Microsoft

Developing an end-to-end Windows Store app using C++ and XAML: Hilo

David Britch
Bob Brumfield
Colin Campbell
Scott Densmore
Thomas Petchel
Rohit Sharma
Blaine Wastell

November 2012

patterns & practices

Contents

Developing an end-to-end Windows Store app using C++ and XAML: Hilo

The Hilo end-to-end photo sample provides guidance to C++ developers that want to create a Windows 8 app using modern C++, XAML, the Windows Runtime, and recommended development patterns. Hilo comes with source code and documentation.

Here's what you'll learn:

- How to use modern C++, asynchronous programming, XAML, and the Windows Runtime to build a world-ready app for the global market. The Hilo source code includes support for four languages and all world calendars.
- How to implement tiles, pages, controls, touch, navigation, file system queries, suspend/resume, and localization.
- How to implement Model-View-ViewModel (MVVM) and Repository patterns.
- How to test your app and tune its performance.

Note If you're new to XAML, read XAML overview to learn more about its purpose and syntax. Read Tutorial: Create your first Windows Store app using C++ to learn how to create a small Windows Store app with C++ and XAML. Then, download Hilo to see a complete app that demonstrates recommended implementation patterns.

Applies to

- Windows Runtime for Windows 8
- XAML
- Visual C++ component extensions (C++/CX)
- Parallel Patterns Library (PPL)

Download

Download Hilo sample

After you download the code, see Getting started with Hilo for instructions.

Prerequisites

- Windows 8
- Microsoft Visual Studio 2012
- An interest in C++ and XAML programming

Visit Windows Store app development to download the latest tools for Windows Store app development.

Table of contents at a glance

Here are the major topics in this guide. For the full table of contents, see Hilo table of contents.

Note This content is available on the web as well. For more info, see Developing an end-to-end Windows Store app using C++ and XAML: Hilo (Windows).

Why XAML?

If you're familiar with Hilo for Windows 7, you might wonder why we chose XAML instead of DirectX for this version. Here is why:

- This version of Hilo is not a port or rewrite of the original. Instead, it carries forward the spirit of creating a modern photo app for Windows using the latest technologies.
- The Windows Runtime provides the features that we wanted. XAML is accelerated by graphics hardware, and provides the necessary performance. Therefore, we didn't need to write infrastructure code with DirectX to enable the experience.
- With DirectX, you have to build all of the UI infrastructure yourself. The Windows Runtime and XAML provide the controls, animation support, and other functionality that supports Windows Store apps.
- C++ is an imperative language. In a DirectX app, you use C++ to explicitly define *what* work needs to be done and *how* that work is done. XAML is a declarative language. We felt that the declarative model enables us to be more productive because we can state how the UI should

work, and the Windows Runtime does the work for us. This way we could focus more of our time more on design and core app logic.

Note Just because we didn't use DirectX in our app doesn't mean it won't work for yours. If you prefer DirectX or your app or game has specific requirements or cannot be written in XAML or JavaScript, see Developing games.

You can also use XAML and DirectX together in your Windows Store app. There are two approaches. You can add XAML to a DirectX app, or you can include DirectX surfaces in a XAML app. Which one to use depends on the nature of the app. For example, an immersive, full-screen 3-D game might use a small amount of XAML for the heads-up display. In contrast, a recipe app for home cooks might use XAML extensively with only a few DirectX surfaces in cases where it needs special visual effects. For more info, see DirectX and XAML interop.

Learning resources

If you're new to C++ programming for Windows Store apps, read Roadmap for Windows Store apps using C++.

We also found Welcome Back to C++ (Modern C++) and C++ and Beyond 2011: Herb Sutter - Why C++? to be helpful resources for learning more about modern C++. The document Writing modern C++ code explains how we applied modern C++ principles to Hilo.

You might also want to read Index of UX guidelines for Windows Store apps and Blend for Visual Studio for user experience guidelines that can help you create a great Windows Store app. The document Designing Hilo's UX explains how we designed the Hilo UX.

Getting started with Hilo (Windows Store apps using C++ and XAML)

Here we explain how to build and run the Hilo Windows Store app, how the C++ and XAML source code is organized, and what tools and languages it uses.

Download

Download Hilo sample

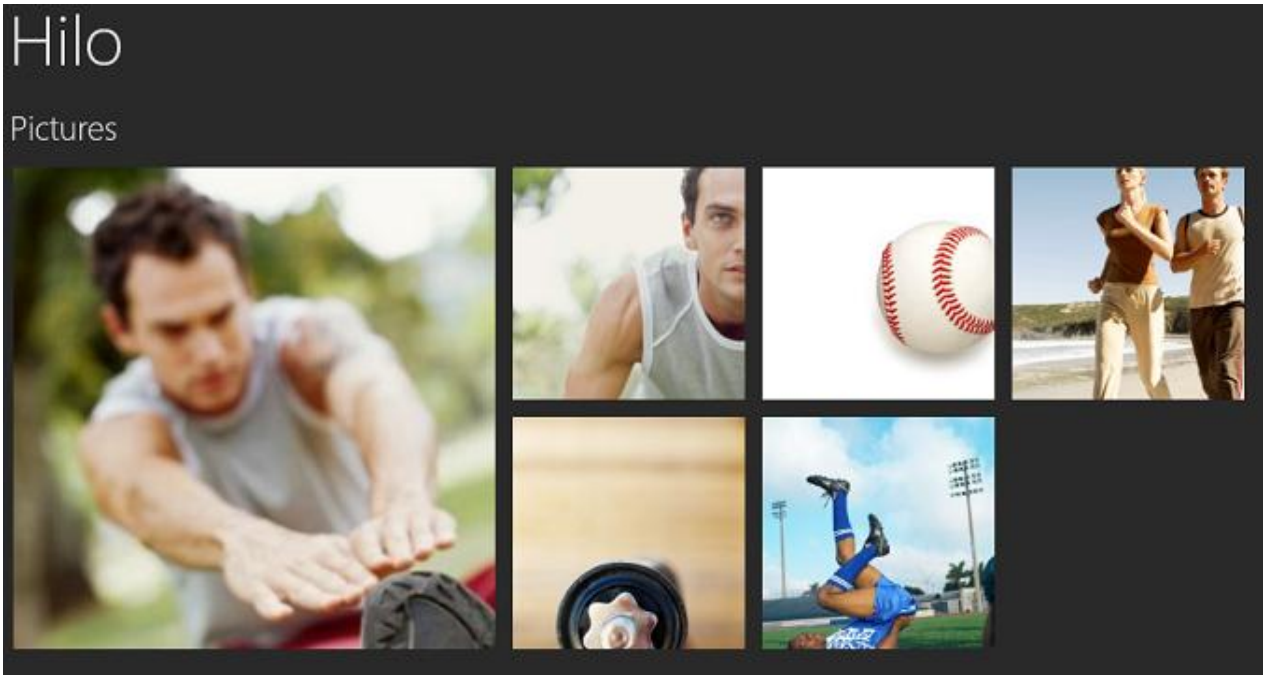
After you download the code, see "Building and running the sample" on this page for instructions.

Important Before you run the sample, ensure that you have at least 1 image in your Pictures library. Having a variety of image formats and sizes will exercise more parts of the code. For example, the rotate operation stores the orientation in Exchangeable Image File (Exif) data for images that support Exif (such as many JPEG and TIFF images). The cartoon effect operation scales down large images to a maximum of 1024x768 pixels.

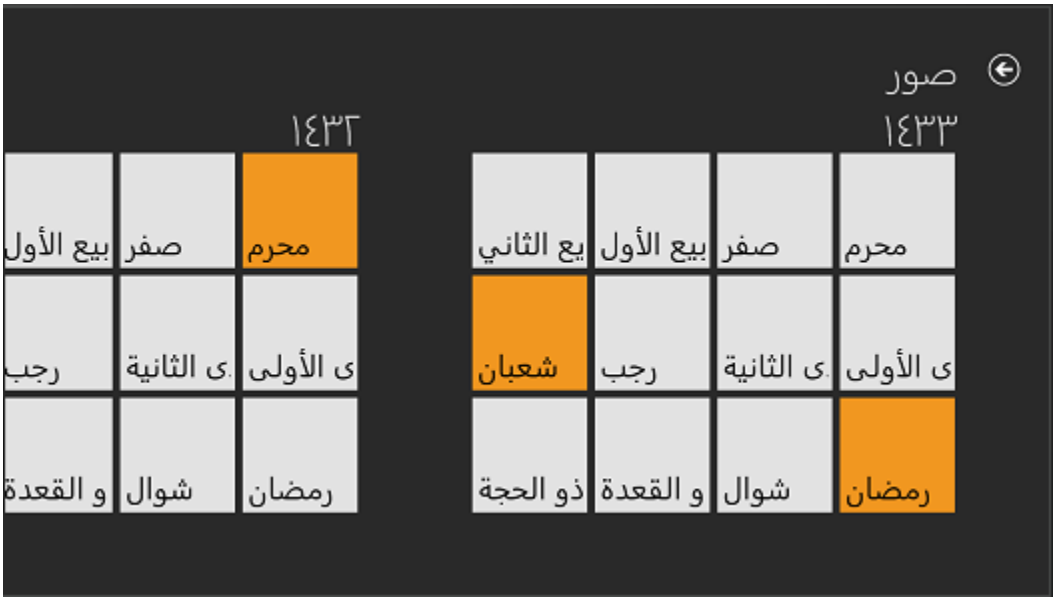
Building and running the sample

Build the Hilo project as you would build a standard project.

1. On the menu bar, choose **Build** > **Build Solution**. The build step compiles the code and also packages it for use as a Windows Store app.
2. After you build the project, you must deploy it. On the menu bar, choose **Build** > **Deploy Solution**. Microsoft Visual Studio also deploys the project when you run the app from the debugger.
3. After you deploy the project, pick the Hilo tile to run the app. Alternatively, from Visual Studio, on the menu bar, choose **Debug** > **Start Debugging**. Make sure that Hilo is the startup project.



You can run Hilo in any of the languages that it supports. Set the desired language from Control Panel. For example, if you set the preferred language to Arabic (Saudi Arabia) before you start Hilo, the app will display Arabic text right-to-left and use the default calendar for that locale. Here is a screen shot of the year groups view localized for Arabic (Saudi Arabia).



The Hilo source code includes localization for Arabic (Saudi Arabia), English (United States), German (Germany) and Japanese (Japan).

Projects and solution folders

The Hilo Visual Studio solution contains three projects: Hilo, HiloTests, and CartoonEffect.

Note The version of Hilo that contains the HiloTests project is available at patterns & practices - Develop Windows Store apps using C++ & XAML: Hilo.

The Hilo project uses Visual Studio solution folders to organize the source code files into these logical categories:

- The **Assets** folder contains the splash screen, tile, and other images.
- The **Common** folder contains common page and navigation functionality for the app.
- The **Strings** folder contains resource strings, with subfolders for each locale.
- The **ExceptionHandling** folder defines the policies and classes for dealing with unhandled exceptions.
- The **Imaging** folder contains imaging extension code and helpers.
- The **Models** folder contains repository code and other classes that are used by viewmodels.
- The **Tile** folder contains the code that updates the app's Start screen tile.
- The **ViewModels** folder contains the application logic that is exposed to XAML controls.
- The **Views** folder contains the app's XAML controls.
- The **XamlExtensions** folder contains data converters and other utilities.

The HiloTests project contains unit tests for Hilo. It shares code with the Hilo project and adds source files that contain unit tests. The CartoonEffect project implements an image processing algorithm that applies a cartoon effect to an image, packaged as a static library.

You can reuse some of the components in Hilo with any app with little or no modification. For your own app, you can adapt the organization and ideas that these files provide. When we consider a coding pattern to be especially applicable in any app, we call it out here.

The development tools and languages

Hilo is a Windows Store app that uses C++ and XAML. This combination is not the only option. You can write Windows Store apps in many ways, using the language of your choice. When we considered which language to use for Hilo, we asked these questions:

- **What kind of app do we want to build?** If you're creating a food, banking, or photo app, you might use HTML5/CSS/JavaScript or XAML/C++/C#/Visual Basic because the Windows Runtime provides enough built-in controls and functionality to create these kinds of apps. However, if you're creating a 3-D app or game and want to take full advantage of graphics hardware, you might choose C++ and DirectX.

- **What is our current skillset?** If you're a XAML expert, then XAML and C++ or .NET might be a natural choice for your app. If you know web development technologies, you might choose HTML5, CSS, and JavaScript.
- **What existing code can we bring forward?** If you have existing code, algorithms, or libraries that work with Windows Store apps, you can bring that code forward. For example, if you have existing app logic written in C++, you might consider creating your Windows Store app with C++.

Tip You can build reusable Windows Runtime components using C++, C#, or Visual Basic. You can use your component in any of your apps, even if they are written in a different programming language. For more info, see Creating Windows Runtime Components.

We chose C++ for performance and because it matched our skillsets. We believed that a full photo app would likely perform compute-intensive image processing such as image filter effects, and we knew that C++ would let us best take advantage of the multicore and GPU capabilities of the hardware platform in this case. For example, Hilo uses C++ Accelerated Massive Parallelism (C++ AMP) to implement a cartoon effect. C++ gave us a direct path to libraries such as C++ AMP and the Parallel Patterns Library (PPL).

We chose XAML for developer productivity. We believed that XAML's declarative UI approach and many built-in features would save us time and let us focus on design and core app logic.

We chose Visual Studio Express as our environment to write, debug, and unit test code, Team Foundation Server to track planned work, manage bug reports, version source code, and automate builds, and Blend for Visual Studio to design animations.

The document Getting started with Windows Store apps orients you to the language options available.

Note Regardless of which language you choose, you'll want to ensure your app is *fast and fluid* to give your users the best possible experience. The Windows Runtime enables this by providing asynchronous operations. Each programming language provides a way to consume these operations. See Asynchronous programming to learn more about how we used C++ to implement a fast and fluid UX.

Designing Hilo's UX (Windows Store apps using C++ and XAML)

Summary

- Focus on the UX and not on what features the app will have.
- Use storyboards to quickly iterate on the UX.
- Use standard Windows features to provide a UX that is consistent with other apps. Also validate the UX with the guidelines index for Windows Store apps.

The document Planning Windows Store apps suggests design guidelines to follow. Here we explain the design process for the Hilo C++ sample app UX and the Windows 8 features that we will use as part of the app. You might want to read Index of UX guidelines for Windows Store apps before you read this document.

The planning process was iterative. We brainstormed the experience goals, ran them past UX designers and customers, gathered feedback, and incorporated that feedback into our planning. Throughout the development process, we regularly checked with our advisors to make sure that we were moving towards our original goals. For example, we received early feedback to demonstrate how to use C++ Accelerated Massive Parallelism (C++ AMP) in Hilo. We had existing code that applies a cartoon effect to an image, so we added this feature. (For more info, see Asynchronous programming.)

To meet our larger overall goal of demonstrating how to develop a Windows Store app using the latest technologies, we added a feature only when it shows something unique about creating Windows Store apps. For example, the app starts on a hub page instead of directly on the image browser page. We considered adding additional sections to the hub page that let you view pictures in different ways (for example, your tagged "favorites"). But we felt that this distracted from the app's core features. We designed Hilo so that you can extend its functionality.

You will learn

- How we tied our "great at" statement to the app flow.
- How storyboards and prototypes drive design.
- Which Windows 8 features to consider as you plan your app.

Deciding the UX goals

Hilo is a photo app, and so we wanted to pick the right experiences for users to relive their memories.

Brainstorming the user experience

Our first step was to brainstorm which aspects of a photo app are the most crucial for a great UX and let these features guide us through the design process. Here are some of the features we came up with:

- Browse photos
- Rotate and crop photos
- Perform red-eye reduction and other image manipulations
- Tag and categorize photos
- Synchronize photos across devices
- Create a photo collage
- Create a photo slideshow
- Share photos on the web

What's Hilo great at?

Planning Windows Store apps suggests that you create a "great at" statement to guide your UX planning. Here's the "great at" statement we came up with for Hilo:

Hilo is great at helping people relive their memories.

There were many things that we *could* have done in Hilo. But we felt that the ability to browse, view, rotate, and crop photos best demonstrate the basics for creating a basic, yet complete, photo app. We focused on the kinds of features that characterize a Windows Store app.

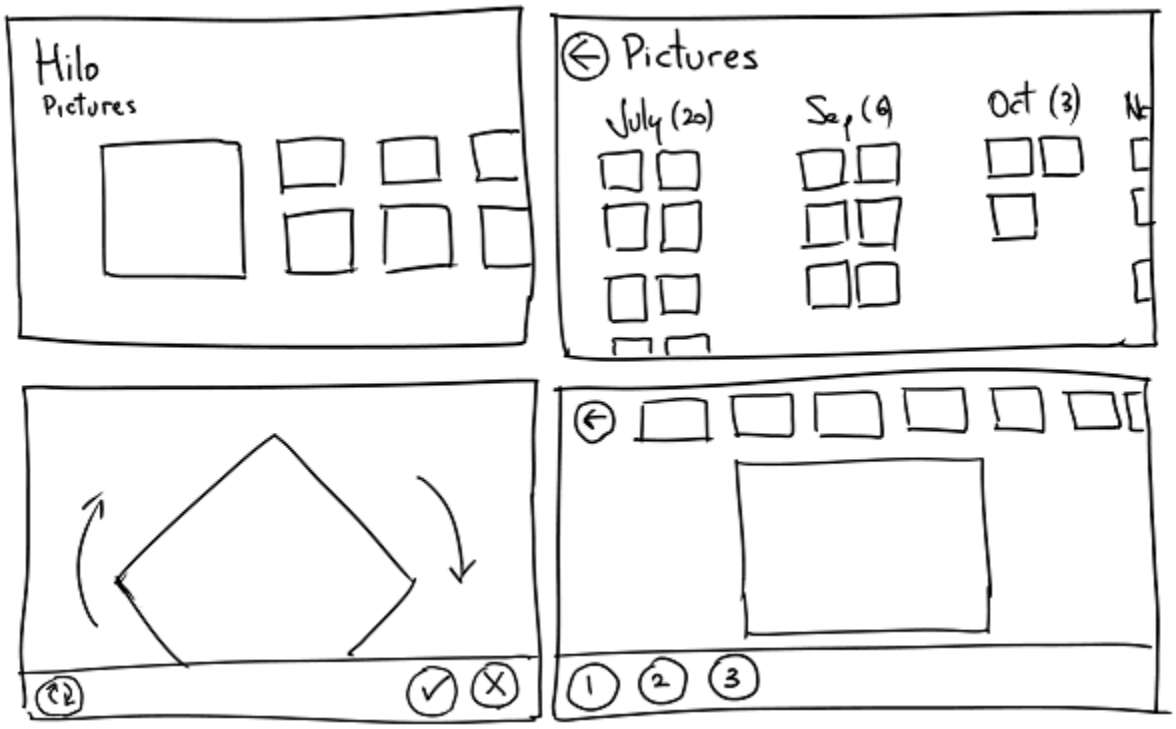
Realistically, the true goal of Hilo is not to provide a great photo app, but to demonstrate the key components and methodologies that make a great app. But we used this statement to guide our design tradeoffs as we built the app. By focusing on the user scenario, and not the individual features, we were better able to focus on what our users can do, and not what the app can do.

Deciding the app flow

The app flow ties to our "great at" statement. A flow defines how the user interacts with the app to perform tasks. Windows Store apps should be intuitive and require the fewest interactions. We used two common techniques to help with this step: creating storyboards and mock-ups.

A *storyboard* defines the flow of an app. Storyboards focus on how we intend the app to behave, and not the specific details of what it will look like. Storyboards help bridge the gap between the idea of the app and its implementation, but are typically faster and cheaper to produce than prototyping the real thing. For Hilo, storyboards were critical to helping us define a UX that naturally flows the user through the experience. This technique has been used in the film industry and is now becoming more common in UX design.

Here is a storyboard.



Note As we created the storyboard, we realized that grouping pictures by month instead of by folder would tie back to our "great at" statement. People typically associate memories with time more than folders on their computers.

A *mockup* also demonstrates the flow of the UX, but more closely resembles what the end product will look like. We created mock-ups based on our storyboards and presented them to our advisors for feedback. These mockups also helped each developer get a feel for what the resulting app should look like.

Here is a prototype of the image view page.

Tip During planning, you can also create small *prototypes* to validate feasibility. A prototype is a small app that either demonstrates only the flow of the UI or demonstrates some minimal functionality. For example, you can create a prototype that contains only the navigation and commands, but doesn't provide any functionality. By making the experience real through software, prototyping enables you to test and validate the flow of your design on devices such as tablets. You can also create prototypes that demonstrate core aspects of the app. For example, we created a prototype that loads one photo and lets you use touch or the mouse to crop that photo. These prototypes allow you to safely explore possibilities before committing changes to your main code base. Although you can prototype during the planning phase of your app, try not to focus too much on writing code. Design the UX you want and then implement that design when it's ready.

Deciding what Windows 8 features to use

Although creativity is key, it's important to provide an experience that's consistent with other Windows Store apps. By doing so, your app will be intuitive to use. We researched the features that the Windows platform provides by looking at MSDN Developer Samples and by talking with others and prototyping. We brainstormed which platform features would best support our app flow and settled on these:

Support for different views and form factors Windows Store apps run on a variety of devices and orientations. We set out with the goal to enable Hilo to run anywhere—from a small tablet device running in either landscape or portrait mode to a large monitor. The XAML **Grid** control can adapt to different displays (screens sizes and pixel density) and orientations because it uses the available screen surface to display the appropriate number of elements, while keeping the same interaction principles. We designed each Hilo page to support snap and fill states.

Tiles An app that feels fresh and engaging keeps your users coming back. We felt that the live tile was a critical component to keep the app feeling fresh and personal to the user.

Tip **Toast notifications** and **Secondary tiles** are also great ways to engage your users and bring them back to your app. Although Hilo doesn't use toast notifications or secondary tiles, you can learn about them by reading Guidelines and checklist for secondary tiles and Guidelines and checklist for toast notifications.

Touch first Touch is more than simply an alternative to using the mouse because it can add a personal connection between the user and the app. We wanted to make touch a first-class part of the app. For example, touch is a very natural way to enable users to crop and rotate their photos. We also realized that Semantic Zoom would be a great way to help users navigate large sets of pictures in a single view. On the image browser page, when you zoom out, the view changes to a calendar-based view. Users can then quickly browse through their photo collection. For more info on how we implemented touch features, see Using touch.

Other features to consider

Although Hilo doesn't use them, here are some other features to include in your planning:

App contracts Contracts declare how your app interacts with other apps and with Windows. For example, the **Share** contract lets users share different elements like text, images or other media across social networks and services. For more info, see App contracts and extensions.

Animations Animations can help make your app engaging or provide visual cues. For more info, see Make animations smooth.

Deciding how to sell the app

Although we don't sell the Hilo app through the Windows Store, we considered the best ways to enable the app to make money. Regardless whether customers pay for the app before they use it, whether there is a trial version, or whether the app includes ads, we felt that making the app world-ready would significantly increase the number of customers who can use the app and have a great experience. Being world-ready not only means supporting localized strings, but it is also being aware of how customers from various cultures use software. (For example, the direction in which they read text.) World-readiness is discussed in greater detail in Making your app world ready in this guide.

For more info about making money with your app, see Plan for monetization.

Making a good first impression

We looked back at our "great at" statement—*Hilo is great at helping people relive their memories*—and realized that connecting the users' personal photos to the app experience was key to helping them relive their memories.

Having a dynamic **live tile and tile notifications** came to mind as the first area to focus on and plan. When the user leaves the app, we want to maintain a good impression by regularly updating the live tile with random recent pictures.

The **splash screen** is important because it expresses your app's personality. Although we want to show the splash screen for as little time as possible, it gives the user a memorable impression of the feel of your app. We chose a splash screen image that fits the Hilo branding and that ties in to UX as a whole.

Note Image assets, including the Hilo logo, are placeholders and meant for training purposes only. They cannot be used as a trademark or for other commercial purposes.

We chose a hub as our **home page** because it immediately shows the primary purpose of the app, to browse and view photos. Because we planned for a great initial experience, users will be sure to explore the rest of the app.

Prototyping and validating the design

To help solidify our planning and feel confident to move on to actual development, we presented our wireframes and prototypes to customers. We also cross-checked our planning against the Index UX guidelines for Windows Store apps to ensure we complied with the official guidelines. Doing so early saved us from making too many core design changes later. Of course, as you develop your app, you will find places where you need to rethink your design. But it is best to harden your design early so that you can move on to developing your next great app.

Writing modern C++ code in Hilo (Windows Store apps using C++ and XAML)

Summary

- Use modern, standard C++ in your app.
- Use Visual C++ component extensions (C++/CX) for interop with XAML and to create Windows Runtime libraries, but you don't need to use C++/CX for the internal implementation of your app.

Here are some tips and coding guidelines for creating Windows Store apps using C++ and XAML, including how to adapt to asynchronous programming using the Parallel Patterns Library (PPL). They arose from questions we ourselves asked as we started developing Hilo C++.

Windows Store apps using C++ combine the best features of C++11, the modern C++ coding style, and C++/CX. If you're new to C++11, consider reading C++11 Features (Modern C++) and Welcome Back to C++ (Modern C++) first.

If you're new to C++/CX, read Visual C++ Language Reference (C++/CX).

You will learn

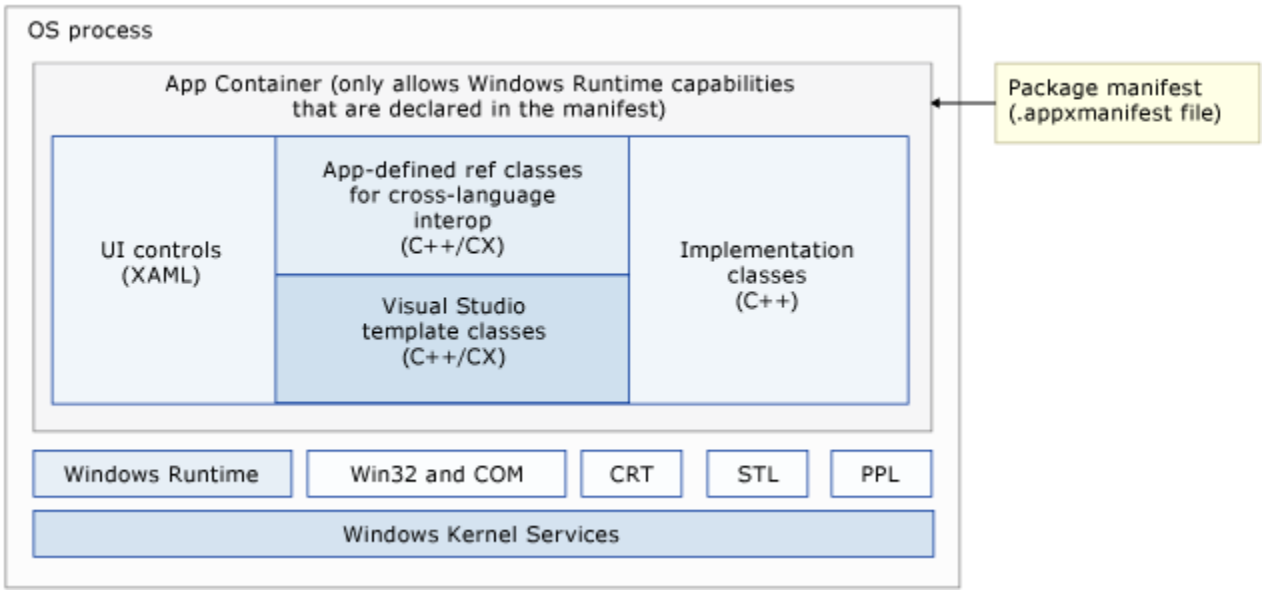
- When to use standard C++ types and libraries in Windows Store apps.
- Best practices for using C++/CX.
- How to port existing C++ libraries for use by Windows Store apps.
- Recommended debugging techniques using Visual Studio.

Download

Download Hilo sample

Understanding the app's environment

Windows Store apps, such as Hilo, include a distinctive visual design and often feature touch-based interaction. In addition, they run in an environment that gives the user more control over what the app is allowed to do compared to desktop and console apps. The extra control helps make the app secure and easier for the user to manage.



These features have implications for the way you implement the app. If you're using C++, here's what establishes your app's static and run-time environment.

- The package manifest
- C++ standard libraries
- Windows Runtime libraries
- Microsoft Win32 and Component Object Model (COM) API
- Parallel Patterns Library (PPL)
- XAML
- Visual Studio project templates
- C++ language extensions for interop
- The C++ compiler and linker
- Certification process of the Windows Store
- Deployment

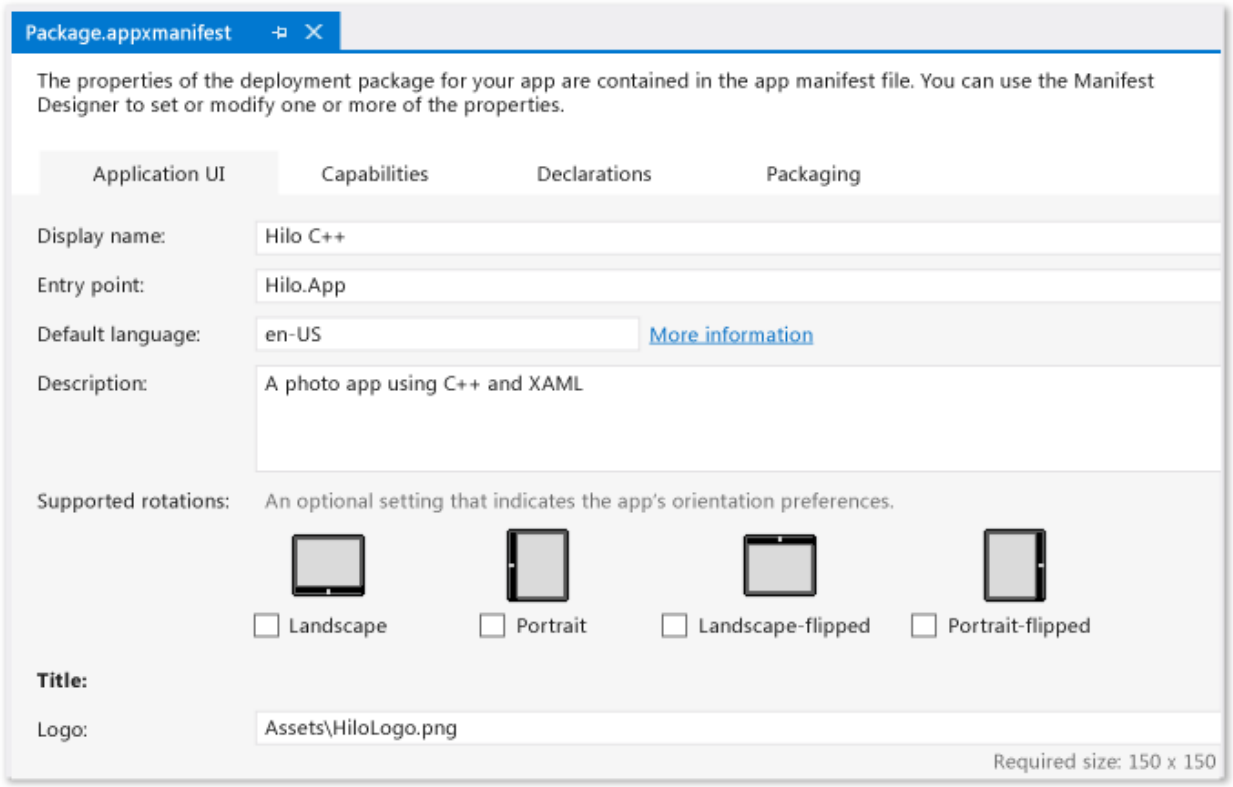
These components and tools determine what platform capabilities are available to your app and how you access these capabilities. If you're new to Windows Store apps, review the overviews of these components and tools to learn how they contribute to your app. You'll see how each component and tool is used in Hilo as you read through this guide.

Note

Another useful general starting point for C++ programmers is Roadmap for Windows Store apps using C++.

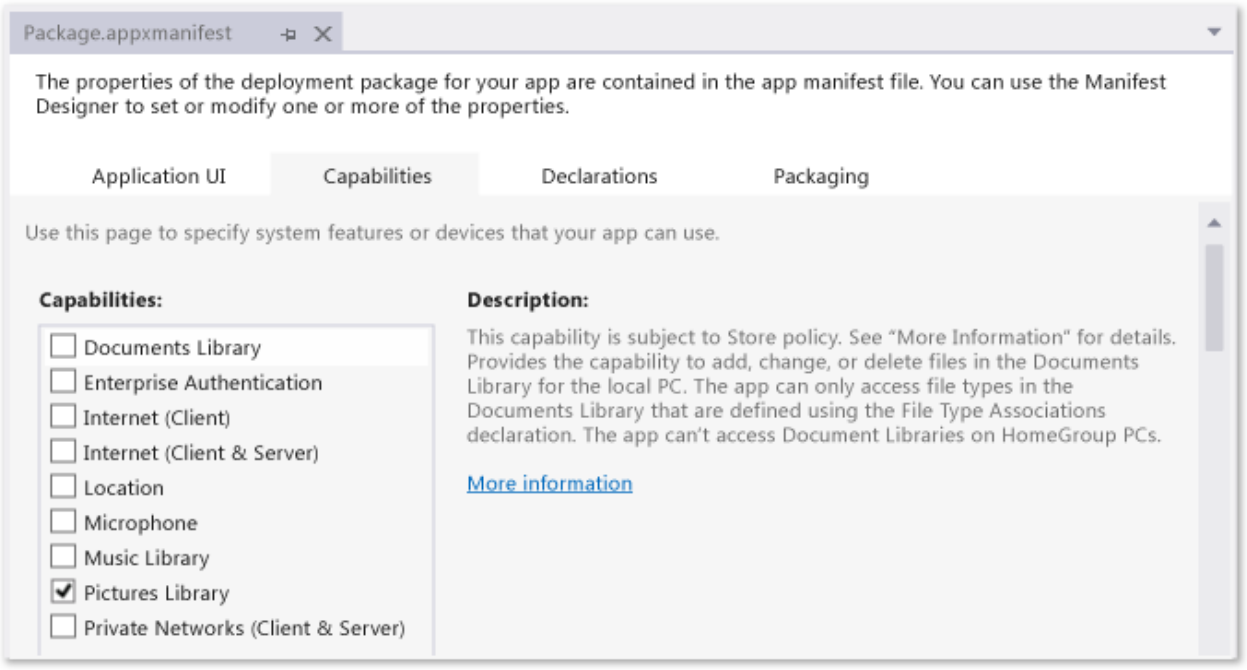
The package manifest

Visual Studio creates a project file named Package.appxmanifest to record settings that affect how the app is deployed and how it runs. The file is known as the package manifest. Visual Studio lets you edit the package manifest using a visual tool that is called the Manifest Designer.



Unlike desktop and console apps, Windows Store apps must declare in advance the environment capabilities that they intend to use, such as accessing protected system resources or user data.

You must declare all capabilities in the package manifest before your app can use them. You can use the **Capabilities** tab in the Manifest Designer to do this. For example, the Hilo manifest includes the ability to access the user's picture library.



Here's the source code of Hilo's package manifest.

Package.appxmanifest

XML

```
<Package xmlns="http://schemas.microsoft.com/appx/2010/manifest">
  <Identity Name="Microsoft.Hilo.Sample.CPP" Publisher="CN=Microsoft Corporation,
O=Microsoft Corporation, L=Redmond, S=Washington, C=US" Version="1.0.0.0" />
  <Properties>
    <DisplayName>ms-resource:DisplayNameProperty</DisplayName>
    <PublisherDisplayName>ms-
resource:PublisherDisplayNameProperty</PublisherDisplayName>
    <Logo>Assets\HiloStoreLogo.png</Logo>
  </Properties>
  <Prerequisites>
    <OSMinVersion>6.2.1</OSMinVersion>
    <OSMaxVersionTested>6.2.1</OSMaxVersionTested>
  </Prerequisites>
  <Resources>
    <Resource Language="x-generate" />
  </Resources>
  <Applications>
    <Application Id="App" Executable="$targetnametoken$.exe" EntryPoint="Hilo.App">
      <VisualElements DisplayName="ms-resource:DisplayName"
Logo="Assets\HiloLogo.png" SmallLogo="Assets\HiloSmallLogo.png" Description="ms-
resource:Description" ForegroundText="light" BackgroundColor="#292929">
        <DefaultTile ShowName="allLogos" WideLogo="Assets\HiloWideLogo.png"
```

```
ShortName="ms-resource:ShortName" />
        <SplashScreen Image="Assets\HiloSplash.png" BackgroundColor="#292929" />
      </VisualElements>
    </Application>
  </Applications>
  <Capabilities>
    <Capability Name="picturesLibrary" />
  </Capabilities>
</Package>
```

Windows Store apps run within a special operating system environment. This environment allows the app to only use features of the Windows Runtime that the package manifest has declared as requested capabilities. To achieve this, the system restricts the functions, data types, and devices that the app can use.

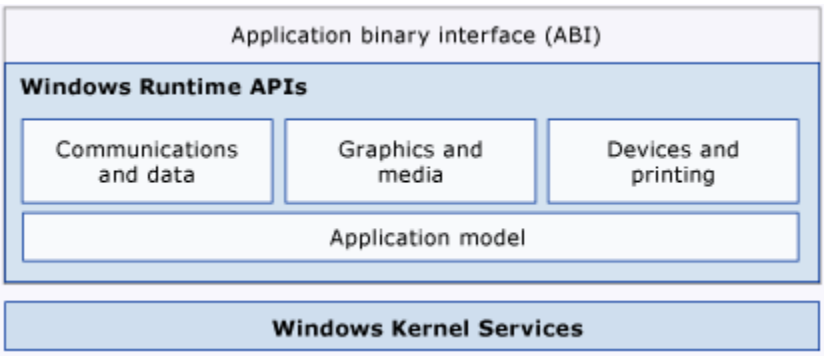
Note The package manifest only restricts calls to the Windows Runtime API. If you use Win32 or the COM API, you'll need to run tools that make sure you're using the API subset that is allowed for apps on the Windows Store. Visual Studio helps you adhere to the correct API subset by filtering functions in the Object Browser.

For a walkthrough of Hilo's use of the manifest, see Testing and deploying the app in this guide. See App capability declarations (Windows Store apps) for more about setting capabilities and Manifest Designer to learn how to edit the package manifest in Visual Studio. For complete documentation of the manifest's XML schema, see Package Manifest.

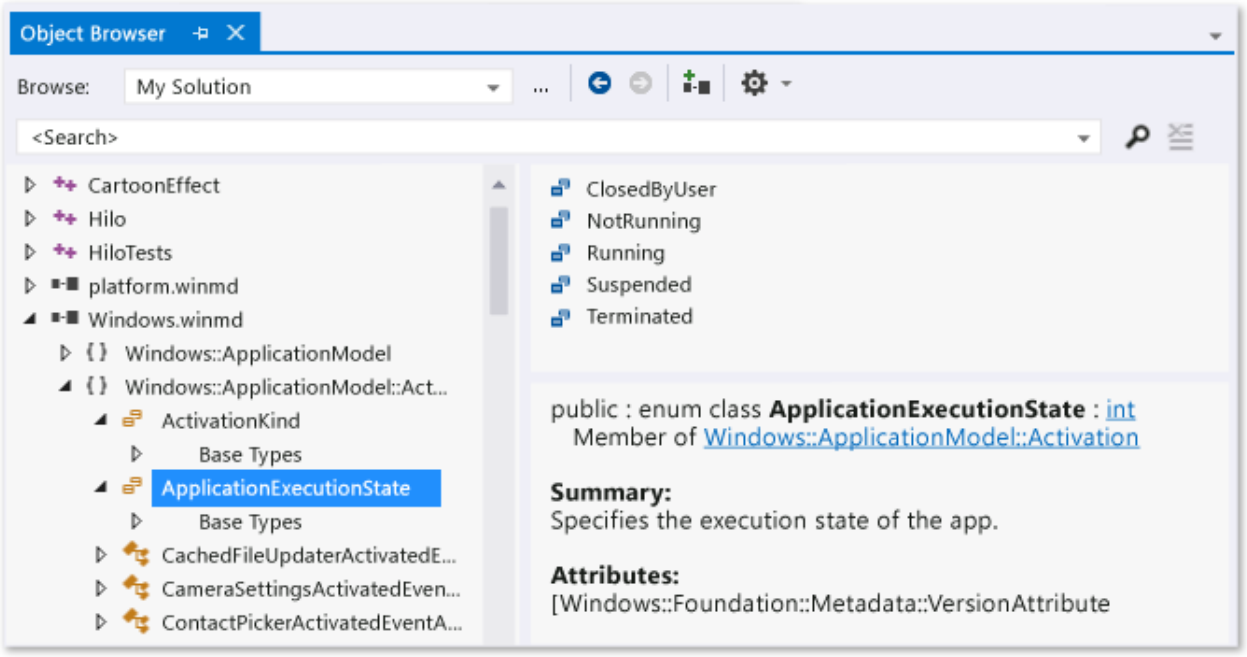
C++ standard libraries

The Visual C++ compiler supports the C++11 standard that lets you use a modern coding style. Your Windows Store app can use the C run-time library (CRT) and the Standard Template Library (STL). There are a few restrictions, such as no access to console I/O. (See Porting existing C++ code in this guide.)

Windows Runtime libraries



The Windows Runtime is a programming interface that you can use to create Windows Store apps. Windows Runtime supports the distinctive visual style and touch-based interaction model of Windows Store apps as well as access to network, disks, devices, and printing. The data types and functions in these libraries use the Windows and Platform namespaces. The Object Browser in Visual Studio lets you examine what types are available.



At the lowest level, the Windows Runtime consists of an *application binary interface* (ABI). The ABI is a binary contract that makes Windows Runtime APIs accessible to multiple programming languages such as JavaScript, the .NET languages, and Microsoft Visual C++. For more info about the Windows Runtime API, see Windows API reference for Windows Store apps.

The structure of a Windows Store app differs from that of a traditional desktop app using the Windows API (Win32). Instead of working with handle types such as **HWND** and functions such as **CreateWindow**, the Windows Runtime provides interfaces such as **Windows::UI::Core::ICoreWindow** so that you can develop Windows Store apps in a more modern, object-oriented way.

Note Part of the richness of the Windows Runtime programming model comes from an extended type system and additional language capabilities. The environment provides properties, delegates, events, interfaces and attributes. See Type system (C++/CX) for an overview.

Win32 and COM API

Windows Store apps can use a subset of the Win32 and COM API. Although the Windows Runtime provides a rich set of functionality, you can continue to use a subset of Win32 and COM to support key

scenarios for Windows Store apps that aren't already covered by the Windows Runtime. For example, **IXMLHTTPRequest2**, a COM interface, is the recommended way to connect to HTTP servers in a Windows Store app using C++. See Win32 and COM API for more info about using Win32 and COM in your Windows Store app. See Porting existing C++ code in this guide for an overview of what's included.

Hilo doesn't call the Win32 API directly. However, it does use COM to work with image pixel data.

Parallel Patterns Library (PPL)

Your app should use the Parallel Pattern Library (PPL) for asynchronous and parallel programming. You use PPL tasks and agents of the Asynchronous Agents Library in places where you might otherwise have used a thread. See Adapting to async programming and Using parallel programming and background tasks in this guide to see how Hilo uses PPL.

XAML

Windows Store apps using Extensible Application Markup Language, or XAML, use a declarative approach for defining the visual presentation of the app. The design and structure of the UX are encoded in XAML, an XML-based markup language. You can use a visual design tool, such as the Visual Studio designer or Microsoft Expression Blend, to define UI elements, or you can write XAML expressions yourself, or both. You use XAML data binding expressions to connect UI elements to data sources in your app.

XAML is the design language for frameworks such as Microsoft Silverlight and Windows Presentation Foundation (WPF). If you're familiar with using XAML with these frameworks or with Expression Blend, you'll be able to continue using your skills when you create Windows Store apps. For more info about XAML as it relates to Windows Store apps, see XAML overview.

Data binding is a convenient way for text boxes, buttons, data grids, and other UI elements to connect to app logic. The Windows Runtime invokes methods and properties of the data sources as needed to supply the UI with information as the app runs. Data binding is also used in cases where the data being passed is a command, such as a request to perform an operation like opening a file. In this case, data binding decouples UI elements that invoke commands from the code that executes the command action. Nearly every object and property of the framework can be bound in XAML.

Visual Studio project templates

We recommend that you use the built-in Visual Studio project templates because they define the compiler settings needed to run a Windows Store app. The project templates also provide code that implements basic functionality, saving you time. Visual Studio puts some files in a solution folder named **Common**, which you shouldn't modify. Visual Studio also creates an App.xaml.cpp file that you can customize with app-specific logic. This file includes the main entry point to the app, the

App::InitializeComponent method. Although the Visual Studio templates save time, they aren't required. You can also modify existing static and dynamic libraries for use in your app. For more info, see Porting existing C++ code in this guide and Building apps and libraries (C++/CX).

For more about Visual Studio's project templates, see Templates to speed up your app development (Windows Store apps using C#/VB/C++ and XAML).

C++ language extensions for interop

The new user interface framework for Windows Store apps uses XAML to interact with native C++. The interop between C++ and XAML, or other languages such as JavaScript and C#, is direct, without calls to an intermediary translation layer. Interop uses a convention for the binary format of objects and function calls that is similar to COM-based programming. The programming environment for Windows Store apps includes C++/CX, which provides C++ language extensions that make interop easy to code. The extensions are only intended to be used with code that deals with interop. The rest of your app should use standards-compliant C++. For more info see Tips for using C++/CX as an interop layer in this guide.

Note You don't have to use the C++/CX for interop. You can also achieve interop with WRL.

The C++ compiler and linker

The **/ZW** compiler option enables C++ source files to use features of the Windows Runtime. It also enables the **__cplusplus_winrt** preprocessor directive that you'll see in some system header files. Your code gets declarations of Windows Runtime types from metadata (.winmd) files instead of from header files. You reference .winmd files with the **#using** directive, or the **/FU** compiler option. Visual Studio configures these options for you if you use one of the C++ project templates for Windows Store apps.

When you link the app, you need to provide two linker options: **/WINMD** and **/APPCONTAINER**. Visual Studio configures these options for you if you use one of the C++ project templates for Windows Store apps.

Certification process of the Windows Store

Publishing to the Windows Store makes your app available for purchase or free download. Publishing to the store is optional. Apps in the store must undergo a certification process that includes an automated structural check. For a description of Hilo's certification experience, see Testing and deploying the app in this guide.

Deployment

Windows Store apps use a simpler installation model than desktop apps. In a Windows Store app, all libraries and resources, other than those provided by the system, are located in the app's installation directory or subdirectories. The installation process for a Windows Store app cannot write to the system registry or define environment variables. In addition, the user can limit the capabilities of the app they're installing. The package manifest lists all the files that are deployed. For more information, see The package manifest.

Using C++11, standard C++ libraries, and a modern coding style

When possible, use C++11 and the standard C++ libraries (for example, the STL and CRT) for your core app logic and the C++/CX syntax only at the boundary where you interact with the Windows Runtime. For example, it's best if you use these techniques:

- Lambda expressions
- Stack semantics, smart pointers, and RAII
- Automatic type deduction
- Range-based for loops
- Standard algorithms and containers
- Free-form iterators
- The pimpl idiom
- Exception handling

Lambda expressions

Use lambda expressions to define the work that's performed by PPL tasks and by STL algorithms. For more info, see Async programming patterns for Windows Store apps using C++ and XAML in this guide.

Stack semantics, smart pointers, and RAII

Use stack semantics, smart pointers, and Resource Acquisition is Initialization (RAII) to automatically control object lifetime and ensure that resources are freed when the current function returns or throws an exception. For more info, see Tips for managing memory in this guide.

Automatic type deduction

Use automatic type deduction to make code easier to read and faster to write. The **auto** and **decltype** keywords direct the compiler to deduce the type of a declared variable from the type of the specified expression. For example, you can use **auto** when you work with STL iterators, whose names can be tedious to type and don't add clarity to your code. Hilo makes extensive use of **auto** when it uses the

concurrency::create_task and **std::make_shared** functions to reduce the need to declare the template type parameter.

C++: ImageBase.cpp

```
auto filePickerTask = create_task(savePicker->PickSaveFileAsync());
```

C++: ThumbnailGenerator.cpp

```
auto decoder = make_shared<BitmapDecoder^>(nullptr);
auto pixelProvider = make_shared<PixelDataProvider^>(nullptr);
```

Note Use the **auto** keyword when readers of your code can understand the type from the context, or when you want to abstract the type. The motivation is readability.

Range-based for loops

Use range-based **for** loops to work with collections of data. Range-based **for** loops have a more succinct syntax than **for** loops and the **std::for_each** algorithm because they don't require you to use iterators or capture clauses. Here's an example:

C++: FileAllPhotosQuery.cpp

```
auto photos = ref new Vector<IPhoto^>();
for (auto file : files)
{
    auto photo = ref new Photo(file, ref new NullPhotoGroup(), policy);
    photos->Append(photo);
}
```

For more info, see Algorithms (Modern C++).

Standard algorithms and containers

Use standard algorithms and containers, such as **std::vector**, **std::for_each**, **std::find_if**, and **std::transform** to take advantage of C++ features. Because Windows RT types are language-neutral, Hilo uses types such as **std::wstring** and **std::wstringstream** to work with strings internally and **Platform::String** only when it interacts with the Windows Runtime. In this example, the returned **Platform::String** is passed to the Windows Runtime.

C++: CalendarExtensions.cpp

```
wstringstream dateRange;
dateRange << L"System.ItemDate:" ;
```

```
cal->Day = cal->FirstDayInThisMonth;
cal->Period = cal->FirstPeriodInThisDay;
cal->Hour = cal->FirstHourInThisPeriod;
cal->Minute = cal->FirstMinuteInThisHour;
cal->Second = cal->FirstSecondInThisMinute;
cal->Nanosecond = 0;
dateRange << GetAqsFormattedDate(cal->GetDateTime());

dateRange << "..";

cal->Day = cal->LastDayInThisMonth;
cal->Period = cal->LastPeriodInThisDay;
cal->Hour = cal->LastHourInThisPeriod;
cal->Minute = cal->LastMinuteInThisHour;
cal->Second = cal->LastSecondInThisMinute;
cal->Nanosecond = 999999;
dateRange << GetAqsFormattedDate(cal->GetDateTime());

return ref new String(dateRange.str().c_str());
```

By using standard functionality, you can write code that's more portable and takes advantage of modern C++ features such as move semantics. (For more info about move semantics, see Rvalue Reference Declarator: &&.)

The same pattern applies to standard containers, such as **std::vector**. Using standard containers enables you to take advantage of C++ features such as move semantics and the ability to work with memory more directly. For example, you might perform internal processing on a **std::vector** object and then need to pass a **Windows::Foundation::Collections::IVector** object to the Windows Runtime. The **Platform::Collections::Vector** class, which is the C++ implementation of **IVector**, has an overloaded constructor that takes an rvalue reference to a **std::vector**. To call the overloaded constructor, use the **std::move** function or directly pass the result of a function that returns **std::vector**. For an example that shows this pattern, see Collections (C++/CX).

Note You don't have to use move semantics. The **Platform::Collections::Vector** class includes a standard copy constructor that takes a **std::vector** object as its argument. In other words, you can keep your old vector data and create a new **Platform::Collections::Vector** instance with a copy of the original data.

As another example, Hilo uses **std::iota** and **std::random_shuffle** to choose random photos (more precisely, indices to an array of photos). This example uses **std::iota** to create a sequence of array indices and **std::random_shuffle** to randomly rearrange the sequence.

C++: RandomPhotoSelector.cpp

```
vector<unsigned int> RandomPhotoSelector::CreateRandomizedVector(unsigned int
vectorSize, unsigned int sampleSize)
{
    // Seed the rand() function, which is used by random_shuffle.
    srand(static_cast<unsigned int>(time(nullptr)));

    // The resulting set of random numbers.
    vector<unsigned int> result(vectorSize);

    // Fill with [0..vectorSize).
    iota(begin(result), end(result), 0);

    // Shuffle the elements.
    random_shuffle(begin(result), end(result));

    // Trim the list to the first sampleSize elements if the collection size is
greater than the sample size.
    if (vectorSize > sampleSize)
    {
        result.resize(sampleSize);
    }

    return result;
}
```

For more info about standard algorithms, see Algorithms (Modern C++)

Free-form iterators

Use the **std::begin** and **std::end** functions to work with ranges. Use these functions, instead of member functions such as **.begin()** and **.end()**, to write more flexible code. For example, you can write a generic algorithm that works with both STL types such as **std::vector**, **std::array**, and **std::list**, which provide **begin** and **ends** member functions, and also Windows Runtime collection types such as **IVector**, which use a different technique to iterate over values. The **std::begin** and **std::end** functions can be overloaded to accommodate different programming styles and also enable you to add iteration to data structures that you cannot alter.

The pimpl idiom

Use the pimpl idiom to hide implementation, minimize coupling, and separate interfaces. Pimpl (short for "pointer to implementation") involves adding implementation details to a .cpp file and a smart pointer to that implementation in the **private** section of the class declaration in the .h file. Doing so can also shorten compile times. Pimpl, when used together with copy construction and move semantics,

also enables you to pass objects by value, which helps eliminate the need to worry about object lifetimes.

You can also take advantage of the pimpl idiom when you use predefined libraries. Pimpl is often related to rvalue references. For example, when you create task-based continuations, you pass **concurrency::task** objects by value and not by reference.

C++: ImageBrowserViewModel.cpp

```
void ImageBrowserViewModel::StartMonthQuery(int queryId, cancellation_token token)
{
    m_runningMonthQuery = true;
    OnPropertyChanged("InProgress");
    m_photoCache->Clear();
    run_async_non_interactive([this, queryId, token]()
    {
        // if query is obsolete, don't run it.
        if (queryId != m_currentQueryId) return;

        m_repository->GetMonthGroupedPhotosWithCacheAsync(m_photoCache, token)
            .then([this, queryId](task<IVectorView<IPhotoGroup^>^> priorTask)
        {
            assert(IsMainThread());
            if (queryId != m_currentQueryId)
            {
                // Query is obsolete. Propagate exception and quit.
                priorTask.get();
                return;
            }

            m_runningMonthQuery = false;
            OnPropertyChanged("InProgress");
            if (!m_runningYearQuery)
            {
                FinishMonthAndYearQueries();
            }
            try
            {
                // Update display with results.
                m_monthGroups->Clear();
                for (auto group : priorTask.get())
                {
                    m_monthGroups->Append(group);
                }
                OnPropertyChanged("MonthGroups");
            }
            // On exception (including cancellation), remove any partially
            // computed results and rethrow.
```

```
            catch (...)
            {
                m_monthGroups = ref new Vector<IPhotoGroup^>();
                throw;
            }
        },
        task_continuation_context::use_current())
            .then(ObserveException<void>(m_exceptionPolicy));
    });
}
```

Passing by value guarantees that the object is valid and eliminates the need to worry about object lifetime. Additionally, passing by value isn't expensive in this case because the **task** class defines a pointer to the actual implementation as its only data member and copies or transfers that pointer in the copy and move constructors.

For more info, see Pimpl For Compile-Time Encapsulation (Modern C++).

Exception handling

Use exception handling to deal with errors. Exception handling has two main advantages over logging or error codes (such as **HRESULT** values):

- It can make code easier to read and maintain.
- It's a more efficient way to propagate an error to a function that can handle that error. The use of error codes typically requires each function to explicitly propagate errors.
- It can make your app more robust because you can't ignore an exception as you might an **HRESULT** or **GetLastError** status.

You can also configure the Visual Studio debugger to break when an exception occurs so that you can stop immediately at the location and context of the error. The Windows Runtime also uses exception handling extensively. Therefore, by using exception handling in your code, you can combine all error handling into one model.

Important Catch only the exceptions that you can safely handle and recover from. Otherwise, don't catch the exception and allow the app to terminate. Don't use **catch(...) {...}** unless you rethrow the exception from the **catch** block.

Here's an example from Hilo that shows the modern C++ coding style C++11 and standard libraries that copy **StorageFile** objects from a **std::vector** object to a **Vector** object so that the collection can be passed to the Windows Runtime. This example uses a lambda expression, automatic type deduction, **std::begin** and **std::end** iterators, and a range-based **for** loop.

C++: ThumbnailGenerator.cpp

```
return when_all(begin(thumbnailTasks), end(thumbnailTasks)).then(
    [](vector<StorageFile^> files)
{
    auto result = ref new Vector<StorageFile^>();
    for (auto file : files)
    {
        if (file != nullptr)
        {
            result->Append(file);
        }
    }

    return result;
});
```

For more info about modern C++ programming, see Welcome Back to C++ (Modern C++).

Adapting to async programming

With async programming you call a function that starts a long-running operation but returns immediately without waiting for the work to be finished. Async operations help improve the responsiveness of the user experience. You'll find many more asynchronous operations in the Windows Runtime than in earlier frameworks.

You can tell if a Windows Runtime function is asynchronous from its name. The names of asynchronous functions end in "Async" such as **ReadTextAsync**. Hilo also follows this naming convention.

The Windows Runtime provides its own data types for interacting with asynchronous operations. These data types are C++/CX interfaces that derive from the **Windows::Foundation::IAsyncInfo** interface.

Each programming language provides native support for asynchronous programming and uses **IAsyncInfo**-derived interfaces for interop. In the case of C++, PPL provides the native support for async programming. In general, you should wrap the Windows Runtime's async interface types using PPL tasks when you get them from a call to a Windows Runtime async method. The only time you should expose these interfaces from your own classes is to create a Windows Runtime component for cross-language interop. For example, you use **IAsyncInfo**-derived types in **public** methods and properties of **public ref** classes that you define.

After wrapping an async operation with a PPL task, you can create additional tasks that the system schedules for execution after the prior task completes. The successor tasks are called *continuation tasks* (or *continuations*). You create continuations with the **task::then** method. Here's an example from Hilo that uses async operations to read an image from a file.

Note The declarations of PPL tasks are located in the ppltasks.h header file.

C++: Photo.cpp

```
task<void> Photo::QueryPhotoImageAsync()
{
    auto imageStreamTask = create_task(m_fileInfo->OpenReadAsync());
    return imageStreamTask.then([this](task<IRandomAccessStreamWithContentType^>
priorTask) -> task<void>
    {
        assert(IsMainThread());
        IRandomAccessStreamWithContentType^ imageData = priorTask.get();
        m_image = ref new BitmapImage();
        m_imageFailedEventToken = m_image->ImageFailed::add(ref new
ExceptionRoutedEventHandler(this, &Photo::OnImageFailedToOpen));
        return create_task(m_image->SetSourceAsync(imageData));
    }).then([this](task<void> priorTask) {
        assert(IsMainThread());
        try
        {
            priorTask.get();
        }
        catch (Exception^ e)
        {
           OnImageFailedToOpen(nullptr, nullptr);
        }
    });
}
```

The **m_fileInfo** member variable is a **Windows::Storage::BulkAccess::FileInformation^** reference. Its **OpenReadAsync** method starts an asynchronous operation function to open and read the file that contains the requested image. The call to **OpenReadAsync** returns an object of type **IAsyncOperation<IRandomAccessStreamWithContentType^>^**.

The async invocation starts the operation. The call itself returns very quickly, without waiting for the file open operation to complete. The return value of the call to **OpenReadAsync** represents the running operation that was started. The return type is parameterized by the asynchronous operation's result type, which in this case is IRandomAccessStreamWithContentType^.

Note The **^** (hat) syntax indicates a Windows Runtime reference. If you're unfamiliar with it, see Classes and structures (C++/CX).

The **Photo::QueryPhotoImageAsync** method then calls the **concurrency::create_task** function to produce a new PPL task that becomes the value of the local variable **imageStreamTask**. The type of the **imageStreamTask** variable is **task<IRandomAccessStreamWithContentType^>**. The effect of passing a

Windows Runtime asynchronous object to the **concurrency::create_task** function is to wrap the asynchronous operation with a PPL task. The newly created PPL task finishes when the **OpenReadAsync** operation completes its work and a random access stream is available.

Async operations of the Windows Runtime do not always require their own threads to run. Windows often manages them as overlapped I/O operations using internal data structures that have less overhead than threads. This detail is internal to the operating system.

After a Windows Runtime operation is wrapped with a PPL task, you can use the **task::then** method to create continuations that run after the previous, or *antecedent*, task completes. Continuations are themselves PPL tasks. They make asynchronous programs easier to read and debug.

The **task::then** method is asynchronous, with it returning a new task very quickly. Unlike other async tasks, the task returned by the **task::then** method doesn't begin to run immediately. Instead, PPL delays the start of the task's execution until the result of the antecedent task becomes available. This means that although the **task::get** method is synchronous when applied to the antecedent task, the value is immediately available. Continuation tasks that aren't ready to run don't block any threads, instead they are managed by the PPL internally until the data they need becomes available.

In the **QueryPhotoImageAsync** method shown above, the argument to the **then** method is a lambda expression that's the work function of the newly created task. (A task's work function is the code that gets invoked when the task runs.) The type of the lambda expression's input parameter matches the type of the **imageStreamTask**. The types match because the **imageStreamTask** is passed as an argument to the continuation's work function when the continuation begins to run. You can think of this as a kind of data flow programming. Continuation tasks begin to run when their inputs become available. When they finish running, they pass their results to the next continuation task in the chain.

Note If you're unfamiliar with lambda expressions in C++, see Lambda Expressions in C++.

In Windows Store apps, continuations of tasks that wrap **IAsyncInfo** objects run by default in the thread context that created the continuation. In most cases, the default context will be the app's main thread. This is appropriate for querying or modifying XAML controls, and you can override the default to handle other cases.

Note In Hilo, we found it useful to clarify our understanding of the thread context for our subroutines by using **assert(IsMainThread()** and **assert(IsBackgroundThread())** statements. In the Debug version of Hilo, these statements break into the debugger if the thread being used is other than the one declared in the assertion. The **IsMainThread** and **IsBackgroundThread** functions are in the Hilo source.

This example didn't need any special synchronization code, such as a lock or a critical section, for the update to the **m_image** member variable. This is because all interactions with view model objects occur on the main thread. Using a single thread automatically serializes any potentially conflicting updates. In

UI programming, it's a good idea to use continuations that run in a known thread context instead of other kinds of synchronization.

This code is an example of a *continuation chain* or a *.then ladder* (pronounced dot-then ladder). This pattern appears frequently in apps that use async APIs from the Windows Runtime. See Asynchronous programming in C++ (Windows Store apps) for more info about continuation chains. For tips and guidance about how to use them, along with more examples, see Async programming patterns for Windows Store apps using C++ and XAML in this guide.

Using parallel programming and background tasks

The goal of asynchronous programming is interactive responsiveness. For example, in Hilo we use asynchronous techniques to make sure that the app's main thread remains unblocked and ready to respond to new requests without delay. However, depending on your app's functional requirements, you might need to ensure the responsiveness of the user experience and the overall throughput of the compute-intensive parts of your app.

C++ is particularly well-suited for apps that need to have both a responsive user interface and high performance for compute-intensive tasks. The concurrency features of Visual C++ can meet the needs of both sets of requirements.

You can increase the throughput of the compute-intensive areas of your app by breaking some tasks into smaller pieces that are performed concurrently by multiple cores of your CPU or by the specialized data-parallel hardware of your computer's graphics processor unit (GPU). These techniques and others are the focus of *parallel programming*. We use parallel programming techniques in several areas of Hilo.

For example, one of Hilo's features is a cartoon effect that you can use to stylize an image using simpler colors and shape outlines. Here's what an image looks like before and after applying this effect.

The cartoon effect is computationally intensive. Hilo's implementation uses C++ AMP to calculate the result quickly on the GPU. On systems that don't have a compute-class GPU available, Hilo uses the PPL, which is part of the Concurrency Runtime. Here's how we use the **accelerator::is_emulated** property to determine whether to use the C++ AMP or the PPL algorithm to perform the cartoon effect:

C++: CartoonizeImageViewModel.cpp

```
void CartoonizeImageViewModel::CartoonizeImage(Object^ parameter)
{
    assert(IsMainThread());
    m_cts = cancellation_token_source();
    auto token = m_cts.get_token();

    // Check for hardware acceleration if we haven't already.
    if (!m_checkedForHardwareAcceleration)
    {
        m_checkedForHardwareAcceleration = true;
        accelerator acc;
        m_useHardwareAcceleration = !acc.is_emulated;
    }

    ChangeInProgress(true);
    EvaluateCommands();
    m_initializationTask = m_initializationTask.then([this, token]() -> task<void>
```

```
    {
        // Use the C++ AMP algorithim if the default accelerator is not an emulator
        // (WARP or reference device).
        if (m_useHardwareAcceleration)
        {
            return CartoonizeImageAmpAsync(token);
        }
        // Otherwise, use the PPL to leverage all available CPU cores.
        else
        {
            return CartoonizeImagePPLAsync(token);
        }
    }, task_continuation_context::use_current()).then([this](task<void> priorTask)
    {
        m_initializationTask = create_empty_task();
        ChangeInProgress(false);
        EvaluateCommands();
        priorTask.get();
    },
        task_continuation_context::use_current())
            .then(ObserveException<void>(m_exceptionPolicy));
}
```

Tip We chose to run the algorithm that uses the PPL when the required hardware is not available because we already had the code available that leverages all CPU cores. However, if you do not have fallback code available, you can still use the WARP or reference device to run your C++ AMP code. Profile your C++ AMP code on multiple configurations to help you determine if you need to consider a similar fallback method.

See the CartoonEffect project in the Hilo source files for details on how we implemented these algorithms.

For more info about C++ AMP, see C++ AMP (C++ Accelerated Massive Parallelism).

When you apply parallel programming techniques you need to think in terms of foreground processing (on the main thread) and background processing (on worker threads). PPL tasks help you control which parts of the app run in the main thread and which parts run in the background. Operations that read or modify XAML controls are always invoked on the app's main thread. However, for compute-intensive or I/O-intensive operations that don't modify the user interface, you can take advantage of the computer's parallel processing hardware by using PPL tasks that run on threads from the system's thread pool. You can see the foreground/background pattern in the Crop action. Here's the code:

C++: CropImageViewModel.cpp

```
task<void> CropImageViewModel::CropImageAsync(float64 actualWidth)
```

```
{
    assert(IsMainThread());
    ChangeInProgress(true);

    // Calculate crop values
    float64 scaleFactor = m_image->PixelWidth / actualWidth;
    unsigned int xOffset = safe_cast<unsigned int>((m_cropOverlayLeft - m_left) *
scaleFactor);
    unsigned int yOffset =
        safe_cast<unsigned int>((m_cropOverlayTop - m_top) * scaleFactor);
    unsigned int newWidth =
        safe_cast<unsigned int>(m_cropOverlayWidth * scaleFactor);
    unsigned int newHeight =
        safe_cast<unsigned int>(m_cropOverlayHeight * scaleFactor);

    if (newHeight < MINIMUMBMPSIZE || newWidth < MINIMUMBMPSIZE)
    {
        ChangeInProgress(false);
        m_isCropOverlayVisible = false;
        OnPropertyChanged("IsCropOverlayVisible");
        return create_empty_task();
    }

    m_cropX += xOffset;
    m_cropY += yOffset;

    // Create destination bitmap
    WriteableBitmap^ destImage = ref new WriteableBitmap(newWidth, newHeight);

    // Get pointers to the source and destination pixel data
    byte* pSrcPixels = GetPointerToPixelData(m_image->PixelBuffer);
    byte* pDestPixels = GetPointerToPixelData(destImage->PixelBuffer);
    auto oldWidth = m_image->PixelWidth;

    return create_task([this, xOffset, yOffset,
          newHeight, newWidth, oldWidth, pSrcPixels, pDestPixels] () {
        assert(IsBackgroundThread());
        DoCrop(xOffset, yOffset, newHeight, newWidth, oldWidth, pSrcPixels,
pDestPixels);
    }).then([this, destImage](){
        assert(IsMainThread());

        // Update image on screen
        m_image = destImage;
        OnPropertyChanged("Image");
        ChangeInProgress(false);
    },
    task_continuation_context::use_current())
        .then(ObserveException<void>(m_exceptionPolicy));
```

```
}
```

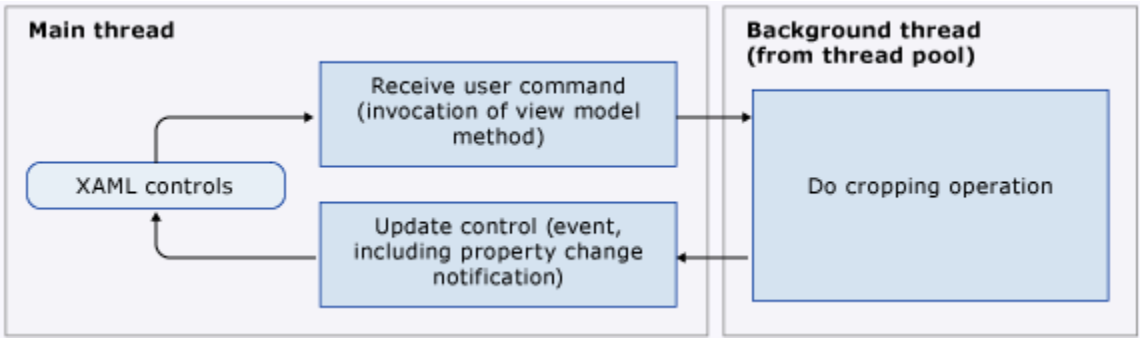
The code shows Hilo's image cropping operation. The UI for the crop operation shows crop handles for the user to manipulate. After specifying the crop region, the user taps the image to generate a preview of the cropped image. The cropping operation's **Grid** control fires a **Tapped** event whose code-behind handler invokes the **CropImageAsync** method in the example.

Because an event handler invokes it, the **CropImageAsync** method runs on the main thread. Internally, it divides its work between the main thread and a background thread in the thread pool. To do this, it uses the **concurrency::create_task** function to schedule the **DoCrop** method in a background thread.

Note While the **DoCrop** method is running in the background, the main thread is free to do other work in parallel. For example, while the **DoCrop** method runs, the main thread continues to animate the progress ring and respond to navigation requests from the user.

After the **DoCrop** method completes, a continuation task begins to run on the main thread to update the XAML controls.

Here's a diagram of the crop operation's use of threads.



The **DoCrop** method runs in a background thread, but it also uses PPL's **parallel_for** function to perform a compute-intensive operation using the computer's multicore hardware. Here's the code.

C++: CropImageViewModel.cpp

```
void CropImageViewModel::DoCrop(uint32_t xOffset, uint32_t yOffset, uint32_t
newHeight, uint32_t newWidth, uint32_t oldWidth, byte* pSrcPixels, byte* pDestPixels)
{
    assert(IsBackgroundThread());
    parallel_for (0u, newHeight, [xOffset, yOffset, newHeight, newWidth, oldWidth,
pDestPixels, pSrcPixels](unsigned int y)
    {
        for (unsigned int x = 0; x < newWidth; x++)
        {
            pDestPixels[(x + y * newWidth) * 4] =
                pSrcPixels[(x +  xOffset + (y + yOffset) * oldWidth) * 4];     // B
            pDestPixels[(x + y * newWidth) * 4 + 1] =
                pSrcPixels[(x +  xOffset + (y + yOffset) * oldWidth) * 4 + 1]; // G
            pDestPixels[(x + y * newWidth) * 4 + 2] =
                pSrcPixels[(x +  xOffset + (y + yOffset) * oldWidth) * 4 + 2]; // R
            pDestPixels[(x + y * newWidth) * 4 + 3] =
                pSrcPixels[(x +  xOffset + (y + yOffset) * oldWidth) * 4 + 3]; // A
        }
    });
}
```

The code is an example of how you can integrate parallel programming techniques into your app. Parallelizing only the outer loop maximizes the benefits of concurrency. If you parallelize the inner loop, you will not receive a gain in performance because the small amount of work that the inner loop performs does not overcome the overhead for parallel processing.

For a modern programming style and best performance, we recommend that you use PPL's parallel algorithms and data types for new code.

For more info, see Parallel Programming in C++.

Tips for using C++/CX as an interop layer

Here are some tips for cross-language interop that we developed during the creation of Hilo. (For complete documentation of the language extensions that are available to a C++ Windows Store app for cross-language interop, see Visual C++ language reference (C++/CX).)

- Be aware of overhead for type conversion.
- Call methods of ref classes from the required thread.
- Mark destructors of public ref classes as virtual.
- Use ref classes only for interop.
- Use techniques that minimize marshaling costs.
- Use the Object Browser to understand your app's .winmd output.

- If C++/CX doesn't meet your needs, consider WRL for low-level interop.
- Don't confuse C++/CX language extensions with C++/CLI.
- Don't try to expose internal types in public ref classes.

Be aware of overhead for type conversion

In order to interact with Windows Runtime features, you sometimes need to create data types from the Platform and Windows namespaces. You should create these types in the most efficient way.

For example, if you create a **Windows::Foundation::Collections::Vector^** reference from a **std::vector** object, the **Vector** constructor may perform a copy. If you allocate a **std::vector** object and know that there are no other references to it, you can create a **Vector** object without copying by calling the **std::move** function on the **std::vector** before passing it to the **Vector** constructor. This works because the **Vector** class provides a move constructor that takes a **std::vector<T>&&** argument.

There are move constructors for the **Platform::Collections** classes **Vector**, **VectorView**, **Map**, and **MapView**. These classes have constructors that take rvalue references to **std::vector** and **std::map** types. Here is an example.

C++: YearGroup.cpp

```
vector<IMonthBlock^> monthBlocks;
monthBlocks.reserve(nMonths);
for (int month = 1; month <= nMonths; month++)
{
    auto monthBlock = ref new MonthBlock(this, month, m_folderQuery, m_repository,
m_exceptionPolicy);
    monthBlocks.push_back(monthBlock);
}
m_months = ref new Vector<IMonthBlock^>(std::move(monthBlocks));
```

The code creates the **m_months** object without copying the **monthBlock** vector.

There are no move constructors for the **Platform::Array** and **Platform::String** class. Instead, you use the **Platform::ArrayReference** class and **Platform::StringReference** class.

In Hilo, we preferred to use the **Vector** and **Map** classes instead of the **Platform::Array** class because of their compatibility with **std::vector** and **std::map**. (We don't use **std::array** because you have to give its size at compile time.)

Call methods of ref classes from the required thread

Some ref classes require their methods, events and properties to be accessed from a specific thread. For example, you must interact with XAML classes from the main thread. If you create a new class that derives from an existing Windows Runtime class, you inherit the context requirements of the base class.

Be careful to respect the threading model of the objects you use.

Reference counts can be decremented as a side effect of operations that occur in any thread. For this reason, you should make sure that destructors for ref classes that you implement can be called from any thread. You must not invoke methods or properties in your destructor that require a specific thread context.

For example, some Windows Runtime classes require you to unregister event handlers in the main thread. Here is a code example of a thread-safe destructor that does this.

C++: ImageView.cpp

```
ImageView::~ImageView()
{
    if (nullptr != PhotosFilmStripGridView)
    {
        // Remove the event handler on the UI thread because GridView methods
        // must be called on the UI thread.
        auto photosFilmStripGridView = PhotosFilmStripGridView;
        auto filmStripLoadedToken = m_filmStripLoadedToken;
        run_async_non_interactive([photosFilmStripGridView, filmStripLoadedToken]()
        {
            photosFilmStripGridView->Loaded::remove(filmStripLoadedToken);
        });
    }
}
```

The **run_async_non_interactive** function is a utility function that is defined in Hilo. It dispatches a function object to the main thread, allowing UI interactions by the user to have higher priority.

Mark destructors of public ref classes as virtual

Destructors of public **ref** classes must be declared **virtual**.

Use ref classes only for interop

You only have to use **^** and **ref new** when you create Windows Runtime objects or create Windows Runtime components. You can use the standard C++ syntax when you write core application code that doesn't use the Windows Runtime.

Hilo uses **^** and **std::shared_ptr** to manage heap-allocated objects and minimize memory leaks. We recommend that you use **^** to manage the lifetime of Windows Runtime variables, **ComPtr** to manage the lifetime of COM variables (such as when you use DirectX), and **std::shared_ptr** or **std::unique_ptr** to manage the lifetime of all other heap-allocated C++ objects.

We recommend that you declare all **ref** classes as public because they're only intended for interop. If you have **private**, **protected**, or **internal ref** classes, it's an indication that you're attempting to use **ref** classes for implementation purposes and not interop across the Abstract Binary Interface (ABI).

Use techniques that minimize marshaling costs

C++/CX is designed for language interop. When you call functions across the ABI, you sometimes incur overhead due to the cost of marshaling (copying) data. Because the XAML UI framework is written in C++, you don't incur marshaling overhead when you interoperate with XAML from a C++ app. If you implement a component in C++ that is called from a language other than C++ or XAML, your app would incur some cost for marshaling.

See Threading and Marshaling (C++/CX) for ways to specify the threading and marshaling behavior of components that you create. For more info about marshaling overhead with other languages, see Keep your app fast when you use interop (Windows Store apps using C#/VB/C++ and XAML).

Use the Object Browser to understand your app's .winmd output

When you build your app, the compiler creates a .winmd file that contains metadata for all the public **ref** types that your app defines. Components such as XAML use the .winmd file to invoke methods of your app's data types across the ABI.

It's often helpful to examine what's in the generated .winmd file. To see this, you can use the Visual Studio **Object Browser**. From the **Object Browser**, navigate to the .winmd file in your project's Debug directory and open it. You'll be able to see all the types that your app exposes to XAML.

Note Be aware of what public **ref** types you're including in your app's .winmd file.

If C++/CX doesn't meet your needs, consider WRL for low-level interop

The language extensions of C++/CX save time, but you don’t have to use them. You can get lower-level access to cross-language interop from standard C++ if you use the Windows Runtime C++ Template Library (WRL). WRL uses conventions that will be familiar to COM programmers.

WRL is a compiler-agnostic way to create and consume Windows Runtime APIs. You can use the WRL instead of the C++/CX syntax. It enables you to optimize your code for performance or for specific scenarios. It also supports app development methodologies that don't use exceptions. For more info, see Windows Runtime C++ Template Library.

In Hilo, we found that C++/CX had the features and performance we needed. We only used the WRL to access a COM interface that enabled us to read pixel data from an image. In general, WRL is a good candidate when you have a COM object that you want to port to be a Windows Runtime object, since WRL roots are in ATL.

Don't confuse C++/CX language extensions with C++/CLI

The syntax of C++/CX language extensions and C++/CLI are similar, but there are two very different execution models. Here's how to think about this.

In order to call Windows Runtime APIs from JavaScript and .NET, those languages require *projections* that are specific to each language environment. When you call a Windows Runtime API from JavaScript or .NET, you're invoking the projection, which in turn calls the underlying **ABI** function. Although you can call the **ABI** functions directly from standard C++, Microsoft provides projections for C++ as well, because they make it much simpler to consume the Windows Runtime APIs, while still maintaining high performance.

Microsoft also provides language extensions to Visual C++ that specifically support the Windows Runtime projections. Many of these language extensions resemble the syntax for the C++/CLI language. However, instead of targeting the common language runtime (CLR), C++ apps use this syntax to generate native code that's compatible with the binary format requirements of the ABI.

Because there's no runtime to manage memory, the system deletes C++/CX objects based on reference counting, in a manner that's similar to **std::shared_ptr**. The handle-to-object operator, or *hat* (^), is an important part of the new syntax because it enables reference counting. Instead of directly calling methods such as **AddRef** and **Release** to manage the lifetime of a Windows Runtime object, the runtime does this for you. It deletes the object when no other component references it, for example, when it leaves scope or you set all references to **nullptr**.

Another important part of using Visual C++ to create Windows Store apps is the **ref new** keyword. Use **ref new** instead of **new** to create reference-counted Windows Runtime objects. For more info, see Type System (C++/CX).

Don't try to expose internal types in public ref classes

Public **ref** classes produce metadata that is exposed in the app's .winmd file. The metadata describes each of the types and the members of each type. In order for the metadata to be complete, every member of a public type must itself be a publicly visible type. As a result, there's a requirement that you can't expose internal types as **public** members of a public **ref** class.

The requirement of metadata can affect the design of your app. In general, try to partition your types so that your public **ref** types are used exclusively for interop and not for other purposes in your app. If you're not careful, it's possible that you'll see an unintended proliferation of public **ref** classes in your app.

Tips for managing memory

Because you don't typically close Windows Store apps and because Windows Store apps run on tablets of varying hardware capabilities, it's important to be aware of the amount of memory your app uses. It's especially important to prevent memory leaks by not allowing objects to remain in memory that are not accessible by code. Also consider the lifetime of every object to ensure you don't keep it in memory longer than it needs to be. For efficient memory management, we recommend that you:

- Use smart pointers.
- Use stack semantics and the RAII pattern.
- Don't keep objects around longer than you need.
- Avoid circular references.

Use smart pointers

Use smart pointers to help ensure that programs are free of memory and resource leaks and are exception-safe. In Windows Store apps, use handle-to-object, **^** (pronounced "hat"), to manage the lifetime of Windows Runtime variables, **Microsoft::WRL::ComPtr** to manage the lifetime of COM variables, (such as when you use DirectX), and **std::shared_ptr** or **std::unique_ptr** to manage the lifetime of all other heap-allocated C++ objects.

It's easy to remember to use **^** because when you call **ref new**, the compiler generates code to allocate a Windows Runtime object and then returns a handle to that object. Windows Runtime methods that don't return value types also return handles. What's important to focus on is the use of standard pointer types and COM objects to eliminate memory and resource leaks.

One way that Hilo uses **std::shared_ptr** is to allow for one task in a continuation chain to write to a variable and another task to read from it. See Assembling the outputs of multiple continuations in this guide for more info.

In Hilo, we needed to directly access pixel data to crop images and to apply the cartoon effect to images. To do so, we needed to convert an **IBuffer** object to its underlying COM interface, **IBufferByteAccess**. We used the **ComPtr** class to manage the pointers so that calls to **AddRef** and **Release** were not necessary. The **ComPtr** class also provides the **As** method, which you can think of as a shortcut for calling **QueryInterface** to retrieve a pointer to a supported interface on an object.

C++: ImageUtilities.cpp

```
// Retrieves the raw pixel data from the provided IBuffer object.
byte* GetPointerToPixelData(IBuffer^ buffer)
{
    // Cast to Object^, then to its underlying IInspectable interface.
    Object^ obj = buffer;
    ComPtr<IInspectable> insp(reinterpret_cast<IInspectable*>(obj));

    // Query the IBufferByteAccess interface.
    ComPtr<IBufferByteAccess> bufferByteAccess;
    ThrowIfFailed(insp.As(&bufferByteAccess));

    // Retrieve the buffer data.
    byte* pixels = nullptr;
    ThrowIfFailed(bufferByteAccess->Buffer(&pixels));
    return pixels;
}
```

Because we're working with COM, we defined the **ThrowIfFailed** function to more easily deal with **HRESULT** values. You can use this utility function in your code when you need to convert a failure code to a Windows Runtime exception.

C++: ImageUtilities.cpp

```
inline void ThrowIfFailed(HRESULT hr)
{
    if (FAILED(hr))
        throw Exception::CreateException(hr);
}
```

For more about smart pointers, see Smart Pointers (Modern C++).

Use stack semantics and the RAII pattern

Stack semantics and the RAII pattern are closely related.

Use stack semantics to automatically control object lifetime and help minimize unnecessary heap allocations. Also use stack semantics to define member variables in your classes and other data structures to automatically free resources when the parent object is freed.

Use the resource acquisition is initialization (RAII) pattern to ensure that resources are freed when the current function returns or throws an exception. Under the RAII pattern, a data structure is allocated on the stack. That data structure initializes or acquires a resource when it's created and destroys or releases that resource when the data structure is destroyed. The RAII pattern guarantees that the destructor is called before the enclosing scope exits. This pattern is useful when a function contains multiple **return** statements. This pattern also helps you write exception-safe code. When a **throw** statement causes the stack to unwind, the destructor for the RAII object is called; therefore, the resource is always correctly deleted or released. Using the **std::shared_ptr** template class for stack-allocated variables is an example of the RAII pattern.

For more info about managing object lifetime, see Object Lifetime And Resource Management (Modern C++). For more info about RAII, see Objects Own Resources (RAII).

Don't keep objects around longer than you need

Use stack semantics or set references to **nullptr** to help ensure that objects are freed when they have no more references.

In Hilo, we preferred the use of stack semantics over member variables for two reasons. First, stack semantics helps ensure that memory is only used in the context of where it's needed. Because Hilo is a photo app, we found it particularly important to free image data as soon as possible to keep memory consumption to a minimum. The same applies to other kinds of apps. For example, in a blog reader, you might want to release network connections when they're no longer needed to allow other apps to use those resources.

Second, because much of the app depends on asynchronous actions, we wanted to restrict variable access to the code that needs it to help make the app concurrency-safe. In traditional multithreading programming that doesn't use lambda expressions, you often need to store state as member variables. Here's an example where we used local variables and capture semantics instead of member variables to asynchronously create a thumbnail from an image on disk.

C++: ThumbnailGenerator.cpp

```
task<InMemoryRandomAccessStream^>
ThumbnailGenerator::CreateThumbnailFromPictureFileAsync(
    StorageFile^ sourceFile,
    unsigned int thumbSize)
{
    (void)thumbSize; // Unused parameter
    auto decoder = make_shared<BitmapDecoder^>(nullptr);
    auto pixelProvider = make_shared<PixelDataProvider^>(nullptr);
    auto resizedImageStream = ref new InMemoryRandomAccessStream();
    auto createThumbnail = create_task(
        sourceFile->GetThumbnailAsync(
        ThumbnailMode::PicturesView,
        ThumbnailSize));

    return createThumbnail.then([](StorageItemThumbnail^ thumbnail)
    {
        IRandomAccessStream^ imageFileStream =
            static_cast<IRandomAccessStream^>(thumbnail);

        return BitmapDecoder::CreateAsync(imageFileStream);

    }).then([decoder](BitmapDecoder^ createdDecoder)
    {
        (*decoder) = createdDecoder;
        return createdDecoder->GetPixelDataAsync(
            BitmapPixelFormat::Rgba8,
            BitmapAlphaMode::Straight,
            ref new BitmapTransform(),
            ExifOrientationMode::IgnoreExifOrientation,
            ColorManagementMode::ColorManageToSRgb);

    }).then([pixelProvider, resizedImageStream](PixelDataProvider^ provider)
    {
        (*pixelProvider) = provider;
        return BitmapEncoder::CreateAsync(
            BitmapEncoder::JpegEncoderId,
            resizedImageStream);

    }).then([pixelProvider, decoder](BitmapEncoder^ createdEncoder)
    {
        createdEncoder->SetPixelData(BitmapPixelFormat::Rgba8,
            BitmapAlphaMode::Straight,
            (*decoder)->PixelWidth,
            (*decoder)->PixelHeight,
            (*decoder)->DpiX,
            (*decoder)->DpiY,
            (*pixelProvider)->DetachPixelData());
```

```
        return createdEncoder->FlushAsync();

    }).then([resizedImageStream]
    {
        resizedImageStream->Seek(0);
        return resizedImageStream;
    });
}
```

If your app requires the use of a member variable, you should set that variable to **nullptr** when you no longer need it.

Avoid circular references

Objects involved in a circular reference can never reach a reference count of zero and are never destroyed. Avoid circular references in your code by replacing one of the references with a weak reference.

For example, Hilo defines the **IPhoto** and **IPhotoGroup** interfaces. **IPhoto** encapsulates info about a photo (its file path, image type, date taken, and so on). **IPhotoGroup** manages collections of photos (for example, all the photos in a given month). These interfaces have a parent-child relationship— you can get an individual photo from a photo group or the parent photo group from a photo. If we were to use standard, or *strong*, references, a photo and its group will always hold one reference to the other, and the memory for neither is ever freed, even after all other references are released.

Instead, Hilo uses *weak references* to avoid circular references like this. A weak reference enables one object to reference another without affecting its reference count.

The Windows Runtime provides the **WeakReference** class. Its **Resolve** method returns the object if it is valid; otherwise it returns **nullptr**. Here is how the **Photo** class, which derives from **IPhoto**, declares a weak reference to its parent group:

C++: Photo.h

```
Platform::WeakReference m_weakPhotoGroup;
```

Here's how the **Photo** class resolves the weak reference in the **Group** property:

C++: Photo.cpp

```
IPhotoGroup^ Photo::Group::get()
{
    return m_weakPhotoGroup.Resolve<IPhotoGroup>();
}
```

Hilo also uses **std::weak_ptr** to break circular references. Use **WeakReference** when you must expose a weak reference across the ABI boundary. Use **std::weak_ptr** in internal code that doesn't interact with the Windows Runtime.

Capturing an object's **this** handle in a lambda expression can also cause a circular reference if you use the lambda as an event handler or pass it to a task that the object references.

Note Within member functions of a **ref** class, the type of the symbol **this** is a const handle (^), not a pointer.

An object's event handlers are detached from the source when that object is destroyed. However, a lambda expression that captures **this** increments that object's reference count. If an object creates a lambda expression that captures **this** and uses that lambda as the handler of an event that is provided by itself or an object it directly or indirectly references, the object's reference count can never reach zero. In this case, the object is never destroyed.

Hilo prevents the circular reference by using member functions instead of lambda expressions to work with events. Here's an example from the **HiloPage** class.

C++: HiloPage.cpp

```
m_navigateBackEventToken = viewModel->NavigateBack::add(ref new
NavigateEventHandler(this, &HiloPage::NavigateBack));
m_navigateHomeEventToken = viewModel->NavigateHome::add(ref new
NavigateEventHandler(this, &HiloPage::NavigateHome));
```

In this example, **NavigateEventHandler** is a delegate type defined by Hilo. A delegate is a ref class that's the Windows Runtime equivalent of a function object in standard C++. When you instantiate a delegate using an object handle and a pointer to a member function, the object's reference count is not incremented. If you invoke the delegate instance after the target object has been destroyed, a **Platform::DisconnectedException** will be raised.

In addition to the two-argument constructor that is shown in this example, delegate types also have an overloaded constructor that accepts a single argument. The argument is a function object such as a lambda expression. If this example captured the object's **this** handle in a lambda expression and passed the lambda as the delegate's constructor argument, then the object's reference count would be

incremented. The reference count would be decremented only when there were no more references to the lambda expression.

If you use lambdas as event handlers, you have two ways to manage memory. You can capture weak references instead of object handles. Or, you can be careful to unsubscribe from events at the right time and break circular references before they cause problems with memory management.

In Hilo, we chose member functions for event callbacks because we felt that this was the easiest to code correctly.

For more info about weak references, see Weak references and breaking cycles (C++/CX) and How to: Create and Use weak_ptr Instances.

Debugging tips and techniques

You can use many of the traditional tools and techniques for Windows Store apps that you use to debug desktop apps. Here are some that we used for Hilo:

- Use breakpoints and tracepoints.
- Use OutputDebugString for "printf" style debugging.
- Break when exceptions are thrown.
- Use the parallel debugging windows.
- Use the simulator and remote debugging to debug specific hardware configurations.

Use breakpoints and tracepoints

A breakpoint tells the debugger that an application should break, or pause, execution at a certain point. A *tracepoint* is a breakpoint with a custom action associated with it. For Hilo, we used breakpoints extensively to examine app state when running PPL continuation tasks.

You can configure breakpoints to trigger when certain conditions are true. For Hilo, we had an interaction between XAML and the C++ code-behind that we needed to debug. However, the issue occurred only after the code had run at least 20 times. So we used the **Hit Count** setting to break after the breakpoint was hit at least 20 times.

For Hilo, we found tracepoints to be especially useful to diagnose from which thread certain PPL tasks were running. The default tracepoint message contained all the info we needed.

```
Function: $FUNCTION, Thread: $TID $TNAME
```

For more info, see Using Breakpoints and Tracepoints.

Tip Use **__debugbreak** to programmatically set a breakpoint from code.

Use OutputDebugString for "printf" style debugging

Use **OutputDebugString** when you don't require breaking into the debugger. **OutputDebugString** also works with tools such as WinDbg and can be used with Release mode.

Caution You can also use **TextBlock** and other XAML controls to perform "printf" style debugging. However, we preferred the use of **OutputDebugString** because using a control can affect the layout of pages in ways you may not expect after you remove them. If you use controls to debug your code, be sure to remove them and then test your code before you deploy your app.

Break when exceptions are thrown

When you break in a **catch** statement, you don't know what code threw the exception. From Visual Studio, choose **Debug**, **Exceptions**, and then choose **Thrown** to break when the exception is thrown. This technique is especially useful when using PPL tasks because you can examine the specific threading context under which the error occurred.

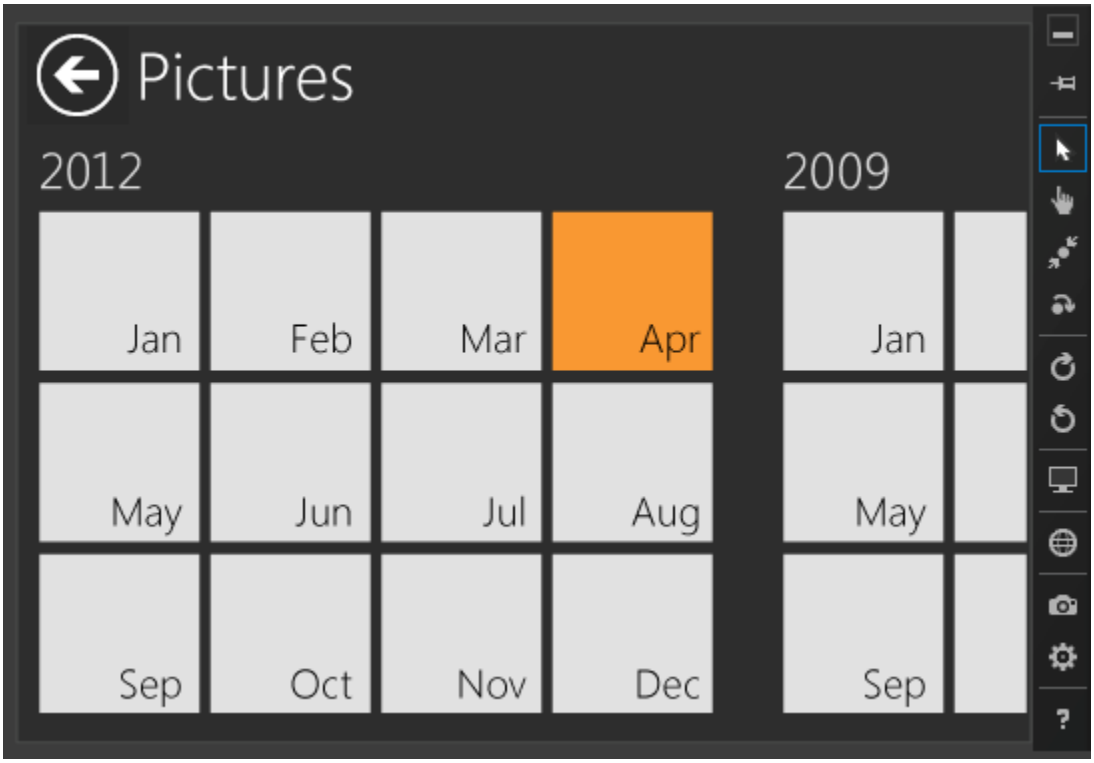
Important It's important to catch every exception that is thrown by PPL tasks. If you don't observe an exception that was thrown by a PPL task, the runtime terminates the app. If your app terminates unexpectedly, enable this feature to help diagnose where the exception occurred.

Use the parallel debugging windows

Use the **Parallel Tasks**, **Parallel Stacks**, and **Parallel Watch** windows to understand and verify the runtime behavior of code that uses the PPL and other Concurrency Runtime features. For Hilo, we used these windows to understand the state of all running tasks at various points in the program. For more info, see Walkthrough: Debugging a Parallel Application and How to: Use the Parallel Watch Window.

Use the simulator and remote debugging to debug specific hardware configurations

For example, if your monitor isn't touch-enabled, you can use the simulator to emulate pinch, zoom, rotation, and other gestures. The simulator also enables you to simulate geolocation and work with different screen resolutions. For more info, see Running Windows Store apps in the simulator and Using the simulator and remote debugger to test devices in this guide.



Use remote debugging to debug your app on a computer that doesn't have Visual Studio. This is useful when you have a computer with a different hardware configuration than your computer that is running Visual Studio. For more info, see Running Windows Store apps on a remote machine.

For more info about debugging, see Debugging Windows Store apps and Debugging Native Code.

Porting existing C++ code

If you have existing code in C++ that you want to port to your Windows Store app, the amount of work that you need to do depends on the code. Code that uses previous UI frameworks needs to be rewritten, while other types of code, such as business logic and numerical routines, can generally be ported without difficulty. If you are using code written by someone else, you might also check to see whether a version already exists for Windows Store apps. In Hilo, we adapted existing code that performs the cartoon effect and ported it for use as a static library (see the CartoonEffect project in the Hilo source files.) We did not introduce any Windows Runtime dependencies in the static library, which allows us to target previous versions of Windows and Windows 8 desktop apps.

Windows Store apps using C++ can reference existing static libraries, DLLs, and COM components, as long as all the existing code that your app calls meets the requirements for Windows Store apps. In Windows Store apps, all components, including static libraries and DLLs, must be local to the app and packaged in the manifest. All the components in the app's package must adhere to the requirements for Windows Store apps.

Overview of the porting process

Here's an outline of how to port existing C++ to make it usable in a Windows Store app. We used this process when porting the existing code that we brought into Hilo.

- Compile and test the code on Windows 8.
- Identify unsupported library functions.
- Use functions from the Window Runtime API reference.
- Replace synchronous library functions with async versions.
- Convert long running operations in your code to async versions.
- Validate the package with the Windows App Certification Kit.

Compile and test the code on Windows 8

The first step for porting is to compile and test your existing code on Windows 8. Code that compiles and runs on Windows 7 should require very few changes on Windows 8, but it's a good idea to identify any platform dependencies before you port your code for use in a Windows Store app.

When writing Hilo, we reused the code that performs the cartoon effect image filtering from an earlier project. This code compiled and ran without change on the Windows 8 platform.

Note Because porting can change the original code, it's a good idea to create unit tests for the original code before starting the port. You can use the new Visual Studio 2012 Native Testing framework to do this.

Identify unsupported library functions

After your code is running, you must make sure that your code only uses functions that are available to Windows Store apps. You can find the list of supported functions in the API reference for Windows Store apps. See Overview of supported functions in this guide for an outline of what's available.

You can also use the compiler preprocessor directive **WINAPI_FAMILY=WINAPI_PARTITION_APP** to help find incompatibilities. This directive causes the C++ compiler to issue errors when it encounters calls to unsupported functions.

Tip Setting the **WINAPI_FAMILY** directive to **WINAPI_PARTITION_APP** during compilation removes the declarations of some Win32 functions and data types, but it doesn't affect the linking step. You can reintroduce some of the missing declarations in a temporary *toremove.h* header file and then work to eliminate references to those functions one at a time. A temporary header file is also a good way to work around the fact that the compiler stops after encountering approximately 100 errors.

Use functions from the Window Runtime API reference

In most cases, you can find a Windows Runtime function that performs the operation you want to do. For example, the Win32 function **InitializeCriticalSectionAndSpinCount** isn't available to Windows Store apps, but the Windows Runtime provides **InitializeCriticalSectionEx** instead. As you work through the problem areas, build your code with the new functions and incrementally test your app. See Alternatives to Windows APIs in Windows Store apps for suggestions of replacement functions to use.

In some cases, you can resolve incompatibilities by using functions from the C++ run-time library and STL. In general, use standard C++ whenever possible. Refactor core logic to use C++ libraries to operate on containers, buffers, and so on. There are many new features in C++ 11 libraries, such as threads, mutexes, and I/O. Be sure to look in the C++ libraries when checking for a replacement for a Win32 API in the C++ library.

Tip Isolate platform-specific code into abstractions to make porting easier.

The remaining code, which formed the core of the cartoon effect image filter, didn't make calls to the Win32 API. It just consisted of numerical functions that manipulated the pixels in the image. The only real porting issues for Hilo were in reconciling the various data formats for bitmaps. For example, when encoding the image prior to saving it, we needed to reorder the color channels to use the BGRA (blue/green/red/alpha) layout that is required by Windows Runtime functions.

Replace synchronous library functions with async versions

After your code compiles and runs using only data types and functions that are available to Windows Store apps, review your code for synchronous library functions that have asynchronous versions available. Refactor the code to consume aysnc operations if it doesn't already do so. Your app should use the async approach whenever possible.

Tip Refactor code to deal with partial data. In cases where partial results are available incrementally, make these results available to the user without delay.

Tip Keep the UI responsive during async operations.

In Hilo, we needed to convert the functions of the original cartoon effect image filter code that loaded the image to be processed to async versions.

Convert long running operations in your code to async versions

If your code has long running operations, consider making these operations asynchronous. An easy way to do this is to schedule long running work on the thread pool, for example, by using a PPL task. You can rewrite code or write a wrapper that runs existing code in thread pool.

In Hilo, we used a PPL continuation task that runs in the thread pool for the ported cartoon effect image filter.

Validate the package with the Windows App Certification Kit

After you have changed your code to use only the functions that are documented in the API reference for Windows Store apps, the final porting step is to create a package using app binaries and then to use the Windows App Certification Kit tool to check that the package is compliant with all the requirements of the Windows Store. The package contains all the app's components, including static libraries and DLLs.

For a description of Hilo's experience with certification, see Testing and deploying the app in this guide.

Overview of supported functions

Here's a summary of what functions and data types a Windows Store app can use. For a complete list, see the API reference for Windows Store apps.

- Porting from Win32-based UI.
- Porting DirectX.
- Porting MFC.
- Using the C++ run-time library (CRT).
- Using the C++ Standard Library.
- Using ATL.

Porting from Win32-based UI

If you have existing code that uses UI functions and data types, such as **HWND**, from **User**, **GDI**, or **MFC**, you'll need to reimplement that code using XAML.

Porting DirectX

DirectX is available in Windows Store apps. If you use DirectX 11, most code ports easily. Use a Visual Studio template for a DirectX project as a starting point. When you initialize a Windows Store app at run time, there is some DirectX setup required. The Visual Studio template sets up this startup code for you. Move your existing code into the new project.

Porting MFC

MFC is not available in Windows Store apps. To replace MFC UI classes, use XAML or DirectX. For dialog apps, you can use XAML data controls with data binding. To replace MFC utility classes such as containers, use STL and C run-time (CRT) data types.

Using the C++ run-time library (CRT)

A large subset of the CRT is available for Windows Store apps. There are a few functions that aren't available.

- **Multi-byte string functions**. The **mb*** and **_ismb*** functions aren't available. Use Unicode strings instead.
- **Process control functions**. The **exec*** functions aren't available. There is no way for you to spawn an application from within a Windows Store app.
- **Thread creation functions**. The **beginthread*** and **endthread*** functions aren't available. Threading is available in Windows Store apps, but it's based on a thread pool. You can also use **std::thread**.
- **Heap and stack functions**. The functions **heapwalk**, **heapmin**, **resetstkoflw**, and others aren't available. You cannot create your own heap in Windows Store apps.
- **Environment variable functions**. The functions **putenv**, **getenv**, **_enviorn**, and related functions aren't available. There are no environment blocks in Windows Store apps.
- **Console functions**. The functions **cprintf**, **cscanf**, and related functions aren't available. There's no console in Windows Store apps.
- **Port functions**. The functions **outp**, **inp**, and other port functions aren't available.
- **Pipe functions**. The functions **popen**, **pclose**, and other pipe functions aren't available.

Note Use the compiler preprocessor directive **WINAPI_FAMILY=WINAPI_PARTITION_APP** to help find incompatibilities.

Some CRT functions that are available to Windows Store apps are blocking I/O functions. We recommend that you replace synchronous I/O functions, such as **open**, **read**, and **write**, with async equivalents from the Windows Runtime.

ANSI string functions of the CRT are available in Windows Store apps, but we recommend that you use Unicode strings.

Using the C++ Standard Library

Nearly all functions and data types of the C++ Standard Library are available in Windows Store apps (console I/O is not available.) We use the C++ Standard Library extensively in Hilo.

Using ATL

A subset of the ATL library is available in Windows Store apps. Here are the available data types.

- **DLL server**.
- **COM objects**. You can create your own COM objects. However, **IDispatch** isn't supported.

- **CStringW**. Only wide strings are supported.
- **ATL container classes**. MFC containers such as map that were ported to ATL are still available.
- **CCriticalSection, CEvent, CMutex, CSemaphore, CMutexLock**. ATL synchronization objects are available.
- **CComVariant**.
- **CComSafeArray**.
- **CComBSTR**.

If you have COM components that are related to business logic and not UI, you'll generally be able to port them.

Porting guidance

Here are some tips for porting existing C++ into a Windows Store app.

- Port all existing code, including libraries.
- Link to static libraries or import libraries as usual.
- Use C++/CX or WRL if your library needs to invoke Windows Runtime functions.
- Use reg-free COM for activation.
- Convert to Windows Runtime types when marshaling cost is an issue.
- Decide between using wrapper code and converting existing code.

Port all existing code, including libraries

Your app's package must include all the binary components that your app needs. This includes static libraries and DLLs.

Link to static libraries or import libraries as usual

You can use libraries that are written in standard C++ in your Windows Store app. Windows Store apps use the same linking as desktop and console apps. If you have binary dependencies, it is important to validate your application package early in the development process so that all library functionality meets the requirements.

Note Although the CartoonEffect library does not directly use any Windows Runtime features, we did need to specify the **/ZW** (Consume Windows Runtime Extension) compiler option because the PPL version of the cartoon effect algorithm uses special semantics in Windows Store apps (for example, in a Windows Store app, you can configure the context on which PPL task continuations run.)

Use C++/CX or WRL if your library needs to invoke Windows Runtime functions

If your library needs to invoke Windows Runtime functions, you can use either C++/CX or WRL. In general terms, C++/CX is easier to use. WRL gives you more fine-grained control.

Note There are some technical issues with exposing public reference classes from within a user-written library. This is outside of the scope of this guide.

Use reg-free COM for activation

Your Windows Store app can use COM components that come from libraries. The library doesn't register COM classes. Instead, your app invokes COM activation using the new function **CoCreateInstanceFromApp**.

Convert to Windows Runtime types when marshaling cost is an issue

Use Windows Runtime types for objects that frequently cross the ABI boundary and are costly to convert.

You can use **String** and **Array**, which can be efficiently converted without copying, as input parameters to the Windows Runtime API. **StringReference** and **ArrayReference** add a Windows Runtime veneer using *borrow* semantics.

For containers and collection types, the conversion from **std::*** to **Platform::*** requires copying. In general, the containers and collections of the **std** namespace are more efficient than the containers and collections of the **Platform** namespace. When to use each of these depends on how often collection contents change compared to how often they cross the ABI.

In Hilo, we spent a lot of time deciding when to use public **ref** classes. For our application, we found that the overhead of type conversion outweighed other concerns.

Decide between using wrapper code and converting existing code

If you have existing code that needs to be called across the ABI, you must decide whether to port that code to C++/CX or leave it in standard C++ and wrap it with a C++/CX layer. In most situations we recommend using a wrapper layer.

Converting existing code and making it use Windows Runtime types and concepts is normally only done to avoid data conversion overhead, for example, when your component must be called very frequently across the ABI. These situations are rare.

If you decide to create a C++/CX wrapper for your classes, the standard way to do this is to define interfaces and in their implementation delegate to your existing C++ code, after any necessary type

conversions. Using interfaces in this way creates a Windows Runtime veneer over your existing code and is sufficient for most cases.

If you have existing COM components, consider exposing them as Windows Runtime types. Doing this makes it easy for Windows Store apps to consume your COM objects. The techniques required to do this are outside the scope of this guide.

For more info about porting

A Channel 9 video is a helpful source of information about porting existing C++ for use in a Windows Store app. See Porting a desktop app to a Windows Store app for more information.

See Win32 and COM for Windows Store apps for the API subset of supported functions. If you can't find a suitable Win32 API function, see Alternatives to Windows APIs in Windows Store apps for suggestions of Windows Runtime functions that can be used instead.

Async programming patterns and tips in Hilo (Windows Store apps using C++ and XAML)

Summary

- Use the Parallel Patterns Library (PPL) for asynchronous operations in your Windows Store app. PPL makes it easier to cancel operations, handle exceptions, and create chains of tasks.
- Don't call blocking operations from your app's main thread to ensure a highly responsive app.

Important APIs

- **concurrency::task**
- **concurrency::task::then**
- **concurrency::create_async**

Hilo contains many examples of continuation chains in C++, which are a common pattern for asynchronous programming in Windows Store apps. Here are some tips and guidance for using continuation chains, and examples of the various ways you can use them in your app.

You will learn

- Best practices for asynchronous programming in Windows Store apps using C++.
- How to cancel pending asynchronous operations.
- How to handle exceptions that occur in asynchronous operations.
- How to use parallel tasks with Windows Store apps that are written in Visual C++.

Ways to use the continuation chain pattern

The asynchronous programming techniques that we use in Hilo fall under the general pattern known as a *continuation chain* or a .then ladder (pronounced dot-then ladder). A continuation chain is a sequence of PPL tasks that are connected by data flow relationships. The output of each task becomes the input of the next continuation in the chain. The start of the chain is usually a task that wraps a Windows Runtime asynchronous operation (an interface that derives from **IAsyncInfo**). We introduced continuation chains in Writing modern C++ code for Windows Store apps in this guide.

Here's an example from Hilo that shows a continuation chain that begins with task that wraps an **IAsyncInfo**-derived interface.

C++: FileSystemRepository.cpp

```
task<IPhoto^> FileSystemRepository::GetSinglePhotoAsync(String^ photoPath)
```

```
{
    String^ query = "System.ParsingPath:=\"" + photoPath + "\"";
    auto fileQuery = CreateFileQuery(KnownFolders::PicturesLibrary, query,
IndexerOption::DoNotUseIndexer);
    auto fileInformationFactory = ref new FileInformationFactory(fileQuery,
ThumbnailMode::PicturesView);
    shared_ptr<ExceptionPolicy> policy = m_exceptionPolicy;
    return create_task(fileInformationFactory->GetFilesAsync(0,
1)).then([policy](IVectorView<FileInformation^>^ files)
    {
        IPhoto^ photo = nullptr;
        auto size = files->Size;
        if (size > 0)
        {
            photo = ref new Photo(files->GetAt(0), ref new NullPhotoGroup(), policy);
        }
        return photo;
    }, task_continuation_context::use_current());
}
```

In this example, the expected calling context for the **FileSystemRepository::GetSinglePhotoAsync** method is the main thread. The method creates a PPL task that wraps the result of the **GetFilesAsync** method. The **GetFilesAsync** method is a helper function that returns an **IAsyncOperation< IVectorView<FileInformation^>^>^** handle.

The **task::then** method creates continuations that run on the same thread or a different thread than the task that precedes them, depending on how you configure them. In this example, there is one continuation.

Note Because the initial task that the **concurrency::create_task** function created wraps an **IAsyncInfo**-derived type, its continuations run by default in the context that creates the continuation chain, which in this example is the main thread.

Some operations, such as those that interact with XAML controls, must occur in the main thread. For example, the **Photo** object can be bound to a XAML control and therefore must be instantiated in the main thread. On the other hand, you must call blocking operations from a background thread only and not your app's main thread. So, to avoid programming errors, you need to know whether each continuation will use the main thread or a background thread from the thread pool.

There are a variety of ways to use the continuation chain pattern depending on your situation. Here are some variations of the basic continuation chain pattern:

- Value-based and task-based continuations.
- Unwrapped tasks.
- Allowing continuation chains to be externally canceled.
- Other ways of signaling cancellation.
- Canceling asynchronous operations that are wrapped by tasks.
- Using task-based continuations for exception handling.
- Assembling the outputs of multiple continuations.
- Using nested continuations for conditional logic.
- Showing progress from an asynchronous operation.
- Creating background tasks with create_async for interop scenarios.
- Dispatching functions to the main thread.
- Using the Asynchronous Agents Library.

Value-based and task-based continuations

The task that precedes a continuation is called the continuation's *antecedent task* or *antecedent*. If an antecedent task is of type **task<T>**, a continuation of that task can either accept type **T** or type **task<T>** as its argument type. Continuations that accept type **T** are *value-based continuations*. Continuations that accept type **task<T>** are *task-based continuations*.

The argument of a value-based continuation is the return value of the work function of the continuation's antecedent task. The argument of a task-based continuation is the antecedent task itself. You can use the **task::get** method to query for the output of the antecedent task.

There are several behavioral differences between value-based and task-based continuations. Value-based continuations don't run if the antecedent task terminated in an exception. In contrast, task-based continuations run regardless of the exception status of the antecedent task. Also, value-based continuations inherit the cancellation token of their antecedent by default while task-based continuations don't.

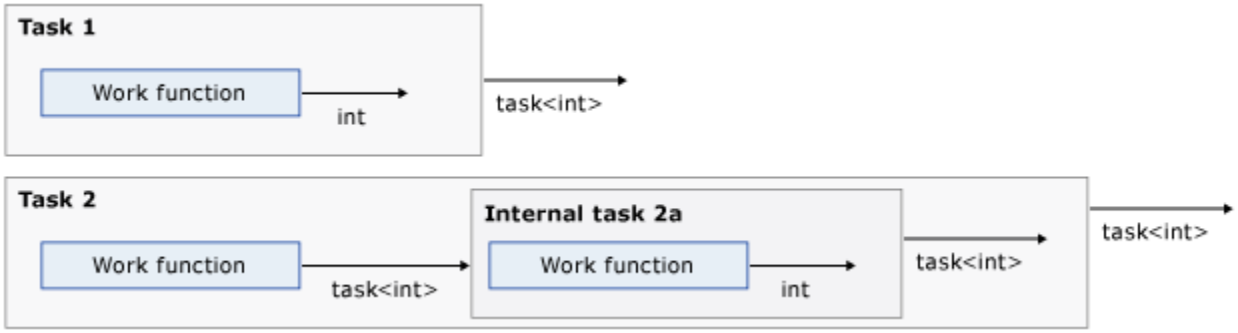
Most continuations are value-based continuations. You use task-based continuations in scenarios that involve exception handling and cancellation. See Allowing continuation chains to be externally canceled and Using task-based continuations for exception handling later in this topic for examples.

Unwrapped tasks

In most cases the work function of a PPL task returns type **T** if the task is of type **task<T>**, but returning type **T** is not the only possibility. You can also return a **task<T>** value from the work function that you pass to the **create_task** or **task::then** functions. In this case, you might expect PPL to create a task with

type **task<task<T>>**, but this is not what happens. Instead, the resulting task has type **task<T>**. The automatic transformation of **task<task<T>>** to **task<T>** is called *unwrapping a task*.

Unwrapping occurs at the type level. PPL schedules the inner task and uses its result as the result of the outer task. The outer task finishes when the inner task completes its work.



Task 1 in the diagram has a work function that returns an int. The type of task 1 is task<int>. Task 2 has a work function that returns task<int>, which is task 2a in the diagram. Task 2 waits for task 2a to finish, and returns the result of task 2a's work function as the result of task 2.

Unwrapping tasks helps you put conditional dataflow logic in networks of related tasks. See Using nested continuations for conditional logic later on this topic for scenarios and examples that require PPL to unwrap tasks. For a table of return types see "Lambda function return types and task return types" in Asynchronous programming in C++.

Allowing continuation chains to be externally canceled

You can create continuation chains that respond to external cancellation requests. For example, when you display the image browser page in Hilo, the page starts an asynchronous operation that creates groups of photos by month and displays them. If you navigate away from the page before the display operation has finished, the pending operation that creates month group controls is canceled. Here are the steps.

1. Create a **concurrency::cancellation_token_source** object.
2. Query the cancellation token source for a **concurrency::cancellation_token** using the **cancellation_token_source::get_token** method.
3. Pass the cancellation token as an argument to the **concurrency::create_task** function of the initial task in the continuation chain or to the **task::then** method of any continuation. You only need to do this once for each continuation chain because subsequent continuation tasks use the cancellation context of their antecedent task by default.
4. To cancel the continuation chain while it's running, call the **cancel** method of the cancellation token source from any thread context.

Here is an example.

C++: ImageBrowserViewModel.cpp

```
void ImageBrowserViewModel::StartMonthAndYearQueries()
{
    assert(IsMainThread());
    assert(!m_runningMonthQuery);
    assert(!m_runningYearQuery);
    assert(m_currentMode == Mode::Active || m_currentMode == Mode::Running ||
m_currentMode == Mode::Pending);

    m_cancellationTokenSource = cancellation_token_source();
    auto token = m_cancellationTokenSource.get_token();
    StartMonthQuery(m_currentQueryId, token);
    StartYearQuery(m_currentQueryId, token);
}
```

The **StartMonthAndYearQueries** method creates the **concurrency::cancellation_token_source** and **concurrency::cancellation_token** objects and passes the cancellation token to the **StartMonthQuery** and **StartYearQuery** methods. Here is the implementation of **StartMonthQuery** method.

C++: ImageBrowserViewModel.cpp

```
void ImageBrowserViewModel::StartMonthQuery(int queryId, cancellation_token token)
{
    m_runningMonthQuery = true;
    OnPropertyChanged("InProgress");
    m_photoCache->Clear();
    run_async_non_interactive([this, queryId, token]()
    {
        // if query is obsolete, don't run it.
        if (queryId != m_currentQueryId) return;

        m_repository->GetMonthGroupedPhotosWithCacheAsync(m_photoCache, token)
            .then([this, queryId](task<IVectorView<IPhotoGroup^>^> priorTask)
        {
            assert(IsMainThread());
            if (queryId != m_currentQueryId)
            {
                // Query is obsolete. Propagate exception and quit.
                priorTask.get();
                return;
            }

            m_runningMonthQuery = false;
            OnPropertyChanged("InProgress");
```

```
            if (!m_runningYearQuery)
            {
                FinishMonthAndYearQueries();
            }
            try
            {
                // Update display with results.
                m_monthGroups->Clear();
                for (auto group : priorTask.get())
                {
                    m_monthGroups->Append(group);
                }
                OnPropertyChanged("MonthGroups");
            }
            // On exception (including cancellation), remove any partially computed
results and rethrow.
            catch (...)
            {
                m_monthGroups = ref new Vector<IPhotoGroup^>();
                throw;
            }
        },
        task_continuation_context::use_current())
            .then(ObserveException<void>(m_exceptionPolicy));
    });
}
```

The **StartMonthQuery** method creates a continuation chain. The head of the continuation chain and the chain's first continuation task are constructed by the the **FileSystemRepository** class's **GetMonthGroupedPhotosWithCacheAsync** method.

C++: FileSystemRepository.cpp

```
task<IVectorView<IPhotoGroup^>^>
FileSystemRepository::GetMonthGroupedPhotosWithCacheAsync(shared_ptr<PhotoCache>
photoCache, concurrency::cancellation_token token)
{
    auto queryOptions = ref new QueryOptions(CommonFolderQuery::GroupByMonth);
    queryOptions->FolderDepth = FolderDepth::Deep;
    queryOptions->IndexerOption = IndexerOption::UseIndexerWhenAvailable;
    queryOptions->Language = CalendarExtensions::ResolvedLanguage();
    auto fileQuery = KnownFolders::PicturesLibrary-
>CreateFolderQueryWithOptions(queryOptions);
    auto fileInformationFactory = ref new FileInformationFactory(fileQuery,
ThumbnailMode::PicturesView);

    m_monthQueryChange = (m_imageBrowserViewModelCallback != nullptr) ? ref new
```

```
QueryChange(fileQuery, m_imageBrowserViewModelCallback) : nullptr;

    shared_ptr<ExceptionPolicy> policy = m_exceptionPolicy;
    auto sharedThis = shared_from_this();
    return create_task(fileInformationFactory->GetFoldersAsync()).then([this,
fileInformationFactory, photoCache, sharedThis,
policy](IVectorView<FolderInformation^>^ folders)
    {
        auto temp = ref new Vector<IPhotoGroup^>();
        for (auto folder : folders)
        {
            auto photoGroup = ref new MonthGroup(photoCache, folder, sharedThis,
policy);
            temp->Append(photoGroup);
        }
        return temp->GetView();
    }, token);
}
```

This code passes a cancellation token to the **task::then** method at the first continuation task.

The year and month queries can take a few seconds to run if there are many pictures to process. It is possible that the user might navigate away from the page while they are running. If this happens the queries are cancelled. The **OnNavigatedFrom** method in the Hilo app calls the **CancelMonthAndYearQueries** method. The **CancelMonthAndYearQueries** of the **ImageBrowserViewModel** class invokes the **cancel** method of the cancellation token source.

C++: ImageBrowserViewModel.cpp

```
void ImageBrowserViewModel::CancelMonthAndYearQueries()
{
    assert(m_currentMode == Mode::Running);

    if (m_runningMonthQuery)
    {
        m_runningMonthQuery = false;
        OnPropertyChanged("InProgress");
    }
    m_runningYearQuery = false;
    m_currentQueryId++;
    m_cancellationTokenSource.cancel();
}
```

Cancellation in PPL is cooperative. You can control the cancellation behavior of tasks that you implement. For the details about what happens when you call the cancellation token's source cancel method, see "Canceling tasks" in Asynchronous programming in C++.

Other ways of signaling cancellation

A value-based continuation won't start if the **cancel** method of its associated cancellation token source has been called. It is also possible that the **cancel** method is called while a task in a continuation chain is running. If you want to know whether an external cancellation request has been signaled while running a task, you can call the **concurrency::is_task_cancellation_requested** function to find out. The **is_task_cancellation_requested** function checks the status of the current task's cancellation token. The function returns **true** if the **cancel** method has been called on the cancellation token source object that created the cancellation token. (For an example see Using C++ in the Bing Maps Trip Optimizer sample.)

Note You can call **is_task_cancellation_requested** from within the body of the work function for any kind of task, including tasks created by **create_task** function, the **create_async** function and **task::then** method.

Tip Don't call **is_task_cancellation_requested** within every iteration of a tight loop. In general, if you want to detect a cancellation request from within in a running task, poll for cancellation frequently enough to give you the response time you want without affecting your app's performance when cancellation is not requested.

After you detect that a running task must process a cancellation request, perform whatever cleanup your app needs and then call the **concurrency::cancel_current_task** function to end your task and notify the runtime that cancellation has occurred.

Cancellation tokens are not the only way to signal cancellation requests. When you call asynchronous library functions, you sometimes receive a special signal value from the antecedent task, such as **nullptr**, to indicate that a cancellation was requested.

Canceling asynchronous operations that are wrapped by tasks

PPL tasks help you provide cancellation support in apps that use the asynchronous operations of the Windows Runtime. When you wrap an **IAsyncInfo**-derived interface in a task, PPL notifies the asynchronous operation (by calling the **IAsyncInfo::Cancel** method) if a cancellation request arrives.

Note PPL's **concurrency::cancellation_token** is a C++ type that can't cross the ABI. This means that you can't pass it as an argument to a public method of a public ref class. If you need a cancellation token in this situation, wrap the call to your public ref class's public async method in a PPL task that uses a cancellation token. Then, call the **concurrency::is_task_cancellation_requested** function within the work function of the **create_async** function inside the async method. The **is_task_cancellation_requested** function has access to the PPL task's token, and returns **true** if the token has been signaled.

Using task-based continuations for exception handling

PPL tasks support deferred exception handling. You can use a task-based continuation to catch exceptions that occurred in any of the steps of a continuation chain.

In Hilo we check for exceptions systematically, at the end of every continuation chain. To make this easy, we created a helper class that looks for exceptions and handles them according to a configurable policy. Here's the code.

C++ TaskExceptionsExtensions.h

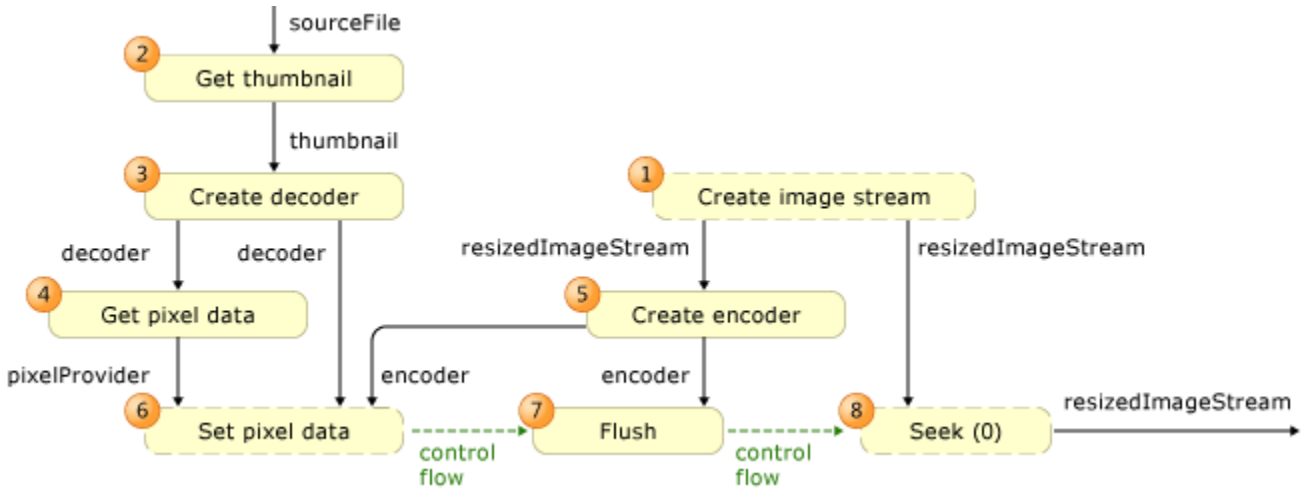
```
    template<typename T>
    struct ObserveException
    {
        ObserveException(std::shared_ptr<ExceptionPolicy> handler) :
m_handler(handler)
        {
        }

        concurrency::task<T> operator()(concurrency::task<T> antecedent) const
        {
            T result;
            try
            {
                result = antecedent.get();
            }
            catch(const concurrency::task_canceled&)
            {
                // don't need to run canceled tasks through the policy
            }
            catch(const std::exception&)
            {
                auto translatedException = ref new Platform::FailureException();
                m_handler->HandleException(translatedException);
                throw;
            }
            catch(Platform::Exception^ ex)
            {
                m_handler->HandleException(ex);
                throw;
            }
            return antecedent;
        }

    private:
        std::shared_ptr<ExceptionPolicy> m_handler;
    };
```

Assembling the outputs of multiple continuations

Continuation chains use a dataflow style of programming where the return value of an antecedent task becomes the input of a continuation task. In some situations the return value of the antecedent task doesn't contain everything you need for the next step. For example, you might need to merge the results of two continuation tasks before a third continuation can proceed. There are several techniques for handling this case. In Hilo we use a shared pointer (**std::shared_ptr**) to temporarily hold intermediate values on the heap.

For example, the next diagram shows the dataflow and control flow relationships of Hilo's **ThumbnailGenerator::CreateThumbnailFromPictureFileAsync** method, which uses a continuation chain to create thumbnail images. Creating a thumbnail from a picture file requires asynchronous steps that don't fit a straight-line dataflow pattern.

In the diagram, the solid ovals represent asynchronous operations in the Windows Runtime. The dashed ovals are tasks that call synchronous functions. The arcs represent inputs and outputs. The dashed arrows are control flow dependencies for operations with side effects. The numbers show the order in which the operations occur in the **CreateThumbnailFromPictureFileAsync** method's continuation chain.



The diagram shows interactions that are too complex for linear dataflow. For example, the synchronous "Set pixel data" operation requires inputs from three separate asynchronous operations, and it modifies one of its inputs, which causes a sequencing constraint for a subsequent asynchronous flush operation. Here's the code for this example.

C++ ThumbnailGenerator.cpp

```
task<InMemoryRandomAccessStream^>
ThumbnailGenerator::CreateThumbnailFromPictureFileAsync(
    StorageFile^ sourceFile,
    unsigned int thumbSize)
```

```
{
    (void)thumbSize; // Unused parameter
    auto decoder = make_shared<BitmapDecoder^>(nullptr);
    auto pixelProvider = make_shared<PixelDataProvider^>(nullptr);
    auto resizedImageStream = ref new InMemoryRandomAccessStream();
    auto createThumbnail = create_task(
        sourceFile->GetThumbnailAsync(
        ThumbnailMode::PicturesView,
        ThumbnailSize));

    return createThumbnail.then([](StorageItemThumbnail^ thumbnail)
    {
        IRandomAccessStream^ imageFileStream =
            static_cast<IRandomAccessStream^>(thumbnail);

        return BitmapDecoder::CreateAsync(imageFileStream);

    }).then([decoder](BitmapDecoder^ createdDecoder)
    {
        (*decoder) = createdDecoder;
        return createdDecoder->GetPixelDataAsync(
            BitmapPixelFormat::Rgba8,
            BitmapAlphaMode::Straight,
            ref new BitmapTransform(),
            ExifOrientationMode::IgnoreExifOrientation,
            ColorManagementMode::ColorManageToSRgb);

    }).then([pixelProvider, resizedImageStream](PixelDataProvider^ provider)
    {
        (*pixelProvider) = provider;
        return BitmapEncoder::CreateAsync(
            BitmapEncoder::JpegEncoderId,
            resizedImageStream);

    }).then([pixelProvider, decoder](BitmapEncoder^ createdEncoder)
    {
        createdEncoder->SetPixelData(BitmapPixelFormat::Rgba8,
            BitmapAlphaMode::Straight,
            (*decoder)->PixelWidth,
            (*decoder)->PixelHeight,
            (*decoder)->DpiX,
            (*decoder)->DpiY,
            (*pixelProvider)->DetachPixelData());
        return createdEncoder->FlushAsync();

    }).then([resizedImageStream]
    {
        resizedImageStream->Seek(0);
        return resizedImageStream;
```

```
    });
}
```

The example calls the **std::make_shared** function to create shared containers for handles to the decoder and pixel provider objects that will be created by the continuations. The ***** (dereference) operator allows the continuations to set the shared containers and read their values.

Using nested continuations for conditional logic

Nested continuations let you defer creating parts of a continuation chain until run time. Nested continuations are helpful when you have conditional tasks in the chain and you need to capture local variables of tasks in a continuation chain for use in subsequent continuation tasks. Nested continuations rely on task unwrapping.

Hilo's image rotation operation uses nested continuations to handle conditional logic. The rotation operation behaves differently depending on whether the file format of the image being rotated supports the Exchangeable Image File Format (EXIF) orientation property. Here's the code.

C++ RotateImageViewModel.cpp

```
concurrency::task<BitmapEncoder^>
RotateImageViewModel::SetEncodingRotation(BitmapEncoder^ encoder,
shared_ptr<ImageEncodingInformation> encodingInfo, float64 rotationAngle,
concurrency::task_continuation_context backgroundContext)
{
    // If the file format supports Exif orientation then update the orientation flag
    // to reflect any user-specified rotation. Otherwise, perform a hard rotate
    // using the BitmapTransform class.
    auto encodingTask = create_empty_task();
    if (encodingInfo->usesExifOrientation)
    {
        // Try encoding with Exif with updated values.
        auto currentExifOrientationDegrees =
ExifExtensions::ConvertExifOrientationToDegreesRotation(ExifRotations(encodingInfo-
>exifOrientation));
        auto newRotationAngleToApply = CheckRotationAngle(safe_cast<unsigned
int>(rotationAngle + currentExifOrientationDegrees));
        auto exifOrientationToApply =
ExifExtensions::ConvertDegreesRotationToExifOrientation(newRotationAngleToApply);
        auto orientedTypedValue = ref new BitmapTypedValue(static_cast<unsigned
short>(exifOrientationToApply), PropertyType::UInt16);

        auto properties = ref new Map<String^, BitmapTypedValue^>();
        properties->Insert(EXIFOrientationPropertyName, orientedTypedValue);

        encodingTask = encodingTask.then([encoder, properties]
```

```
        {
            assert(IsBackgroundThread());
            return encoder->BitmapProperties->SetPropertiesAsync(properties);
        }, backgroundContext).then([encoder, encodingInfo] (task<void>
setPropertiesTask)
        {
            assert(IsBackgroundThread());

            try
            {
                setPropertiesTask.get();
            }
            catch(Exception^ ex)
            {
                switch(ex->HResult)
                {
                case WINCODEC_ERR_UNSUPPORTEDOPERATION:
                case WINCODEC_ERR_PROPERTYNOTSUPPORTED:
                case E_INVALIDARG:
                    encodingInfo->usesExifOrientation = false;
                    break;
                default:
                    throw;
                }
            }

        }, backgroundContext);
    }

    return encodingTask.then([encoder, encodingInfo, rotationAngle]
    {
        assert(IsBackgroundThread());
        if (!encodingInfo->usesExifOrientation)
        {
            BitmapRotation rotation =
static_cast<BitmapRotation>((int)floor(rotationAngle / 90));
            encoder->BitmapTransform->Rotation = rotation;
        }
        return encoder;
    });
}
```

Showing progress from an asynchronous operation

Although the Hilo app doesn't report the progress of ongoing operations using a progress bar, you may want to. To learn more, see the blog post Keeping apps fast and fluid with asynchrony in the Windows

Runtime, which has some examples of how to use the IAsyncActionWithProgress<TProgress> and the IAsyncOperationWithProgress<TProgress, TResult> interfaces.

Note Hilo shows the animated progress ring control during long-running asynchronous operations. For more info about how Hilo displays progress, see ProgressRing in this guide.

Creating background tasks with create_async for interop scenarios

Sometimes you must directly use the interfaces that are derived from **IAsyncInfo** in classes that you define. For example, the signatures of public methods of public ref classes in your app cannot include non-ref types such as **task<T>**. Instead, you expose asynchronous operations with one of the interfaces that derive from the **IAsyncInfo** interface. (A public ref class is a C++ class that has been declared with public visibility and the C++/CX ref keyword.)

You can use the **concurrency::create_async** function to expose a PPL task as an **IAsyncInfo** object.

See Creating Asynchronous Operations in C++ for Windows Store apps for more info about the **create_async** function.

Dispatching functions to the main thread

The continuation chain pattern works well for operations that the user initiates by interacting with XAML controls. Most operations begin on the main thread with an invocation of a view model property that is bound to a property of the XAML control. The view model creates a continuation chain that runs some tasks on the main thread and some tasks in the background. Updating the UI with the result of the operation takes place on the main thread.

But not all updates to the UI are in response to operations that begin on the main thread. Some updates can come from external sources, such as network packets or devices. In these situations, your app may need to update XAML controls on the main thread outside of a continuation context. A continuation chain might not be what you need.

There are two ways to handle this situation. One way is to create a task that does nothing and then schedule a continuation of that task that runs in the main thread using the result of the **concurrency::task_continuation_context::use_current** function as an argument to the **task::then** method. This approach makes it easy to handle exceptions that arise during the operation. You can handle exceptions with a task-based continuation. The second option is familiar to Win32 programmers. You can use the **CoreDispatcher** class's **RunAsync** method to run a function in the main thread. You can get access to a **CoreDispatcher** object from the **Dispatcher** property of the **CoreWindow** object. If you use the **RunAsync** method, you need to decide how to handle exceptions. By default, if you invoke **RunAsync** and ignore its return value, any exceptions that occur during the operation will be lost. If you

don't want to lose exceptions you can wrap the **IAsyncAction^** handle that the **RunAsync** method returns with a PPL task and add a task-based continuation that observes any exceptions of the task.

Here's an example from Hilo.

C++: TaskExtensions.cpp

```
void run_async_non_interactive(std::function<void ()>&& action)
{
    Windows::UI::Core::CoreWindow^ wnd =
Windows::ApplicationModel::Core::CoreApplication::MainView->CoreWindow;
    assert(wnd != nullptr);

    wnd->Dispatcher->RunAsync(
        Windows::UI::Core::CoreDispatcherPriority::Low,
        ref new Windows::UI::Core::DispatchedHandler([action]()
    {
        action();
    }));
}
```

This helper function runs functions that are dispatched to the main thread at a low priority, so that they don't compete with actions on the UI.

Using the Asynchronous Agents Library

Agents are a useful model for parallel programming, especially in stream-oriented apps such as UIs. Use message buffers to interact with agents. For more info see Asynchronous Agents Library.

Tips for async programming in Windows Store apps using C++

Here are some tips and guidelines that can help you write effective asynchronous code in Windows Store apps that use C++.

- Don’t program with threads directly.
- Use "Async" in the name of your async functions.
- Wrap all asynchronous operations of the Windows Runtime with PPL tasks.
- Return PPL tasks from internal async functions within your app.
- Return IAsyncInfo-derived interfaces from public async methods of public ref classes.
- Use public ref classes only for interop.
- Use modern, standard C++, including the std namespace.
- Use task cancellation consistently.
- Handle task exceptions using a task-based continuation.
- Handle exceptions locally when using the when_all function.

- Call view model objects only from the main thread.
- Use background threads whenever possible.
- Don't call blocking operations from the main thread.
- Don't call task::wait from the main thread.
- Be aware of special context rules for continuations of tasks that wrap async objects.
- Be aware of special context rules for the create_async function.
- Be aware of app container requirements for parallel programming.
- Use explicit capture for lambda expressions.
- Don't create circular references between ref classes and lambda expressions.
- Don't use unnecessary synchronization.
- Don't make concurrency too fine-grained.
- Watch out for interactions between cancellation and exception handling.
- Use parallel patterns.
- Be aware of special testing requirements for asynchronous operations.
- Use finite state machines to manage interleaved operations.

Don’t program with threads directly

Although Windows Store apps cannot create new threads directly, you still have the full power of C++ libraries such as the PPL and the Asynchronous Agents Library. In general, use features of these libraries for all new code instead of programming with threads directly.

Note When porting existing code, you can continue to use thread pool threads. See Thread pool sample for a code example.

Use "Async" in the name of your async functions

When you write a function or method that operates asynchronously, use "Async" as part of its name to make this immediately apparent. All async functions and methods in Hilo use this convention. The Windows Runtime also uses it for all async functions and methods.

Wrap all asynchronous operations of the Windows Runtime with PPL tasks

In Hilo, we wrap every async operation that we get from the Windows Runtime in a PPL task by using the **concurrency::create_task** function. We found that PPL tasks are more convenient to work with than the lower-level async interfaces of the Windows Runtime. Unless you use PPL (or write custom wrapping code that checks **HRESULT** values), your app will ignore any errors that occur during the async operations of the Windows Runtime. In contrast, an async operation that is wrapped with a task will automatically propagate runtime errors by using PPL deferred exceptions.

Also, PPL tasks give you an easy-to-use syntax for the continuation chain pattern.

The only exception to this guidance is the case of fire-and-forget functions where you don't want notification of exceptions, which is an unusual situation.

Return PPL tasks from internal async functions within your app

We recommend that you return PPL tasks from all asynchronous functions and methods in your app, unless they are public, protected, or public protected methods of public ref classes. The **IAsyncInfo**-derived interfaces only for public async methods of public ref classes.

Return IAsyncInfo-derived interfaces from public async methods of public ref classes

The compiler requires that any public, protected or public protected async methods of your app's public ref classes return an **IAsyncInfo** type. The actual interface must be a handle to one of the 4 interfaces that derive from the **IAsyncInfo** interface:

- **IAsyncAction^**
- **IAsyncOperation<TResult>^**
- **IAsyncActionWithProgress<TProgress>^**
- **IAsyncOperationWithProgress<TProgress, TResult>^**

In general, apps will expose public async methods in public ref classes only in cases of application binary interface (ABI) interop.

Use public ref classes only for interop

Use public ref classes for interop across the ABI. For example, you must use public ref classes for view model classes that expose properties for XAML data binding.

Don't declare classes that are internal to your app using the ref keyword. There are cases where this guidance doesn't apply, for example, when you are creating a private implementation of an interface that must be passed across the ABI. In this situation, you need a private ref class. Such cases are rare.

Similarly, we recommend that you use the data types in the **Platform** namespace such as **Platform::String** primarily in public ref classes and to communicate with Windows Runtime functions. You can use **std::wstring** and **wchar_t *** within your app and convert to a **Platform::String^** handle when you cross the ABI.

Use modern, standard C++, including the std namespace

Windows Store apps should use the latest C++ coding techniques, standards and libraries. Use a modern programming style, and include the data types operations of the std namespace. For example, Hilo uses the **std::shared_ptr** and **std::vector** types.

Only the outermost layer of your app needs to use C++/CX. This layer interacts with XAML across the ABI and perhaps with other languages such as JavaScript or C#.

Use task cancellation consistently

There are several ways to implement cancellation by using PPL tasks and **IAsyncInfo** objects. It's a good idea to choose one way and use it consistently throughout your app. A consistent approach to cancellation helps to avoid coding mistakes and makes code reviews easier. Your decision is a matter of taste, but here's what worked well for Hilo.

- We determine support for external token-based cancellation for each continuation chain and not for each task in a continuation chain. In other words, if the first task in a continuation supports external cancellation using a cancellation token, then all tasks in that chain, including nested tasks, support token-based cancellation. If the first task doesn't support token-based cancellation, then no task in the chain supports token-based cancellation.
- We create a **concurrency::cancellation_token_source** object whenever we start building a new continuation chain that needs to support cancellation that is initiated outside of the continuation chain itself. For example, in Hilo we use cancellation token source objects to allow pending async operations to be canceled when the user navigates away from the current page.
- We call the **get_token** method of the cancellation token source. We pass the cancellation token as an argument to the **create_task** function that creates the first task in the continuation chain. If there are nested continuations, we pass the cancellation token to the first task of each nested continuation chain. We don't pass a cancellation token to the **task::then** invocations that create continuations. Value-based continuations inherit the cancellation token of their antecedent task by default and don't need a cancellation token argument to be provided. Task-based continuations don't inherit cancellation tokens, but they generally perform cleanup actions that are required whether or not cancellation has occurred.
- When long-running tasks can be canceled, we call the **concurrency::is_task_cancellation_requested** function periodically to check if a request for cancellation is pending. Then, after performing cleanup actions, we call the **concurrency::cancel_current_task** function to exit the task and propagate the cancellation along the continuation chain.
- When an antecedent task uses alternative signaling techniques to indicate cancellation, such as returning **nullptr**, we call the **cancel_current_task** function in the continuation to propagate the cancellation along the continuation chain. Sometimes receiving a null pointer from the antecedent task is the only mechanism for cancelling a continuation chain. In other words, not all continuation chains can be canceled externally. For example, in Hilo, we sometimes cancel a continuation chain when the user selects the Cancel button of the file picker. When this happens the async file picking operation returns **nullptr.**

- We use a **try/catch** block in a task-based continuation whenever we need to detect whether a previous task in a continuation chain was canceled. In this situation, we catch the **concurrency::task_canceled** exception.
- There is a **try/catch** block in Hilo's **ObserveException** function object that prevents the **task_canceled** exception from being unobserved. This stops the app from being terminated when a cancellation occurs and no other specific handling for the exception exists.
- If a Hilo component creates a cancelable task on behalf of another part of the app, we make sure that the task-creating function accepts a **cancellation_token** as an argument. This allows the caller to specify the cancellation behavior.

Handle task exceptions using a task-based continuation

It's a good idea to have a continuation at the end of each continuation chain that catches exceptions with a final lambda. We recommend only catching exceptions that you know how to handle. Other exceptions will be caught by the runtime and cause the app to terminate using the "fail fast" **std::terminate** function.

In Hilo, every continuation chain ends with a continuation that calls Hilo's **ObserveException** function object. See Using task-based continuations for exception handling in this guide for more info and a code walkthrough.

Handle exceptions locally when using the when_all function

The **when_all** function returns immediately when any of its tasks throws an exception. If this happens, some of the tasks may still be running. When an exception occurs, your handler must wait for the pending tasks.

To avoid the complexity of having to test for exceptions and wait for tasks to complete, handle exceptions locally within the tasks that are managed by **when_all**.

For example, Hilo uses **when_all** for loading thumbnail images. The exceptions are handled in the tasks and not propagated back to **when_all**. Hilo passes a configurable exception policy object to the tasks to ensure that the tasks can use the app's global exception policy. Here's the code.

C++: ThumbnailGenerator.cpp

```
task<Vector<StorageFile^>^> ThumbnailGenerator::Generate(
    IVector<StorageFile^>^ files,
    StorageFolder^ thumbnailsFolder)
{
    vector<task<StorageFile^>> thumbnailTasks;

    unsigned int imageCounter = 0;
```

```
    for (auto imageFile : files)
    {
        wstringstream localFileName;
        localFileName << ThumbnailImagePrefix << imageCounter++ << ".jpg";

        thumbnailTasks.push_back(
            CreateLocalThumbnailAsync(
            thumbnailsFolder,
            imageFile,
            ref new String(localFileName.str().c_str()),
            ThumbnailSize,
            m_exceptionPolicy));
    }

    return when_all(begin(thumbnailTasks), end(thumbnailTasks)).then(
        [](vector<StorageFile^> files)
    {
        auto result = ref new Vector<StorageFile^>();
        for (auto file : files)
        {
            if (file != nullptr)
            {
                result->Append(file);
            }
        }

        return result;
    });
}
```

Here's the code for each task.

C++: ThumbnailGenerator.cpp

```
task<StorageFile^> ThumbnailGenerator::CreateLocalThumbnailAsync(
    StorageFolder^ folder,
    StorageFile^ imageFile,
    String^ localFileName,
    unsigned int thumbSize,
    std::shared_ptr<ExceptionPolicy> exceptionPolicy)
{
    auto createThumbnail = create_task(
        CreateThumbnailFromPictureFileAsync(imageFile, thumbSize));

    return createThumbnail.then([exceptionPolicy, folder, localFileName](
        task<InMemoryRandomAccessStream^> createdThumbnailTask)
    {
        InMemoryRandomAccessStream^ createdThumbnail;
```

```
        try
        {
            createdThumbnail = createdThumbnailTask.get();
        }
        catch(Exception^ ex)
        {
            exceptionPolicy->HandleException(ex);
            // If we have any exceptions we won't return the results
            // of this task, but instead nullptr.  Downstream
            // tasks will need to account for this.
            return create_task_from_result<StorageFile^>(nullptr);
        }

        return InternalSaveToFile(folder, createdThumbnail, localFileName);
    });
}
```

Call view model objects only from the main thread

Some methods and properties of view model objects are bound to XAML controls using data binding. When the UI invokes these methods and properties, it uses the main thread. Also, by convention, call all methods and properties of view model objects on the main thread.

Use background threads whenever possible

If the first task in a continuation chain wraps an asynchronous operation of the Windows Runtime, then by default the continuations run in the same thread as the **task::then** method. This is usually the main thread. This default is appropriate for operations that interact with XAML controls, but is not the right choice for many kinds of app-specific processing, such as applying image filters or performing other compute-intensive operations.

Be sure to use the value that the **task_continuation_context::use_arbitrary** function returns as an argument to the **task::then** method when you create continuations that should run in the background on thread pool threads.

Don't call blocking operations from the main thread

Don't call long-running operations in the main thread. In general, a good rule of thumb is to not call user-written functions that block the main thread for more than 50 milliseconds at a time. If you need to call a long-running function, wrap it in a task that runs in the background.

Note This tip concerns only the amount of time that you block the main thread, not the time that is required to complete an async operation. If an aysnc operation is very long, give the user a visual indication of its progress while keeping the main thread unblocked.

For more info see Keep the UI thread responsive (Windows Store apps using C#/VB/C++ and XAML).

Don't call task::wait from the main thread

Don't call the **task::wait** method from the main thread (or any STA thread). If you need to do something in the main thread after a PPL task completes, use the **task::then** method to create a continuation task.

Be aware of special context rules for continuations of tasks that wrap async objects

Windows Store apps using C++ use the **/ZW** compiler switch that affects the behavior of several functions. If you use these functions for desktop or console applications, be aware of the differences caused by whether you use the switch or not.

With the **/ZW** switch, when you invoke the **task::then** method on a task that wraps an **IAsyncInfo** object, the default run-time context of the continuation is the current context. The current context is the thread that invoked the then method. This default applies to all continuations in a continuation chain, not just the first one. This special default for Windows Store apps makes coding easier. In most cases continuations of Windows Runtime operations need to interact with objects that require you to run from the app's main thread. For an example see Ways to use the continuation chain pattern in this guide.

When a task doesn't wrap one of the **IAsyncInfo** interfaces, the then method produces a continuation that runs by default on a thread that the system chooses from the thread pool.

You can override the default behavior by passing an additional continuation context parameter to the **task::then** method. In Hilo, we always provide this optional parameter when the first task of the continuation chain has been created by a separate component or a helper method of the current component.

Note In Hilo, we found it useful to clarify our understanding of the thread context for our subroutines by using **assert(IsMainThread())** and **assert(IsBackgroundThread())** statements. In the Debug version of Hilo, these statements throw an exception if the thread being used is other than the one declared in the assertion. The implementations of the **IsMainThread** and **IsBackgroundThread** functions are in the Hilo source.

Be aware of special context rules for the create_async function

With the **/ZW** switch, the ppltasks.h header file provides the **concurrency::create_async** function that allows you to create tasks that are automatically wrapped by one of the **IAsyncInfo** interfaces of the Windows Runtime. The **create_task** and **create_async** functions have similar syntax and behavior, but there are some differences between them.

When you pass a work function as an argument to the **create_async** function, that work function can return **void**, an ordinary type **T**, **task<T>**, or a handle to any of the **IAsyncInfo**-derived interfaces such as **IAsyncAction^** and **IAsyncOperation<T>^**.

When you call the **create_async** function, the work function runs synchronously in the current context or asynchronously in the background, depending on the return type of the work function. If the return type of the work function is void or an ordinary type **T**, the work function runs in the background on a thread-pool thread. If the return type of the work function is a task or a handle to one of the interfaces that derive from **IAsyncInfo**, then the work function runs synchronously in the current thread. Running the work function synchronously is helpful in cases where it is a small function that does a small amount of set up before invoking another async function.

Note Unlike the **create_async** function, when you call the **create_task** function, it always runs in the background on a thread pool thread, regardless of the return type of the work function.

Note In Hilo, we found it useful to clarify our understanding of the thread context for our subroutines by using **assert(IsMainThread())** and **assert(IsBackgroundThread())** statements. In the Debug version of Hilo, these statements throw an exception if the thread being used is other than the one declared in the assertion. The implementations of the **IsMainThread** and **IsBackgroundThread** functions are in the Hilo source.

Be aware of app container requirements for parallel programming

You can use the PPL and the Asynchronous Agents Library in a Windows Store app, but you can't use the Concurrency Runtime's Task Scheduler or the Resource Manager components.

Use explicit capture for lambda expressions

It's a good idea to be explicit about the variables you capture in lambda expressions. For that reason we don't recommend that you use the [=] or [&] options for lambda expressions.

Also, be aware of the object lifetime when you capture variables in lambda expressions. To prevent memory errors, don't capture by reference any variables that contain stack-allocated objects. In addition, don't capture member variables of transient classes, for the same reason.

Don't create circular references between ref classes and lambda expressions

When you create a lambda expression and capture the **this** handle by value, the reference count of the **this** handle is incremented. If you later bind the newly created lambda expression to a member variable of the class referenced by **this**, you create a circular reference that can result in a memory leak.

Use *weak references* to capture the **this** reference when the resulting lambda would create a circular reference. For a code example see Weak references and breaking cycles (C++/CX).

Don't use unnecessary synchronization

The programming style of Windows Store apps can make some forms of synchronization, such as locks, needed less often. When multiple continuation chains are running at the same time, they can easily use the main thread as a synchronization point. For example, the member variables of view model objects are always accessed from the main thread, even in a highly asynchronous app, and therefore you don't need to synchronize them with locks.

In Hilo, we are careful to access member variables of view model classes only from the main thread.

Don't make concurrency too fine-grained

Use concurrent operations only for long-running tasks. There is a certain amount of overhead in setting up an asynchronous call.

Watch out for interactions between cancellation and exception handling

PPL implements some parts of its task cancellation functionality by using exceptions. If you catch exceptions that you don't know about, you may interfere with PPL's cancellation mechanism.

Note If you only use cancellation tokens to signal cancellation and don't call the **cancel_current_task** function, no exceptions will be used for cancellation.

Note If a continuation needs to check for cancellations or exceptions that occurred in tasks earlier in the continuation chain, make sure that the continuation is a task-based continuation.

Use parallel patterns

If your app performs heavy computations it is very likely that you need to use parallel programming techniques. There are a number of well-established patterns for effectively using multicore hardware. Parallel Programming with Microsoft Visual C++ is a resource for some of the most common patterns, with examples that use PPL and the Asynchronous Agents Library.

Be aware of special testing requirements for asynchronous operations

Unit tests require special handling for synchronization when there are async calls. Most testing frameworks don't allow tests to wait for asynchronous results. You need special coding to work around this issue.

In addition, when you test components that require some operations to occur in the main thread, you need a way to make sure the testing framework executes in the required thread context.

In Hilo we addressed both of these issues with custom testing code. See Testing the app in this guide for a description of what we did.

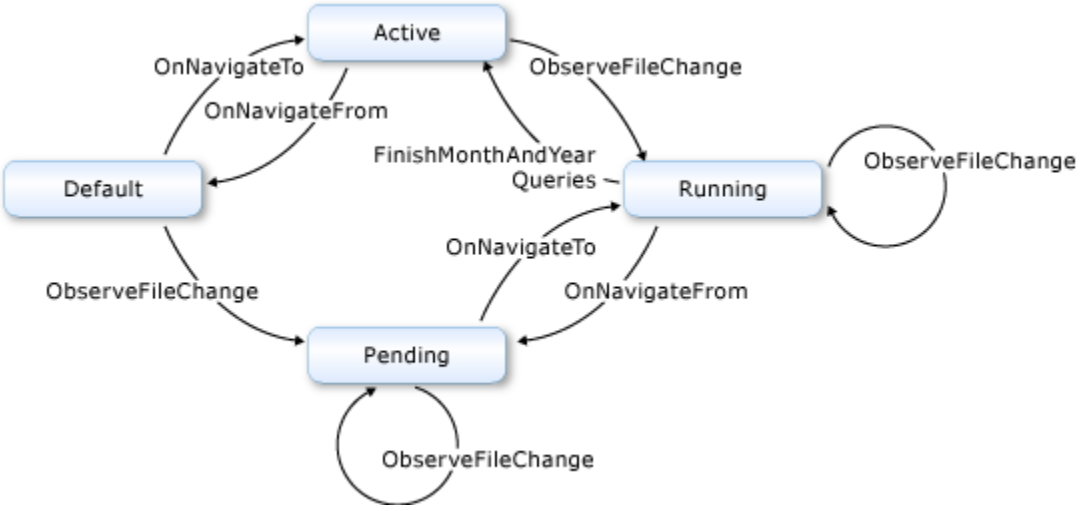
Use finite state machines to manage interleaved operations

Async programs have more possible execution paths and potential feature interactions than synchronous programs. For example, if your app allows the user to press the back button while an async operation such as loading a file is in progress, you must account for this in your app's logic. You might cancel the pending operation and restore the app's state to what it was before the operation started.

Allowing your app's operations to be interleaved in a flexible way helps increase the user's perception of the app's speed and responsiveness. It is an important characteristic of Windows Store apps, but if you're not careful it can be a source of bugs.

Correctly handling the many execution paths and interactions that are possible with an async UI requires special programming techniques. Finite state machines are a good choice that can help you implement robust handling of interleaved operations of async operations. With a finite state machine, you explicitly describe what happens in each possible situation.

Here is a diagram of a finite state machine from Hilo's image browser view model.



In the diagram, each oval represents a distinct operational mode that an instance of the **ImageBrowserViewModel** class can have at run time. The arcs are named transitions between modes. Some state transitions occur when an async operation finishes. You can recognize them because they contain the word Finish in their name. All of the others occur when an async operation starts.

The diagram shows what can happen in each operational mode. For example, the if the image browser is in the **Running** mode and an **OnNavigatedFrom** action occurs, then the resulting mode is **Pending**. Transitions are the result of

- user actions such as navigation requests
- program actions such as the completion of a previously started async operation
- external events such as notifications of network or file system events

Not all transitions are available in each mode. For example, the transition labeled **FinishMonthAndYearQueries** can occur only in the **Running** mode.

In the code, the named transitions are member functions of the **ImageBrowserViewModel** class. The image browser's current operational mode is the value of the **m_currentMode** member variable, which stores values of the **ImageBrowserViewModel::Mode** enumeration.

C++: ImageBrowserViewModel.h

```
enum class Mode {
    Default,        /* (0, 0, 0): no pending data changes, not updating, not visible
*/
    Active,         /* (0, 0, 1): no pending data changes, not updating, visible */
    Pending,        /* (1, 0, 0): pending data changes, not updating, not visible */
    Running,        /* (0, 1, 1): no pending data changes, updating, visible  */
    NotAllowed      /* error state */
};
```

Here is an example of the code that implements the **ObserveFileChange** transition in the diagram.

C++: ImageBrowserViewModel.cpp

```
// State transition occurs when file system changes invalidate the result of the
current query.
void ImageBrowserViewModel::ObserveFileChange()
{
    assert(IsMainThread());

    switch (m_currentMode)
    {
    case Mode::Default:
        m_currentMode = Mode::Pending;
        break;

    case Mode::Pending:
        m_currentMode = Mode::Pending;
        break;
```

```
    case Mode::Active:
        StartMonthAndYearQueries();
        m_currentMode = Mode::Running;
        break;

    case Mode::Running:
        CancelMonthAndYearQueries();
        StartMonthAndYearQueries();
        m_currentMode = Mode::Running;
        break;
    }
}
```

The **ObserveFileChange** method responds to operating system notifications of external changes to the image files that the user is viewing. There are 4 cases to consider, depending on what is happening in the app at the time of notification. For example, if the image browser is currently running an async query, its mode is **Running**. In this mode, the **ObserveFileChange** action cancels the running query because the results are no longer needed and then starts a new query. The operational mode remains **Running**.

Working with tiles and the splash screen in Hilo (Windows Store apps using C++ and XAML)

Summary

- Create tiles for your app early in development.
- Every app must have a square tile. Consider when to also enable the wide tile
- Use a tile template to update the contents of your app's tile.

Important APIs

- **TileUpdateManager**
- **TileUpdater**

Using tiles and the splash screen effectively can give your users a great first-impression of your Windows Store app using C++ and XAML.

You will learn

- How we incorporated an app tile that displays the user's photos.
- How we added the splash screen.
- What considerations to make in your own app.

Why are tiles important?

Traditional Windows desktop apps use icons. Icons help visually connect an app or file type with its brand or use. Because an icon is a static resource, you can often wait until the end of the development cycle to incorporate it. However, tiles are different from icons. Tiles add life and personality and can create a personal connection between the app and the user. The goal of tiles is to keep your users coming back by offering a personal connection.

Tip For Hilo, we knew early that we wanted to display the user's photos on the tile. But for other apps, it might not be apparent until later what content will keep your users coming back. We recommend that you add support for tile updates when you first create your project, even if you're not yet sure what the tile will show. When you decide later what to show on the tile, the infrastructure will already be in place. This way, you won't need to retrofit your code to support tile updates later.

Choosing a tile strategy

You have a few options when choosing a tile strategy. You can provide a wide tile, which your user can then change to a square tile if they prefer. You can also display badges and notifications on your tile.

Note Because Hilo is a photo app, we wanted to show the user's photos on the tile. You can show images on both the square and wide tile. The wide tile enables us to show multiple images, so we decided to support both. See Choosing between a square and wide tile size for info on how to choose the right tiles for your app.

We settled on the following rules for the tile behavior:

- Display wide default tile that shows the Hilo logo before the app is ever launched.
- Each time the app is launched, update the square and wide tiles according to these rules:
    - If the user has less than 5 pictures in the Pictures folder, display the default tile that shows the Hilo logo.
    - Otherwise, randomly choose 15 pictures from the most recent 30 and set up the notification queue to cycle among 3 batches of 5 pictures. (If the user chooses the square tile, it will show only the first picture from each batch.)

Read Guidelines and checklist for tiles to learn how tile features relate to the different tile styles. Although you can provide notifications to the square tile, we wanted to also enable the wide tile so that we could display multiple images.

Tip

You can also enable secondary tiles for your app. A secondary tile enables your users to pin specific content or experiences from an app to the Start screen to provide direct access to that content or experience. For more info about secondary tiles, read Pinning secondary tiles.

Designing the logo images

Our UX designer created the small, square, and wide logos according to the pixel size requirements for each. The designer suggested a theme that fitted the Hilo brand. Choosing a small logo that represents your app is important so that users can identify your app when the tile displays custom content. This is especially important when the contents of your tile changes frequently—you want your users to be able to easily find and identify your app. The small Hilo logo has a transparent background so that it looks good when it appears on top of a tile notification or other background.

The **Assets** folder contains the small, square, and wide logo images. For more info about working with image resources, see Quickstart: Using file or image resources and How to name resources using qualifiers.

30 x 30 pixels

150 x 150 pixels

310 x 150 pixels

Important Because the small logo appears on top of a tile notification, consider the color scheme that you use for the foreground color versus the background color. For Hilo, this decision was challenging because we display users' photos and cannot know what colors will appear. We experimented with several foreground colors and chose white because we felt it looked good when displayed over most pictures. (We also considered the fact that most photos do not have white as the dominant color.)

Note Image assets, including the Hilo logo, are placeholders and meant for training purposes only. They cannot be used as a trademark or for other commercial purposes.

Placing the logos on the default tiles

The Visual Studio manifest editor makes the process of adding the default tiles relatively easy. To learn how, read Quickstart: Creating a default tile using the Visual Studio manifest editor.

Updating tiles

You use tile templates to update the tiles. Tile templates are an XML-based approach to specify the images and text used to customize the tile. The **Windows::UI::Notifications** namespace provides classes to update Start screen tiles. For Hilo, we used the **TileUpdateManager** and **TileUpdater** classes to get the tile template and queue the notifications. Hilo defines the **TileUpdateScheduler** and

WideFiveImageTile classes to choose the images to show on the tile and form the XML that is provided to Windows Runtime.

Note Each tile template enables you to display images, text, or both. We chose **TileTemplateType::TileWideImageCollection** for the wide tile because it shows the greatest number of images. We also chose it because we did not need to display additional text on the tile. We also use **TileSquareImage** to display the first image in the user's collection when they choose to show the square tile. For the complete list of options, see **TileTemplateType**.

Tile updates occur in the **App::OnLaunched** method, which is called during app initialization.

C++

```
m_tileUpdateScheduler = std::make_shared<TileUpdateScheduler>();
m_tileUpdateScheduler->ScheduleUpdateAsync(m_repository, m_exceptionPolicy);
```

Note We considered updating the tiles when the app suspends, instead of when the app initializes, to make the updates more frequent. However, every Windows Store app should suspend as quickly as possible. Therefore, we did not want to introduce additional overhead when the app suspends. For more info, read Minimize suspend/resume time.

The **TileUpdateScheduler::ScheduleUpdateAsync** method performs the following steps to update the tile in the background:

- Create a local folder that will store a copy of the thumbnails to be displayed on the tile.
- If there are at least 30 photos in the user's collection:
  - Randomly select 15 of the user's 30 most recent photos.
  - Create 3 batches of 5 photos.
  - Generate thumbnails for the randomly selected photos in the local app folder.
- Create the notification and update the tile.

Verify your URLs describes the ways you can reference images that appear in your tile notification. We use ms-appdata:///local/ for Hilo because we copy thumbnails to a local app folder.

Note The number of bytes consumed by the thumbnails is small, so even if very large pictures are chosen to use on the tile, copying the thumbnails doesn't take much time or disk space.

The following example shows the **TileUpdateScheduler::ScheduleUpdateAsync** method. This code controls the process that creates the thumbnails folder, selects the images, and updates the tile. Each call to the Windows Runtime is asynchronous; therefore, we create a chain of continuation tasks that perform the update operations.

C++: TileUpdateScheduler.cpp

```
task<void> TileUpdateScheduler::ScheduleUpdateAsync(std::shared_ptr<Repository>
repository, std::shared_ptr<ExceptionPolicy> policy)
{
    // The storage folder that holds the thumbnails.
    auto thumbnailStorageFolder = make_shared<StorageFolder^>(nullptr);

    return create_task(
        // Create a folder to hold the thumbnails.
        // The ReplaceExisting option specifies to replace the contents of
        // any existing folder with a new, empty folder.
        ApplicationData::Current->LocalFolder->CreateFolderAsync(
            ThumbnailsFolderName,
            CreationCollisionOption::ReplaceExisting)).then([repository,
thumbnailStorageFolder](StorageFolder^ createdFolder)
    {
        assert(IsBackgroundThread());
        (*thumbnailStorageFolder) = createdFolder;

        // Collect a multiple of the batch and set size of the most recent photos
        // from the library.
        // Later a random set is selected from this collection for thumbnail image
generation.
        return repository->GetPhotoStorageFilesAsync("", 2 * BatchSize * SetSize);
    }, task_continuation_context::use_arbitrary())
        .then([](IVectorView<StorageFile^>^ files) -> task<IVector<StorageFile^>^>
    {
        assert(IsBackgroundThread());
        // If we received fewer than the number in one batch,
        // return the empty collection.
        if (files->Size < BatchSize)
        {
            return create_task_from_result(static_cast<IVector<StorageFile^>^>(
                ref new Vector<StorageFile^>()));
        }
        auto copiedFileInfos = ref new Vector<StorageFile^>(begin(files),
end(files));
        return RandomPhotoSelector::SelectFilesAsync(copiedFileInfos->GetView(),
SetSize * BatchSize);
```

```
    }, task_continuation_context::use_arbitrary()).then([this,
thumbnailStorageFolder, policy](IVector<StorageFile^>^ selectedFiles) ->
task<Vector<StorageFile^>^>
    {
        assert(IsBackgroundThread());
        // Return the empty collection if the previous step did not
        // produce enough photos.
        if (selectedFiles->Size == 0)
        {
            return create_task_from_result(ref new Vector<StorageFile^>());
        }
        ThumbnailGenerator thumbnailGenerator(policy);
        return thumbnailGenerator.Generate(selectedFiles, *thumbnailStorageFolder);
    }, task_continuation_context::use_arbitrary())
        .then([this](Vector<StorageFile^>^ files)
    {
        assert(IsBackgroundThread());
        // Update the tile.
        UpdateTile(files);
    }, concurrency::task_continuation_context::use_arbitrary())
        .then(ObserveException<void>(policy));
}
```

Note We considered multiple options for how the pictures are chosen. Some alternatives we considered were to choose the most recent pictures or enable the user to select them. We went with randomly choosing 15 of the most recent 30 to get both variety and recent pictures. We felt that having users choose the pictures would be inconvenient and might not entice them to come back to the app later.

To create a thumbnail image, the **ThumbnailGenerator::CreateThumbnailFromPictureFileAsync** method gets the thumbnail from the image, decodes it, and then encodes it as a .jpg image. Because tile notifications only support the .jpg/.jpeg, .png, and .gif image formats, we re-encode each image to enable the app to use the .bmp, and .tiff image formats. We chose .jpg as the target format because it produces the smallest images, while still providing the desired image quality.

C++: ThumbnailGenerator.cpp

```
task<InMemoryRandomAccessStream^>
ThumbnailGenerator::CreateThumbnailFromPictureFileAsync(
    StorageFile^ sourceFile,
    unsigned int thumbSize)
{
    (void)thumbSize; // Unused parameter
    auto decoder = make_shared<BitmapDecoder^>(nullptr);
    auto pixelProvider = make_shared<PixelDataProvider^>(nullptr);
    auto resizedImageStream = ref new InMemoryRandomAccessStream();
```

```
    auto createThumbnail = create_task(
        sourceFile->GetThumbnailAsync(
        ThumbnailMode::PicturesView,
        ThumbnailSize));

    return createThumbnail.then([](StorageItemThumbnail^ thumbnail)
    {
        IRandomAccessStream^ imageFileStream =
            static_cast<IRandomAccessStream^>(thumbnail);

        return BitmapDecoder::CreateAsync(imageFileStream);

    }).then([decoder](BitmapDecoder^ createdDecoder)
    {
        (*decoder) = createdDecoder;
        return createdDecoder->GetPixelDataAsync(
            BitmapPixelFormat::Rgba8,
            BitmapAlphaMode::Straight,
            ref new BitmapTransform(),
            ExifOrientationMode::IgnoreExifOrientation,
            ColorManagementMode::ColorManageToSRgb);

    }).then([pixelProvider, resizedImageStream](PixelDataProvider^ provider)
    {
        (*pixelProvider) = provider;
        return BitmapEncoder::CreateAsync(
            BitmapEncoder::JpegEncoderId,
            resizedImageStream);

    }).then([pixelProvider, decoder](BitmapEncoder^ createdEncoder)
    {
        createdEncoder->SetPixelData(BitmapPixelFormat::Rgba8,
            BitmapAlphaMode::Straight,
            (*decoder)->PixelWidth,
            (*decoder)->PixelHeight,
            (*decoder)->DpiX,
            (*decoder)->DpiY,
            (*pixelProvider)->DetachPixelData());
        return createdEncoder->FlushAsync();

    }).then([resizedImageStream]
    {
        resizedImageStream->Seek(0);
        return resizedImageStream;
    });
}
```

The **TileUpdateScheduler::UpdateTile** method uses the **TileUpdateManager** class to create a **TileUpdater** object. The **TileUpdater** class updates the content of the app's tile. The **TileUpdateScheduler::UpdateTile** method calls the **TileUpdater::EnableNotificationQueue** method to queue notifications for each batch. The **TileUpdateScheduler::UpdateTile** method then builds a list of image paths for each batch of pictures and passes that list to a **WideFiveImageTile** object. The **WideFiveImageTile** object formats the XML for the tile update and for each batch of pictures, and then the **TileUpdateScheduler::UpdateTile** method calls the **TileUpdater::Update** method to update the tile.

C++: ThumbnailGenerator.cpp

```
void TileUpdateScheduler::UpdateTile(IVector<StorageFile^>^ files)
{
    // Create a tile updater.
    TileUpdater^ tileUpdater = TileUpdateManager::CreateTileUpdaterForApplication();
    tileUpdater->Clear();

    unsigned int imagesCount = files->Size;
    unsigned int imageBatches = imagesCount / BatchSize;

    tileUpdater->EnableNotificationQueue(imageBatches > 0);

    for(unsigned int batch = 0; batch < imageBatches; batch++)
    {
        vector<wstring> imageList;

        // Add the selected images to the wide tile template.
        for(unsigned int image = 0; image < BatchSize; image++)
        {
            StorageFile^ file = files->GetAt(image + (batch * BatchSize));
            wstringstream imageSource;
            imageSource << L"ms-appdata:///local/"
                << ThumbnailsFolderName->Data()
                << L"/"
                << file->Name->Data();
            imageList.push_back(imageSource.str());
        }

        WideFiveImageTile wideTile;
        wideTile.SetImageFilePaths(imageList);

        // Create the notification and update the tile.
        auto notification = wideTile.GetTileNotification();
        tileUpdater->Update(notification);
    }
}
```

The **WideFiveImageTile** class, which is defined in WideFiveImageTile.cpp, encapsulates the creation of the XML for the tile update. It builds upon the **TileTemplateType::TileWideImageCollection** tile template by inserting the provided list of file names into the template's XML content. It also uses the **TileSquareImage** tile template to show just the first picture if the user chooses the square tile. The **WideFiveImageTile** class then creates a **TileNotification** object using the updated XML content.

This code example shows how the **WideFiveImageTile::UpdateContentWithValues** method updates the template's XML content.

C++: WideFiveImageTile.cpp

```
void WideFiveImageTile::UpdateContentWithValues(XmlDocument^ content)
{
    if (m_fileNames.size() == 0) return;

    // Update wide tile template with the selected images.
    for(unsigned int image = 0; image < m_fileNames.size(); image++)
    {
        IXmlNode^ tileImage = content->GetElementsByTagName("image")->GetAt(image);
        tileImage->Attributes->GetNamedItem("src")->InnerText = ref new String(
            m_fileNames[image].c_str());
    }

    // Update square tile template with the first image.
    TileTemplateType squareTileTemplate = TileTemplateType::TileSquareImage;
    XmlDocument^ squareTileXml =
TileUpdateManager::GetTemplateContent(squareTileTemplate);

    IXmlNode^ tileImage = squareTileXml->GetElementsByTagName("image")->First()-
>Current;
    tileImage->Attributes->GetNamedItem("src")->InnerText = ref new String(
        m_fileNames[0].c_str());

    auto node = content->ImportNode(squareTileXml->GetElementsByTagName("binding")-
>First()->Current, true);
    content->GetElementsByTagName("visual")->First()->Current->AppendChild(node);
}
```

The XML for a typical tile notification looks like this:

XML

```
<tile>
  <visual>
    <binding template="TileWideImageCollection">
      <image id="1" src="ms-appdata:///local/thumbnails/thumbImage_0.jpg"/>
      <image id="2" src="ms-appdata:///local/thumbnails/thumbImage_1.jpg"/>
```

```
      <image id="3" src="ms-appdata:///local/thumbnails/thumbImage_2.jpg"/>
      <image id="4" src="ms-appdata:///local/thumbnails/thumbImage_3.jpg"/>
      <image id="5" src="ms-appdata:///local/thumbnails/thumbImage_4.jpg"/>
    </binding>
  <binding template="TileSquareImage">
      <image id="1" src="ms-appdata:///local/thumbnails/thumbImage_0.jpg"/>
    </binding>
  </visual>
</tile>
```

Note The wide tile template also contains a template for the square tile. This way, both the square and wide tiles are covered by a single template.

If you use text with your tile, or your images are sensitive to different languages and cultures, read Globalizing tile and toast notifications to learn how to globalize your tile notifications.

Use these resources to learn more about tile templates:

- The tile template catalog
- Using tile update templates
- Tile schema

Adding the splash screen

All Windows Store apps must have a splash screen, which is a composite of a splash screen image and a background color, both of which you can customize. The splash screen is shown as your app loads. As with tiles, our designer created the splash screen image. We chose an image that resembled the default tile logos and fits the Hilo brand. It was straight forward to add the splash screen to the app. Read Quickstart: Adding a splash screen to learn how.

You might need to display the splash screen for a longer duration, to show real-time loading information to your users, if your app needs more time to load. This involves creating a page that mimics the splash screen by showing the splash screen image and any additional info. For Hilo, we considered using an extended splash screen, but because the app loads the hub page very quickly, we didn't have to add it. For more info about extending the duration of the splash screen, see How to extend the splash screen.

Use these resources to learn more about splash screens:

- Adding a splash screen
- Guidelines and checklist for splash screens

Using the Model-View-ViewModel (MVVM) pattern in Hilo (Windows Store apps using C++ and XAML)

Summary

- Use the MVVM architecture pattern to make your app easier to maintain and test.
- Let your app's MVVM components communicate asynchronously for best performance.

Important APIs

- **Binding**
- **ICommand**
- **INotifyPropertyChanged**

Early in the project we decided to adopt the Model-View-ViewModel (MVVM) pattern for Hilo's architecture. We were attracted by the fact that the MVVM pattern makes it easier to maintain and test your Windows Store app using C++ and XAML, especially as it grows. MVVM is a relatively new pattern for C++ apps.

You will learn

- How Windows Store apps can benefit from MVVM.
- Recommended techniques for applying the MVVM pattern to Windows Store apps.
- How to map views to UI elements.
- How to share view models across views.
- How to execute commands in a view model.

What is MVVM?

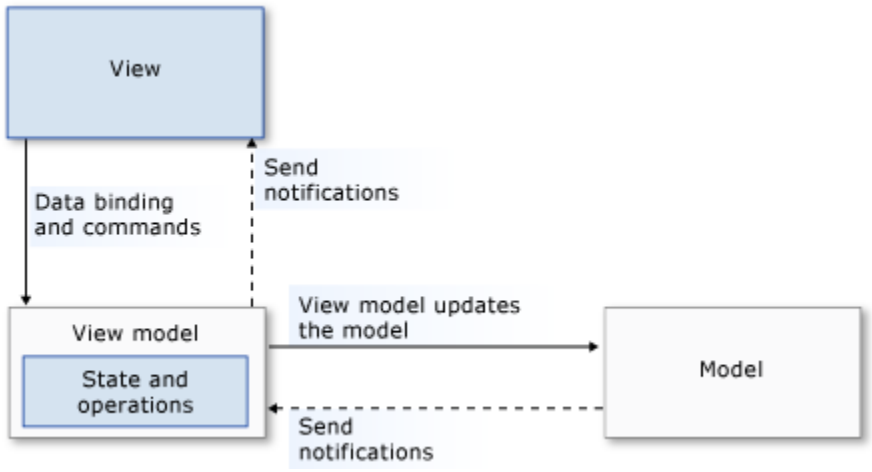
MVVM is an architectural pattern. It is a specialization of the presentation model pattern that was introduced by Martin Fowler. It is also related to the model-view-controller pattern (MVC) and the model view presenter (MVP) pattern that you may already know.

An app that uses MVVM separates business logic, UI, and presentation behavior.

- **Models** represent the state and operations of business objects that your app manipulates. For example, Hilo reads and modifies image files, so it makes sense that data types for image files and operations on image files are part of Hilo’s model.
- **Views** contain UI elements, and they include any code that implements the app’s user experience. A view defines the structure, layout, and appearance of what the user sees on the screen. Grids, pages, buttons, and text boxes are examples of the elements that view objects manage.

- **View models** encapsulate the app's state, actions, and operations. A view model serves as the decoupling layer between the model and the view. It provides the data in a format that the view can consume and updates the model so that the view does not need to interact with the model. View models respond to commands and trigger events. They also act as data sources for any data that views display. View models are built specifically to support a view. You can think of a view model as the app, minus the UI. In Windows Store apps, you can declaratively bind views to their corresponding view models.

Here are the relationships between a view, a view model and a model.

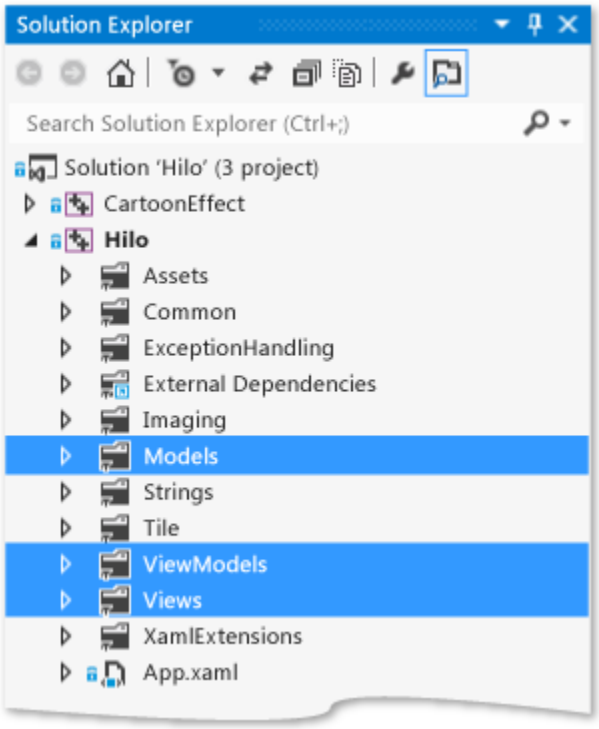


MVVM in Hilo

In Hilo, there is one view class per page of the UI. (A page is an instance of the **Windows::UI::Xaml::Controls::Page** class.) Each view has a corresponding view model class. All of the view models in Hilo share the app's domain model, which is often just called the model. The model consists of classes that the view models use to implement the app's functionality.

Note If you want to jump directly to a walkthrough of the Hilo's view and view model code, see Creating and navigating pages in Hilo.

In the Visual Studio solution Hilo.sln there are solution folders that are named for each of the MVVM layers.



- The Models folder contains .cpp (C++) and .h (C++ header) files that make up the Hilo model.
- The Views folder contains the UI classes and XAML files.
- The ViewModels folder contains .cpp and .h files for the app's view model classes.

With the MVVM pattern, XAML data binding lets a view model act as a page's data context. A data context is responsible for providing properties that provide the view with data for the UI elements on the page.

Note You don't have to use data binding to connect views and view models. It's also possible to use a code-behind file containing C++ code that is associated with page classes. You can recognize code-behind files because they use the .xaml.cpp suffix. For example, in Hilo the file MainHubView.xaml.cpp is the code-behind file for the page that the MainHubView.xaml file defines. Many visual design tools such as Microsoft Expression are optimized for use with data binding.

View models connect to the app's underlying model by calling instance methods. You need no special binding to make these calls. If you want a strong separation between the model and the app's view models, you can package model classes in a separate library. Hilo doesn't use a separate library for its model. Instead, it just keeps the files that define model classes in a separate folder in the Hilo Visual Studio project.

Why use MVVM for Hilo?

There are two main implementation approaches for a UI: using a code-behind file for the presentation logic, or separating UI structure and presentation logic with a pattern such as the MVVM pattern. After analyzing our needs, we chose the MVVM approach for Hilo because:

- We wanted to test our presentation logic. MVVM makes it easy to cleanly separate view logic from UI controls, which is important for test automation.
- We wanted to make sure that the view and presentation logic could evolve independently and reduce dependencies between UX designers and developers. MVVM, used with XAML's data binding, makes this possible.

For more info

You can find more info about MVVM online. Here are some examples in managed code, but the concepts also apply to C++:

- MVVM Quickstart.
- Implementing the MVVM Pattern and Advanced MVVM Scenarios in Prism.
- Developing a Windows Phone Application using the MVVM Pattern.

Variations of the MVVM pattern

You can customize the MVVM pattern in various ways. Let's look at some of them.

- Mapping views to UI elements other than pages
- Sharing view models among multiple views
- Executing commands in a view model
- Using a view model locator object to bind views to view models

Mapping views to UI elements other than pages

In Hilo, each page class is a MVVM view object, and all MVVM views are pages. But you don't have to do the same. For example, a view could be a **DataTemplate** for an object in an **ItemsControl.**

Sharing view models among multiple views

A view can have its own view model, or it can share the view model of another view. The choice depends on whether views share a lot of common functionality. In Hilo, each view is associated with a unique view model for simplicity.

Executing commands in a view model

You can use data binding for buttons and other UI controls that cause the app to perform operations. If the control is a Command Source, the control's **Command** property is data-bound to an **ICommand** property on the view model. When the control's command is invoked, the code in the view model is executed. We recommend that you use data binding for commands when you use MVVM.

Here is an example of executing a command from Hilo. The rotate image page contains a UI element for the Save File button. This XAML code comes from the RotateImageView.xaml file.

XAML: RotateImageView.xaml

```
<Button x:Name="SaveButton"
        x:Uid="AcceptAppBarButton"
        Command="{Binding SaveCommand}"
        Style="{StaticResource AcceptAppBarButtonStyle}"
        Tag="Save" />
```

The expression "Command={Binding SaveCommand}" creates a binding between the **Command** property of the button and the **SaveCommand** property of the **RotateImageViewModel** class. The **SaveCommand** property contains a handle to an **ICommand** object. The next code comes from the RotateImageViewModel.cpp file.

C++: RotateImageViewModel.cpp

```
ICommand^ RotateImageViewModel::SaveCommand::get()
{
    return m_saveCommand;
}
```

The **m_saveCommand** member variable is initialized in the constructor of the **RotateImageViewModel** class. Here is the code from the RotateImageViewModel.cpp file.

C++: RotateImageViewModel.cpp

```
m_saveCommand = ref new DelegateCommand(ref new ExecuteDelegate(this,
&RotateImageViewModel::SaveImage), nullptr);
```

Hilo's **DelegateCommand** class is an implementation of the **ICommand** interface. The class defines the **ExecuteDelegate** delegate type. Delegates let you use a pointer to a C++ member function as a callable Windows Runtime object. Hilo invokes the **ExecuteDelegate** when the UI triggers the command.

Note For more info about the delegate language extension in C++/CX, see Delegates (C++/CX).

Because we used data binding, changing the action of the save command is a matter of assigning a different **DelegateCommand** object to the **m_saveCommand** member variable. There's no need to change the view's XAML file.

In this example, the underlying member function comes from the RotateImageViewModel.cpp file.

C++: RotateImageViewModel.cpp

```
void RotateImageViewModel::SaveImage(Object^ parameter)
{
   // Asynchronously save image file
}
```

Using a view model locator object to bind views to view models

In Hilo, each view (page) has a corresponding view model.

If you use MVVM, the app needs to connect its views to its view models. This means that each view must have a view model assigned to its **DataContext** property. For Hilo, we used a single **ViewModelLocator** class because we needed set up code to execute before UI elements were bound to the view model. The **ViewModelLocator** class has properties that retrieve a view model object for each page of the app. See Creating and navigating between pages for a description of how the **ViewModelLocator** class binds views and view models in Hilo.

You don't have to use a view model locator class. In fact, there are several ways to bind a view to its corresponding view model object. If you don’t use a view model locator class, you can connect the creation and destruction of view model instances to the lifetime of the corresponding view object. For example, you could create a new view model instance every time the page loads.

You can also connect views to view models in a code-behind file. The code in a code-behind file can instantiate a new view model instance and assign it to the view’s **DataContext** property. You can instantiate the view model in the page's **Initialize** method or in its **OnNavigatedTo** method.

Tips for designing Windows Store apps using MVVM

Here are some tips for applying the MVVM pattern to Windows Store apps in C++.

- Keep view dependencies out of the view model.
- Centralize data conversions in the view model or a conversion layer.
- Expose operational modes in the view model.
- Ensure that view models have the Bindable attribute.
- Ensure that view models implement the INotifyProperyChanged interface for data binding to work.

- Keep views and view models independent.
- Use asynchronous programming techniques to keep the UI responsive.
- Always observe threading rules for Windows Runtime objects.

Keep view dependencies out of the view model

When designing a Windows Store app with the MVVM pattern, you need to decide what is placed in the model, what is placed in the views, and what is placed in the view models. This division is often a matter of taste, but some general principles apply. Ideally, you define the view with XAML, with only limited code-behind that doesn't contain business logic. We also recommend that you keep the view model free of dependencies on the data types for UI elements or views. Don't include view header files in view model source files.

Centralize data conversions in the view model or a conversion layer

The view model provides data from the model in a form that the view can easily use. To do this the view model sometimes has to perform data conversion. Placing this data conversion in the view model is a good idea because it provides properties in a form that the UI can bind to.

It is also possible to have a separate data conversion layer that sits between the view model and the view. This might occur, for example, when data types need special formatting that the view model doesn’t provide.

Expose operational modes in the view model

The view model may also be responsible for defining logical state changes that affect some aspect of the display in the view, such as an indication that some operation is pending or whether a particular command is available. You don't need code-behind to enable and disable UI elements—you can achieve this by binding to a view model property.

Here's an example.

XAML

```
<Grid Background="{Binding HasPhotos, Converter={StaticResource BrushConverter}}"
      Height="150"
      IsTapEnabled="{Binding HasPhotos}"
      PointerEntered="OnZoomedOutGridPointerEntered"
      Margin="0"
      Width="150">
```

In the example the **IsTapEnabled** attribute is bound to the view model's **HasPhoto** property.

Ensure that view models have the Bindable attribute

For the view model to participate in data binding with the view, view model classes must have the **Windows::UI::Xaml::Data::Bindable** attribute to ensure that the type is included in XAML's generated file.

In addition, you also need to include the header for your view model in an App.xaml.h header file, either directly or indirectly. In Hilo all the view models header files are included in the ViewModelLocator.h file, which is included in the App.xaml.h file. This ensures that the types necessary to work with XAML get properly generated at compile time.

Note For more info about the attribute language extension of C++/CX, see User-defined attributes (C++/CX).

Here's is an example of the **Bindable** attribute.

C++

```
[Windows::UI::Xaml::Data::Bindable]
[Windows::Foundation::Metadata::WebHostHiddenAttribute]
public ref class MainHubViewModel sealed : public ViewModelBase
{
  // ...
}
```

Ensure that view models implement the INotifyProperyChanged interface for data binding to work

View models that need to notify clients that a property value has changed must raise the **PropertyChanged** event. To do this, view model classes need to implement the **Windows::UI::Xaml::Data::INotifyPropertyChanged** interface. The Windows Runtime registers a handler for this event when the app runs. Visual Studio provides an implementation of the **INotifyPropertyChanged** interface in the **BindableBase** template class that you can use as a base class for any XAML data source. Here is the generated header file, from BindableBase.h:

C++: BindableBase.h

```
[Windows::Foundation::Metadata::WebHostHidden]
public ref class BindableBase : Windows::UI::Xaml::DependencyObject,
Windows::UI::Xaml::Data::INotifyPropertyChanged,
Windows::UI::Xaml::Data::ICustomPropertyProvider
{
  public:
    virtual event Windows::UI::Xaml::Data::PropertyChangedEventHandler^
```

```
PropertyChanged;

   // ...

  protected:
    virtual void OnPropertyChanged(Platform::String^ propertyName);
};
```

The generated implementation invokes the handler when the event is raised.

C++

```
void BindableBase::OnPropertyChanged(String^ propertyName)
{
    PropertyChanged(this, ref new PropertyChangedEventArgs(propertyName));
}
```

Note C++/CX has events and properties as part of the programming language. It includes the event and property keywords. Windows Runtime types declare events in their public interfaces that your app can subscribe to. The subscriber performs custom actions when the publisher fires the event. For more info about C++/CX features that support the Windows Runtime, see Creating Windows Runtime Components in C++. For more info about the event language extension of C++/CX that is used in this code example, see Events (C++/CX).

View model classes can inherit the **INotifyPropertyChanged** implementation by deriving from the **BindableBase** class. For example, here is the declaration of the **ViewModelBase** class in Hilo. The code comes from the ViewModelBase.h file.

C++: ViewModelBase.h

```
public ref class ViewModelBase : public Common::BindableBase
{
  // ...
}
```

Whenever view models need to tell the UI that a bound property has changed, they call the **OnPropertyChanged** method that they inherited from the **BindableBase** class. For example, here is a property set method defined in the **RotateImageViewModel** class.

C++

```
void RotateImageViewModel::RotationAngle::set(float64 value)
{
    m_rotationAngle = value;
```

```
    // Derive margin so that rotated image is always fully shown on screen.
    Thickness margin(0.0);
    switch (safe_cast<unsigned int>(m_rotationAngle))
    {
    case 90:
    case 270:
        margin.Top = 110.0;
        margin.Bottom = 110.0;
        break;
    }
    m_imageMargin = margin;
    OnPropertyChanged("ImageMargin");
    OnPropertyChanged("RotationAngle");
}
```

Note Property notifications to XAML must occur in the UI thread. This means that the **OnPropertyChanged** method and any of its callers must also occur in the app's UI thread. In general, a useful convention is that all view model methods and properties must be called in the app's UI thread.

Keep views and view models independent

If you follow the principles we outlined here, you will be able to re-implement a view model without any change to the view. The binding of views to a particular property in its data source should be a view's principal dependency on its corresponding view model. (If you rename a bound property in the view model you need to rename it in the XAML data binding expression as well.)

Use asynchronous programming techniques to keep the UI responsive

Windows Store apps are about a fast and fluid user experience. For that reason Hilo keeps the UI thread unblocked. Hilo uses asynchronous library methods for I/O operations and parallel tasks when operations perform a significant amount of computation. Hilo raises events to asynchronously notify the view of a property change.

See Asynchronous programming for Windows Store apps using C++ and XAML for more info.

Always observe threading rules for Windows Runtime objects

Objects that are created by calls into the Windows Runtime are sometimes single-threaded. This means that you must invoke the methods, properties and event handlers from the thread context that was used to create the object. In most cases the context is the app's UI thread.

To avoid errors, Hilo was designed so that calls into its view models occur on the app's UI thread. (Model classes perform time-consuming operations such as image processing on worker threads.)

See Programming patterns for asynchronous UI for more info. See Controlling the Execution Thread for info about the threading model that Windows Store apps use.

For a walkthrough of Hilo's use of asynchronous programming, see Asynchronous programming for Windows Store apps using C++ and XAML.

Using the Repository pattern in Hilo (Windows Store apps using C++ and XAML)

Summary

- Use the Repository pattern to make it easier to write unit tests and maintain your app over time.
- Access the file system using the Repository pattern.

For Hilo C++ we decided to use the Repository pattern to separate the logic that retrieves data from the Windows 8 file system from the presentation logic that acts on that data. We wanted to make the app easier to maintain and unit test which is why we selected the Repository pattern.

You will learn

- How to implement the Repository pattern for a Windows Store app using C++.
- How to use the Repository pattern for unit testing your app.

Introduction

The Repository pattern separates the logic that retrieves and persists data and maps it to the underlying model, from the business logic that acts upon that data. The repository is responsible for:

- Mediating between the data source and the business layers of the app.
- Querying the data source for data.
- Mapping the data from the data source to the model.
- Persisting changes in the model to the data source.

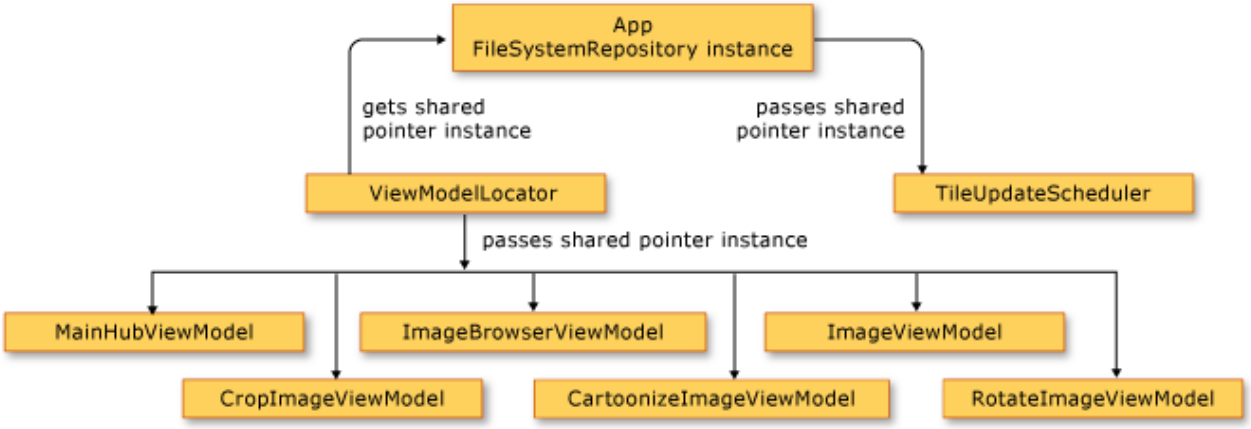
The separation of business logic from the data source offers these benefits:

- Centralizing access to the underlying data source via a data access layer.
- Isolating the data access layer to support unit testing.
- Improving the code's maintainability by separating business logic from data access logic.

For more info, see The Repository Pattern.

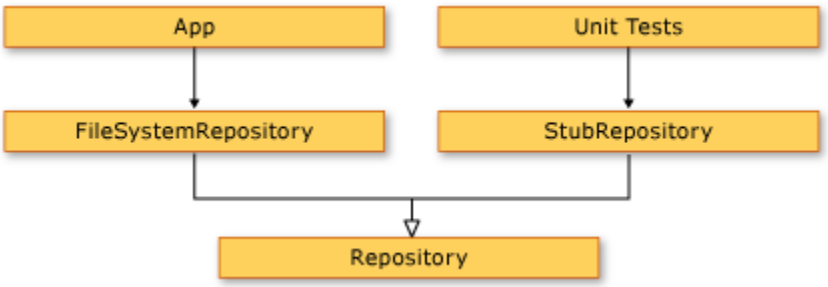
Hilo implements the Repository pattern by defining an abstract base class that has pure virtual member functions. These functions provide a base class from which other repository classes must inherit. A pure virtual member function is a member function that is defined as virtual and is assigned to 0. Such an abstract base class can't be used to instantiate objects and serves only to act as an interface. Therefore, if a subclass of this abstract class needs to be instantiated, it has to implement each of the virtual functions, which results in the subclass supporting the interface defined by the abstract base class.

The **Repository** abstract base class defines a number of pure virtual member functions that must be overridden by member functions that have the same signatures in a derived class. The **FileSystemRepository** class derives from the **Repository** class and overrides the pure virtual member functions to query the file system for photos. A shared instance of the **FileSystemRepository** class is then used by the **TileUpdateScheduler** class and view model classes to read photos from the file system. The following illustration shows this relationship.



The **StubRepository** class also implements the pure virtual member functions defined in the **Repository** abstract base class, in order to provide a mock repository implementation for unit testing. An instance of the **StubRepository** class can then be passed into the view model classes from unit tests. Rather than return photos from the file system, this mock class simply returns mock photo objects. For more info about unit testing, see Testing the app.

The following illustration shows an overview of the repository architecture. The main advantage of this architecture is that a repository that accesses a data source, such as the file system, can easily be swapped out for a repository that accesses a different data source, such as the cloud.



In order to implement a new repository, which derives from the **Repository** abstract base class, you must also implement the **IPhoto**, **IPhotoGroup**, **IYearGroup**, and **IMonthBlock** interfaces. These interfaces are implemented in Hilo by the **Photo**, **HubPhotoGroup**, **MonthGroup**, **YearGroup**, and **MonthBlock** classes, respectively.

In order to explain how Hilo implements and uses the **Repository** class, we've provided a code walkthrough that demonstrates how the Rotate page retrieves a photo from the file system by using the **Repository** class.

Code walkthrough

In Hilo, the **App** class contains a member variable, **m_repository**, of type **Repository**, which is instantiated as a shared pointer of type **FileSystemRepository** in the class constructor. This instance is created as shared pointer so that there's only a single instance of the **FileSystemRepository** class in the application, which is passed between the required classes.

C++: App.xaml.cpp

```
m_exceptionPolicy = ExceptionPolicyFactory::GetCurrentPolicy();
m_repository = std::make_shared<FileSystemRepository>(m_exceptionPolicy);
```

The **OnLaunched** method of the **App** class passes the shared pointer instance of the **FileSystemRepository** to the **ScheduleUpdateAsync** method of the **TileUpdateScheduler** class. The **ScheduleUpdateAsync** method uses this shared pointer instance to retrieve photos, from which thumbnails are generated for the app's live tile.

The **App** class exposes a singleton instance of the **FileSystemRepository** class through a **GetRepository** method, which is invoked by the **ViewModelLocator** class constructor to retrieve and store the shared pointer instance of the **FileSystemRepository** class.

C++: ViewModelLocator.cpp

```
m_repository = safe_cast<Hilo::App^>(Windows::UI::Xaml::Application::Current)-
>GetRepository();
```

As both the **ViewModelLocator** and **TileUpdateScheduler** classes expect repositories of type **shared_ptr<Repository>**, any additional repository implementations that implement the **Repository** abstract base class can be used instead.

The **ViewModelLocator** class has properties that retrieve a view model object for each page of the app. For more info, see Using the MVVM pattern. The following code example shows the **RotateImageVM** property of the **ViewModelLocator** class.

C++: ViewModelLocator.cpp

```
RotateImageViewModel^ ViewModelLocator::RotateImageVM::get()
{
    return ref new RotateImageViewModel(m_repository, m_exceptionPolicy);
```

```
}
```

The **RotateImageVM** property creates a new instance of the **RotateImageViewModel** class and passes in the shared pointer instance of the **FileSystemRepository** class, to the **RotateImageViewModel** constructor. The shared pointer instance of the **FileSystemRepository** class is then stored in the **m_repository** member variable of the **RotateImageViewModel** class, as shown in the following code example.

C++: RotateImageViewModel.cpp

```
RotateImageViewModel::RotateImageViewModel(shared_ptr<Repository> repository,
shared_ptr<ExceptionPolicy> exceptionPolicy) :
    ImageBase(exceptionPolicy), m_repository(repository),
m_imageMargin(Thickness(0.0)), m_getPhotoAsyncIsRunning(false),
    m_inProgress(false), m_isSaving(false), m_rotationAngle(0.0)
```

The **Image** control in the **RotateImageView** class binds to the **Photo** property of the **RotateImageViewModel** class. In turn, the **Photo** property invokes the **GetImagePhotoAsync** method to retrieve the photo for display, as shown in the following code example.

C++: RotateImageViewModel.cpp

```
concurrency::task<IPhoto^> RotateImageViewModel::GetImagePhotoAsync()
{
    assert(IsMainThread());
    return m_repository->GetSinglePhotoAsync(m_photoPath);
}
```

GetImagePhotoAsync invokes **GetSinglePhotoAsync** on the shared pointer instance of the **FileSystemRepository** class that is stored in the **m_repository** member variable.

Querying the file system

Hilo uses the **FileSystemRepository** class to query the Pictures library for photos. The view models use this class to provide photos for display.

For instance, the **RotateImageViewModel** class uses the **GetImagePhotoAsync** method to return a photo for display on the rotate page. This method, in turn, invokes the **GetSinglePhotoAsync** method in the **FileSystemRepository**. Here's the code.

C++: FileSystemRepository.cpp

```
task<IPhoto^> FileSystemRepository::GetSinglePhotoAsync(String^ photoPath)
```

```
{
    String^ query = "System.ParsingPath:=\"" + photoPath + "\"";
    auto fileQuery = CreateFileQuery(KnownFolders::PicturesLibrary, query,
IndexerOption::DoNotUseIndexer);
    auto fileInformationFactory = ref new FileInformationFactory(fileQuery,
ThumbnailMode::PicturesView);
    shared_ptr<ExceptionPolicy> policy = m_exceptionPolicy;
    return create_task(fileInformationFactory->GetFilesAsync(0,
1)).then([policy](IVectorView<FileInformation^>^ files)
    {
        IPhoto^ photo = nullptr;
        auto size = files->Size;
        if (size > 0)
        {
            photo = ref new Photo(files->GetAt(0), ref new NullPhotoGroup(), policy);
        }
        return photo;
    }, task_continuation_context::use_current());
}
```

The **GetSinglePhotoAsync** method calls **CreateFileQuery** to create a file query that will be used by a **FileInformationFactory** object, to query the file system using Advanced Query Syntax. **GetFilesAsync** is then called on the **FileInformationFactory** object, which returns an **IVectorView** collection of **FileInformation** objects. Each **FileInformation** object represents a file in the file system that matches the query. However, in this case the **IVectorView** collection will contain a maximum of one **FileInformation** object, because **GetFilesAsync** only returns one file that matches the query. The value-based continuation, which runs on the main thread, then creates an instance of the **Photo** class to represent the single file returned by **GetFilesAsync**. The **Photo** instance is then returned from **GetSinglePhotoAsync** to **GetImagePhotoAsync** in the **RotateImageViewModel** class that then returns it to the **Photo** property in that class for display.

Detecting file system changes

Some of the pages in the app respond to file system changes in the Pictures library while the app is running. The hub page, image browser page, and image view page refresh after detecting an underlying file system change to the Pictures library. The hub page and image browser page refresh no more than one time every 30 seconds, even if file change notifications arrive faster than that. This is to avoid pages being updated too many times in response to images being added to the Pictures library while the app is running.

To implement this, the **HubPhotoGroup**, **ImageBrowserViewModel**, and **ImageViewModel** classes each create a **function** object in their constructors that will be invoked when photos are added, deleted, or otherwise modified in the folders that have been queried. The **function** object is then passed into the **AddObserver** method of the **FileSystemRepository** class. Here's the code.

C++: ImageViewModel.cpp

```
auto wr = WeakReference(this);
function<void()> callback = [wr] {
    auto vm = wr.Resolve<ImageViewModel>();
    if (nullptr != vm)
    {
        vm->OnDataChanged();
    }
};
m_repository->AddObserver(callback, PageType::Image);
```

This code allows the **OnDataChanged** method in the **ImageViewModel** to be invoked if the underlying file system changes while the app is running. When the **OnDataChanged** method is invoked, it re-queries the file system to update the photo thumbnails on the filmstrip. The **AddObserver** method, in the **FileSystemRepository** class, simply stores the **function** object in a member variable for later use.

When the **QueryPhotosAsync** method in the **ImageViewModel** class is invoked to get the photos for display on the filmstrip, the **GetPhotosForDateRangeQueryAsync** method will be invoked in the **FileSystemRepository** class. This method queries the Pictures library for photos that fall within a specific date range, and returns any matching photos. After the query is created, a **QueryChange** object will be created to register a **function** object to be invoked if the results of the file system query changes. Here's the code.

C++: FileSystemRepository.cpp

```
m_allPhotosQueryChange = (m_imageViewModelCallback != nullptr) ? ref new
QueryChange(fileQuery, m_imageViewModelCallback) : nullptr;
```

The **QueryChange** class is used to monitor the query for file system changes. The constructor accepts a file query and a **function** object. It then registers a handler for the **ContentsChanged** event that fires when an item is added, deleted or modified in the folder being queried. When this event fires, the **function** object passed into the class is executed.

The overall effect of this is that the image view page displays a collection of photo thumbnails on the filmstrip that fall within a specific date range and the full photo for the selected thumbnail. If photos from that date range are added to the file system, deleted from the file system, or otherwise modified, the app will be notified that the relevant file system content has changed and the thumbnails will be refreshed.

Note The handler for this event is still registered when the app is suspended. However, the app won't receive **ContentsChanged** events while it is suspended. Instead, when the app resumes, it refreshes the current view.

When the app resumes from a suspended state, the **OnResume** method in the **App** class invokes the **NotifyAllObservers** method in the **FileSystemRepository** class, which in turn invokes the **function** objects for the hub, image browser, and image view pages so that the relevant content is refreshed.

Creating and navigating between pages in Hilo (Windows Store apps using C++ and XAML)

Summary

- Use Blend for Microsoft Visual Studio 2012 for Windows 8 to create pages and custom controls, change templates and styles, and create animations. Use Microsoft Visual Studio to fine tune your UI, write code, and debug your app.
- Create pages using the MVVM pattern if appropriate for your requirements. When using MVVM, use XAML data binding to link each page to a view model object.
- Use the **LayoutAwarePage** class to provide navigation, state management, and view management.

In Windows Store apps such as Hilo, there is one page for each screen that a user can navigate to. The app creates the first page on startup and then creates subsequent pages in response to navigation requests.

You Will Learn

- How pages were designed in the Hilo app.
- How the Hilo app creates pages and their data sources at run time.
- How Hilo supports application view states such as the snapped view.

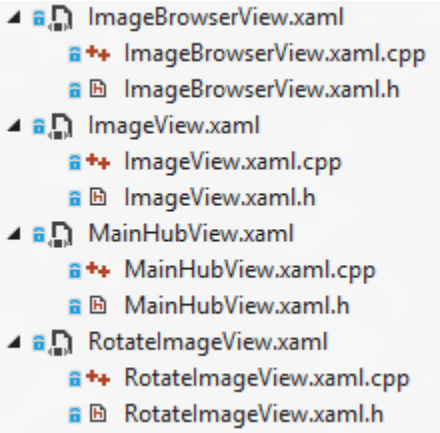
Understanding the tools

For Hilo, we used Blend and the Visual Studio XAML Designer to work with XAML, because these tools made it straightforward to quickly add and modify page layout. Blend was useful to initially define pages and controls; we used Visual Studio to "fine-tune" their appearances. These tools also enabled us to iterate quickly through design choices because they give immediate visual feedback.

In many cases, our UX designer was able to work in parallel with the developers because changing the visual appearance of a page does not affect its behavior.

We recommend that you use Visual Studio to work with the code-focused aspects of your app. Visual Studio is best suited for writing code, running, and debugging your app.

Tip Visual Studio groups XAML design files with its code behind. From **Solution Explorer**, expand any .xaml file to see its backing .h and .cpp files.



We recommend that you use Blend to work on the visual appearance of your app. You can use Blend to create pages and custom controls, change templates and styles, and create animations. Blend comes with minimal code-behind support.

For more info about XAML editing tools, see Blend for Visual Studio and Creating a UI by using XAML Designer.

Adding new pages to the project

There are six pages in the Hilo app. These are

- Main hub page
- Browser page
- View image page
- Crop page
- Rotate page
- Cartoon effect page

The page classes are the views of the MVVM pattern:

- **MainHubView**
- **ImageBrowserView**
- **ImageView**
- **CropImageView**
- **RotateImageView**
- **CartoonizerImageView**

When we built the app, we added each page to the project by using the **Add New Item** dialog box in Visual Studio. For the project template we used the Visual C++ / Windows Store / Grid App (XAML) template. Each page is a separate XAML tree in its own code file.

The Grid App (XAML) template creates pages that are based on Visual Studio's **LayoutAwarePage** class, which provides navigation, state management, and view management.

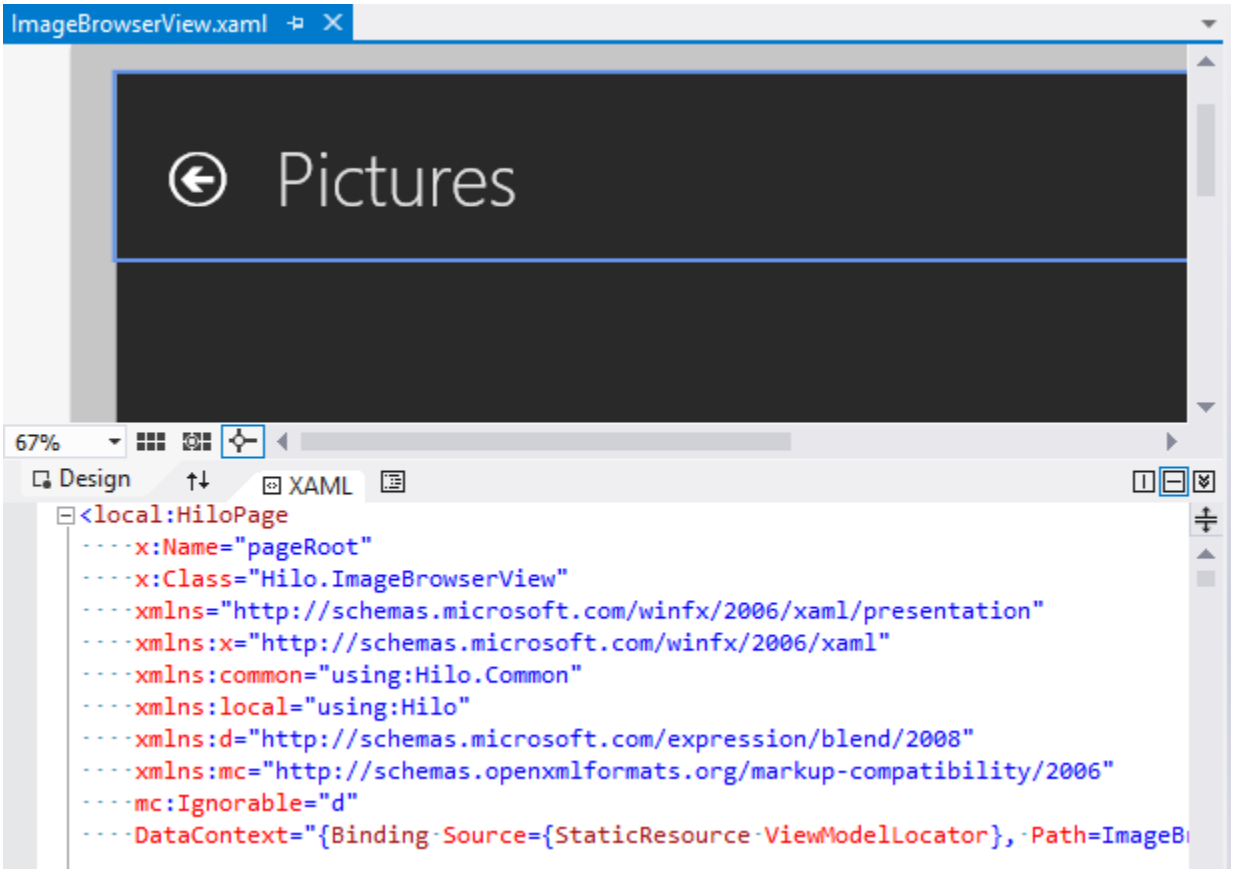
In Hilo, we specialized the functionality of Visual Studio's **LayoutAwarePage** template class with the **HiloPage** class. This class customizes navigation behavior and state management for Hilo's view models. Hilo pages derive from **HiloPage**.

See Tutorial: Create your first Windows Store app using C++ for an introduction that walks you through creating pages in a Windows Store app. See Part 1: Create a "Hello, world" app (Windows Store apps using C#/VB and XAML) for more about the **LayoutAwarePage** class.

Creating pages in the designer view

Pages in Windows Store apps are user controls that support navigation. Each page object represents content that can be navigated to by the user. Pages contain other controls. Page classes are laid out visually in the designer. All page classes are subtypes of the **Windows::UI::Xaml::Page** class. In Hilo, page classes also derive from the **HiloPage** class that in turn derives from the **LayoutAwarePage** class.

For example, here is what the image browser page looks like in the Visual Studio designer.



Each page has an associated XAML file that is created by Visual Studio, and a .h and .cpp file for page-related code written by the app developer.

Visual Studio uses the **partial** keyword, which is part of the C++/CX language extensions, to split the declaration of a page class into several files. For example, for the hub page, the **MainHubView** class is split into four files.

- **MainHubView.xaml**. This file contains the XAML markup that describe the page's controls and their visual layout.
- **MainHubView.g.h**. This header file contains declarations that Visual Studio creates from the XAML source. You should not modify the contents of this file. (The "g" suffix in the file name stands for generated.)
- **MainHubView.xaml.h**. This header contains declarations of member functions whose implementations you can customize. You can also add your own declarations of member functions and member variables to this file.
- **MainHubView.xaml.cpp**. This is the code-behind file. Due to the use of the Model-View-ViewModel (MVVM) pattern, this file only contains event handling code that delegates operations to the page's view model class, **MainHubViewModel**.

Tip Hilo uses the MVVM pattern that abstracts the user interface for the application. With MVVM you rarely need to customize the code-behind .cpp files. Instead, the controls of the user interface are bound to properties of a view model object. If page-related code is required, it should be limited to conveying data to and from the page's view model object.

If you are interested in Hilo's interaction model and more info about how we designed the Hilo UX, see Designing the UX for Windows Store apps.

Establishing the data binding

Data binding links each page to a view model class that is part of the Hilo implementation. The view model class gives the page access to the underlying app logic by using the conventions of the MVVM pattern. For more info see Using the MVVM pattern in this guide.

The data context of each page class in the Hilo app is set to the Hilo app's **ViewModelLocator** class. The data binding specifies the view model locator as a static resource, which means the resource has previously been defined. In this case, the resource is a singleton instance of the **ViewModelLocator** class.

Here is how the data context of the Hilo hub page is set with a data binding, taken from the MainHubView.xaml file:

```
DataContext="{Binding Source={StaticResource ViewModelLocator}, Path=MainHubVM}"
```

When the Windows Runtime creates a Hilo page, it binds to the property specified in the **Path** field of the data context. The property's get method is a method of Hilo's **ViewModelLocator** class, with the method creating a new view model object. For example, the **ViewModelLocator** class's **MainHubVM** method creates an instance of the **MainHubViewModel** class.

C++ ViewModelLocator.cpp

```
MainHubViewModel^ ViewModelLocator::MainHubVM::get()
{
    auto vector = ref new Vector<HubPhotoGroup^>();
    // Pictures Group
    auto loader = ref new ResourceLoader();
    auto title = loader->GetString("PicturesTitle");
    auto emptyTitle = loader->GetString("EmptyPicturesTitle");
    auto picturesGroup = ref new HubPhotoGroup(title, emptyTitle, m_repository,
m_exceptionPolicy);
    vector->Append(picturesGroup);
    return ref new MainHubViewModel(vector, m_exceptionPolicy);
}
```

The **MainHubViewModel** object also exposes other properties that the Hub page class can use.

For reference, here is a summary of the pages, binding paths, and view model classes found in the Hilo app:

App page	Page class	Data binding path	View model class
Hub	MainHubView	MainHubVM	MainHubViewModel
Browse	ImageBrowserView	ImageBrowserVM	ImageBrowserViewModel
Image	ImageView	ImageVM	ImageViewModel
Crop	CropImageView	CropImageVM	CropImageViewModel
Rotate	RotateImageView	RotateImageVM	RotateImageViewModel
Cartoon Effect	CartoonizerImageView	CartoonizerImageVM	CartoonizerImageViewModel

Adding design time data

When you use a visual designer to create a data bound UI, you can display sample data to view styling results and layout sizes. To display data in the designer you must declare it in XAML. This is necessary because the designer parses the XAML for a page but does not run its code-behind. In Hilo, we wanted to display design time data in order to support the designer-developer workflow.

Sample data can be displayed at design time by declaring it in XAML by using the various data attributes from the designer XML namespace. This XML namespace is typically declared with a **d:** prefix.

```
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

Attributes with **d:** prefixes are then interpreted only at design time and are ignored at run time. For example, in a **CollectionViewSource** the **d:Source** attribute is used for design time sample data, and the **Source** attribute is used for run time data.

XAML ImageBrowserView.xaml

```
<CollectionViewSource
    x:Name="MonthGroupedItemsViewSource"
    d:Source="{Binding MonthGroups, Source={d:DesignInstance
Type=local:DesignTimeData, IsDesignTimeCreatable=True}}"
    Source="{Binding MonthGroups}"
    IsSourceGrouped="true"
    ItemsPath="Items"/>
```

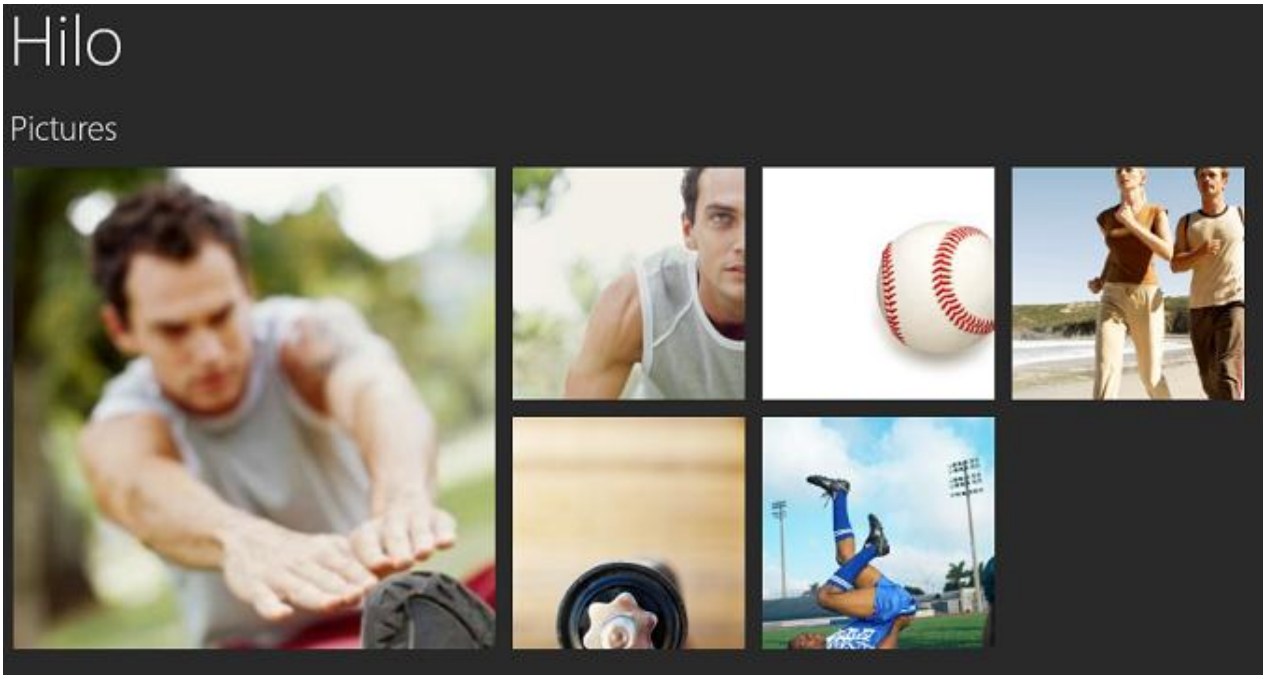
The **d:DesignInstance** attribute indicates that the design time source is a designer created instance based on the **DesignTimeData** type. The **IsDesignTimeCreateable** setting indicates that the designer will instantiate that type directly, which is necessary to display the sample data generated by the type constructor.

For more information, see Data binding overview.

Creating the main hub page

The XAML UI framework provides a built-in navigation model that uses **Frame** and **Page** elements and works much like the navigation in a web browser. The **Frame** control hosts **Page**s, and has a navigation history that you can use to go back and forward through pages you've visited. You can also pass data between pages as you navigate. In the Visual Studio project templates, a **Frame** named **rootFrame** is set as the content of the app window.

When the Hilo app starts up, the **OnLaunched** method of the **Hilo::App** class navigates to the app's hub page.



The **App** class derives from the **Windows::UI:Xaml::Application** class and overrides the**OnLaunched** method. Here is the relevant code from the **OnLaunched** method.

C++ App.xaml.cpp

```
if (rootFrame->Content == nullptr)
{
    // When the navigation stack isn't restored navigate to the first page,
    // configuring the new page by passing required information as a navigation
    // parameter.  See http://go.microsoft.com/fwlink/?LinkId=267278 for a
walkthrough of how
    // Hilo creates pages and navigates to pages.
    if (!rootFrame->Navigate(TypeName(MainHubView::typeid)))
    {
        throw ref new FailureException((ref new LocalResourceLoader())-
>GetString("ErrorFailedToCreateInitialPage"));
    }
}
```

This code shows how Hilo calls the **Navigate** method of a **Windows::UI::Xaml::Controls::Frame** object to load content that is specified by the data type, in this case the **MainHubView** class. The **Frame** control helps maintain application context. The code tests whether the frame object's **Content** property is null as a way of determining whether the app is resuming from a previous state or starting in its default navigation state. (For more info on resuming from previous states, see Handling suspend, resume, and activation in this guide.)

See Tutorial: Create your first Windows Store app using C++ for additional info and code examples of creating pages.

Navigating between pages

The **LayoutAwarePage** class provides methods that use frames to help the app navigate between pages. These are the **GoHome**, **GoForward**, and **GoBack** methods. Hilo calls these methods to initiate page navigation.

For example, the filmstrip control's Back button on the image view page invokes the **GoBack** method.

XAML: ImageView.xaml

```
<Button x:Name="FilmStripBackButton"
        AutomationProperties.AutomationId="FilmStripBackButton"
        Click="GoBack"
        IsEnabled="{Binding Frame.CanGoBack, ElementName=pageRoot}"
        Margin="26,53,36,36"
        Style="{StaticResource BackButtonStyle}"
        VerticalAlignment="Top"/>
```

Direct navigation by the user is not the only possibility. For example, when the rotate image page loads, it locates the image in the file system. If the image is no longer there due to file system changes outside of the Hilo app, the rotate operation navigates back to the main hub page programmatically. Here's the code.

C++: RotateImageViewModel.cpp

```
void RotateImageViewModel::Initialize(String^ photoPath)
{
    assert(IsMainThread());
    m_photo = nullptr;
    m_photoPath = photoPath;

    GetImagePhotoAsync().then([this](IPhoto^ photo)
    {
        assert(IsMainThread());
        // Return to the hub page if the photo is no longer present
        if (photo == nullptr)
        {
            GoHome();
        }
    });
}
```

See Tutorial: Create your first Windows Store app using C++ for additional info about navigating between pages.

Supporting portrait, snap, and fill layouts

We designed Hilo to be viewed full-screen in the landscape orientation. Windows Store apps such as Hilo must adapt to different application view states, including both landscape and portrait orientations. Hilo supports *FullScreenLandscape*, *Filled*, *FullScreenPortrait*, and *Snapped* layouts. Hilo uses the **VisualState** class to specify changes to the visual display to support each layout. The **VisualStateManager** class, used by the **LayoutAwarePage** class, manages states and the logic for transitioning between states for controls. For example, here is the XAML specification of the layout changes for the crop image page.

XAML: CropImageView.xaml

```
<VisualStateManager.VisualStateGroups>
    <!-- Visual states reflect the application's view state -->
    <VisualStateGroup x:Name="ApplicationViewStates">
        <VisualState x:Name="FullScreenLandscape"/>
        <VisualState x:Name="Filled"/>
        <VisualState x:Name="FullScreenPortrait">
            <Storyboard>
                <ObjectAnimationUsingKeyFrames Storyboard.TargetName="CancelButton"

Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0"
                                            Value="Collapsed"/>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames
Storyboard.TargetName="CancelButtonNoLabel"

Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0"
                                            Value="Visible"/>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames Storyboard.TargetName="SaveButton"

Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0"
                                            Value="Collapsed"/>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames
Storyboard.TargetName="SaveButtonNoLabel"

Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0"
                                            Value="Visible"/>
```

```
                </ObjectAnimationUsingKeyFrames>
            </Storyboard>
        </VisualState>
        <VisualState x:Name="Snapped">
            <Storyboard>
                <ObjectAnimationUsingKeyFrames Storyboard.TargetName="CancelButton"

Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0"
                                            Value="Collapsed"/>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames
Storyboard.TargetName="CancelButtonNoLabel"

Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0"
                                            Value="Visible"/>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames Storyboard.TargetName="SaveButton"

Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0"
                                            Value="Collapsed"/>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames
Storyboard.TargetName="ResumeCropStackPanel"

Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0"
                                            Value="Visible"/>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames Storyboard.TargetName="CropCanvas"

Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0"
                                            Value="Collapsed"/>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames Storyboard.TargetName="CropImageGrid"
                                               Storyboard.TargetProperty="Margin">
                    <DiscreteObjectKeyFrame KeyTime="0"
                                            Value="20,50,20,0"/>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames Storyboard.TargetName="PageTitle"

Storyboard.TargetProperty="Visibility">
                    <DiscreteObjectKeyFrame KeyTime="0"
                                            Value="Visible"/>
                </ObjectAnimationUsingKeyFrames>
            </Storyboard>
```

```
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Note We typically set the **Style** property when we need to update multiple properties or when there is a defined style that does what we want. We often update individual properties directly when we only need to update one property. Although styles enable you to control multiple properties and also provide a consistent appearance throughout your app, providing too many can make your app difficult to maintain. Use styles only when it makes sense to do so. For more info about styling controls, see Quickstart: styling controls.

In most cases you can provide complete support for the app's view states by using XAML. In Hilo, there is only one place where we needed to change the C++ code to support layout. This is in the crop image view model, where the app adjusts the position of the crop overlay to match the new layout. To do this, the app provides a handler for the **SizeChanged** event.

XAML: CropImageView.xaml

```
<Image x:Name="Photo"
       AutomationProperties.AutomationId="ImageControl"
       HorizontalAlignment="Center"
       VerticalAlignment="Center"
       SizeChanged="OnSizeChanged"
       Source="{Binding Image}"/>
```

Hilo binds the **SizeChanged** event to the **CropImageView::OnSizeChanged** member function.

C++: CropImageView.cpp

```
void CropImageView::OnSizeChanged(Object^ sender, SizeChangedEventArgs^ e)
{
    m_cropImageViewModel->CalculateInitialCropOverlayPosition(
        Photo->TransformToVisual(CropImageGrid),
        Photo->RenderSize.Width, Photo->RenderSize.Height);

    if (!m_sizeChangedAttached)
    {
        SizeChanged += ref new SizeChangedEventHandler(this,
&CropImageView::OnSizeChanged);
        m_sizeChangedAttached = true;
    }
}
```

When the image size changes as a result of a crop operation, the handler invokes the crop image view model's **CalculateInitialCropOverlayPosition** method, which makes the crop overlay fit the new layout. In addition, when the size of the page changes, such as when switching from the *FullScreenLandscape* view state to the *Filled* view state, the handler will again make the crop overlay fit the new layout.

Tip When you develop an app in Visual Studio, you can use the Simulator debugger to test layouts. To do this, press F5 and use the debugger tool bar to debug with the Simulator.

See Adapting to different layouts for more info and a walkthrough of a sample app that responds to layout changes.

Using controls in Hilo (Windows Store apps using C++ and XAML)

Summary

- Use binding to declaratively connect your UI to data.
- Create custom controls to combine the functionality of existing controls.
- Use control templates to control the appearance of predefined controls, avoiding the need to write custom controls.

Controls are the core of Windows 8 and XAML. Learn about some of the controls, including Image, Grid and GridView, ProgressRing, Button, TextBlock, AppBar, StackPanel, ListView, SemanticZoom, Canvas and ContentControl, and Popup, and how we used binding and styles to implement the Hilo C++ UI.

You will learn

- How to use binding to link the UI to code-behind.
- How to apply data templates.
- How to write custom controls.
- How to handle invalid images when using binding.
- How to convert your model data to correctly display.

You might notice the declarative, rather than imperative, nature of XAML. XAML forms a tree of objects, which you can think of like a DOM in HTML. If you're just getting started, see Quickstart: Create a UI with XAML and then read Quickstart: adding controls and handling events.

Note The topic Creating and navigating between pages explains how we used Blend for Microsoft Visual Studio 2012 for Windows 8 and the Visual Studio XAML Designer to work with XAML pages and controls. Although these tools are easy to use, don't overstyle your controls. For example, not every button needs a gradient background or rounded corners. See Index of UX guidelines for Windows Store apps for the recommended guidelines.

Here are a few important concepts in XAML that may be unfamiliar to C++ developers:

- *Code-behind* connects XAML markup with compiled code that implements the logic of your app. For example, for a Button control, you implement the handler for the **Click** event in your C++ code. This C++ code is an example of code-behind.

  Although you can add logic directly into code behind, in Hilo, code behind connects the XAML to view models that define its logic.

- A *template* or *data template* defines the structure, layout, and animations that are associated with a control, plus its styles. A template affects the look and feel of a control, but not its behaviors. For example, list boxes and most other controls use strings to represent their data elements. Templates enable you to use graphics and other visual features to represent data. You use the **DataTemplate** class to define the visual representation of data from XAML.
- *Binding* provides a data-bound property value such that the value is deferred until run time. You use the **Binding** class to connect your UI to data.
- A *custom control* aggregates functionality of other controls.
- A *data converter* transforms one value into another. Use data converters together with binding to change the value or format of bound data to something that your app will display.
- A *resource* refers to functionality that can be reused. For example, styles, brushes, and text that appear on a page are all examples of resources.

The way that you program Windows Store apps using XAML closely resembles Windows Presentation Foundation (WPF) and Silverlight. The main differences are that Windows Store apps are focused on touch and provide several new controls. Read Porting Silverlight or WPF XAML/code to a Windows Store app to learn the differences between Silverlight or WPF and Windows Store apps using XAML. Controls list and Controls by function list all available controls, including AppBar, **GridView**, SemanticZoom, and other controls that are new to Windows 8.

Note A Windows Store app using C++ and XAML runs completely as native code. XAML controls are also accelerated by graphics hardware. The C++/CX syntax helps make it easier to target the Windows Runtime, and doesn't use the .NET Framework. Your C++ app targets .NET only when it uses a Windows Runtime component that uses .NET. For more info about C++/CX, see Writing modern C++ code.

Data binding

Binding provides a data-bound property value such that the value is deferred until run time. Binding is key to effectively using XAML because it enables you to declaratively connect your UI to data. Binding also helps minimize the need for code behind. Because data is resolved at run time, you can clearly separate your UI from your data when you design your app. For Hilo, binding data at run time is critical because we don't know anything about the user's picture library at design time. For more info about binding, see Data binding overview and Binding markup extension.

Binding also supports the MVVM pattern. For more info about how we use binding in Hilo, see Using the MVVM pattern. Later sections on this page also describe how we used binding to link to both static and dynamic content.

Data converters

A *data converter* transforms a value. A converter can take a value of one type and return a value of another type or simply change a value to produce the same type.

For Hilo, we use data converters to change the value or format of bound data when data types need special formatting that the view model doesn’t provide. Here are the data converters we defined for Hilo.

Converter type	Converts from	Converts to	Description
BooleanToBrushConverter	**bool**	**Platform::String**	On the calendar view, used to show orange for months that have pictures and grey for months that don't have pictures. The value **true** maps to orange (#F19720) and **false** maps to grey (#E2E2E2). This works because a type converter is present on the **Brush** class to convert the string value to a **Brush**.
FileSizeConverter	**unsigned long long**	**Platform::String**	Used to convert file sizes in bytes to kilobytes or megabytes to make the file size more human-readable.

The Visual Studio project templates also provide the **BooleanNegationConverter** and **BooleanToVisibilityConverter** classes, which negate **bool** values and convert **bool** values to **Visibility** enumeration values, respectively. You can find these classes in the Common folder of the solution.

To create a data converter, you create a type that derives from **IValueConverter**. You then implement the **Convert**, **ConvertBack** methods, and any additional methods that you need. Here's the declaration for the **FileSizeConverter** class.

C++: FileSizeConverter.h

```
[Windows::Foundation::Metadata::WebHostHidden]
public ref class FileSizeConverter sealed : public
Windows::UI::Xaml::Data::IValueConverter
{
public:
    FileSizeConverter();
    FileSizeConverter(IResourceLoader^ loader);

    virtual Object^ Convert(Object^ value, Windows::UI::Xaml::Interop::TypeName
targetType, Object^ parameter, Platform::String^ language);
    virtual Object^ ConvertBack(Object^ value, Windows::UI::Xaml::Interop::TypeName
targetType, Object^ parameter, Platform::String^ language);

private:
    float64 ToTwoDecimalPlaces(float64 value);
    IResourceLoader^ m_loader;
};
```

And here's its implementation.

C++: FileSizeConverter.cpp

```
Object^ FileSizeConverter::Convert(Object^ value, TypeName targetType, Object^
parameter, String^)
{
    float64 size = static_cast<float64>(safe_cast<uint64>(value));
    std::array<String^, 3> units =
    {
        m_loader->GetString("BytesUnit"),
        m_loader->GetString("KilobytesUnit"),
        m_loader->GetString("MegabytesUnit")
    };
    unsigned int index = 0;

    while (size >= 1024)
    {
        size /= 1024;
        index++;
    }

    return ToTwoDecimalPlaces(size) + " " + units[index];
}

float64 FileSizeConverter::ToTwoDecimalPlaces(float64 value)
{
    float64 f;
```

```
    float64 intpart;
    float64 fractpart;
    fractpart = modf(value, &intpart);
    f = floor(fractpart * 100 + 0.5) / 100.0;
    return intpart + f;
}

Object^ FileSizeConverter::ConvertBack(Object^ value, TypeName targetType, Object^
parameter, String^)
{
    throw ref new NotImplementedException();
}
```

In the **FileSizeConverter::Convert** method, we use **safe_cast** to convert from **Object^** to the value type **uint64** (**safe_cast** throws **Platform::InvalidCastException** if the conversion fails). We cast to **uint64** because we bind the converter to the **IPhoto::FileSize** property, which returns **uint64**. We then convert the **uint64** value to **float64** because the computation uses floating-point arithmetic.

We must implement the **ConvertBack** method because it's part of the **IValueConverter** interface. But we throw **NotImplementedException** to indicate that we have not implemented this functionality. You should throw this exception instead of returning a default value when you don't expect the method to be called. That way, during development, you can verify that the method is never called. If you return a default value, the app can behave in ways you don't expect. In Hilo, we don't expect **ConvertBack** to be called because we're not performing two-way binding. (In some applications, a field that is bound to a date might allow the user to type a new date value. This two-way binding would require both the **Convert** and **ConvertBack** methods to work properly.)

To use your converter from XAML, first add it to your resource dictionary. Here's how we did so for the image view.

XAML: ImageView.xaml

```
<Page.Resources>
    <local:FileSizeConverter x:Key="FileSizeConverter" />
    <Style x:Key="FilmStripGridViewItemStyle"
           BasedOn="{StaticResource HiloGridViewItemStyle}"
           TargetType="GridViewItem">
        <Setter Property="Margin" Value="0,0,2,4" />
    </Style>
</Page.Resources>
```

You must also include the header file for the converter in the code-behind for the page.

C++: ImageView.xaml.h

```
#include "FileSizeConverter.h" // Required by generated header
#include "ImageView.g.h"
```

To convert a bound value, use the **Binding::Converter** property from XAML. This example uses **FileSizeConverter** to make the image size on disk more human-readable on the popup that displays additional file info. (For more info about this popup, see Press and hold to learn in this guide.)

XAML: ImageView.xaml

```
<TextBlock Foreground="{StaticResource ApplicationForegroundThemeBrush}"
           Text="{Binding Path=SelectedItem.FileSize, Converter={StaticResource
FileSizeConverter}}" />
```

For more info about converters, see Data binding overview.

Common controls used in Hilo

Here are some of the common controls that we used in Hilo. You can also refer to the .xaml files that appear in the **Views** folder in the Visual Studio project.

Tip See Controls list and Controls by function for all available controls. Bookmark these pages so that you can come back to them when you want to add another feature to your app. Also use the **Toolbox** window in Visual Studio and the **Assets** tab in Blend to browse and add controls to your pages.

- Image
- Grid and GridView
- ProgressRing
- Button
- TextBlock
- AppBar
- StackPanel
- ListView
- SemanticZoom
- Canvas and ContentControl
- Popup

Image

We use the **Image** control to display pictures in the app. We found it relatively straightforward to use the standard properties to set the size, alignment, and other display properties. At times, we don't

specify properties such as the size to allow the framework to fit the image to the available size. For example, for a **Grid** control, if a child's size is not set explicitly, it stretches to fill the available space in the grid cell. For more info about the layout characteristics of common controls, see Quickstart: Adding layout controls.

Here's the XAML for the **Image**.

XAML: CropImageView.xaml

```
<Image x:Name="Photo"
       AutomationProperties.AutomationId="ImageControl"
       HorizontalAlignment="Center"
       VerticalAlignment="Center"
       SizeChanged="OnSizeChanged"
       Source="{Binding Image}"/>
```

Because the majority of images are of the user's photos, we found binding to be a very useful way to specify what to load at runtime without the need to write any code-behind. We also needed to handle image files that cannot be loaded (for example, when the image file is corrupt.) The Hilo **Photo** class, which is bound to the view model and provides the image data, subscribes to the **Image::ImageFailed** event, which is called when an error occurs during image retrieval.

C++: Photo.cpp

```
m_image = ref new BitmapImage();
m_imageFailedEventToken = m_image->ImageFailed::add(ref new
ExceptionRoutedEventHandler(this, &Photo::OnImageFailedToOpen));
```

The handler for the **Image::ImageFailed** event, **Photo::OnImageFailedToOpen**, uses the **ms-appx://** syntax to load a default image when an error occurs during image retrieval.

C++: Photo.cpp

```
void Photo::OnImageFailedToOpen(Object^ sender, ExceptionRoutedEventArgs^ e)
{
    assert(IsMainThread());

    // Load a default image.
    m_image = ref new BitmapImage(ref new Uri("ms-appx:///Assets/HiloLogo.png"));
}
```

For more info about **ms-appx://** and other URI schemes, see How to load file resources.

The next section describes how we used binding to display the filmstrip that appears on the image page.

For more info about images, see Quickstart: Image and ImageBrush, XAML images sample, and XAML essential controls sample.

Grid and GridView

Grids are the heart of every page in the Hilo app because they enabled us to position child controls where we wanted them. A **Grid** is a layout panel that supports arranging child elements in rows and columns. A **GridView** presents a collection of items in rows and columns that can scroll horizontally. A **GridView** can also display variable-sized collections.

We used **Grid** to display fixed numbers of items that all fit on the screen. We used **GridView** to display variable-sized collections. For example, we used **GridView** on the image browser page to group photos by month. We also used **GridView** to display the filmstrip that appears in the top app bar on the image page.

Here's the XAML for the **GridView** that represents the filmstrip.

XAML: ImageView.xaml

```
<GridView x:Name="PhotosFilmStripGridView"
          Grid.Column="1"
          AutomationProperties.AutomationId="PhotosFilmStripGridView"
          IsItemClickEnabled="False"
          ItemContainerStyle="{StaticResource FilmStripGridViewItemStyle}"
          ItemsSource="{Binding Photos}"
          SelectedItem="{Binding Path=SelectedItem, Mode=TwoWay}"
          SelectionMode="Single"
          VerticalAlignment="Center">
    <GridView.ItemsPanel>
        <ItemsPanelTemplate>
            <VirtualizingStackPanel Height="138" Orientation="Horizontal" />
        </ItemsPanelTemplate>
    </GridView.ItemsPanel>
    <GridView.ItemTemplate>
        <DataTemplate>
            <Border>
                <Image Source="{Binding Path=Thumbnail}"
                       Height="138"
                       Width="200"
                       Stretch="UniformToFill" />
            </Border>
        </DataTemplate>
    </GridView.ItemTemplate>
</GridView>
```

Note The **x:Name** attribute specifies the name of the control instance to the code. In this example, the compiler generates a member variable named **PhotosFilmStripGridView** for the **ImageView** class. This member variable appears in the file ImageView.g.h, which is generated by the compiler.

This **GridView** uses **FilmStripGridViewItemStyle**, which is defined in ApplicationStyles.xaml, to customize the appearance of the **GridViewItem**s that are displayed in the **GridView**. Each **GridViewItem**

is an **Image** control, wrapped in a **Border** control, and is defined in the **DataTemplate** of the **GridView**. In order to support UI virtualization, a **VirtualizingStackPanel** is used to display the **GridViewItem**s, and is defined in the **ItemsPanelTemplate** of the **GridView**. For more information about UI virtualization, see UI virtualization for working with large data sets in this topic.

The next example shows how the **ImageView::OnFilmStripLoaded** method scrolls the selected item into view.

C++: ImageView.xaml.cpp

```
// Scrolls the selected item into view after the collection is likely to have loaded.
void ImageView::OnFilmStripLoaded(Object^ sender, RoutedEventArgs^ e)
{
    auto vm = dynamic_cast<ImageViewModel^>(DataContext);
    if (vm != nullptr)
    {
        PhotosFilmStripGridView->ScrollIntoView(vm->SelectedItem);
    }

    PhotosFilmStripGridView->Loaded::remove(m_filmStripLoadedToken);
}
```

An interesting aspect of **GridView** is how it binds to data. To enable MVVM on the image view page, we set the pages'**DataContext** to the **ImageVM** property of the **ViewModelLocator** class (for more info on MVVM, see Using the MVVM pattern.)

XAML: ImageView.xaml

```
<local:HiloPage
    x:Name="pageRoot"
    x:Uid="Page"
    x:Class="Hilo.ImageView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Hilo"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    DataContext="{Binding Source={StaticResource ViewModelLocator}, Path=ImageVM}">
```

As a result, child controls of the page bind to the **ImageViewModel** class (the **ViewModelLocator::ImageVM** property returns an **ImageViewModel** object.) For the filmstrip, we bind to the **ImageViewModel::Photos** property to display the photos for the current month.

XAML: ImageView.xaml

```
<GridView x:Name="PhotosFilmStripGridView"
          Grid.Column="1"
          AutomationProperties.AutomationId="PhotosFilmStripGridView"
          IsItemClickEnabled="False"
          ItemContainerStyle="{StaticResource FilmStripGridViewItemStyle}"
          ItemsSource="{Binding Photos}"
          SelectedItem="{Binding Path=SelectedItem, Mode=TwoWay}"
          SelectionMode="Single"
          VerticalAlignment="Center">
```

The **ImageViewModel::Photos** property is a collection of **IPhoto** objects. To display a single photo, we use an **Image** element that is bound to the **IPhoto::Thumbnail** property.

XAML: ImageView.xaml

```
<GridView.ItemTemplate>
    <DataTemplate>
        <Border>
            <Image Source="{Binding Path=Thumbnail}"
                   Height="138"
                   Width="200"
                   Stretch="UniformToFill" />
        </Border>
    </DataTemplate>
</GridView.ItemTemplate>
```

At run time, the binding between the view and the view model occurs when the page is created. When a user navigates to the page, the **ImageViewModel::OnNavigatedTo** method sets the file system query for the current month's pictures. When the **GridView** first displays the photos for the current month, the **ImageViewModel::Photos** property invokes the **ImageViewModel::OnDataChanged** method, which in turn invokes the **ImageViewModel::QueryPhotosAsync** method. The **ImageViewModel::QueryPhotosAsync** method invokes **GetPhotosForDateRangeQueryAsync** on the shared pointer instance of the **FileSystemRepository** class, which was passed into the **ImageViewModel** constructor, and passes the **m_query** member variable which contains the file system query for the current month's pictures.

For more info about **GridView**, see Adding ListView and GridView controls and Item templates for grid layouts.

ProgressRing

We use the **ProgressRing** control to indicate that an operation is in progress. We perform any long-running operations in the background, so the user can interact with the app, and we use progress rings to show that an operation is occurring and will complete shortly. Use progress controls to indicate operations that take 2 or more seconds to complete.

Here's the XAML for the image browser page. The progress ring appears when the OS is loading thumbnail data.

XAML: ImageBrowserView.xaml

```
<ProgressRing x:Name="ProgressRing"
              IsActive="{Binding InProgress}"
              Height="60"
              Width="60"
              Foreground="{StaticResource HiloHighlightBrush}"/>
```

We bind the **IsActive** property to the **ImageBrowserViewModel::InProgress** property (a Boolean value). The **ImageBrowserViewModel::InProgress** property returns **true** when the view model is querying the OS for images. After the query finishes, the **ImageBrowserViewModel::InProgress** property returns **false** and the view model object raises the **PropertyChanged** event to instruct the XAML runtime to hide the progress ring.

We use the **ProgressRing** control instead of the **ProgressBar** control to indicate that an operation is in progress because the time remaining to complete that operation is not known. Use **ProgressBar** to provide more detailed info about when an operation will complete. Progress bars and progress rings should not prevent the user from canceling the operation or navigating to another page. For more info about performing and canceling background operations, see Async programming patterns in C++.

Note Always set the **ProgressRing::IsActive** and **ProgressBar::IsIndeterminate** properties to **false** when you've finished showing progress info. Even when these controls are not visible, their animations can still consume CPU cycles and thus cause performance problems or shorten battery life. Also, when **ProgressRing::IsActive** is **false**, the **ProgressRing** is not shown, but space is reserved for it in the UI layout. To not reserve space for the **ProgressRing**, set its **Visibility** property to **Collapsed**.

For more progress control guidelines, see Guidelines and checklist for progress controls.

Button

A **Button** control responds to user input and raises a **Click** event. In Hilo, we use buttons to:

- Define a back button on each page.
- Enable the user to navigate to the browse pictures page from the hub page.
- Enable the user to navigate to all photos for a given month.
- Add commands to app bars.

Here's the XAML for the **Button** control that enables the user to navigate to all photos for a given month.

XAML: ImageBrowserView.xaml

```
<Button AutomationProperties.AutomationId="MonthGroupTitle"
        Content="{Binding Title}"
        Command="{Binding ElementName=MonthPhotosGridView,
Path=DataContext.GroupCommand}"
        CommandParameter="{Binding}"
        Style="{StaticResource HiloTextButtonStyle}" />
```

An important part of this implementation is the fact that the button's **Command** binds to the **GroupCommand** property on the view model. Also, we bind the button's **Content** property to a text string. However, you can bind the button's content to any class that derives from **Panel**. The **Button** is styled using the **HiloTextButtonStyle**, which uses the **SubheaderTextStyle** specified in StandardStyles.xaml, with no other adornment.

The **ImageBrowserViewModel** constructor sets the **m_groupCommand** (the backing **ICommand** value for the **GroupCommand** property) to point to the **NavigateToGroup** method.

C++: ImageBrowserViewModel.cpp

```
m_groupCommand = ref new DelegateCommand(ref new ExecuteDelegate(this,
&ImageBrowserViewModel::NavigateToGroup), nullptr);
```

The **NavigateToGroup** method navigates to the image browser page for the current month.

C++: ImageBrowserViewModel.cpp

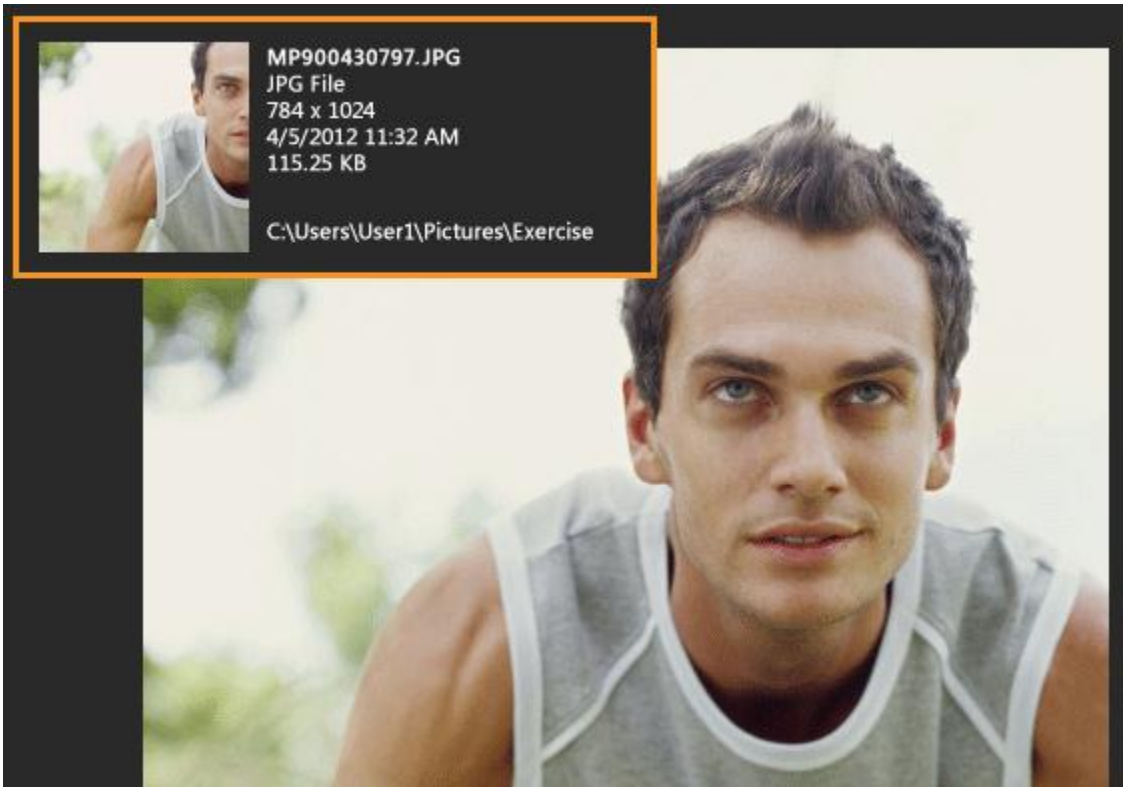
```
void ImageBrowserViewModel::NavigateToGroup(Object^ parameter)
{
    auto group = dynamic_cast<IPhotoGroup^>(parameter);
    assert(group != nullptr);
    if (group->Items->Size > 0)
    {
        auto photo = dynamic_cast<IPhoto^>(group->Items->GetAt(0));
        assert(photo != nullptr);
        ImageNavigationData imageData(photo);
        ViewModelBase::GoToPage(PageType::Image, imageData.SerializeToString());
    }
}
```

See Executing commands in a view model in this guide for more info on binding commands to the view model.

For more info about buttons, see Adding buttons and XAML essential controls sample.

TextBlock

The **TextBlock** control displays text. We use text blocks to display the title of each page and to fill the **Popup** control that displays additional info about a picture when the user presses and holds the current picture.



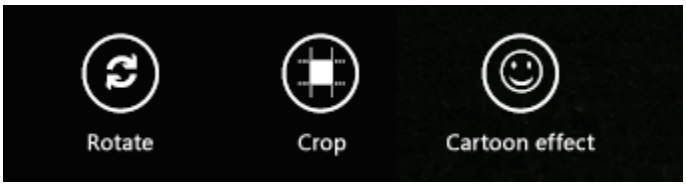
Binding and the use of resources are two important aspects of displaying text. Binding helps defer the value of a text block until run time (for example, to display info about a picture.) Resources are important because they enable you to more easily localize your app. For more info about localization, see Making your app world ready in this guide.

For more info about text blocks, see Quickstart: Displaying text and XAML text display sample.

AppBar

The **AppBar** control is a toolbar for displaying app-specific commands. Hilo displays a bottom app bar on every page, and a top app bar on the image view page that displays a filmstrip view of all photos for the current month. The **Page::BottomAppBar** property can be used to define the bottom app bar and the **Page::TopAppBar** property can be used to define the top app bar.

Here's what the buttons look like on the bottom app bar on the image view page.



You can add buttons that enable navigation or critical app features on a page. Place buttons that are not critical to the navigation and use of your app in an app bar. For example, on the image browser page, we enable the user to click on the text for a given month to jump to all photos for that month because we felt that it was crucial to navigation. We added the rotate, crop, and cartoon effect commands to the app bar because these are secondary commands and we didn't want them to distract the user.

Here's the XAML for the bottom app bar that appears on the image view page. This app bar contains **Button** elements that enable the user to enter rotate, crop, and cartoon effect modes.

XAML: ImageView.xaml

```
<local:HiloPage.BottomAppBar>
    <AppBar x:Name="ImageViewBottomAppBar"
            x:Uid="AppBar"
            AutomationProperties.AutomationId="ImageViewBottomAppBar"
            Padding="10,0,10,0">
        <Grid>
            <StackPanel HorizontalAlignment="Left"
                        Orientation="Horizontal">
                <Button x:Name="RotateButton"
                        x:Uid="RotateAppBarButton"
                        Command="{Binding RotateImageCommand}"
                        Style="{StaticResource RotateAppBarButtonStyle}"
                        Tag="Rotate" />
                <Button x:Name="CropButton"
                        x:Uid="CropAppBarButton"
                        Command="{Binding CropImageCommand}"
                        Style="{StaticResource CropAppBarButtonStyle}"
                        Tag="Crop" />
                <Button x:Name="CartoonizeButton"
                        x:Uid="CartoonizeAppBarButton"
                        Command="{Binding CartoonizeImageCommand}"
                        Style="{StaticResource CartoonEffectAppBarButtonStyle}"
                        Tag="Cartoon effect" />
                <Button x:Name="RotateButtonNoLabel"
                        Command="{Binding RotateImageCommand}"
                        Style="{StaticResource RotateAppBarButtonNoLabelStyle}"
                        Tag="Rotate"
                        Visibility="Collapsed">
                    <ToolTipService.ToolTip>
                        <ToolTip x:Uid="RotateAppBarButtonToolTip" />
                    </ToolTipService.ToolTip>
                </Button>
                <Button x:Name="CropButtonNoLabel"
                        Command="{Binding CropImageCommand}"
                        Style="{StaticResource CropAppBarButtonNoLabelStyle}"
                        Tag="Crop"
```

```
                        Visibility="Collapsed">
                    <ToolTipService.ToolTip>
                        <ToolTip x:Uid="CropAppBarButtonToolTip" />
                    </ToolTipService.ToolTip>
                </Button>
                <Button x:Name="CartoonizeButtonNoLabel"
                        Command="{Binding CartoonizeImageCommand}"
                        Style="{StaticResource
CartoonEffectAppBarButtonNoLabelStyle}"
                        Tag="Cartoon effect"
                        Visibility="Collapsed">
                    <ToolTipService.ToolTip>
                        <ToolTip x:Uid="CartoonizeAppBarButtonToolTip" />
                    </ToolTipService.ToolTip>
                </Button>
            </StackPanel>
        </Grid>
    </AppBar>
</local:HiloPage.BottomAppBar>
```

We define two sets of buttons—one set for landscape mode and one for portrait mode. Per UI guidelines for app bars, we display labels on the buttons in landscape mode and hide the labels in portrait mode. Here's the XAML that defines the visibility for the buttons in portrait mode.

XAML: ImageView.xaml

```
<VisualState x:Name="FullScreenPortrait">
    <Storyboard>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="BackButton"
                                       Storyboard.TargetProperty="Style">
            <DiscreteObjectKeyFrame KeyTime="0"
                                    Value="{StaticResource
PortraitBackButtonStyle}"/>
        </ObjectAnimationUsingKeyFrames>

        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="ItemGridView"
                                       Storyboard.TargetProperty="ItemsPanel">
            <DiscreteObjectKeyFrame KeyTime="0"
                                    Value="{StaticResource
HubVerticalVirtualizingStackPanelTemplate}" />
        </ObjectAnimationUsingKeyFrames>

        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="ItemGridView"
                                       Storyboard.TargetProperty="Padding">
            <DiscreteObjectKeyFrame KeyTime="0"
                                    Value="96,0,10,56"/>
        </ObjectAnimationUsingKeyFrames>
```

```
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="RotateButton"
                                       Storyboard.TargetProperty="Visibility">
            <DiscreteObjectKeyFrame KeyTime="0"
                                    Value="Collapsed"/>
        </ObjectAnimationUsingKeyFrames>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="CropButton"
                                       Storyboard.TargetProperty="Visibility">
            <DiscreteObjectKeyFrame KeyTime="0"
                                    Value="Collapsed"/>
        </ObjectAnimationUsingKeyFrames>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="CartoonizeButton"
                                       Storyboard.TargetProperty="Visibility">
            <DiscreteObjectKeyFrame KeyTime="0"
                                    Value="Collapsed"/>
        </ObjectAnimationUsingKeyFrames>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="RotateButtonNoLabel"
                                       Storyboard.TargetProperty="Visibility">
            <DiscreteObjectKeyFrame KeyTime="0"
                                    Value="Visible"/>
        </ObjectAnimationUsingKeyFrames>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="CropButtonNoLabel"
                                       Storyboard.TargetProperty="Visibility">
            <DiscreteObjectKeyFrame KeyTime="0"
                                    Value="Visible"/>
        </ObjectAnimationUsingKeyFrames>
        <ObjectAnimationUsingKeyFrames
Storyboard.TargetName="CartoonizeButtonNoLabel"
                                       Storyboard.TargetProperty="Visibility">
            <DiscreteObjectKeyFrame KeyTime="0"
                                    Value="Visible"/>
        </ObjectAnimationUsingKeyFrames>
    </Storyboard>
</VisualState>
```

Tip We authored this XAML by hand. However, many developers prefer to use Blend to define visual states.

For more info about how we used app bars in Hilo, see Swipe from edge for app commands in this guide.

StackPanel

The **StackPanel** control arranges its children into a single line that can be oriented horizontally or vertically. Our use of **StackPanel** includes:

- Arranging buttons on the app bar.
- Displaying a progress ring next to another control, such as a **Button**.
- Displaying elements in a **Popup** control to implement press and hold. For more info, see Press and hold to learn in this guide.

Here's an example of how we used **StackPanel** to arrange buttons on the bottom app bar of the hub page.

XAML: MainHubView.xaml

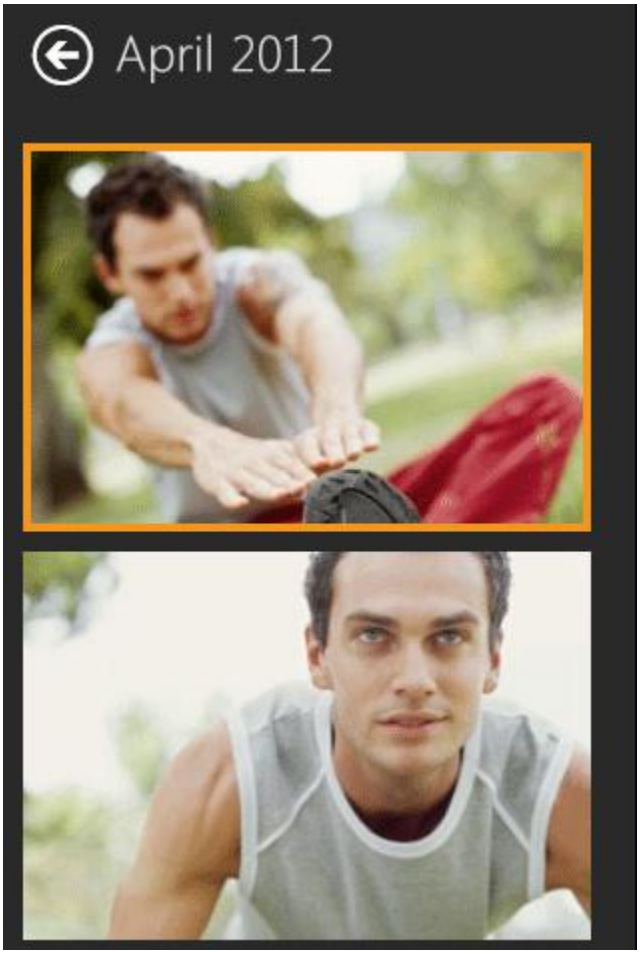
```
<local:HiloPage.BottomAppBar>
    <AppBar x:Name="MainHubViewBottomAppBar"
            x:Uid="AppBar"
            AutomationProperties.AutomationId="MainHubViewBottomAppBar"
            IsOpen="{Binding Path=IsAppBarOpen, Mode=TwoWay}"
            IsSticky="{Binding Path=IsAppBarSticky, Mode=TwoWay}"
            Padding="10,0,10,0"
            Visibility="{Binding Path=IsAppBarEnabled, Mode=TwoWay,
Converter={StaticResource BoolToVisConverter}}">
        <Grid>
            <StackPanel HorizontalAlignment="Left"
                        Orientation="Horizontal">
                <Button x:Name="RotateButton"
                        x:Uid="RotateAppBarButton"
                        Command="{Binding RotateImageCommand}"
                        Style="{StaticResource RotateAppBarButtonStyle}"
                        Tag="Rotate" />
                <Button x:Name="CropButton"
                        x:Uid="CropAppBarButton"
                        Command="{Binding CropImageCommand}"
                        Style="{StaticResource CropAppBarButtonStyle}"
                        Tag="Crop" />
                <Button x:Name="CartoonizeButton"
                        x:Uid="CartoonizeAppBarButton"
                        Command="{Binding CartoonizeImageCommand}"
                        Style="{StaticResource CartoonEffectAppBarButtonStyle}"
                        Tag="Cartoon effect" />
                <Button x:Name="RotateButtonNoLabel"
                        Command="{Binding RotateImageCommand}"
                        Style="{StaticResource RotateAppBarButtonNoLabelStyle}"
                        Tag="Rotate"
                        Visibility="Collapsed">
```

```
                    <ToolTipService.ToolTip>
                        <ToolTip x:Uid="RotateAppBarButtonToolTip" />
                    </ToolTipService.ToolTip>
                </Button>
                <Button x:Name="CropButtonNoLabel"
                        Command="{Binding CropImageCommand}"
                        Style="{StaticResource CropAppBarButtonNoLabelStyle}"
                        Tag="Crop"
                        Visibility="Collapsed">
                    <ToolTipService.ToolTip>
                        <ToolTip x:Uid="CropAppBarButtonToolTip" />
                    </ToolTipService.ToolTip>
                </Button>
                <Button x:Name="CartoonizeButtonNoLabel"
                        Command="{Binding CartoonizeImageCommand}"
                        Style="{StaticResource
CartoonEffectAppBarButtonNoLabelStyle}"
                        Tag="Cartoon effect"
                        Visibility="Collapsed">
                    <ToolTipService.ToolTip>
                        <ToolTip x:Uid="CartoonizeAppBarButtonToolTip" />
                    </ToolTipService.ToolTip>
                </Button>
            </StackPanel>
        </Grid>
    </AppBar>
</local:HiloPage.BottomAppBar>
```

We set the **Orientation** property to **Horizontal** to arrange the children of the **StackPanel** from left to right. Set **Orientation** to **Vertical** to arrange children from top to bottom.

ListView

The **ListView** control displays data vertically. One way that we use this control is to display photos vertically on the image view page when the app is in snapped view.



Here's the XAML for the **ListView**.

XAML: ImageView.xaml

```
<ListView x:Name="SnappedPhotosFilmStripListView"
          Grid.Row="1"
          AutomationProperties.AutomationId="SnappedPhotosFilmStripListView"
          Margin="0,-10,0,0"
          Padding="10,0,0,60"
          ItemsSource="{Binding Photos}"
          IsItemClickEnabled="False"
          ItemContainerStyle="{StaticResource HiloListViewItemStyle}"
          SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
          SelectionMode="Single"
```

```
          Visibility="Collapsed">
    <ListView.ItemsPanel>
        <ItemsPanelTemplate>
            <VirtualizingStackPanel Orientation="Vertical" />
        </ItemsPanelTemplate>
    </ListView.ItemsPanel>
    <ListView.ItemTemplate>
        <DataTemplate>
            <Border>
                <Image Source="{Binding Path=Thumbnail}"
                       Stretch="UniformToFill" />
            </Border>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

This **ListView** uses **HiloListViewItemStyle**, which is defined in ApplicationStyles.xaml, to customize the appearance of the **ListViewItem** instances that are displayed in the **ListView**. Each **ListViewItem** is an **Image** control, wrapped in a **Border** control, and is defined in the **DataTemplate** of the **ListView**. In order to support UI virtualization, a **VirtualizingStackPanel** is used to display the **ListViewItem** instances, and is defined in the **ItemsPanelTemplate** of the **ListView**. For more information about UI virtualization, see UI virtualization for working with large data sets in this guide.

When the app enters snapped view, we hide the photo grid that contains the **Image** control, and show the photos as a vertical film strip.

XAML: ImageView.xaml

```
<ObjectAnimationUsingKeyFrames Storyboard.TargetName="PhotoGrid"
                               Storyboard.TargetProperty="Visibility">
    <DiscreteObjectKeyFrame KeyTime="0"
                            Value="Collapsed"/>
</ObjectAnimationUsingKeyFrames>
<ObjectAnimationUsingKeyFrames Storyboard.TargetName="SnappedPhotosFilmStripListView"
                               Storyboard.TargetProperty="Visibility">
    <DiscreteObjectKeyFrame KeyTime="0"
                            Value="Visible"/>
</ObjectAnimationUsingKeyFrames>
```

We felt that the vertical layout that **ListView** provides made it a natural way to display groups of photos in snapped view. (**ListView** resembles **GridView**—they both inherit **ListViewBase**. The main difference is how they define their layout.)

For more info about **ListView**, see Adding ListView and GridView controls, XAML ListView and GridView essentials sample, and XAML ListView and GridView customizing interactivity sample.

SemanticZoom

The **SemanticZoom** control lets the user zoom between two views of a collection of items. For more info about how we use this control to help navigate between large sets of pictures, see Pinch and stretch to zoom in this guide.

Canvas and ContentControl

The **Canvas** control supports absolute positioning of child elements relative to the top left corner of the canvas. We used **Canvas** to control the position of the rectangle that you adjust to crop a photo.

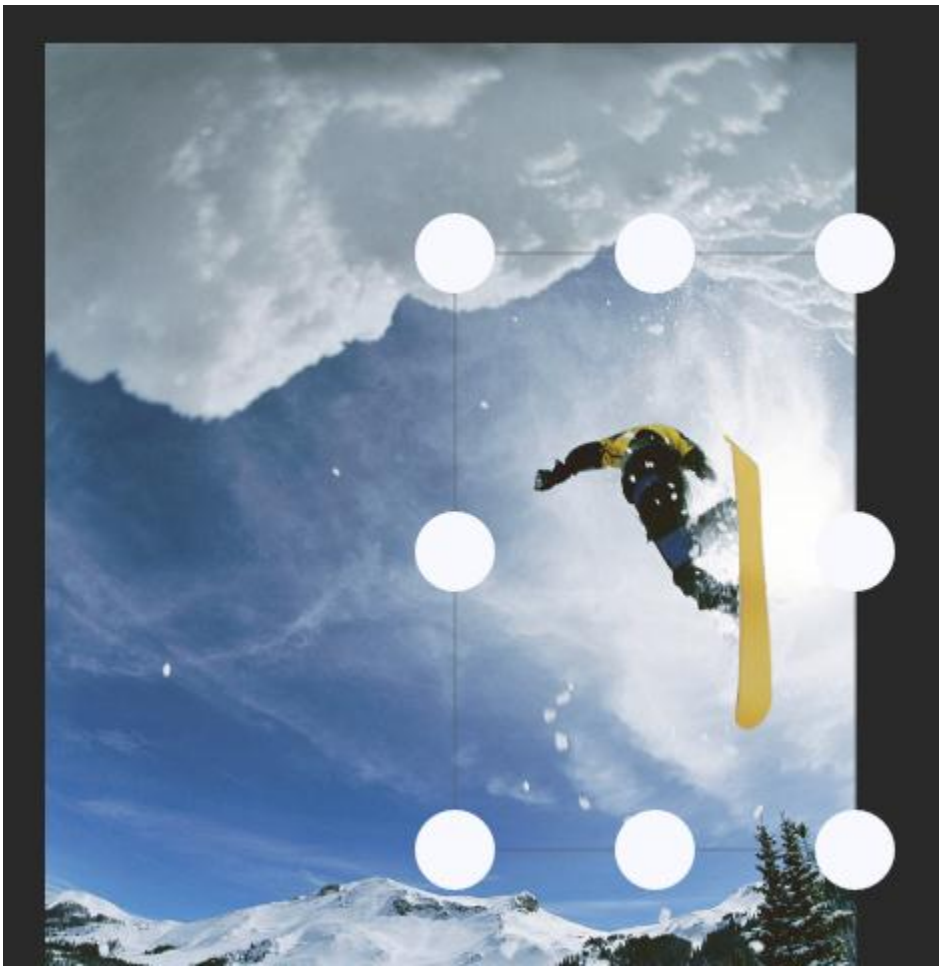
Tip Use a dynamic layout control, such as a **Grid**, when you don't need absolute positioning. A dynamic layout adapts automatically to different screen resolutions and orientations. Fixed layout controls, such as **Canvas** enable greater control, but you must manually update the layout when the orientation or page layout changes.

Here's the XAML for the **Canvas**.

XAML: CropImageView.xaml

```
<Canvas x:Name="CropCanvas" HorizontalAlignment="Left" VerticalAlignment="Top">
    <ContentControl x:Name="CropOverlay"
                    Canvas.Left="{Binding CropOverlayLeft}"
                    Canvas.Top="{Binding CropOverlayTop}"
                    Height="{Binding CropOverlayHeight}"
                    MinHeight="100"
                    MinWidth="100"
                    Template="{StaticResource HiloCroppingOverlayTemplate}"
                    Visibility="{Binding Path=IsCropOverlayVisible,
Converter={StaticResource BoolToVisConverter}}"
                    Width="{Binding CropOverlayWidth}">
        <Grid Background="Transparent"
              HorizontalAlignment="Stretch"
              VerticalAlignment="Stretch"
              Tapped="OnCropRectangleTapped">
        </Grid>
    </ContentControl>
</Canvas>
```

Here's what the Hilo crop UX looks like.



The **Canvas** control contains one child element—a **ContentControl** that implements the draggable points that enable the user to position the bounds of the crop rectangle. A **ContentControl** represents a control with a single piece of content. **Button** and other standard controls inherit from **ContentControl**. We use **ContentControl** to define custom content. The **ControlTemplate**, **HiloCroppingOverlayTemplate**, is defined in the **Resources** section of CropImageView.xaml. This **ControlTemplate** defines one **Thumb** element for each of the 8 draggable points. **HiloCroppingOverlayThumbStyle** defines the color, shape, and other style elements for each point, and is defined in ApplicationStyles.xaml.

XAML: CropImageView.xaml

```
<ControlTemplate x:Key="HiloCroppingOverlayTemplate" TargetType="ContentControl">
    <Grid>
        <Thumb Canvas.ZIndex="1"
               DragDelta="OnThumbDragDelta"
               HorizontalAlignment="Stretch"
               Style="{StaticResource HiloCroppingOverlayThumbStyle}"
               VerticalAlignment="Top" />
        <Thumb Canvas.ZIndex="1"
               DragDelta="OnThumbDragDelta"
               HorizontalAlignment="Left"
               Style="{StaticResource HiloCroppingOverlayThumbStyle}"
               VerticalAlignment="Stretch" />
        <Thumb Canvas.ZIndex="1"
               DragDelta="OnThumbDragDelta"
               HorizontalAlignment="Right"
               Style="{StaticResource HiloCroppingOverlayThumbStyle}"
               VerticalAlignment="Stretch" />
        <Thumb Canvas.ZIndex="1"
               DragDelta="OnThumbDragDelta"
               HorizontalAlignment="Stretch"
               Style="{StaticResource HiloCroppingOverlayThumbStyle}"
               VerticalAlignment="Bottom" />
        <Thumb Canvas.ZIndex="1"
               DragDelta="OnThumbDragDelta"
               HorizontalAlignment="Left"
               Style="{StaticResource HiloCroppingOverlayThumbStyle}"
               VerticalAlignment="Top" />
        <Thumb Canvas.ZIndex="1"
               DragDelta="OnThumbDragDelta"
               HorizontalAlignment="Right"
               Style="{StaticResource HiloCroppingOverlayThumbStyle}"
               VerticalAlignment="Top" />
        <Thumb Canvas.ZIndex="1"
               DragDelta="OnThumbDragDelta"
               HorizontalAlignment="Left"
               Style="{StaticResource HiloCroppingOverlayThumbStyle}"
               VerticalAlignment="Bottom" />
        <Thumb Canvas.ZIndex="1"
               DragDelta="OnThumbDragDelta"
               HorizontalAlignment="Right"
               Style="{StaticResource HiloCroppingOverlayThumbStyle}"
               VerticalAlignment="Bottom" />
        <ContentPresenter Content="{TemplateBinding ContentControl.Content}" />
    </Grid>
</ControlTemplate>
```

For more info about the crop UX, see Using touch in this guide.

Popup

We used the **Popup** control to display additional info about a picture when the user presses and holds the current picture. For more info on how we used this control, see Press and hold to learn in this guide.

Styling controls

Hilo's appearance was customized by styling and templating the controls used in the app.

Styles enable you to set control properties and reuse those settings for a consistent appearance across multiple controls. Styles are defined in XAML either inline for a control, or as a reusable resource. Resources can be defined in a page's XAML file, in the App.xaml file, or in a separate resource dictionary. A resource dictionary can be shared across apps, and more than one resource dictionary can be merged in a single app. For more information, see Quickstart: styling controls.

The structure and appearance of a control can be customized by defining a new **ControlTemplate** for the control. Templating a control often helps to avoid having to write custom controls. For more information, see Quickstart: control templates.

UI virtualization for working with large data sets

Because Hilo is a photo app, we needed to consider how to most efficiently load and display the user's pictures. We understood that the user might have hundreds of pictures for any given month, or just a few, and we wanted to enable a great UX for either case.

We also wanted to keep the amount of memory the app consumes to a minimum. The chances of a suspended or inactive app being terminated rises with the amount of memory it uses.

Because only a subset of photos are displayed at the same time, we make use of *UI virtualization*. UI virtualization enables controls that derive from **ItemsControl** (that is, controls that can be used to present a collection of items) to only load into memory those UI elements that are near the viewport, or visible region of the control. As the user scrolls through the list, elements that were previously near the viewport are unloaded from memory and new elements are loaded.

Note When used with data binding, UI virtualization stores the item containers in memory only for visible items, but still holds the entire data structure in memory. *Data virtualization* is a related concept where data structures for only the visible elements are held in memory. In Hilo, we wanted to use **FileInformationFactory::GetVirtualizedFilesVector** to bind to image files. We could bind to the returned data, but we couldn't manipulate it. Because we couldn't group the data by month as we desired, we didn't use data virtualization. If you want to use data virtualization in your app, you can use methods

such as **FileInformationFactory::GetVirtualizedFilesVector** if you don't need to manipulate data. You can also use the incremental loading feature of **ListView** and **GridView**. Otherwise, consider implementing your own data virtualization strategy.

Controls that derive from **ItemsControl**, such as **ListView** and **GridView**, perform UI virtualization by default. XAML generates the UI for the item and holds it in memory when the item is close to being visible on screen. When the item is no longer being displayed, the control reuses that memory for another item that is close to being displayed.

If you restyle an **ItemsControl** to use a panel other than its default panel, the control continues to support UI virtualization as long as it uses a virtualizing panel. Standard virtualizing panels include **WrapGrid** and **VirtualizingStackPanel**. Using standard non-virtualizing panels, which include **VariableSizedWrapGrid** and **StackPanel**, disables UI virtualization for that control.

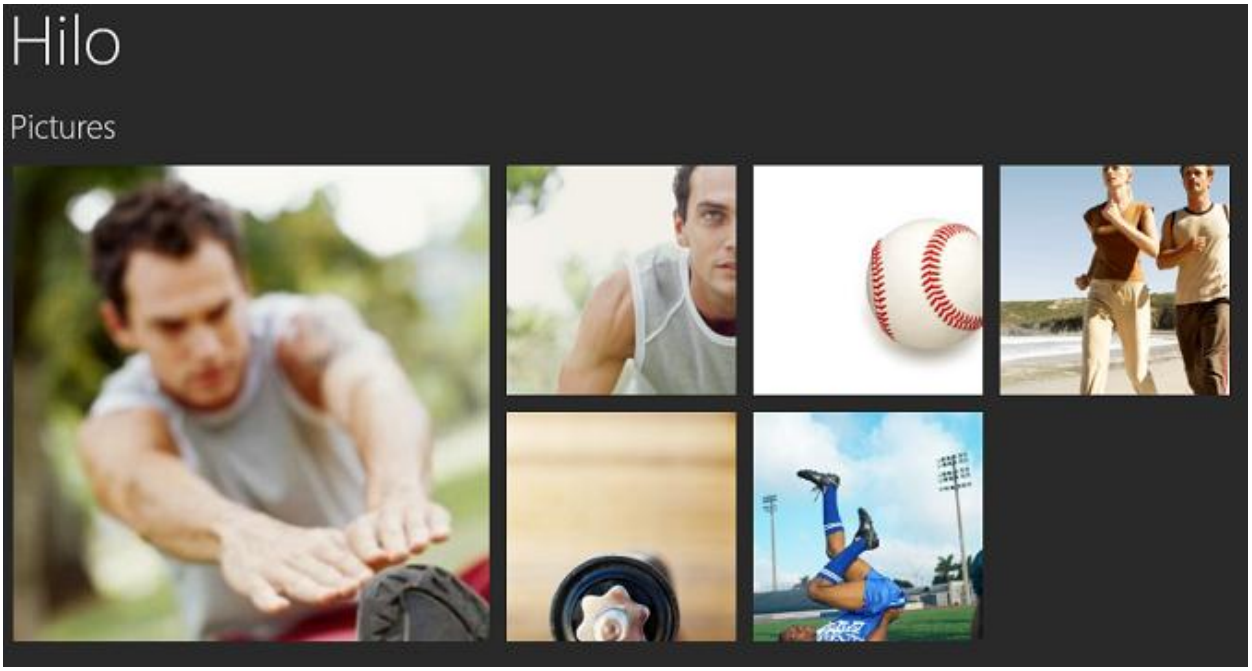
Note UI virtualization is not supported for grouped data. Therefore, in Hilo, we limited the size of groups to reduce the memory footprint of the app.

Tip You can also use Semantic Zoom to more efficiently work with large data sets. For more info about how we use Semantic Zoom in Hilo, see Pinch and stretch to zoom.

For more info about working with large data sets, see Using virtualization with a list or grid and Load, store, and display large sets of data efficiently. Read Tuning performance to learn about other performance considerations we made in Hilo.

Overriding built-in controls

On the hub page, we wanted the first picture to take the dimensions of four thumbnails with the remaining thumbnails being the normal width and height.



It can be challenging to define this kind of layout by using styles alone (in other words, to detect the first item in a collection and lay it out correctly). Instead, we defined the **VariableGridView** class, which derives from **GridView**.

C++: VariableGridView.h

```
[Windows::Foundation::Metadata::WebHostHidden]
public ref class VariableGridView sealed : public
Windows::UI::Xaml::Controls::GridView
{
protected:
    virtual void PrepareContainerForItemOverride(Windows::UI::Xaml::DependencyObject^
element, Platform::Object^ item) override;
};
```

We overrode the **PrepareContainerForItemOverride** method to enable the first image to span multiple rows and columns.

C++: VariableGridView.cpp

```
void VariableGridView::PrepareContainerForItemOverride(DependencyObject^ element,
Object^ item)
{
    auto model = dynamic_cast<IResizable^>(item);

    if (model != nullptr)
    {
        element->SetValue(VariableSizedWrapGrid::ColumnSpanProperty, model-
>ColumnSpan);
        element->SetValue(VariableSizedWrapGrid::RowSpanProperty, model->RowSpan);
    }

    GridView::PrepareContainerForItemOverride(element, item);
}
```

We define the **IResizable** interface to allow implementors to define the row and column spans in the parent container.

C++: IResizable.h

```
public interface class IResizable
{
    property int ColumnSpan
    {
        int get();
        void set(int value);
    }

    property int RowSpan
    {
        int get();
        void set(int value);
    }
};
```

The **Photo** class implements both **IPhoto** and **IResizable**. The constructor sets the default row and column span values to 1.

C++: Photo.h

```
[Windows::UI::Xaml::Data::Bindable]
[Windows::Foundation::Metadata::WebHostHidden]
public ref class Photo sealed : public IResizable, public IPhoto, public
Windows::UI::Xaml::Data::INotifyPropertyChanged
```

The **HubPhotoGroup::QueryPhotosAsync** method, which loads photos for the hub page, sets the column and row span values to 2 for the first photo in the collection.

C++: HubPhotoGroup.cpp

```
bool firstPhoto = true;
for (auto photo : photos)
{
    if (firstPhoto)
    {
        IResizable^ resizable = dynamic_cast<IResizable^>(photo);
        if (nullptr != resizable)
        {
            resizable->ColumnSpan = 2;
            resizable->RowSpan = 2;
        }
        firstPhoto = false;
    }
    m_photos->Append(photo);
}
```

The first photo in the hub view now occupies two rows and two columns.

Note In this solution, our model has control over the view and its layout. Although this violates the MVVM pattern (for more info, see Using the MVVM pattern), we tried to abstract this solution in such a way that it remains flexible. Because this solution targets a fairly narrow problem, we found the tradeoff between getting the app to work as we wanted and introducing some coupling between the view and the model, acceptable.

Touch and gestures

The XAML runtime provides built-in support for touch. Because the XAML runtime uses a common event system for many user interactions, you get automatic support for mouse, pen, and other pointer-based interactions.

Tip Design your app for the touch experience, and mouse and pen support come for free.

For more info about how we used touch in Hilo, see Using touch.

Testing controls

When you test your app, ensure that each control behaves as you expect in different configurations and orientations. When we used a control to add a new feature to the app, we ensured that the control behaved correctly in snap and fill views and under both landscape and portrait orientations.

If your monitor isn't touch-enabled, you can use the simulator to emulate pinch, zoom, rotation, and other gestures. You can also simulate geolocation and work with different screen resolutions. For more info, see Running Windows Store apps in the simulator.

You can test your app on a computer that doesn't have Visual Studio but has hardware you need to test. For more info, see Running Windows Store apps on a remote machine.

For more info on how we tested Hilo, see Testing apps.

Using touch in Hilo (Windows Store apps using C++ and XAML)

Summary

- Consider how each user interaction works on touch-enabled devices.
- Use the standard touch gestures and controls that Windows 8 provides when possible.
- Use XAML to bind standard Windows controls to the view model that implements the gesture behavior.

Hilo provides examples of tap, slide, swipe, pinch and stretch, and turn gestures. We use XAML data binding for connecting standard Windows controls that use touch gestures to the view model that implements those gestures. Here we briefly explain how we applied the Windows 8 touch language to Hilo to provide a great experience on any device.

You will learn

- How the Windows 8 touch language was used in Hilo.
- How the Windows Runtime supports non-touch devices.

Part of providing a great experience means that the app is accessible and intuitive to use on a traditional desktop computer, and on a small tablet. For Hilo, we put touch at the forefront of our UX planning because we felt that touch was key to providing an engaging interaction between the user and the app.

As described in Designing the Hilo UX, touch is more than simply an alternative to using the mouse. We wanted to make touch a first-class part of the app because touch can add a personal connection between the user and the app. Touch is also a very natural way to enable users to crop and rotate their photos. We also realized that Semantic Zoom would be a great way to help users navigate large sets of pictures in a single view. When the user uses the pinch gesture on the browse page, the app switches to a more calendar-based view. Users can then browse photos more quickly.

Hilo uses the Windows 8 touch language. We use the standard touch gestures that Windows provides for these reasons:

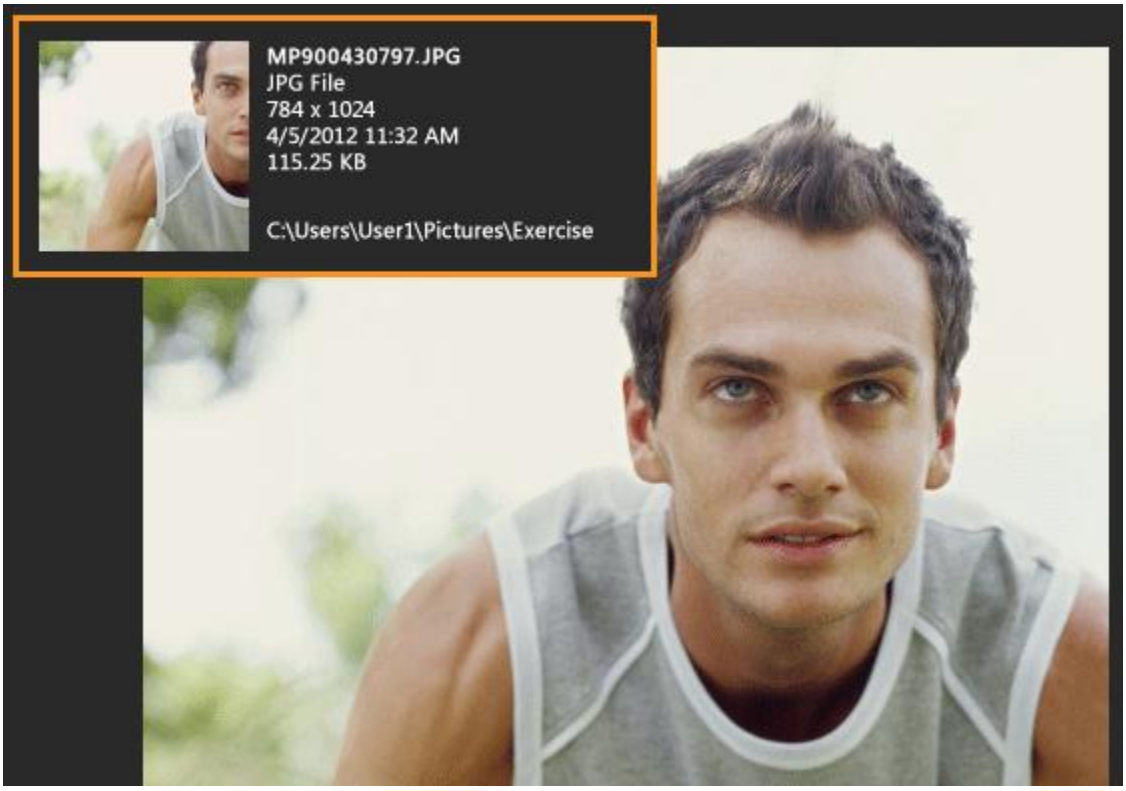
- The Windows Runtime provides an easy way to work with them.
- We didn't want to confuse users by creating custom interactions.
- We want users to use the gestures they already know to explore the app, and not need to learn new gestures.

Note Using the Model-View-ViewModel (MVVM) pattern plays an important role in defining user interaction. We use views to implement the app UI and models and view models to encapsulate the app's state and actions. For more info about MVVM, see Using the MVVM pattern.

The document Touch interaction design (Windows Store apps) explains the Windows 8 touch language. The following sections describe how we applied the Windows 8 touch language to Hilo.

Press and hold to learn

This touch interaction enables you to display a tooltip, context menu, or similar element without committing the user to an action. We use press and hold on the image view page to enable the user to learn about a photo, for example, its filename, dimensions, and date taken.



We used the **Popup** control to implement press and hold. The resulting popup menu contains a **Grid** that binds to photo data. We didn't have to set any properties to initially hide the popup because popups don't appear by default.

Here's the XAML for the **Popup**.

XAML: ImageView.xaml

```
<Popup x:Name="ImageViewFileInformationPopup"
       AutomationProperties.AutomationId="ImageViewFileInformationPopup">
    <Popup.Child>
        <Border Background="{StaticResource GreySplashScreenColor}"
                BorderBrush="{StaticResource HiloBorderBrush}"
                BorderThickness="3">
            <Grid Margin="10">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto"/>
                    <ColumnDefinition Width="Auto"/>
                </Grid.ColumnDefinitions>
                <Image Source="{Binding Path=SelectedItem.Thumbnail}"
                       Height="105"
                       Width="105"
                       Stretch="UniformToFill" />
                <StackPanel Grid.Column="1"
                            Margin="10,0,10,0">
                    <TextBlock Foreground="{StaticResource
ApplicationForegroundThemeBrush}"
                               FontWeight="Bold"
                               Text="{Binding Path=SelectedItem.Name}"
                               TextWrapping="Wrap"/>
                    <TextBlock Foreground="{StaticResource
ApplicationForegroundThemeBrush}"
                               Text="{Binding Path=SelectedItem.DisplayType}"/>
                    <TextBlock Foreground="{StaticResource
ApplicationForegroundThemeBrush}"
                               Text="{Binding Path=SelectedItem.Resolution}" />
                    <StackPanel Orientation="Horizontal">
                        <TextBlock Foreground="{StaticResource
ApplicationForegroundThemeBrush}"
                                   Text="{Binding
Path=SelectedItem.FormattedDateTaken}" />
                        <TextBlock Foreground="{StaticResource
ApplicationForegroundThemeBrush}"
                                   Margin="4,0,0,0"
                                   Text="{Binding
Path=SelectedItem.FormattedTimeTaken}" />
                    </StackPanel>
                    <TextBlock Foreground="{StaticResource
ApplicationForegroundThemeBrush}"
                               Text="{Binding Path=SelectedItem.FileSize,
Converter={StaticResource FileSizeConverter}}" />
                    <TextBlock Foreground="{StaticResource
ApplicationForegroundThemeBrush}"
```

```
                                Margin="0,20,0,0"
                                Text="{Binding Path=SelectedItem.FormattedPath}" />
                </StackPanel>
            </Grid>
        </Border>
    </Popup.Child>
</Popup>
```

The **OnImagePointerPressed**, **OnImagePointerReleased**, and **OnImagePointerMoved** methods handle the **PointerPressed**, **PointerReleased**, and **PointerMoved** events, respectively. The **OnImagePointerPressed** method sets a flag to indicate that the pointer is pressed. If the left button is pressed (the **IsLeftButtonPressed** method always returns **true** for touch input), the popup is placed 200 pixels to the left and top of the current touch point.

C++: ImageView.xaml.cpp

```
void Hilo::ImageView::OnImagePointerPressed(Object^ sender, PointerRoutedEventArgs^
e)
{
    m_pointerPressed = true;
    PointerPoint^ point = e->GetCurrentPoint(PhotoGrid);
    m_pointer = point->Position;
    if (point->Properties->IsLeftButtonPressed)
    {
        ImageViewFileInformationPopup->HorizontalOffset = point->Position.X - 200;
        ImageViewFileInformationPopup->VerticalOffset = point->Position.Y - 200;
        ImageViewFileInformationPopup->IsOpen = true;
    }
}
```

Note We present the popup to the left of the pointer because most people are right-handed. If the user is touching with the index finger, the user's right hand could block any popup that appears to the right of the pointer.

The **OnImagePointerReleased** method closes the popup. It's called when the user releases the pointer.

C++: ImageView.xaml.cpp

```
void Hilo::ImageView::OnImagePointerReleased(Object^ sender, PointerRoutedEventArgs^
e)
{
    if (m_pointerPressed)
    {
        ImageViewFileInformationPopup->IsOpen = false;
        m_pointerPressed = false;
    }
}
```

The **OnImagePointerMoved** method also closes the popup. The **OnImagePointerPressed** method sets a flag and saves the current pointer position because the **OnImagePointerMoved** is also called when the pointer is initially pressed. Therefore, we need to compare the current position to the saved position and close the popup only if the position has changed.

C++: ImageView.xaml.cpp

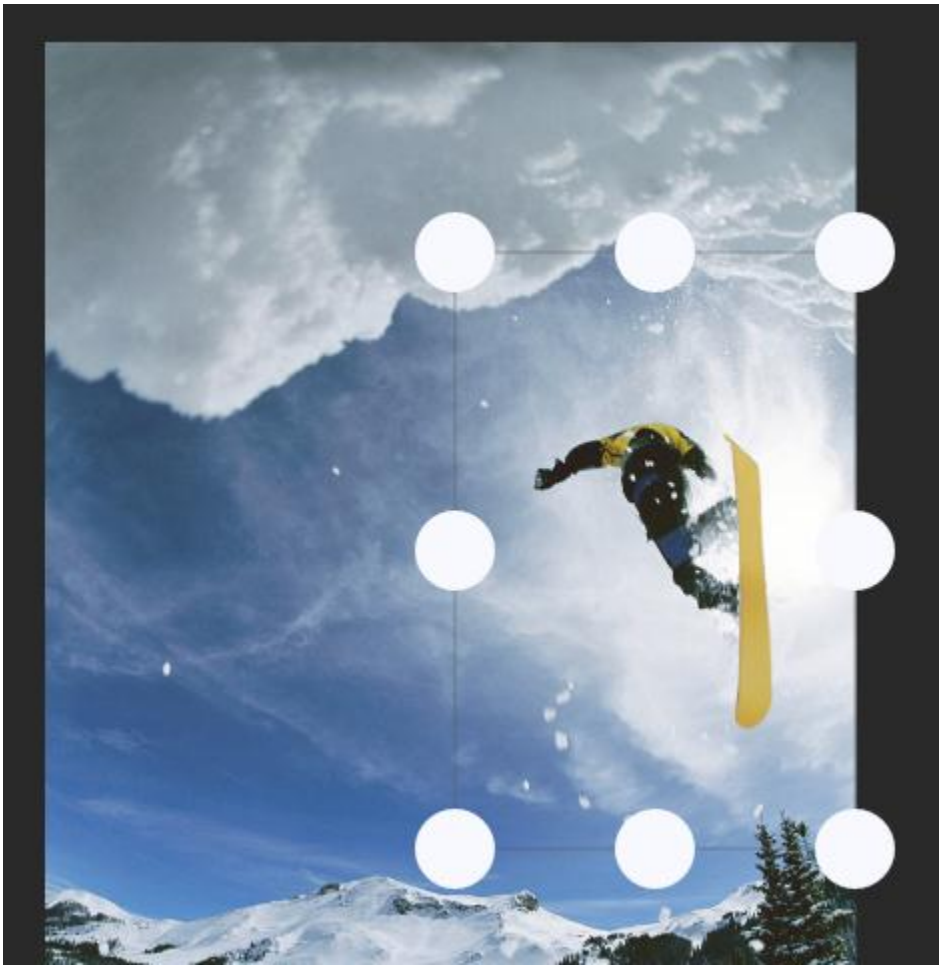
```
void Hilo::ImageView::OnImagePointerMoved(Object^ sender, PointerRoutedEventArgs^ e)
{
    if (m_pointerPressed)
    {
        PointerPoint^ point = e->GetCurrentPoint(PhotoGrid);
        if (point->Position != m_pointer)
        {
            OnImagePointerReleased(sender, e);
        }
    }
}
```

The **Popup** also defines a **Converter** object that converts from bytes to kilobytes or megabytes to make the image size on disk more human-readable. We found converters to be a convenient way to bind to data, that we want to display in different ways. For more info about converters, see Data converters in this guide.

Why not use the Holding event? We considered using the **UIElement::Holding** event. However, this event is for apps that respond only to touch input, and not other pointer input such as the mouse. We also looked at the **GestureRegognizer::Holding** event. However, **GestureRecognizer** isn't recommended for XAML apps because XAML controls provide gesture recognition functionality for you.

Tap for primary action

Tapping an element invokes its primary action. For example, in crop mode, you tap the crop area to crop the image.



To implement crop, we placed a **Grid** element on top of the crop control and implemented the **Tapped** event. We use a **Grid** control so that the entire crop area receives tap events, not just the control that defines the crop rectangle.

Here's the XAML for the **Grid**.

XAML: CropImageView.xaml

```
<Grid Background="Transparent"
      HorizontalAlignment="Stretch"
      VerticalAlignment="Stretch"
      Tapped="OnCropRectangleTapped">
</Grid>
```

The view for the crop image page handles the **Tapped** event by forwarding the operation to the view model.

C++: CropImageView.xaml.cpp

```
void CropImageView::OnCropRectangleTapped(Object^ sender, TappedRoutedEventArgs^ e)
{
    if (!m_cropImageViewModel->InProgress)
    {
        m_cropImageViewModel->CropImageAsync(Photo->ActualWidth);
    }
}
```

The view model for the crop image page crops the image and then calls Hilo's **BindableBase::OnPropertyChanged** method to communicate to the view that the operation has completed. Using the MVVM pattern, in this guide, explains how Hilo uses property changed notifications to communicate state changes.

C++: CropImageViewModel.cpp

```
task<void> CropImageViewModel::CropImageAsync(float64 actualWidth)
{
    assert(IsMainThread());
    ChangeInProgress(true);

    // Calculate crop values
    float64 scaleFactor = m_image->PixelWidth / actualWidth;
    unsigned int xOffset = safe_cast<unsigned int>((m_cropOverlayLeft - m_left) *
scaleFactor);
    unsigned int yOffset = safe_cast<unsigned int>((m_cropOverlayTop - m_top) *
scaleFactor);
    unsigned int newWidth = safe_cast<unsigned int>(m_cropOverlayWidth *
scaleFactor);
    unsigned int newHeight = safe_cast<unsigned int>(m_cropOverlayHeight *
scaleFactor);

    if (newHeight < MINIMUMBMPSIZE || newWidth < MINIMUMBMPSIZE)
    {
        ChangeInProgress(false);
        m_isCropOverlayVisible = false;
        OnPropertyChanged("IsCropOverlayVisible");
        return create_empty_task();
    }

    m_cropX += xOffset;
    m_cropY += yOffset;

    // Create destination bitmap
    WriteableBitmap^ destImage = ref new WriteableBitmap(newWidth, newHeight);
```

```
    // Get pointers to the source and destination pixel data
    byte* pSrcPixels = GetPointerToPixelData(m_image->PixelBuffer);
    byte* pDestPixels = GetPointerToPixelData(destImage->PixelBuffer);
    auto oldWidth = m_image->PixelWidth;

    return create_task([this, xOffset, yOffset,
        newHeight, newWidth, oldWidth, pSrcPixels, pDestPixels] () {
        assert(IsBackgroundThread());
        DoCrop(xOffset, yOffset, newHeight, newWidth, oldWidth, pSrcPixels,
pDestPixels);
    }).then([this, destImage](){
        assert(IsMainThread());

        // Update image on screen
        m_image = destImage;
        OnPropertyChanged("Image");
        ChangeInProgress(false);
    }, task_continuation_context::use_current())
        .then(ObserveException<void>(m_exceptionPolicy));
}
```

Slide to pan

Hilo uses the slide gesture to navigate between images in a collection. For example, when you browse to an image, you can use the slide gesture to navigate to the previous or next image in the collection.

When you view an image, you can also quickly pan to any image in the current collection by using the filmstrip that appears when you activate the app bar. We used the **GridView** control to implement the filmstrip.

Here's the XAML for the **GridView**.

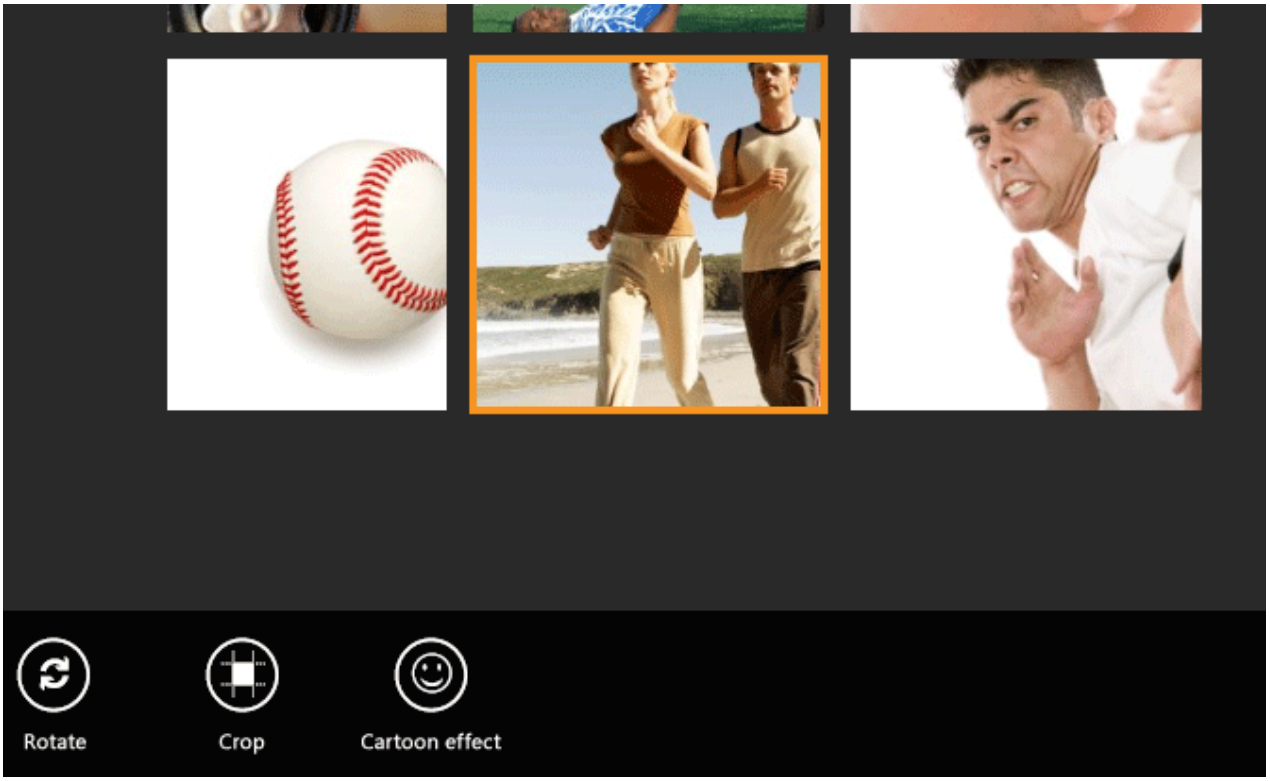
XAML: ImageView.xaml

```
<GridView x:Name="PhotosFilmStripGridView"
          Grid.Column="1"
          AutomationProperties.AutomationId="PhotosFilmStripGridView"
          IsItemClickEnabled="False"
          ItemContainerStyle="{StaticResource FilmStripGridViewItemStyle}"
          ItemsSource="{Binding Photos}"
          SelectedItem="{Binding Path=SelectedItem, Mode=TwoWay}"
          SelectionMode="Single"
          VerticalAlignment="Center">
    <GridView.ItemsPanel>
        <ItemsPanelTemplate>
            <VirtualizingStackPanel Height="138" Orientation="Horizontal" />
        </ItemsPanelTemplate>
    </GridView.ItemsPanel>
    <GridView.ItemTemplate>
        <DataTemplate>
            <Border>
                <Image Source="{Binding Path=Thumbnail}"
                       Height="138"
                       Width="200"
                       Stretch="UniformToFill" />
            </Border>
        </DataTemplate>
    </GridView.ItemTemplate>
</GridView>
```

A benefit of using the **GridView** control is that it has touch capabilities built in, removing the need for additional code or logic.

Swipe to select, command, and move

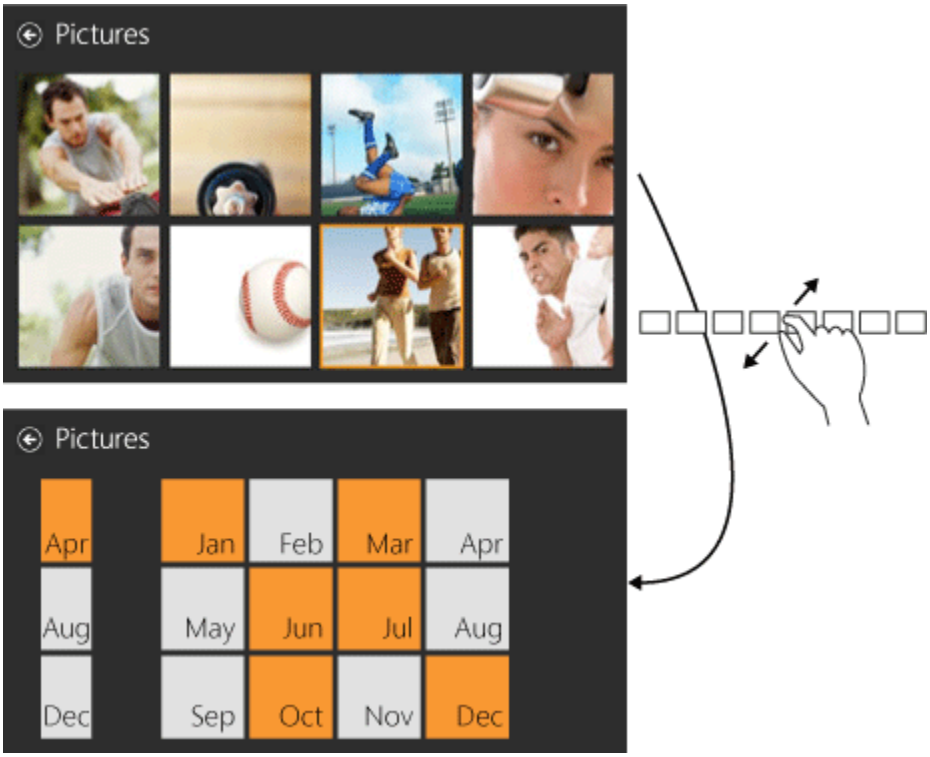
With the swipe gesture, you slide your finger perpendicular to the panning direction to select objects. In Hilo, when a page contains multiple images, you can use this gesture to select one image. When you display the app bar, the commands that appear apply to the selected image. **ListView**, **GridView**, and other controls provide built-in support for selection. You can use the **SelectedItem** property to retrieve or set the selected item.



Pinch and stretch to zoom

Pinch and stretch gestures are not just for magnification, or performing "optical" zoom. Hilo uses *Semantic Zoom* to help navigate between large sets of pictures. Semantic Zoom enables you to switch between two different views of the same content. You typically have a main view of your content and a second view that allows users to quickly navigate through it. (Read Adding SemanticZoom controls for more info on Semantic Zoom.)

On the image browser page, when you zoom out, the view changes to a calendar-based view. The calendar view highlights those months that contain photos. You can then zoom back in to the image-based month view.



To implement Semantic Zoom, we used the **SemanticZoom** control. You provide **ZoomedInView** and **ZoomedOutView** sections in your XAML to define the zoomed-in and zoomed-out behaviors, respectively.

For the zoomed-in view, we display a **GridView** that binds to photo thumbnails that are grouped by month. The grid view also shows a title (the month and year) for each group. The **ImageBrowserViewModel::MonthGroups** property and Hilo's **MonthGroup** class define which photos are displayed on the grid. The **MonthGroup** class also defines the title of the group.

Here's the XAML for the **ZoomedInView**.

XAML: ImageBrowserView.xaml

```
<SemanticZoom.ZoomedInView>
    <GridView x:Name="MonthPhotosGridView"
              AutomationProperties.AutomationId="MonthPhotosGridView"
              AutomationProperties.Name="Month Grouped Photos"
              ItemsSource="{Binding Source={StaticResource
MonthGroupedItemsViewSource}}"
              ItemContainerStyle="{StaticResource HiloGridViewItemStyle}"
              ItemClick="OnPhotoItemClicked"
              IsItemClickEnabled="True"
              Padding="116,0,40,46"
              SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
```

```
                SelectionMode="Single">
```

The **ItemsSource** property specifies the items for the control. **MonthGroupedItemsViewSource** is a **CollectionViewSource** that provides the source data for the control.

XAML: ImageBrowserView.xaml

```
<CollectionViewSource
    x:Name="MonthGroupedItemsViewSource"
    d:Source="{Binding MonthGroups, Source={d:DesignInstance
Type=local:DesignTimeData, IsDesignTimeCreatable=True}}"
    Source="{Binding MonthGroups}"
    IsSourceGrouped="true"
    ItemsPath="Items"/>
```

The **MonthGroups** property on the view model **(ImageBrowserViewModel**) specifies the data that is bound to the grid view for the zoomed-in view. **MonthGroups** is a collection of **IPhotoGroup** objects. Hilo defines the **IPhotoGroup** interface to provide the title of the group and info about each photo.

For the zoomed-out view, we display a **GridView** that binds to filled rectangles for calendar months that are grouped by year (Hilo's **ImageBrowserViewModel::YearGroups** property, **YearGroup**, and **MonthBlock** classes define the title and which months contain photos).

Here's the XAML for the **ZoomedOutView**.

XAML: ImageBrowserView.xaml

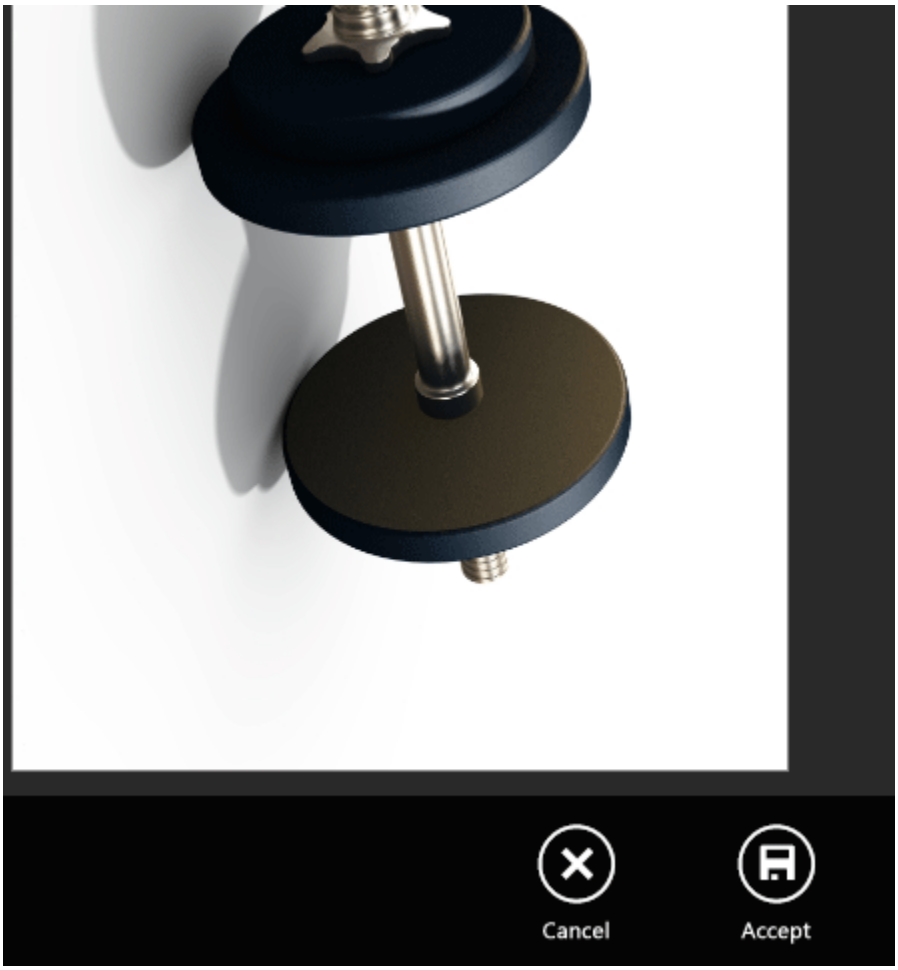
```
<SemanticZoom.ZoomedOutView>
    <GridView x:Name="YearPhotosGridView"
              AutomationProperties.AutomationId="YearPhotosGridView"
              AutomationProperties.Name="Year Grouped Photos"
              ItemsSource="{Binding Source={StaticResource
YearGroupedItemsViewSource}}"
              IsItemClickEnabled="True"
              Padding="116,0,40,46"
              SelectionMode="None">
```

The **YearGroupedItemsViewSource** is similar to **MonthGroupedItemsViewSource**, except that it binds to data that is grouped by year.

For more info about Semantic Zoom, see Quickstart: adding SemanticZoom controls, Adding SemanticZoom controls, Guidelines for Semantic Zoom, and XAML GridView grouping and SemanticZoom sample.

Turn to rotate

On the rotate image view, you can use two fingers to rotate the image. When you release your fingers, the image snaps to the nearest 90-degree rotation.



The **RotateImageView** class defines the UX for rotating an image and the **RotateImageViewModel** class defines its view model. Because **RotateImageView** inherits from **Control**, it is set up to receive events when the user manipulates objects. To receive manipulation events on the image control for the rotation gesture, Hilo sets the **ManipulationMode** property in the XAML for the **Image**.

Here's the XAML for the **Image**.

XAML: RotateImageView.xaml

```
<Image x:Name="Photo"
       AutomationProperties.AutomationId="ImageControl"
       HorizontalAlignment="Center"
       ManipulationMode="Rotate"
```

```
        Margin="{Binding ImageMargin}"
        RenderTransformOrigin="0.5, 0.5"
        Source="{Binding Photo.Image}"
        VerticalAlignment="Center">
```

The **RotateImageView** class overrides the **OnManipulationDelta** and **OnManipulationCompleted** methods to handle rotation events. The **OnManipulationDelta** method updates the current rotation angle and the **OnManipulationCompleted** method snaps the rotation to the nearest 90-degree value.

Caution These event handlers are defined for the entire page. If your page contains more than one item, you would need additional logic to determine which object to manipulate.

C++: RotateImageView.xaml.cpp

```
void RotateImageView::OnManipulationDelta(ManipulationDeltaRoutedEventArgs^ e)
{
    m_viewModel->RotationAngle += e->Delta.Rotation;
}

void RotateImageView::OnManipulationCompleted(ManipulationCompletedRoutedEventArgs^
e)
{
    m_viewModel->EndRotation();
}
```

C++: RotateImageViewModel.cpp

```
void RotateImageViewModel::EndRotation()
{
    auto quarterTurns = (RotationAngle / 90);
    auto nearestQuarter = (int)floor(quarterTurns + 0.5) % 4;
    RotationAngle = (float64)nearestQuarter * 90;
}
```

A **RenderTransform** on the image binds the **RotationAngle** property on the view model to the displayed image.

XAML: RotateImageView.xaml

```
<Image.RenderTransform>
    <RotateTransform
        x:Name="ImageRotateTransform"
        Angle="{Binding RotationAngle}" />
</Image.RenderTransform>
```

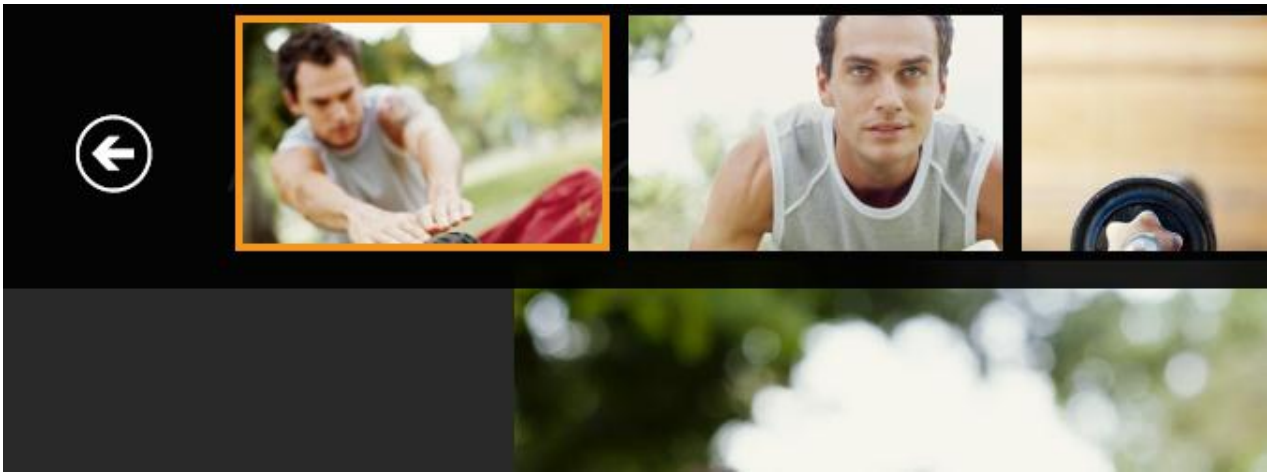
Hilo uses a view model locator to associate views with their view models. For more info about how we use view model locators in Hilo, see Using the MVVM pattern.

Swipe from edge for app commands

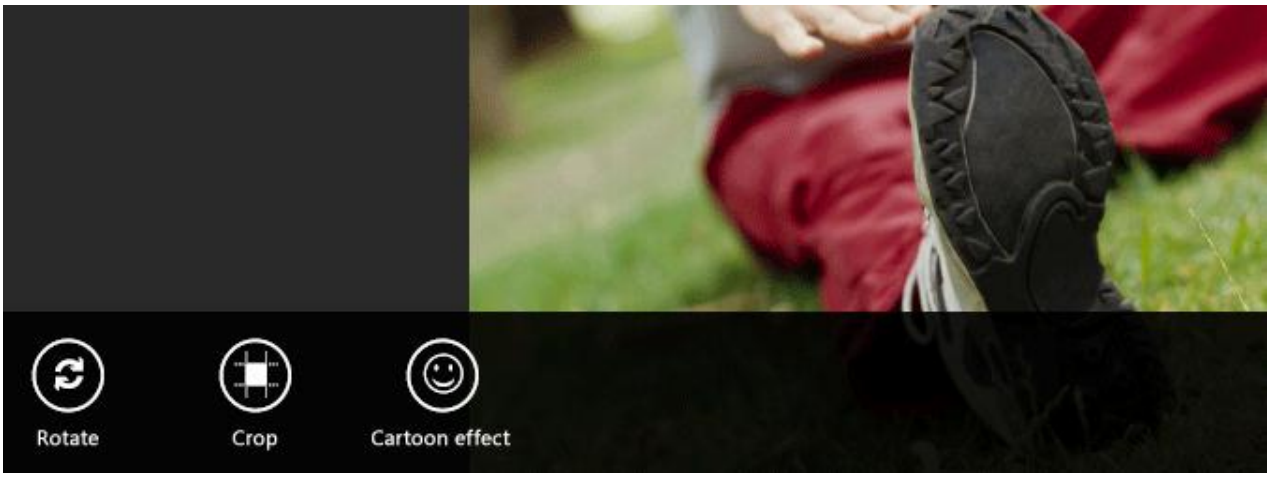
When there are relevant commands to show, Hilo shows the app bar when the user swipes from the bottom or top edge of the screen.

Every page can define a top app bar, a bottom app bar, or both. For instance, Hilo shows both when you view an image and activate the app bar.

Here's the top app bar:



Here's the bottom app bar for the same photo.



From XAML, use the **Page::TopAppBar** property to define the top app bar and the **Page::BottomAppBar** property to define the bottom app bar. Each of these contains an **AppBar** control that holds the app

bar's UI components. The following shows the XAML for the bottom app bar for the image view page. This app bar contains the commands to enter the modes to rotate, crop, or apply the cartoon effect to the image.

XAML: ImageView.xaml

```
<local:HiloPage.BottomAppBar>
    <AppBar x:Name="ImageViewBottomAppBar"
            x:Uid="AppBar"
            AutomationProperties.AutomationId="ImageViewBottomAppBar"
            Padding="10,0,10,0">
        <Grid>
            <StackPanel HorizontalAlignment="Left"
                        Orientation="Horizontal">
                <Button x:Name="RotateButton"
                        x:Uid="RotateAppBarButton"
                        Command="{Binding RotateImageCommand}"
                        Style="{StaticResource RotateAppBarButtonStyle}"
                        Tag="Rotate" />
                <Button x:Name="CropButton"
                        x:Uid="CropAppBarButton"
                        Command="{Binding CropImageCommand}"
                        Style="{StaticResource CropAppBarButtonStyle}"
                        Tag="Crop" />
                <Button x:Name="CartoonizeButton"
                        x:Uid="CartoonizeAppBarButton"
                        Command="{Binding CartoonizeImageCommand}"
                        Style="{StaticResource CartoonEffectAppBarButtonStyle}"
                        Tag="Cartoon effect" />
                <Button x:Name="RotateButtonNoLabel"
                        Command="{Binding RotateImageCommand}"
                        Style="{StaticResource RotateAppBarButtonNoLabelStyle}"
                        Tag="Rotate"
                        Visibility="Collapsed">
                    <ToolTipService.ToolTip>
                        <ToolTip x:Uid="RotateAppBarButtonToolTip" />
                    </ToolTipService.ToolTip>
                </Button>
                <Button x:Name="CropButtonNoLabel"
                        Command="{Binding CropImageCommand}"
                        Style="{StaticResource CropAppBarButtonNoLabelStyle}"
                        Tag="Crop"
                        Visibility="Collapsed">
                    <ToolTipService.ToolTip>
                        <ToolTip x:Uid="CropAppBarButtonToolTip" />
                    </ToolTipService.ToolTip>
                </Button>
                <Button x:Name="CartoonizeButtonNoLabel"
```

```
                        Command="{Binding CartoonizeImageCommand}"
                        Style="{StaticResource
CartoonEffectAppBarButtonNoLabelStyle}"
                        Tag="Cartoon effect"
                        Visibility="Collapsed">
                    <ToolTipService.ToolTip>
                        <ToolTip x:Uid="CartoonizeAppBarButtonToolTip" />
                    </ToolTipService.ToolTip>
                </Button>
            </StackPanel>
        </Grid>
    </AppBar>
</local:HiloPage.BottomAppBar>
```

Caution In most cases, don't display the app bar if there are no relevant commands to show. For example, the main hub page binds the **AppBar::IsOpen** property to the **MainHubViewModel::IsAppBarOpen** property on the view model. **AppBar::IsOpen** controls whether the app bar is visible or not, and is set when an object is selected and cleared when no object is selected.

For more info about app bars, see Adding app bars and XAML AppBar control sample.

Swipe from edge for system commands

Because touch interaction can be less precise than other pointing devices, such as the mouse, we maintained a sufficient margin between the app controls and the edge of the screen. We also strived to use the same margin on all pages. We relied on the project templates that come with Visual Studio to provide suggested margins. Because we didn't "crowd the edges" of the screen, the user can easily swipe from the edge of the screen to reveal the app bars, charms, or display previously used apps.

What about non-touch devices?

We also wanted Hilo to be intuitive for users who use a mouse or similar pointing device. The built-in controls work equally well with the mouse and other pointing devices as they do with touch. So when you design for touch, you also get the mouse and pen/stylus for free.

For example, you can use the left mouse button to invoke commands. The Windows Runtime also provides mouse and keyboard equivalents for many commands. (For example, you can use the right mouse button to activate the app bar; holding the Ctrl key down while scrolling the mouse scroll wheel controls Semantic Zoom interaction.)

We used the **PointerPoint** class to get basic properties for the mouse, pen/stylus, and touch input. Here's an example where we use **PointerPoint** to get the pointer's position.

C++: ImageView.xaml.cpp

```
void Hilo::ImageView::OnImagePointerPressed(Object^ sender, PointerRoutedEventArgs^
e)
{
    m_pointerPressed = true;
    PointerPoint^ point = e->GetCurrentPoint(PhotoGrid);
    m_pointer = point->Position;
    if (point->Properties->IsLeftButtonPressed)
    {
        ImageViewFileInformationPopup->HorizontalOffset = point->Position.X - 200;
        ImageViewFileInformationPopup->VerticalOffset = point->Position.Y - 200;
        ImageViewFileInformationPopup->IsOpen = true;
    }
}
```

Handling suspend, resume, and activation in Hilo (Windows Store apps using C++ and XAML)

Summary

- Save application data when the app is being suspended.
- Release exclusive resources and file handles when the app is being suspended.
- Use the saved application data to restore the app when needed.

Hilo provides examples of how to suspend and resume an app that uses C++ and XAML. You can use these examples to write an app that fully manages its execution life cycle. Suspension can happen at any time, and when it does you need to save your app's data so that the app can resume correctly.

You will learn

- How the app's activation history affects its behavior.
- How to implement support for suspend and resume using C++ and XAML.

Tips for implementing suspend/resume

You should design your app to suspend correctly when the user switches away from it, or when there is a low power state. You should also design the app to resume correctly when the user switches back to it, or when Windows leaves the low power state. Here are some points to remember.

- Save application data when the app is being suspended.
- Resume your app to the state the user left it in, rather than starting afresh.
- Allow views and view models to save and restore state that is relevant to each. For example, if the user has typed text into a text box but has not yet tabbed out of the text box, you may want to save the partially entered text as view state. In Hilo, only view models need to save state.
- Release exclusive resources when the app is being suspended.
- When the app resumes, update the UI if the content has changed.
- When the app resumes after it was terminated, use the saved application data to restore the app state.

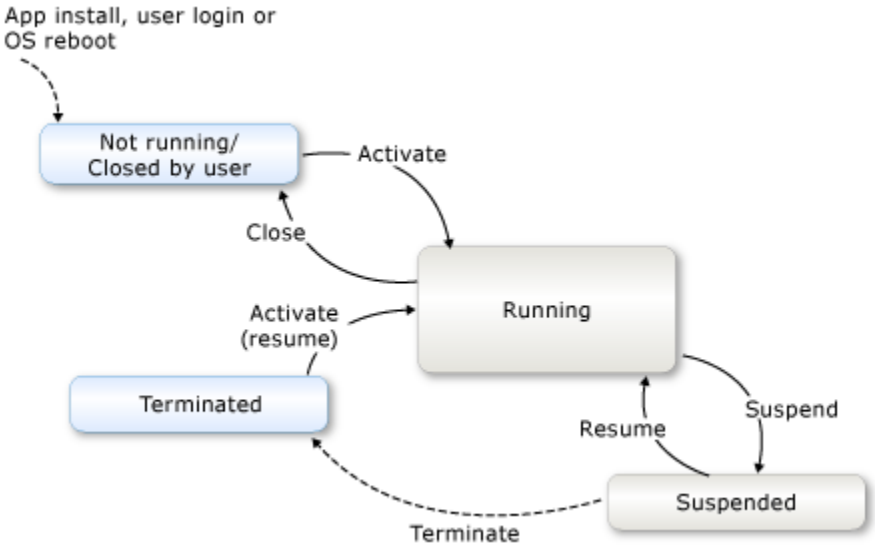
See Guidelines for app suspend and resume (Windows Store apps).

Understanding possible execution states

Which events occur when you activate an app depends on the app's execution history. There are five cases to consider. The cases correspond to the values of the **Windows::ActivationModel::Activation::ApplicationExecutionState** enumeration.

- **NotRunning**
- **Terminated**
- **ClosedByUser**
- **Suspended**
- **Running**

Here's a diagram that shows how Windows determines the app's execution state. In the diagram, the blue rectangles indicate that the app is not loaded into system memory. The white rectangles indicate that the app is in memory. The dashed arcs are changes that occur without any notification to your running app. The solid arcs are actions that include app notification.



The execution state depends on the app's history. For example, when the user launches the app for the first time after installing it or after restarting Windows, the previous execution state is **NotRunning**, and the state after activation is **Running**. The activation event arguments include a **PreviousExecutionState** property that tells you the state your app was in before it was activated.

If the user switches to a different app or if the system enters a low power mode of operation, Windows notifies the app that it is being suspended. At this time, you must save the navigation state and all user data that represents the user's session. You should also free exclusive system resources such as open files and network connections.

Windows allows 5 seconds for the app to handle the **Suspending** event. If the **Suspending** event handler does not complete within that amount of time, Windows assumes that the app has stopped responding and terminates it.

After the app responds to the **Suspending** event, its state is **Suspended**. If the user switches back to the app, Windows resumes it and allows it to run again.

Windows may terminate the app (without notification) after it has been suspended. For example, if the system is low on resources it might decide to reclaim resources that are held by suspended apps. If the user launches your app after Windows has terminated it, the app's previous execution state at the time of activation is **Terminated**.

You can use the previous execution state to determine whether your app needs to restore the data that it saved when it was last suspended, or whether you must load your app's default data. In general, if the app crashes or the user closes it, restarting the app should take the user to the app's default initial navigation state. When an app determines that it is activated after being terminated, it should load the application data that it saved during suspension so that the app appears as it did when it was suspended. You can see how this works in the code walkthrough of Hilo's support for suspend and resume on this page.

Note When the app is in suspended, but hasn't yet been terminated, you can resume the app without restoring any state. The app will still be in memory. In this situation, you might need to reacquire resources and update the UI to reflect any changes to the environment that have occurred while the app was suspended. For example, Hilo updates its view of the user's Pictures library when resuming from the suspended state.

See ApplicationExecutionState enumeration for more info about each of the possible previous execution states. See Application lifecycle (Windows Store apps) for a description of the suspend/resume process. You may also want to consult Guidelines for app suspend and resume (Windows Store apps) for info on the recommended user experience for suspend and resume.

Implementation approaches for suspend and resume in C++ and XAML

For Windows Store apps such as Hilo that use a Visual Studio project template, implementing suspend/resume involves four components of the system.

- **Windows Core**. The **Windows::ApplicationModel::Core::CoreApplicationView** class's **Activated** event allows an app to receive activation-related notifications.
- **XAML**. The **Windows::UI::Xaml::Application** class provides the **OnLaunched** method that your app's **App** class should override to perform application initialization and to display its initial content. XAML's **Application** class invokes the **OnLaunched** method when the user starts the app. When you create a new project for a Windows Store app using one of the Microsoft Visual

Studio project templates for C++ and XAML, Visual Studio creates an **App** class that derives from **Windows::UI::Xaml::Application** and overrides the **OnLaunched** method. You can specialize the code that Visual Studio creates, or use it without changes.

- **Visual Studio template classes**. When you create a new project for a Windows Store app, Visual Studio creates several classes for you in the Common solution folder. One of these classes is the **SuspensionManager** class. This class provides methods that work with the **Windows::UI::Xaml::Application** class to make it convenient to save and restore app state. The **SuspensionManager** class interacts with the **Windows::UI::Xaml::Frame** class to save and restore the app's navigation stack for you. The **SuspensionManager** class interacts with the **LayoutAwarePage** template class to integrate saving page state at the appropriate times.
- **Your app's page classes**. Hilo saves navigation state with each invocation of their **OnNavigatedFrom** method. Hilo pages delegate saving state to their associated view model classes. Hilo view models implement the **LoadState** and **SaveState** methods.

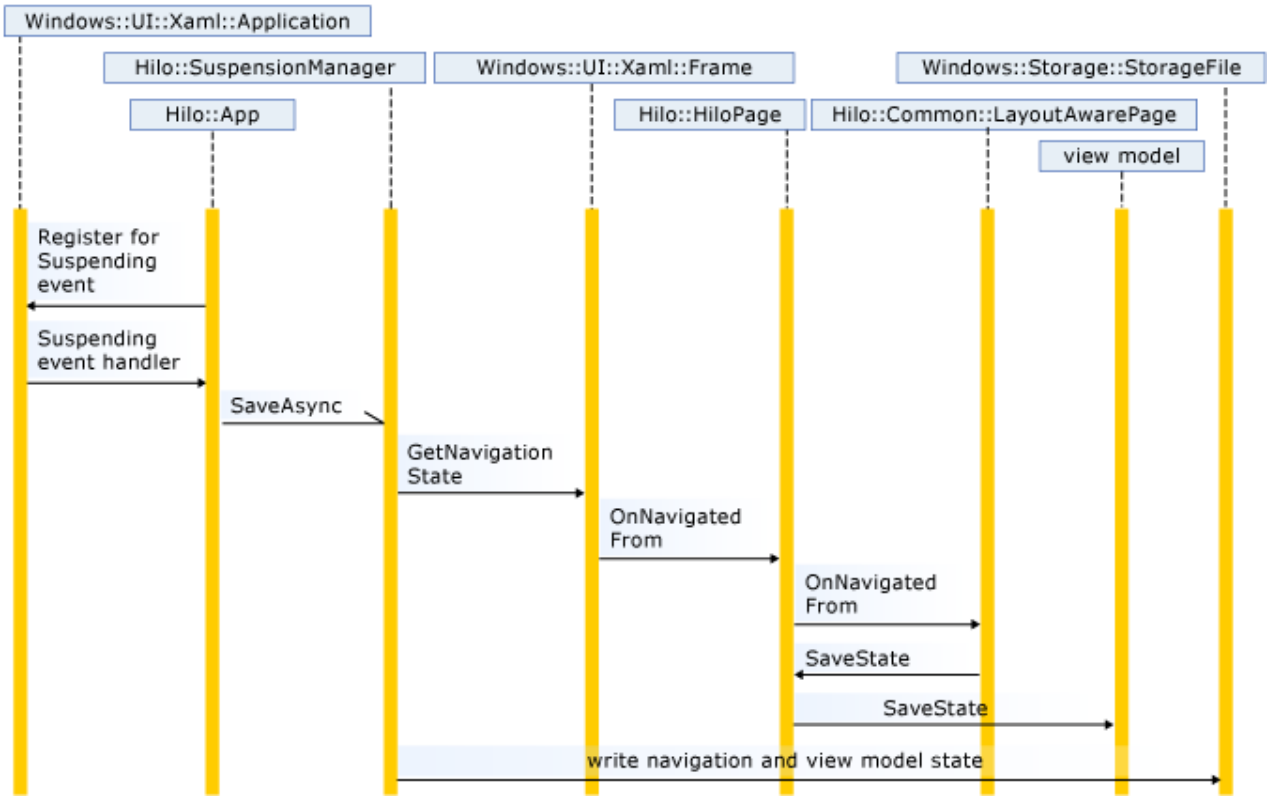
Note Hilo does not directly interact with the **CoreApplicationView** class's activation-related events. We mention them here in case your app needs access to these lower-level notifications.

Note A user can activate an app through a variety of contracts and extensions. The **Application** class only calls the **OnLaunched** method in the case of a normal launch. See the **Windows::UI::Xaml::Application** class for more info about how to detect other kinds of activation events. In Hilo we only needed to handle normal launch.

The rest of this page is a walkthrough of how Hilo implements suspend and resume by using these layers of the system.

Code walkthrough of suspend

This diagram shows the interaction of the classes that implement the suspend operation in Hilo.



When Hilo starts, it registers a handler for the **Suspending** event that is provided by the **Application** base class.

C++ App.xaml.cpp

```
Suspending += ref new SuspendingEventHandler(this, &App::OnSuspending);
```

Windows invokes the **App::OnSuspending** event handler before it suspends the app. Hilo uses the event handler to save relevant app and user data to persistent storage.

C++ App.xaml.cpp

```
void App::OnSuspending(Object^ sender, SuspendingEventArgs^ e)
{
    (void) sender; // Unused parameter
    assert(IsMainThread());

    auto deferral = e->SuspendingOperation->GetDeferral();
    HiloPage::IsSuspending = true;
```

```
    SuspensionManager::SaveAsync().then([=](task<void> antecedent)
    {
        HiloPage::IsSuspending = false;
        antecedent.get();
        deferral->Complete();
    });
}
```

Hilo's **Suspending** event's handler is asynchronous. If a **Suspending** event's handler is asynchronous, it must notify its caller when its work has finished. To do this, the handler invokes the **GetDeferral** method that returns a **SuspendingDeferral** object and calls this object's **Complete** method at the end of the async operation.

The **SuspensionManager** class's **SaveAsync** method persists the app's navigation and user data to disk. After the save operation has finished, a continuation task notifies the deferral object that the app is ready to be suspended.

Note In Hilo, we defined the **HiloPage::IsSuspending** static property to establish context when the app receives callbacks from the **SaveAsync** method. We need to know if the **OnNavigatedFrom** method is called in the case of normal operation, or if it is being called while saving state for suspend/resume. For example, when the user navigates away from a page, the app sometimes needs to disconnect event handlers to that page. We don't want to disconnect event handlers when suspending.

The **SaveAsync** method of the **SuspensionManager** class writes the current session state to disk. The **SaveAsync** method calls the **GetNavigationState** method of each registered **Frame** object. In Hilo, there is only a registered frame, and it corresponds to the current page. The **GetNavigationState** method invokes **OnNavigatedFrom** method of the frame's associated page object.

In Hilo, each page is an instance of the **HiloPage** class. Here is this class's **OnNavigatedFrom** method.

C++ HiloPage.cpp

```
void
Hilo::HiloPage::OnNavigatedFrom(Windows::UI::Xaml::Navigation::NavigationEventArgs^
e)
{
    ViewModelBase^ viewModel = dynamic_cast<ViewModelBase^>(DataContext);
    if (!HiloPage::IsSuspending)
    {
        // Since we never receive destructs, this is the only place to unsubscribe.
        DetachNavigationHandlers(viewModel);
        viewModel->OnNavigatedFrom(e);
    }
```

```
   LayoutAwarePage::OnNavigatedFrom(e);
}
```

The **OnNavigatedFrom** method of the **HiloPage** class updates the view models and then invokes the **OnNavigatedFrom** method of its base class, **LayoutAwarePage**.

C++ LayoutAwarePage.cpp

```
void LayoutAwarePage::OnNavigatedFrom(NavigationEventArgs^ e)
{
    auto frameState = SuspensionManager::SessionStateForFrame(Frame);
    auto pageState = ref new Map<String^, Object^>();
    SaveState(pageState);
    frameState->Insert(_pageKey, pageState);
}
```

The **OnNavigatedFrom** method of the **LayoutAwarePage** class gets an object from the suspension manager and sets it as the value of the **frameState** variable. This object implements the **Windows::Foundation::Collections::IMap<String^, Object^>** interface. It represents all of the state that has been collected so far during the **SaveAsync** operation. Next, the code calls the page's **SaveState** method to augment the frameState object with information for the current page.

Note The **OnNavigatedFrom** method is called during the suspend operation but also for ordinary navigation while the app is running. Hilo also uses the **SaveState** and **LoadState** methods to support navigating back to a known state. For example, after the user crops an image and returns to the image view page, restoring the state on navigation lets the image view page display the correct photo.

Here is the **SaveState** method.

C++ HiloPage.cpp

```
void HiloPage::SaveState(IMap<String^, Object^>^ pageState)
{
    auto vm = dynamic_cast<ViewModelBase^>(DataContext);
    if (vm != nullptr)
    {
        auto vmStateMap = ref new Map<String^, Object^>();
        vm->SaveState(vmStateMap);

        pageState->Insert(viewModelStateKey, vmStateMap);
    }
}
```

In Hilo, the page delegates the saving of state to its associated view model, and uses a single key/value pair for the entire page's data. For example, if the currently displayed page is the image view page, the applicable method is the **SaveState** method of Hilo's **ImageViewModel** class.

C++ ImageViewModel.cpp

```
void ImageViewModel::SaveState(IMap<String^, Object^>^ stateMap)
{
    if (m_photo != nullptr)
    {
        std::wstringstream stringSerialization;
        stringSerialization << m_photo->DateTaken.UniversalTime ;

        stateMap->Insert(FilePathMapKey, m_photo->Path);
        stateMap->Insert(FileDateMapKey, ref new
String(stringSerialization.str().c_str()));
        stateMap->Insert(QueryMapKey, m_query);
    }
}
```

This code shows how the image view model class saves three pieces of information: the path and name of the current image, the file date, and the query that was used. Other view models save states that are relevant to their operations.

You can use any serialization technique that you want, but the end result must be converted into a **Platform::Object^** reference. You can choose any keys that you want for the key/value pairs. Just be sure to use keys that are unique within the current page, and use the same keys when you later restore the data.

Code walkthrough of resume

When an app is resumed from the **Suspended** state, it enters the **Running** state and continues from where it was when it was suspended. No application data is lost, as it was stored in memory, so most apps don't need to do anything when they are resumed.

It is possible that the app being resumed has been suspended for hours or even days. So, if the app has content or network connections that may have gone stale, these should be refreshed when the app resumes. When an app is suspended, it does not receive network or file event items that it registered to receive. These events are not queued, but are simply ignored. In this situation, your app should test the network or file status when it resumes. The app might also refresh location-dependent state such as a map position.

If an app registered an event handler for the **Resuming** event, it is called when the app resumes from the **Suspended** state. You can refresh your content by using this event handler. Hilo subscribes to the **Resuming** event to refresh its view of the Pictures library.

C++ App.xaml.cpp

```
Resuming += ref new EventHandler<Platform::Object^>(this, &App::OnResume);
```

Here is the **App::OnResume** member function.

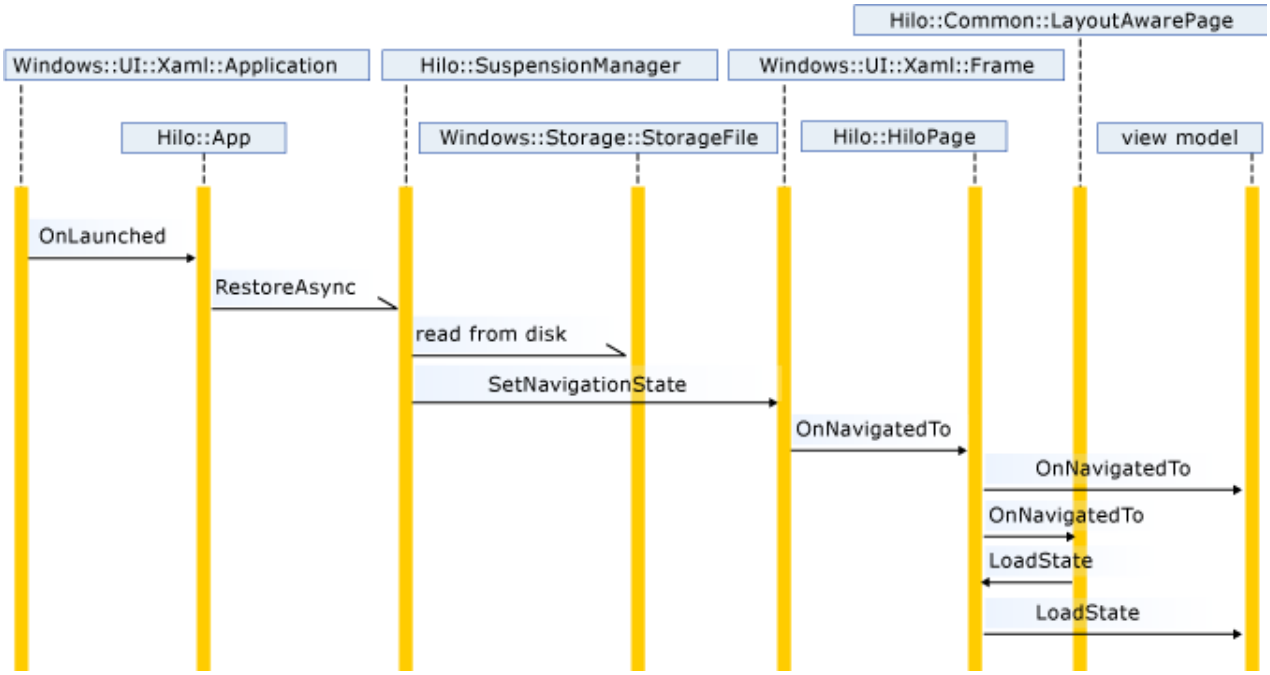
C++ App.xaml.cpp

```
void App::OnResume(Object^ sender, Platform::Object^ e)
{
    (void) sender; // Unused parameter
    (void) e;      // Unused parameter
    assert(IsMainThread());

    if (m_repository != nullptr)
    {
        // Hilo does not receive data change events when suspended. Create these
events on resume.
        m_repository->NotifyAllObservers();
    }
}
```

This code tells the app's file respository object that it should generate data-change events for view models that need to respond to file system changes. Hilo doesn't attempt to detect data changes that occurred while it was suspended.

Code walkthrough of activation after app termination

If Windows has terminated a suspended app, the **Application** base class calls the **OnLaunched** method when the app becomes active again. This diagram shows the interaction of classes in Hilo that restore the app after it has been terminated.



Hilo's **App** class overrides the **OnLaunched** method of **Windows::UI::Xaml::Application** base class. When your **App** class's **OnLaunched** method runs, its argument is a **LaunchActivatedEventArgs** object. This object contains an **ApplicationExecutionState** enumeration that tells you the app's previous execution state. Here's the code.

C++: App.xaml.cpp

```
void App::OnLaunched(LaunchActivatedEventArgs^ args)
{
    assert(IsMainThread());
    auto rootFrame = dynamic_cast<Frame^>(Window::Current->Content);

    // Do not repeat app initialization when the Window already has content,
    // just ensure that the window is active
    if (rootFrame == nullptr)
    {
        // Create a Frame to act as the navigation context and associate it with
        // a SuspensionManager key. See http://go.microsoft.com/fwlink/?LinkId=267280
for more info
        // on Hilo's implementation of suspend/resume.
        rootFrame = ref new Frame();
        SuspensionManager::RegisterFrame(rootFrame, "AppFrame");

        auto prerequisite = task<void>([](){});
        if (args->PreviousExecutionState == ApplicationExecutionState::Terminated)
        {
            // Restore the saved session state only when appropriate, scheduling the
            // final launch steps after the restore is complete
            prerequisite = SuspensionManager::RestoreAsync();
        }
        prerequisite.then([=](task<void> prerequisite)
        {
            try
            {
                prerequisite.get();
            }
            catch (Platform::Exception^)
            {
                //Something went wrong restoring state.
                //Assume there is no state and continue
            }

            if (rootFrame->Content == nullptr)
            {
                // When the navigation stack isn't restored navigate to the first
page,
                // configuring the new page by passing required information as a
navigation
                // parameter.  See http://go.microsoft.com/fwlink/?LinkId=267278 for
a walkthrough of how
                // Hilo creates pages and navigates to pages.
                if (!rootFrame->Navigate(TypeName(MainHubView::typeid)))
                {
                    throw ref new FailureException((ref new LocalResourceLoader())-
```

```
>GetString("ErrorFailedToCreateInitialPage"));
                }
            }

            // Place the frame in the current Window
            Window::Current->Content = rootFrame;
            // Ensure the current window is active
            Window::Current->Activate();

        }, task_continuation_context::use_current());
    }
    else
    {
        if (rootFrame->Content == nullptr)
        {
            // When the navigation stack isn't restored navigate to the first page,
            // configuring the new page by passing required information as a
navigation
            // parameter. See http://go.microsoft.com/fwlink/?LinkId=267278 for a
walkthrough of how
            // Hilo creates pages and navigates to pages.
            if (!rootFrame->Navigate(TypeName(MainHubView::typeid)))
            {
                throw ref new FailureException((ref new LocalResourceLoader())-
>GetString("ErrorFailedToCreateInitialPage"));
            }
        }
        // Ensure the current window is active
        Window::Current->Activate();
    }

    // Schedule updates to the tile. See
http://go.microsoft.com/fwlink/?LinkId=267275 for
    // info about how Hilo manages tiles.
    m_tileUpdateScheduler = std::make_shared<TileUpdateScheduler>();
    m_tileUpdateScheduler->ScheduleUpdateAsync(m_repository, m_exceptionPolicy);
}
```

The code checks its argument to see whether the previous state was **Terminated**. If so, the method calls the **SuspensionManager** class's **RestoreAsync** method to recover saved settings from the _sessionState.dat file.

The **RestoreAsync** method reads the saved state info and then calls the **SetNavigationState** method for each registered frame that was previously saved. See Code walkthrough of suspend in this guide to see how the save happens.

The **SetNavigationState** method of each frame calls the **OnNavigatedTo** method of its associated page. Here is how the **HiloPage** class overrides the **OnNavigatedTo** method.

C++: HiloPage.cpp

```
void HiloPage::OnNavigatedTo(NavigationEventArgs^ e)
{

    ViewModelBase^ viewModel = dynamic_cast<ViewModelBase^>(DataContext);
    this->AttachNavigationHandlers(viewModel);
    if (viewModel != nullptr)
    {
        viewModel->OnNavigatedTo(e);
    }

    LayoutAwarePage::OnNavigatedTo(e);
}
```

The Hilo page calls the **OnNavigatedTo** method of its associated view model. For example, if the currently displayed page is the image view page, the applicable method is the **OnNavigatedTo** method of Hilo's **ImageViewModel** class. Here is the code.

ImageViewModel.cpp

C++

```
void ImageViewModel::OnNavigatedTo(NavigationEventArgs^ e)
{
    auto data = dynamic_cast<String^>(e->Parameter);
    ImageNavigationData imageData(data);
    Initialize(imageData.GetFilePath(), imageData.GetFileDate(),
imageData.GetDateQuery());
}
```

In this example, the code extracts the saved navigation information from the argument and updates the view model.

After the view model's **OnNavigatedTo** method has finished, the **HiloPage** object invokes the **OnNavigatedTo** method of its base class, **LayoutAwarePage.**

C++: LayoutAwarePage.cpp

```
void LayoutAwarePage::OnNavigatedTo(NavigationEventArgs^ e)
{
    // Returning to a cached page through navigation shouldn't trigger state loading
    if (_pageKey != nullptr) return;

    auto frameState = SuspensionManager::SessionStateForFrame(Frame);
    _pageKey = "Page-" + Frame->BackStackDepth;

    if (e->NavigationMode == NavigationMode::New)
    {
        // Clear existing state for forward navigation when adding a new page to the
        // navigation stack
        auto nextPageKey = _pageKey;
        int nextPageIndex = Frame->BackStackDepth;
        while (frameState->HasKey(nextPageKey))
        {
            frameState->Remove(nextPageKey);
            nextPageIndex++;
            nextPageKey = "Page-" + nextPageIndex;
        }

        // Pass the navigation parameter to the new page
        LoadState(e->Parameter, nullptr);
    }
    else
    {
        // Pass the navigation parameter and preserved page state to the page, using
        // the same strategy for loading suspended state and recreating pages
discarded
        // from cache
        LoadState(e->Parameter, safe_cast<IMap<String^, Object^>^>(frameState-
>Lookup(_pageKey)));
    }
}
```

The **LayoutAwarePage** gets the previously saved state for the page from the **SuspensionManager** class. It then calls the page's **LoadState** method to deserialize the page's saved state and restore it. Here is the code for the **LoadState** method from the **HiloPage** class and the **ImageViewModel** class.

C++: HiloPage.cpp

```
void HiloPage::LoadState(Object^ navigationParameter, IMap<String^, Object^>^
pageState)
{
    auto vm = dynamic_cast<ViewModelBase^>(DataContext);
    if (vm != nullptr && pageState != nullptr)
    {
        IMap<String^, Object^>^ state = nullptr;
        state = dynamic_cast<IMap<String^, Object^>^>(pageState-
>Lookup(viewModelStateKey));

        vm->LoadState(state);
    }
}
```

C++: ImageViewModel.cpp

```
void ImageViewModel::LoadState(IMap<String^, Object^>^ stateMap)
{
    if (stateMap != nullptr)
    {
        auto filePath = dynamic_cast<String^>(stateMap->Lookup(FilePathMapKey));

        auto fileDateString = dynamic_cast<String^>(stateMap-
>Lookup(FileDateMapKey));
        DateTime fileDate;
        fileDate.UniversalTime = _wtoi64(fileDateString->Data());

        auto query = dynamic_cast<String^>(stateMap->Lookup(QueryMapKey));

        Initialize(filePath, fileDate, query);
    }
}
```

This code deserializes the previously saved information for the image view model. You can compare it with the code that originally saved the data, which appears above in Code walkthrough of suspend in this guide.

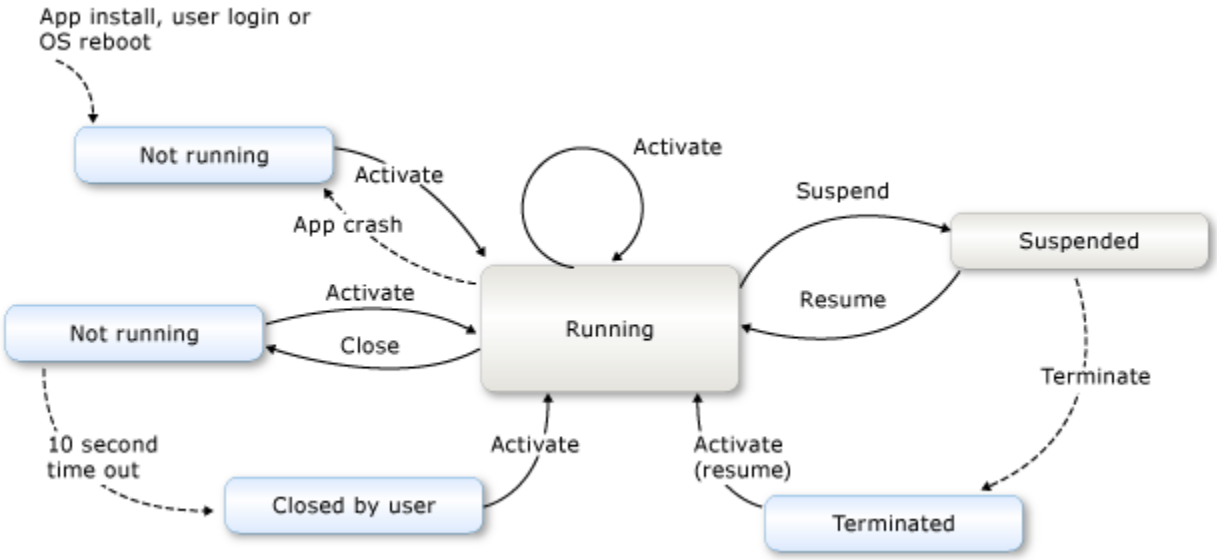
Other ways to exit the app

Apps do not contain UI for closing the app, but users can choose to close an app by pressing Alt+F4, dragging the app from the Start Page, or selecting the **Close** context menu. When an app has been closed using any of these methods, it enters the **NotRunning** state for approximately 10 seconds and then transitions to the **ClosedByUser** state.

Apps shouldn't close themselves programmatically as part of normal execution. When you close an app programmatically, Windows treats this as an app crash. The app enters the **NotRunning** state and remains there until the user activates it again.

Your app must follow the system crash experience, which is to simply return to the Start Page.

Here's a diagram that shows how Windows determines the app's execution state. Windows takes app crashes and user close actions into account, as well as the suspend or resume state. In the diagram, the blue rectangles indicate that the app is not loaded into system memory. The white rectangles indicate that the app is in memory. The dashed arcs are changes that occur without any notification to your running app. The solid arcs are actions that include app notification.



Improving performance in Hilo (Windows Store apps using C++ and XAML)

Summary

- Actual app performance is different from perceived app performance.
- Use performance tools to measure, evaluate, and target performance-related issues in your app.
- Keep the UI thread unblocked by running compute-intensive operations on a background thread.

The Hilo C++ team spent time learning what works and what doesn't when building a fast and fluid app. We identified areas in the app that we needed to improve perceived performance and where we had to improve the actual performance. Here are some tips and coding guidelines for creating a well-performing, responsive app.

You will learn

- The differences between performance and perceived performance.
- Recommended strategies when profiling an app.
- Tips that help create a fast and fluid app.
- How to keep the app's UI responsive by running compute-intensive operations on a background thread.

Improving performance with app profiling

Users have a number of expectations for apps. They want immediate responses to touch, clicks, gestures, and key-presses. They expect animations to be smooth and fluid, and that users will never have to wait for the app to catch up with them.

Performance problems show up in various ways. They can reduce battery life, cause panning and scrolling to lag behind the user’s finger, or even make the app appear unresponsive for a period of time. One technique for determining where code optimizations have the greatest effect in reducing performance problems is to perform app profiling.

The profiling tools for Windows Store apps let you measure, evaluate, and target performance-related issues in your code. The profiler collects timing info for apps by using a sampling method that collects CPU call stack info at regular intervals. Profiling reports display info about the performance of your app and help you navigate through the execution paths of your code and the execution cost of your functions so that you can find the best opportunities for optimization. For more info see How to profile Visual C++, Visual C#, and Visual Basic code in Windows Store apps on a local machine. To see how to

analyze the data returned from the profiler see Analyzing performance data for Visual C++, Visual C#, and Visual Basic code in Windows Store apps.

Optimizing performance is more than just implementing efficient algorithms. Performance can also be thought of as the user's perception of app performance, while they use it. The user's app experience can be separated into three categories – perception, tolerance, and responsiveness.

- **Perception**. A user's perception of performance can be defined as how favorably they recall the time it took to perform their tasks within the app. This perception doesn't always match reality. Perceived performance can be improved by reducing the amount of time between activities that the user needs to perform to accomplish their task in an app.
- **Tolerance**. A user's tolerance for delay depends on how long the user expects an operation to take. For example, a user might find cropping an image intolerable if the app becomes unresponsive during the cropping process, even for a few seconds. A user's tolerance for delay can be increased by identifying areas of your app that require substantial processing time, and limiting or eliminating user uncertainty during those scenarios by providing a visual indication of progress. In addition, async APIs can be used to avoid blocking the UI thread and making the app appear frozen.
- **Responsiveness**. Responsiveness of an app is relative to the activity being performed. To measure and rate the performance of an activity, there must be a time interval to compare it against. The Hilo team used the heuristic that if an activity takes longer than 500ms, the app might need to provide feedback to the user in the form of a visual indication of progress.

Profiling tips

When profiling your app, follow these tips to ensure that reliable and repeatable performance measurements are taken:

- Windows 8 runs on a wide variety of devices, and taking performance measurements on one hardware item won't always show the performance characteristics of other form factors.
- Make sure the machine that is capturing performance measurements is plugged in, rather than running from a battery. Many systems conserve power when running from a battery, and so operate differently.
- Make sure that the total memory utilization on the system is less than 50%. If it's higher, close apps until you reach 50% to make sure that you're measuring the impact of your app, rather than other processes.
- When remotely profiling an app, it's recommended that you interact with your app directly on the remote device. While you can interact with your app via Remote Desktop Connection, it can significantly alter the performance of your app and the performance data that you collect. For more info, see How to profile Visual C++, Visual C#, and Visual Basic code in Windows Store apps on a remote machine.

- To collect the most accurate performance results, profile a Release build of your app. See How to: Set Debug and Release Configurations.
- Avoid profiling your app in the simulator because the simulator can distort the performance of your app.

Other performance tools

In addition to using profiling tools to measure app performance, the Hilo team also used the Windows Reliability and Performance Monitor (perfmon) and XPerf.

Perfmon can be used to examine how programs you run affect your computer's performance, both in real time and by collecting log data for later analysis. The Hilo team used this tool for a general diagnosis of the app's performance. For more info about perfmon, see Windows Reliability and Performance Monitor.

XPerf is part of Windows Performance Tools (WPT). WPT contains performance analysis tools that can be used to diagnose a wide range of performance problems including application start times, system responsiveness issues, and application resource usage. The Hilo team used XPerf for investigating the memory usage of the app.

Performance tips

The Hilo team spent time learning what works and what doesn't when building a fast and fluid app. Here are some points to remember.

- Keep the launch times of your app fast
- Emphasize responsiveness in your apps by using asynchronous API calls on the UI thread
- Use thumbnails for quick rendering
- Prefetch thumbnails
- Trim resource dictionaries
- Optimize the element count
- Use independent animations
- Use parallel patterns for heavy computations
- Use techniques that minimize marshaling costs
- Keep your app's memory usage low when suspended
- Minimize the amount of resources your app uses by breaking down intensive processing into smaller operations

Keep the launch times of your app fast

Defer loading large in-memory objects while the app is activating. If you have large tasks to complete, provide a custom splash screen so that your app can accomplish these tasks in the background.

Emphasize responsiveness in your apps by using asynchronous API calls on the UI thread

Don't block the UI thread with synchronous APIs. Instead, use asynchronous APIs or call synchronous APIs in a non-blocking context. In addition, intensive processing operations should be moved to a thread pool thread. This is important because users will most likely notice delays longer than 100ms. Intensive processing operations should be broken down into a series of smaller operations, allowing the UI thread to listen for user input in-between.

Use thumbnails for quick rendering

The file system and media files are an important part of most apps, and also one of the most common sources of performance issues. File access is traditionally a key performance bottleneck for apps that display gallery views of files, such as photo albums. Accessing images can be slow, because it takes memory and CPU cycles to store, decode, and display the image.

Instead of scaling a full size image to display as a thumbnail, use the Windows Runtime thumbnail APIs. The Windows Runtime provides a set of APIs backed by an efficient cache that allows the app to quickly get a smaller version of an image to use for a thumbnail.

Prefetch thumbnails

As well as providing APIs for retrieving thumbnails, the Windows Runtime also includes a **SetThumbnailPrefetch** method in its API. This method specifies the thumbnail to retrieve for each file or folder based on the purpose of the thumbnail, its requested size, and the desired behavior to use to retrieve the thumbnail image.

In Hilo, the **FileSystemRepository** class queries the file system for photos that meet a specific date criteria, and returns any photos that meet that criteria. The **CreateFileQuery** method uses the **SetThumbnailPrefetch** method to return thumbnails for the files in the query result set.

C++: FileSystemRepository.cpp

```
inline StorageFileQueryResult^
FileSystemRepository::CreateFileQuery(IStorageFolderQueryOperations^ folder, String^
query, IndexerOption indexerOption)
{
    auto fileTypeFilter = ref new Vector<String^>(items);
    auto queryOptions = ref new QueryOptions(CommonFileQuery::OrderByDate,
fileTypeFilter);
    queryOptions->FolderDepth = FolderDepth::Deep;
    queryOptions->IndexerOption = indexerOption;
    queryOptions->ApplicationSearchFilter = query;
    queryOptions->SetThumbnailPrefetch(ThumbnailMode::PicturesView, 190,
ThumbnailOptions::UseCurrentScale);
```

```
    queryOptions->Language = CalendarExtensions::ResolvedLanguage();
    return folder->CreateFileQueryWithOptions(queryOptions);
}
```

In this case, the code prefetches thumbnails that display a preview of each photo, up to 190 pixels wide, and increases the requested thumbnail size based upon the pixels per inch (PPI) of the display. Using the **SetThumbnailPrefetch** method can result in improvements of 70% in the time taken to show a view of photos from the user's Pictures library.

Trim resource dictionaries

App-wide resources should be stored in the **Application** object to avoid duplication, but resources specific to single pages should be moved to the resource dictionary of the page.

Optimize the element count

The XAML framework is designed to display thousands of objects, but reducing the number of elements on a page will make your app render faster. You can reduce a page's element count by avoiding unnecessary elements, and collapsing elements that aren't visible.

Use independent animations

An independent animation runs independently from the UI thread. Many of the animation types used in XAML are composed by a composition engine that runs on a separate thread, with the engine's work being offloaded from the CPU to the GPU. Moving animation composition to a non-UI thread means that the animation won't jitter or be blocked by the app working on the UI thread. Composing the animation on the GPU greatly improves performance, allowing animations to run at a smooth and consistent frame rate.

You don't need additional markup to make your animations independent. The system determines when it's possible to compose the animation independently, but there are some limitations for independent animations. Here are some common problems.

- Animating the **Height** and **Width** properties of a **UIElement** results in a dependent animation because these properties require layout changes that can only be accomplished on the UI thread. To have a similar effect to animating **Height** or **Width**, you can animate the scale of the control instead.
- If you set the **CacheMode** property of an element to **BitmapCache** then all animations in the visual subtree are run dependently. The solution is to simply not animate cached content.
- The **ProgressRing** and **ProgressBar** controls have infinite animations that can continue running even if the control isn't visible on the page, which may prevent the CPU from going into low

power or idle mode. Set the **ProgressRing::IsActive** and **ProgressBar::IsIndeterminate** properties to false when they aren't being shown on the page.

Hilo uses the **ObjectAnimationUsingKeyFrames** type, which is an independent animation.

Use parallel patterns for heavy computations

If your app performs heavy computations, it's very likely that you need to use parallel programming techniques. There are a number of well-established patterns for effectively using multicore hardware. Parallel Programming with Microsoft Visual C++ is a resource for some of the most common patterns, with examples that use PPL and the Asynchronous Agents Library. See Concurrency Runtime for comprehensive documentation of the APIs, along with examples.

Hilo contains some compute-intensive operations for manipulating images. For these operations we used parallel programming techniques that take advantage of the computer's parallel processing hardware. See Adapting to async programming, Using parallel programming and background tasks and Async programming patterns in C++ in this guide for more info.

Be aware of the overhead for type conversion

In order to interact with Windows Runtime features you sometimes need to create data types from the Platform and Windows namespaces. In some cases, creating objects of these types incurs overhead for type conversion. Hilo performs type conversion at the ABI to minimize this overhead. See Writing modern C++ code in this guide for more info.

Use techniques that minimize marshaling costs

If your code communicates with languages other than C++ and XAML, you can incur costs for marshaling data across runtime environments. Hilo interacts only with C++ and XAML, so this wasn't a consideration for us. See Writing modern C++ code in this guide for more info.

Keep your app's memory usage low when suspended

When your app resumes from suspension, it reappears nearly instantly. But when your app restarts from termination, it may take longer to appear. Therefore, preventing your app from being terminated when it's suspended is a technique for managing the user's perception and tolerance of app responsiveness. This can be accomplished by keeping your app's memory usage low when suspended.

When your app begins the suspension process, it should free any large objects that can be easily rebuilt on resume. This helps to keep your app's memory footprint low, and reduces the likelihood that the OS will terminate your app after suspension. For more info see Handling suspend, resume and activation in this guide.

Minimize the amount of resources your app uses by breaking down intensive processing into smaller operations

Windows has to accommodate the resource needs of all Windows Store apps by terminating suspended apps to allow other apps to run. A side effect of this is that if your app requests a large amount of memory, other apps might be terminated, even if the app then frees that memory soon after requesting it. Be a good citizen so that the user doesn't begin to attribute any perceived latencies in the system to your app. You can do this by breaking down intensive processing operations into a series of smaller operations.

Testing and deploying Windows Store apps: Hilo (C++ and XAML)

Summary

- Use multiple modes of testing for best results.
- Use unit tests to identify bugs at their source.
- Apps must undergo certification before they can be listed in the Windows Store.

We designed Hilo C++ for testability and recommend that you do the same when designing your apps. For Hilo C++ we created and conducted unit tests, integration tests, user experience tests, security tests, localization tests, performance tests, and device tests.

You will learn

- How to use the testing tools that are available in Microsoft Visual Studio 2012.
- How the various modes of testing contribute to reliability and correctness of an app.

Ways to test your app

You can test your app in many ways. Here's what we chose for Hilo.

- **Unit testing** tests individual functions in isolation. The goal of unit testing is to check that each unit of functionality performs as expected so that errors don't propagate throughout the system. Detecting a bug where it occurs is more efficient than observing the effect of a bug indirectly at a secondary point of failure.
- **Integration testing** verifies that the components of the app work together correctly. Integration tests exercise app functionality in a realistic way. In Hilo, you can recognize this kind of test because it invokes methods of the view model. The separation of views from the view model makes integration tests possible.
- **User experience (UX) testing** interacts directly with the user interface. This kind of testing often requires human intervention. Automated integration tests can be substituted for some UX testing but cannot eliminate it completely.
- **Security testing** focuses on potential security vulnerabilities. It is based on a threat model that identifies possible classes of attack.
- **Localization testing** makes sure that the app works in all language environments.
- **Performance testing** identifies how the app spends its time when it's running. In many cases, performance testing can locate bottlenecks, or routines that take a large percentage of the app's CPU time.

- **Device testing** checks that the app works properly on the range of hardware that it supports. For example, it's important to test that the app works with various screen resolutions and touch-input capabilities.

The rest of this page describes the tools we used to test Hilo.

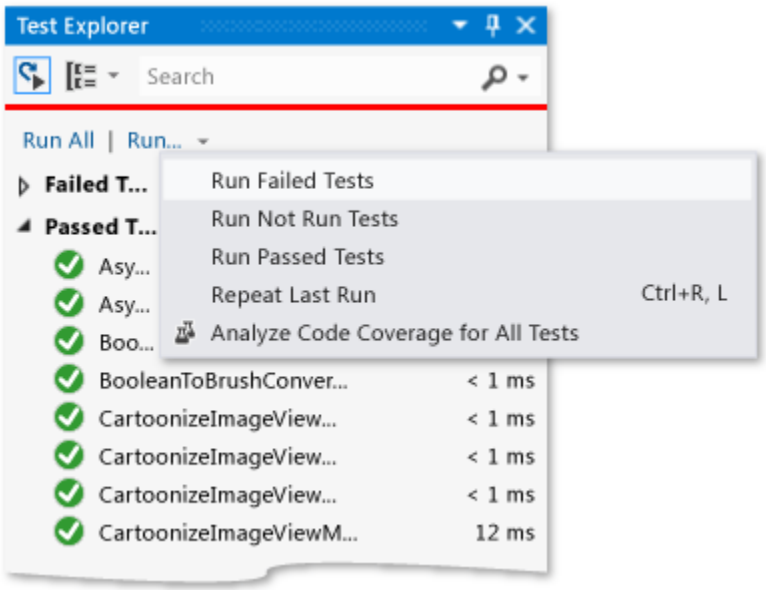
Using the Visual Studio unit testing framework

In Hilo, we use Visual Studio's unit-testing framework for unit tests and integration tests. The HiloTests project of the Hilo Visual Studio solution contains all the code that supports Hilo testing.

Note The version of Hilo that contains the HiloTests project is available at patterns & practices - Develop Windows Store apps using C++ & XAML: Hilo.

You can examine the unit tests by opening the Hilo Visual Studio solution. On the menu bar, choose **Test** > **Windows** > **Test Explorer**. The **Test Explorer** window lists all of Hilo's unit tests.

Here's the Hilo solution in Visual Studio that shows the **Test Explorer** window.



Here is an example of a test.

C++: RotateImageViewModelTests.cpp

```
TEST_METHOD(RotateImageViewModelShouldSaveAndLoadRotationAngle)
{
    auto vm = std::make_shared<RotateImageViewModel^>(nullptr);
    auto newVm = std::make_shared<RotateImageViewModel^>(nullptr);
```

```
    TestHelper::RunUISynced([this, vm, newVm]()
    {
        (*vm) = ref new RotateImageViewModel(m_repository, m_exceptionPolicy);
        (*vm)->RotationAngle = 90;
        auto state = ref new Platform::Collections::Map<String^, Object^>();
        (*vm)->SaveState(state);

        (*newVm) = ref new RotateImageViewModel(m_repository, m_exceptionPolicy);
        (*newVm)->LoadState(state);
    });

    Assert::AreEqual((*vm)->RotationAngle,  (*newVm)->RotationAngle);
}
```

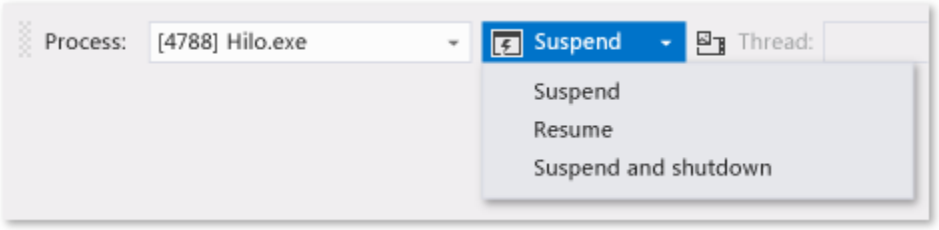
The code checks that the **SaveState** and **LoadState** methods of the **RotateImageViewModel** class save and restore the value of the **RotationAngle** property. The **RunUISynced** function is defined in the Hilo test project to enable asynchronous operations to be tested through unit tests.

The **TEST_METHOD** preprocessor macro is defined by the Microsoft unit test framework for C++.

For more info about the unit test tools in Microsoft Visual Studio, see Verifying Code by Using Unit Tests. For specifics about testing C++ code, see Writing Unit tests for C/C++ with the Microsoft Unit Testing Framework for C++.

Using Visual Studio to test suspending and resuming the app

When you debug a Windows Store app, the **Debug Location** toolbar contains a drop-down menu that enables you to suspend, resume, or suspend and shut down (terminate) the running app. You can use this feature to test that your app behaves as expected when the system suspends or resumes it or activates it after a suspend/terminate sequence.



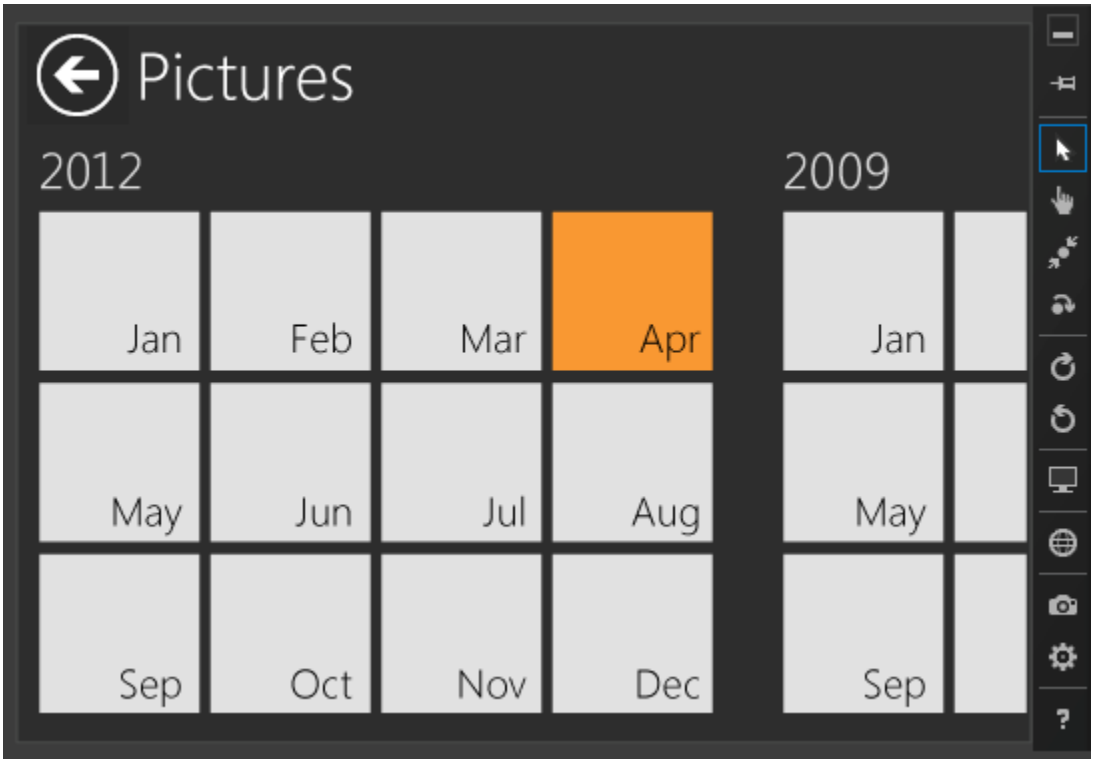
If you want to demonstrate suspending from the debugger, run Hilo in the Visual Studio debugger and set breakpoints in Hilo's **App:OnSuspending** and **App::OnLaunched** methods. Then select **Suspend and shutdown** from the **Debug Location** toolbar. The app will exit. Restart the app in the debugger, and the

app will follow the code path for resuming from the **Terminated** state. In Hilo, this logic is in the **App::OnLaunched** method. For more info, see Handling suspend, resume and activation.

Using the simulator and remote debugger to test devices

Visual Studio includes a simulator you can use to run your Windows Store app in various device environments. For example, with the simulator, you can check that your app works correctly under a variety of screen resolutions and with a variety of input hardware. You can simulate touch gestures even if you are developing the app on a computer that does not support touch.

Here's Hilo running in the simulator.



To start the simulator, select **Simulator** from the drop-down menu on the **Debug** toolbar in Visual Studio. Other choices on the drop-down menu are **Local machine** and **Remote machine**.

In addition to using the simulator, we also tested Hilo on a variety of hardware. You can use remote debugging to test your app on a computer that doesn't have Visual Studio. For more info about remote debugging, see Running Windows Store apps on a remote machine.

Using pseudo-localized versions for testing

We used pseudo-localized versions of Hilo for localization testing. See Localizability Testing for more info.

Security testing

We used the STRIDE methodology for threat modeling as a basis for security testing in Hilo. For more info about that methodology, see Uncover Security Design Flaws Using The STRIDE Approach.

Using Xperf for performance testing

In Hilo, we used Xperf.exe for performance testing. For more info, see Achieving high performance in this guide.

Using WinDbg for debugging

For debugging during normal development we used the Visual Studio debugger. We also used the Windows Debugger (WinDbg) for debugging in some test scenarios.

Using the Visual C++ compiler for testing

You can also use the Visual C++ compiler to help with testing your app. For example, use **static_assert** to verify that certain conditions are true at compile time and **assert** for conditions that must be true at run time. Assertions can help you to constrain your unit tests to given types and inputs.

Making your app world ready

Preparing for international markets can help you reach more users. Globalizing your app provides you with guides, checklists, and tasks to help you create a user experience that reaches users by helping you to globalize and localize your Windows Store app. Hilo supports all world calendar formats. Its resource strings have been localized in 4 languages: Arabic (Saudi Arabia), English (United States), German (Germany), and Japanese.

Here are some of the issues we had to consider while developing Hilo.

- **Think about localization early.** We considered how flow direction and other UX elements affect users across various locales. We also incorporated string localization early to make the entire process more manageable.
- **Separate resources for each locale.** We maintain separate solution folders for each locale. For example, **Strings** > **en-US** > **Resources.resw** defines the strings for the en-US locale. For more info, see Quickstart: Using string resources, and How to name resources using qualifiers.

- **Localize the app manifest.** We followed the steps in Localizing the package manifest, which explains how to use the **Manifest Designer** to localize the name, description, and other identifying features of your app.
- **Ensure that each piece of text that appears in the UI is defined by a string resource.** We use the x:Uid directive to provide a unique name for the localization process to associate localized strings with text that appears on screen. For example, here's the XAML that defines the app title that appears on the main page:

XAML: MainHubView.xaml

```
<TextBlock x:Name="PageTitle"
           x:Uid="AppName"
           Grid.Column="1"
           Style="{StaticResource PageHeaderTextStyle}"
           Text="{Binding AppName}"/>
```

For the en-US locale, in the resource file we define **AppName.Text** as "Hilo". We specify the **.Text** part so that the XAML runtime can associate the string with the **Text** property of the **TextBlock** control. We also use this technique to set tooltip text (**ContentControl::Content**).

- **Add contextual comments to the app resource file.** Comments in the resource file help localizers more accurately translate strings. For example, for the **DisplayNameProperty** string, we provided the comment "DisplayName property value for the package manifest." to give the localizer a better idea of where the string is used. For more info, see How to prepare for localization.
- **Define the flow direction for all pages.** We define **Page.FlowDirection** in the string resources file to set the flow direction for all pages. For languages that use left-to-right reading order, such as English and German, we define "LeftToRight" as its value. For languages that read right-to-left, such as Arabic and Hebrew, you define this value as "RightToLeft." We also defined the flow direction for all app bars by defining **AppBar.FlowDirection** in the resource files.
- **Ensure exception messages are read from the resource file.** It's important to localize exception message strings because these strings for unhandled exceptions can appear to the user. Hilo defines the **IResourceLoader** interface to load string resources.

C++: IResourceLoader.h

```
public interface class IResourceLoader
{
    Platform::String^ GetString(Platform::String^ value);
};
```

The **LocalResourceLoader** class derives from **IResourceLoader** and uses the **ResourceLoader** class to load strings from the resource file.

C++: LocalResourceLoader.cpp

```
String^ LocalResourceLoader::GetString(String^ value)
{
    auto loader = ref new ResourceLoader();
    return loader->GetString(value);
}
```

When we provide a message when we throw a Windows Runtime exception, we use the **LocalResourceLoader** class to read the message text. Here's an example.

C++: WideFiveImageTile.cpp

```
void WideFiveImageTile::SetImageFilePaths(const vector<wstring>& fileNames)
{
    if (fileNames.size() > MaxTemplateImages)
    {
        throw ref new FailureException(m_loader-
>GetString("ErrorWideTileTooBig"));
    }

    m_fileNames = fileNames;
}
```

For the en-US locale, we define "ErrorWideTileTooBig" as "Wide tile can only take up to 5 images." Other locales have messages that convey the same error. For example, the Strings/ja-JP/Resources.resw file contains an equivalent Japanese string, which is "ワイドタイルは５イメージまでしか".

Note We define the **IResourceLoader** interface for testing. In the HiloTests project, we define the **StubResourceLoader** class, which uses hard-coded strings instead of localized strings. Using mock data in this way can make it easier to isolate tests to focus on specific functionality.

- **Use the Calendar class to support world calendars.** Hilo uses methods and properties of the **Windows::Globalization::Calendar** class. For example, Hilo gets the **Calendar::NumberOfMonthsInThisYear** property to determine the number of months in the current year, rather than assuming that there are always 12 months in every year. Hilo's calendar logic can be found in the CalendarExtensions.cpp file.

You can test your app's localization by configuring the list of preferred languages in Control Panel. For more info about making your app world-ready, see How to prepare for localization, Guidelines and checklist for application resources, and Quickstart: Translating UI resources.

Testing your app with the Windows App Certification Kit

To give your app the best chance of being certified, validate it by using the Windows App Certification Kit. The kit performs a number of tests to verify that your app meets certain certification requirements for the Windows Store. These tests include:

- Examining the app manifest to verify that its contents are correct.
- Inspecting the resources defined in the app manifest to ensure that they are present and valid.
- Testing the app's resilience and stability.
- Determining how quickly the app starts and how fast it suspends.
- Inspecting the app to verify that it calls only APIs for Windows Store apps.
- Verifying that the app uses Windows security features.

You must run the Windows App Certification Kit on a release build of your app; otherwise, validation fails. For more info, see How to: Set Debug and Release Configurations.

In addition, it's possible to validate your app whenever you build it. If you're running Team Foundation Build, you can modify settings on your build machine so that the Windows App Certification Kit runs automatically every time your app is built. For more info, see Validating a package in automated builds.

For more info, see How to test your app with the Windows App Certification Kit.

Creating a Windows Store certification checklist

You'll use the Windows Store as the primary method to sell your apps or make them available. For info about how to prepare and submit your app, see Selling apps.

As you plan your app, we recommend that you create a publishing-requirements checklist to use later when you test your app. This checklist can vary depending on the kind of app you're building and on how you plan to monetize it. Here's our checklist:

1. **Open a developer account**. You must have a developer account to upload apps to the Windows Store. For more info, see Registering for a Windows Store developer account.
2. **Reserve an app name**. You can reserve an app name for one year, and if you don’t submit the app within the year, the reservation expires. For more info, see Naming and describing your app.
3. **Acquire a developer license**. You need a developer license to develop a Windows Store app. For more info, see Getting a Developer License for Windows 8.
4. **Edit your app manifest**. Modify the app manifest to set the capabilities of your app and provide items such as logos. For more info, see Manifest Designer.
5. **Associate your app with the Store**. When you associate your app with the Windows Store, your app manifest file is updated to include Store-specific data.
6. **Copy a screenshot of your app to the Windows Store**.

7. **Create your app package**. The simplest way to do this is through Visual Studio. For more info, see Packaging your app using Visual Studio 2012. An alternative way is to create your app package at the command prompt. For more info, see Building an app package at a command prompt.
8. **Upload your app package to the Windows Store**. During the upload process, your app's packages are checked for technical compliance with the certification requirements. If your app passes these tests, you'll see a successful upload message. If a package fails an upload test, you'll see an error message. For more info, see Resolving package upload errors.

Although we didn't actually upload Hilo to the Windows Store, we did perform the necessary steps to ensure that it would pass validation.

Before creating your app package for upload to the Windows Store, be sure to do the following:

- Review the app-submission checklist. This checklist indicates the information that you must provide when you upload your app. For more info, see App submission checklist.
- Ensure that you have validated a release build of your app with the Windows App Certification Kit. For more info, see Testing your app with the Windows App Certification Kit on this page.
- Take some screen shots that show off the key features of your app.
- Have other developers test your app. For more info, see Sharing an app package locally.

In addition, if your app collects personal data or uses software that is provided by others, you must also include a privacy statement or additional license terms.

Meet the Hilo team (Windows Store apps using C++ and XAML)

About patterns & practices

The goal of patterns & practices is to enhance developer success through guidance on designing and implementing software solutions. We develop content, reference implementations, samples, and frameworks that explain how to build scalable, secure, robust, maintainable software solutions. We work with community and industry experts on every project to ensure that some of the best minds in the industry have contributed to and reviewed the guidance as it develops.

Visit the patterns & practices Developer Center to learn more about patterns & practices and what we do.

Meet the team

This guide was produced by:

- **Program Management:** Blaine Wastell
- **Development and written guidance:** David Britch (Content Master Ltd.), Bob Brumfield, Colin Campbell (Modeled Computation LLC), Scott Densmore, Kate Gregory (Partner, Gregory Consulting), Thomas Petchel
- **Test:** Swetha Dyachavaram (Technology analyst, Infosys Ltd.), Carlos Farre, Rohit Sharma, Aviraj Singh (Sr. Technology Architect, Infosys Limited)

- **UX and graphic design:** Deborah Steinke (Graphic Designer, adaQuest, Inc.), Howard Wooten
- **Editorial support:** Marzena Makuta, Mike Matteson, John Osborne, Kristi Rasmussen

We want to thank the customers, partners, and community members who have patiently reviewed our early content and drafts. We especially want to recognize Marcelo Hideaki Azuma (ITGROUP), Chad Carter (CTO, GlobalCove Technologies), Tony Champion (Owner, Champion DS), Carlos dos Santos (CDS Informática Ltda.), Genevieve Fernandes, Alon Fliess (Chief Architect, CodeValue), Timo Heinäpurola, Tim Heuer, Robert Hogg (Black Marble), Hong Hong, Mike Kenyon (Senior Principal Software Engineer, IHS, Inc.), Artur Laksberg, Michael B. McLaughlin, Harry Pierson, Caio Proiete (Senior Consultant/Trainer, CICLO), Andy Rich, Martin Salias, Herb Sutter, Jose Miguel Torres (Software Engineer, Xamarin), and J. Andre Tournier (Sr. Software Developer, WebMD) for their technical insights and support throughout this project.

We hope that you enjoy working with the Hilo source files and this guide as much as we did creating it. Happy coding!