices.Scanners.ScannerDeviceContract", 1, 0);
```

Win32 APIs in the UWP

A UWP app or Windows Runtime Component written in C++/CX has access to the Win32 APIs that are part of the UWP. These Win32 APIs are implemented by all Windows 10 device families. Link your app with Windowsapp.lib. Windowsapp.lib is an "umbrella" lib that provides the exports for the UWP APIs. Linking to Windowsapp.lib will add to your app dependencies on dlls that are present on all Windows 10 device families.

For the full list of Win32 APIs available to UWP apps, see API Sets for UWP apps and Dlls for UWP apps.

User experience

A Universal Windows app allows you to take advantage of the unique capabilities of the device on which it is running. Your app can make use of all of the power of a desktop device, the natural interaction of direct manipulation on a tablet (including touch and pen input), the portability and convenience of mobile devices, the collaborative power of Surface Hub, and other devices that support UWP apps.

Good design is the process of deciding how users will interact with your app, as well as how it will look and function. User experience plays a huge part in determining how happy people will be with your app, so don't skimp on this step. Design basics introduce you to designing a Universal Windows app. See the Introduction to Universal Windows Platform (UWP) apps for designers for information on designing UWP apps that delight your users. Before you start coding, see the device primer to help you think through the interaction experience of using your app on all the different form factors you want to target.

In addition to interaction on different devices, plan your app to embrace the benefits of working across multiple devices. For example:

- Use cloud services to sync across devices. Learn how to connect to web services in support of your app experience.
- Consider how you can support users moving from one device to another, picking up where they left off. Include notifications and in-app purchases in your planning. These features should work across devices.
- Design your workflow using Navigation design basics for UWP apps to accommodate mobile, small-screen, and large-screen devices. Lay out your user interface to respond to different screen sizes and resolutions.
- Consider whether there are features of your app that don't make sense on a small mobile screen. There may also be areas that don't make sense on a stationary desktop machine and require a mobile device to light up. For example, most scenarios around location imply a mobile device.
- Consider how you'll accommodate multiple input modalities. See the Guidelines for interactions to learn how users can interact with your app by using Cortana, Speech, Touch interactions, the Touch keyboard and more.

  See the Guidelines for text and text input for more tradition interaction experiences.

Submit a Universal Windows app through your Dashboard

The new unified Windows Dev Center dashboard lets you manage and submit all of your apps for Windows devices in one place. New features simplify processes while giving you more control. You'll also find detailed analytic reports combined payout details, ways to promote your app and

engage with your customers, and much more.

See Using the unified Windows Dev Center dashboard to learn how to submit your apps for publication in the Windows Store.

.NET for UWP apps

.NET for UWP apps provides a set of managed types that you can use to create Universal Windows Platform apps for Windows 10 using C# or Visual Basic. The following list displays the namespaces in .NET for UWP apps. Note that .NET for UWP apps includes a subset of the types provided in the full .NET Framework for each namespace. For information about individual namespaces, see the linked topics.

For more information, see the .NET for Windows Store apps overview.

UWP only: APIs for UWP apps that are expressed as HTML or XAML elements are supported only in UWP apps; they are not supported in desktop apps or Windows desktop browsers.

Namespaces

Namespace	Description
System	Contains the fundamental classes and base classes that define commonly used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
System.CodeDom.Compiler	Contains the types for managing the generation and compilation of source code in supported programming languages.
System.Collections	Contains the interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hash tables and dictionaries.
System.Collections.Concurrent	Contains the interfaces and classes that define various collection objects for concurrent processing.
System.Collections.Generic	Contains the interfaces and classes that define generic collections, which enable users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.
System.Collections.ObjectModel	Contains the classes that can be used as collections in the object model of a reusable library. Use these classes when properties or methods return collections.
System.Collections.Specialized	Contains specialized and strongly typed collections; for example, a linked list dictionary, a bit vector, and collections that contain only strings.
System.ComponentModel	Provides the classes that are used to implement the run-time and design-time behavior of components and controls.
System.ComponentModel.DataAnnotations	Provides the attribute classes that are used to define metadata for controls.
System.ComponentModel.DataAnnotations.Schema	Provides support for attribute classes that are used to define metadata for controls.

System.Composition	To install the System.Composition namespaces, open your project in Visual Studio 2015 or later, choose **Manage NuGet Packages** from the **Project** menu, and search online for the Microsoft.Composition package.
System.Composition.Convention	To install the System.Composition namespaces, open your project in Visual Studio 2015 or later, choose **Manage NuGet Packages** from the **Project** menu, and search online for the Microsoft.Composition package.
System.Composition.Hosting	To install the System.Composition namespaces, open your project in Visual Studio 2015 or later, choose **Manage NuGet Packages** from the **Project** menu, and search online for the Microsoft.Composition package.
System.Composition.Hosting.Core	To install the System.Composition namespaces, open your project in Visual Studio 2015 or later, choose **Manage NuGet Packages** from the **Project** menu, and search online for the Microsoft.Composition package.
System.Data	Provides access to the classes that represent the ADO.NET architecture. ADO.NET lets you build components that efficiently manage data from multiple data sources.
System.Data.Common	Contains classes shared by the .NET Framework data providers.
System.Diagnostics	Provides the classes that enable you to interact with system processes, event logs, and performance counters.
System.Diagnostics.CodeAnalysis	Contains the classes for interaction with code analysis tools.
System.Diagnostics.Contracts	Contains the static classes for representing program constructs such as preconditions, postconditions, and invariants.
System.Diagnostics.Tracing	Provides the types and members that allow developers to create strongly typed events to be captured by Event Tracing for Windows (ETW).
System.Dynamic	Provides the classes and interfaces that support the dynamic language runtime (DLR).
System.Globalization	Contains the classes that define culture-related information, including the language, the country/region, the calendars in use, the format patterns for dates, currency, and numbers, and the sort order for strings.
System.IO	Contains the types that enable synchronous and asynchronous reading and writing on data streams and files.
System.IO.IsolatedStorage	Contains types that allow the creation and use of isolated stores.
System.IO.Compression	Contains the classes that provide basic compression and decompression for streams.
System.Linq	Contains the classes and interfaces that support queries that use Language-Integrated Query (LINQ).
System.Linq.Expressions	Contains the types that enable language-level code expressions to be represented as objects in the form of expression trees.
System.Net	Provides a simple programming interface for many of the protocols that are used on networks today.
System.Net.Http	Provides a programming interface for modern HTTP applications.
System.Net.Http.Headers	Provides support collections of HTTP headers that are used by the System.Net.Http namespace.
System.Net.NetworkInformation	Provides access to network traffic data, network address information, and notification of address changes for the local computer.
System.Net.Security	Provides network streams for secure communications between hosts.

System.Net.Sockets	Provides a managed implementation of the Windows Sockets (Winsock) interface for developers who need to tightly control access to the network.
System.Numerics	Contains the types that complement the numeric primitives that are defined by the .NET Framework.
System.Reflection	Contains the classes and interfaces that provide a managed view of loaded types, methods, and fields, with the ability to dynamically create and invoke types.
System.Reflection.Context	Contains the classes that enable customized reflection contexts.
System.Reflection.Emit	Contains the classes that allow a compiler or tool to emit metadata and Microsoft intermediate language (MSIL) and optionally generate a PE file on disk. The primary clients of these classes are script engines and compilers.
System.Resources	Provides the classes and interfaces that enable developers to create, store, and manage various culture-specific resources that are used in an application.
System.Runtime	Contains the advanced types that support diverse namespaces such as System, the Runtime namespaces, and the Security namespaces.
System.Runtime.CompilerServices	Provides functionality for compiler writers who use managed code to specify attributes in metadata that affect the run-time behavior of the common language runtime.
System.Runtime.ExceptionServices	Provides the classes for advanced exception handling.
System.Runtime.InteropServices	Provides a wide variety of members that support COM interop and platform invoke services.
System.Runtime.InteropServices.ComTypes	Contains the methods that are definitions of COM functions for managed code.
System.Runtime.InteropServices.WindowsRuntime	Contains the classes that support interoperation between managed code and the Windows Runtime, and that enable the creation of Windows Runtime types with managed code.
System.Runtime.Serialization	Contains the classes that can be used for serializing and deserializing objects.
System.Runtime.Serialization.Json	Contains the types for serializing objects to JavaScript Object Notation (JSON) and deserializing objects from JSON.
System.Runtime.Versioning	Contains the advanced types that support versioning in side-by-side implementations of the .NET Framework.
System.Security	Provides the underlying structure of the .NET Framework security system, including base classes for permissions.
System.Security.Authentication	Provides a set of enumerations that describe the security of a connection.
System.Security.Authentication.ExtendedProtection	Provides support for authentication using extended protection for applications.
System.Security.Claims	Contains classes that implement claims-based identity in the .NET Framework, including classes that represent claims, claims-based identities, and claims-based principals.

System.Security.Cryptography	Provides cryptographic services, including secure encoding and decoding of data, as well as many other operations, such as hashing, random number generation, and message authentication.
System.Security.Cryptography.X509Certificates	Contains the common language runtime implementation of the Authenticode X.509 v.3 certificate. This certificate is signed with a private key that uniquely and positively identifies the holder of the certificate.
System.Security.Principal	Defines a principal object that represents the security context under which code is running.
System.ServiceModel	Contains the types that are required to build Windows Communication Foundation (WCF) service and client applications that can be used to build widely distributed applications.
System.ServiceModel.Channels	Contains the types that are required to construct and modify the messages that are used by clients and services to communicate with each other, the types of channels that are used to exchange messages, the channel factories and channel listeners that are used to construct those channels, and the binding elements that are used to configure them.
System.ServiceModel.Description	Contains the types that are required to construct and modify descriptions of services, contracts, and endpoints that are used to build service runtimes and to export metadata.
System.ServiceModel.Dispatcher	Contains the types that are required to modify the run-time execution behavior of service and client applications.
System.ServiceModel.Security	Contains the classes that support general Windows Communication Foundation (WCF) security.
System.ServiceModel.Security.Tokens	Contains the types that represent security tokens and certificates for Windows Communication Foundation (WCF) security.
System.Text	Contains the classes that represent character encodings; and a helper class that manipulates and formats String objects without creating intermediate instances of String.
System.Text.RegularExpressions	Contains the classes that provide access to the .NET Framework regular expression engine.
System.Threading	Provides the classes and interfaces that enable multithreaded programming.
System.Threading.Tasks	Provides the types that simplify the work of writing concurrent and asynchronous code.
System.Threading.Tasks.Dataflow	Provides an actor-based programming model that provides in-process message passing for coarse-grained dataflow and pipelining tasks. To install the System.Threading.Tasks.Dataflow namespace, open your project in Visual Studio 2015 or later, choose **Manage NuGet Packages** from the **Project** menu, and search online for the Microsoft.Tpl.Dataflow package.
System.Windows.Input	Contains the types that enable custom commands.
System.Xml	Provides standards-based support for processing XML.
System.Xml.Linq	Contains the types for LINQ to XML, which is an in-memory XML programming interface that enables you to modify XML documents efficiently and easily.
System.Xml.Schema	Contains the XML classes that provide standards-based support for XML Schema definition language (XSD) schemas.
System.Xml.Serialization	Contains the classes that are used to serialize objects into XML format documents or streams.

System.Xml.XPath	Contains the classes that define a cursor model for navigating and editing XML information items as instances of the XQuery 1.0 and XPath 2.0 Data Model.
Windows.Foundation	Enables fundamental Windows Runtime functionality, including managing asynchronous operations, accessing property stores, and working with images and URIs.
Windows.UI	Provides a Windows 8.x Store app with access to core system functionality and run-time information about its UI.
Windows.UI.Xaml	Provides general framework API and application model API, and a variety of support classes that are commonly used by many different feature areas.
Windows.UI.Xaml.Controls.Primitives	Defines classes that represent the component parts of UI controls, or otherwise support the control composition model. Also defines interfaces for control patterns such as snapping and selection.
Windows.UI.Xaml.Media	Provides basic media support, graphics primitives, and brush-drawing APIs.
Windows.UI.Xaml.Media.Animation	Provides animation and storyboard API for transition animations, visual states, or animated UI components.
Windows.UI.Xaml.Media.Media3D	Contains the types that support matrix/perspective transformation.
Microsoft.CSharp.RuntimeBinder	Contains the types that support interoperation between the dynamic language runtime (DLR) and C#.
Microsoft.VisualBasic	Contains the classes that support compilation and code generation using the Visual Basic language.
Microsoft.VisualBasic.CompilerServices	Contains the internal-use only types that support the Visual Basic compiler.
Microsoft.Win32.SafeHandles	Contains classes that are abstract derivations of safe handle classes that provide common functionality supporting file and operating system handles.

.NET for Windows 8.x Store apps

The .NET for Windows 8.x Store apps provide a set of managed types that you can use to create Windows 8.x Store apps for Windows using C# or Visual Basic. The following list displays the namespaces in the .NET for Windows 8.x Store apps. Note that the .NET for Windows 8.x Store apps include a subset of the types provided in the full .NET Framework for each namespace. For information about individual namespaces, see the linked topics.

For more information, see .NET for Windows Store apps overview.

Windows 8.x Store apps only: APIs for Windows 8.x Store apps that are expressed as HTML or XAML elements are supported only in Windows 8.x Store apps; they are not supported in desktop apps or Windows desktop browsers.

Namespaces

Namespace	Description
System	Contains fundamental classes and base classes that define commonly used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
System.CodeDom.Compiler	Contains types for managing the generation and compilation of source code in supported programming languages.
System.Collections	Contains interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hash tables and dictionaries.
System.Collections.Concurrent	Contains interfaces and classes that define various collection objects for concurrent processing.
System.Collections.Generic	Contains interfaces and classes that define generic collections, which enable users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.
System.Collections.ObjectModel	Contains classes that can be used as collections in the object model of a reusable library. Use these classes when properties or methods return collections.
System.Collections.Specialized	Contains specialized and strongly typed collections; for example, a linked list dictionary, a bit vector, and collections that contain only strings.
System.ComponentModel	Provides classes that are used to implement the run-time and design-time behavior of components and controls.
System.ComponentModel.DataAnnotations	Provides attribute classes that are used to define metadata for controls.
System.ComponentModel.DataAnnotations.Schema	Provides support for attribute classes that are used to define metadata for controls.
System.Compositi	To install the System.Composition namespaces, open your project in Visual Studio 2012 or later, choose **Manage**

on	**NuGet Packages** from the **Project** menu, and search online for the Microsoft.Composition package.
System.Composition.Convention	To install the System.Composition namespaces, open your project in Visual Studio 2012 or later, choose **Manage NuGet Packages** from the **Project** menu, and search online for the Microsoft.Composition package.
System.Composition.Hosting	To install the System.Composition namespaces, open your project in Visual Studio 2012 or later, choose **Manage NuGet Packages** from the **Project** menu, and search online for the Microsoft.Composition package.
System.Composition.Hosting.Core	To install the System.Composition namespaces, open your project in Visual Studio 2012 or later, choose **Manage NuGet Packages** from the **Project** menu, and search online for the Microsoft.Composition package.
System.Diagnostics	Provides classes that enable you to interact with system processes, event logs, and performance counters.
System.Diagnostics.CodeAnalysis	Contains classes for interaction with code analysis tools.
System.Diagnostics.Contracts	Contains static classes for representing program constructs such as preconditions, postconditions, and invariants.
System.Diagnostics.Tracing	Provides the types and members that allow developers to create strongly typed events to be captured by Event Tracing for Windows (ETW).
System.Dynamic	Provides classes and interfaces that support the dynamic language runtime (DLR).
System.Globalization	Contains classes that define culture-related information, including the language, the country/region, the calendars in use, the format patterns for dates, currency, and numbers, and the sort order for strings.
System.IO	Contains types that enable synchronous and asynchronous reading and writing on data streams and files.
System.IO.Compression	Contains classes that provide basic compression and decompression for streams.
System.Linq	Contains classes and interfaces that support queries that use Language-Integrated Query (LINQ).
System.Linq.Expressions	Contains types that enable language-level code expressions to be represented as objects in the form of expression trees.
System.Net	Provides a simple programming interface for many of the protocols used on networks today.
System.Net.Http	Provides a programming interface for modern HTTP applications.
System.Net.Http.Headers	Provides support collections of HTTP headers used by the System.Net.Http namespace.
System.Net.NetworkInformation	Provides access to network traffic data, network address information, and notification of address changes for the local computer.
System.Numerics	Contains types that complement the numeric primitives that are defined by the .NET Framework.
System.Reflection	Contains classes and interfaces that provide a managed view of loaded types, methods, and fields, with the ability to dynamically create and invoke types.
System.Reflection.Context	Contains classes that enable customized reflection contexts.
System.Reflection.Emit	Contains classes that allow a compiler or tool to emit metadata and Microsoft intermediate language (MSIL) and optionally generate a PE file on disk. The primary clients of these classes are script engines and compilers.
System.Resources	Provides classes and interfaces that enable developers to create, store, and manage various culture-specific resources used in an application.
System.Runtime	Contains advanced types that support diverse namespaces such as System, the Runtime namespaces, and the Security namespaces.
System.Runtime.C	Provides functionality for compiler writers who use managed code to specify attributes in metadata that affect the

ompilerServices	run-time behavior of the common language runtime.
System.Runtime.ExceptionServices	Provides classes for advanced exception handling.
System.Runtime.InteropServices	Provides a wide variety of members that support COM interop and platform invoke services.
System.Runtime.InteropServices.ComTypes	Contains methods that are definitions of COM functions for managed code.
System.Runtime.InteropServices.WindowsRuntime	Contains classes that support interoperation between managed code and the Windows Runtime, and that enable the creation of Windows Runtime types with managed code.
System.Runtime.Serialization	Contains classes that can be used for serializing and deserializing objects.
System.Runtime.Serialization.Json	Contains types for serializing objects to JavaScript Object Notation (JSON) and deserializing objects from JSON.
System.Runtime.Versioning	Contains advanced types that support versioning in side-by-side implementations of the .NET Framework.
System.Security	Provides the underlying structure of the .NET Framework security system, including base classes for permissions.
System.Security.Principal	Defines a principal object that represents the security context under which code is running.
System.ServiceModel	Contains the types necessary to build Windows Communication Foundation (WCF) service and client applications that can be used to build widely distributed applications.
System.ServiceModel.Channels	Contains the types required to construct and modify the messages used by clients and services to communicate with each other, the types of channels used to exchange messages, the channel factories and channel listeners used to construct those channels, and the binding elements used to configure them.
System.ServiceModel.Description	Contains the types requires to construct and modify descriptions of services, contracts, and endpoints that are used to build service runtimes and to export metadata.
System.ServiceModel.Dispatcher	Contains the types necessary to modify the run-time execution behavior of service and client applications.
System.ServiceModel.Security	Contains classes that support general Windows Communication Foundation (WCF) security.
System.ServiceModel.Security.Tokens	Contains types that represent security tokens and certificates for Windows Communication Foundation (WCF) security.
System.Text	Contains classes representing character encodings; and a helper class that manipulates and formats String objects without creating intermediate instances of String.
System.Text.RegularExpressions	Contains classes that provide access to the .NET Framework regular expression engine.
System.Threading	Provides classes and interfaces that enable multithreaded programming.
System.Threading.Tasks	Provides types that simplify the work of writing concurrent and asynchronous code.
System.Threading.Tasks.DataFlow	Provides an actor-based programming model that provides in-process message passing for coarse-grained dataflow and pipelining tasks. To install the System.Threading.Tasks.Dataflow namespace, open your project in Visual Studio 2012 or later, choose **Manage NuGet Packages** from the **Project** menu, and search online for the Microsoft.Tpl.Dataflow package.

System.Windows.Input	Contains types that enable custom commands.
System.Xml	Provides standards-based support for processing XML.
System.Xml.Linq	Contains the types for LINQ to XML, which is an in-memory XML programming interface that enables you to modify XML documents efficiently and easily.
System.Xml.Schema	Contains the XML classes that provide standards-based support for XML Schema definition language (XSD) schemas.
System.Xml.Serialization	Contains classes that are used to serialize objects into XML format documents or streams.
Windows.Foundation	Enables fundamental Windows Runtime functionality, including managing asynchronous operations, accessing property stores, and working with images and URIs.
Windows.UI	Provides a Windows 8.x Store app with access to core system functionality and run-time information about its UI.
Windows.UI.Xaml	Provides general framework API and application model API, and a variety of support classes that are commonly used by many different feature areas.
Windows.UI.Xaml.Controls.Primitives	Defines classes that represent the component parts of UI controls, or otherwise support the control composition model. Also defines interfaces for control patterns such as snapping and selection.
Windows.UI.Xaml.Media	Provides basic media support, graphics primitives, and brush-drawing APIs.
Windows.UI.Xaml.Media.Animation	Provides animation and storyboard API for transition animations, visual states, or animated UI components.
Windows.UI.Xaml.Media.Media3D	Contains types that support matrix/perspective transformation.
Microsoft.CSharp.RuntimeBinder	Contains types that support interoperation between the dynamic language runtime (DLR) and C#.
Microsoft.VisualBasic	Contains classes that support compilation and code generation using the Visual Basic language.
Microsoft.VisualBasic.CompilerServices	Contains internal-use only types that support the Visual Basic compiler.