Developing a Metro style enabled desktop browser

XXX 0, 0000

Abstract

Windows 8 Consumer Preview introduces a new programming paradigm called Metro style. Metro style apps offer a clean, polished user experience that push app experiences to the forefront, and immerse the user in a full screen environment that's tailored to the user's hardware and context. Windows 8 Consumer Preview also continues to offer desktop app experiences as found in previous versions of Windows. In Windows 8 Consumer Preview, the browser that the user sets as the "default" for handling web pages and associated protocols may be designed to access both the Metro style experience as well as the traditional desktop experience. This type of browser is called a "Metro style enabled desktop browser". This guide describes how to create such a browser. The information in this document applies specifically to browsers that will be made available to end-users as the default viewer for the http:// protocol and associated web pages and protocols on the x86/64 architecture.

This information applies to the following operating systems:

Windows 8 Consumer Preview

This document will be updated at the time of the final release of Windows 8, if necessary.

Contents

Presentations of browsers

The desktop browser ecosystem is well established. Users expect that popular Windows 7 desktop browsers will continue to work on Windows 8 Consumer Preview, and that is the case. Beyond the desktop, Windows 8 Consumer Preview introduces a new app programming model which enables development of apps that are touch optimized, behave in the manner customers expect (maximize battery life, emphasize reliability, adhere to customer preferences for privacy and potentially sensitive devices like GPS, handle system start-up and shutdown without changing the state or impacting other apps, etc.), take on the enhanced new look and feel of Windows 8, and integrate with the richness of the PC via contracts. Apps that do these things are said to participate in the *Metro style user experience*.

In Windows Consumer Preview, a web browser may be built as a Metro style app, a desktop app, or a Metro style enabled desktop browser.

- **Metro style app.** A Metro style app adheres to the principles of the new app model– it runs in an App Container, uses APIs found in the Windows Software Development Kit (SDK) for Metro style apps, is packaged as an .appx file, and is made available via the Windows Store.
- **Desktop browsers.** Same model as Windows 7.
- **Metro style enabled desktop browser.** A desktop browser that chooses to participate in the new Metro style experience when the user has expressed preference for the browser to do so. Such a browser can provide HTML5 rendering for webpages and service HTTP / HTTPS requests. By definition, such a browser has full access to Win32 APIs for rendering HTML5, including the ability to use multiple background processes, JIT compiling, and other distinctly browser-related functionality (like background downloading of files). Desktop browsers typically run at medium or low integrity level.

This guide focuses on developing a Metro style enabled desktop browser.

User experience framing

The following design and user experience principles provide framing for how to think about web browsers in Windows 8 Consumer Preview:

- **The user is in control of browser preference.** Browser choice is one of personal preference, and Windows 8 Consumer Preview continues to honor user selection. Users can select the default via a “new app installed” system notification, and they can also configure via other means, such as Set User Defaults (SUD) and Set Program Access and User Defaults (SPAD).
- **There is only one default browser on the system.** Any and all browsers can register their intent to be considered as the default (http:// protocol and the other consolidated attributes which define a browser in SUD / SPAD), and the user can select one browser as the default. This is the same model as Windows 7.
- **A Metro style enabled desktop browser may participate in the Metro style user experience only if it is the default browser.** Desktop apps (typically packaged as .MSI, medium integrity level) run in the desktop. Metro style apps (.appx packaged, run in App Containers, API set restricted to the Windows SDK for

Metro style apps, acquired via the Windows Store) run in the Metro style experience. A Metro style enabled desktop browser can be thought of as a desktop browser that can also participate in the new Metro style experience. The restriction to limit Metro style user experience participation to the user's default browser is rooted in preserving the Metro style user experience. Note that this limitation applies to all browsers, including Internet Explorer.

Implementation

The following sections detail the design and code required to build a Metro style enabled desktop browser.

Technology	Description
Installation	Update installation code to create correctly formatted tiles, splash screen, and other associated content which indicate to the system that the browser is a Metro style enabled desktop browser.
Becoming the Default Browser	Understand the changes to how users select file and protocol handlers for apps in Windows Consumer Preview.
Tiles	Provide a square tile for the Start screen and implement execute commands for secondary tile invocation.
Activation (Launch)	Implement the app activation contract to launch the browser in the Metro style user experience.
Windowing	Understand the rules governing window creation in the Metro style user experience.
App Contracts	Implement and participate in app contracts: Search, Share, Play To, and Print.
Process Lifetime Management	Save and restore state in the Metro style user experience.
Roaming Browser Application Data	Keep browser application data (e.g. settings) synchronized between multiple Windows 8 devices.

Installation

No change is required to browser discovery and acquisition mechanisms:

- Metro style enabled desktop browsers may be distributed via existing channels, for example, web download, network share, OEM pre-install, or systems management software.
- Metro style enabled desktop browsers may be deployed via existing desktop app deployment methods, for example, MSI packaged, Click-Once installer, or .ZIP archive extraction with file copy.

  Note: We strongly recommend that you digitally sign all the installation executables, including those that might be used as "bootstrap files" during a Click-Once installation, with an Authenticode Certificate issued by a Certificate Authority (CA) that is a member of the Windows Root Certificate Program. For more information on this best practice and the role of Windows SmartScreen in Windows 8, see Protecting you from malware.

However, Windows must be able to identify that a browser is a Metro style enabled desktop browser, otherwise the operating system will assume that the browser is

desktop-capable only and will treat it accordingly. This means no interaction with the Metro style user experience. A Metro style enabled desktop browser must perform the following:

- Author a “VisualElementsManifest.xml” file that adheres to the schema and example included in this document.
- Include the .xml file as part of the Metro style enabled desktop browser’s installation contents.
- Include the supporting files as referenced by the .xml file (e.g. tile logo, splash screen images, Resources.PRI file) as part of the Metro style enabled desktop browser’s installation contents.
- During installation, copy the .xml file to the same directory as the browser executable.

  C:\Program Files\YourBrowser\browser.exe

  C:\Program Files\YourBrowser\VisualElementsManifest.xml
- During installation, copy the supporting files to the same directory as the browser executable (or relative to it).

  C:\Program Files\YourBrowser\browser.exe

  C:\Program Files\YourBrowser\VisualElementsManifest.xml

  C:\Program Files\YourBrowser\logo.jpg

  C:\Program Files\YourBrowser\logosmall.jpg

  C:\Program Files\YourBrowser\imagefile.jpg

  C:\Program Files\YourBrowser\Resources.PRI

  Note: Should you wish to update the resources associated with the browser after the initial install (for example: during an update), you must also update the app’s shortcut timestamp. Doing so signals Windows to reread and reload the visual elements.
- During installation, create a Start shortcut to your app populated with the following properties:

  PKEY_AppUserModel_IsDualMode set as TRUE (VT_BOOL)

  PKEY_AppUserModel_ID set as the browser’s AppUserModelID (VT_LPWSTR). For additional information relevant to this specific property, see Register an Application User Model ID (AppUserModelID), in this document.

Sample “VisualElementsManifest.xml” file

```
<Application
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <VisualElements
        DisplayName="displayname"
        Logo="logo.jpg"
        SmallLogo="logosmall.jpg"
        ForegroundText="light"
        BackgroundColor="blue">
        <DefaultTile
            ShortName="shortname"
            ShowName="allLogos"
```

```
            />
        <SplashScreen
            Image="imagefile.jpg" />
    </VisualElements>
</Application>
```

Note: Windows uses the DisplayName attribute under /Application/VisualElements for the tile and splash screen. The BackgroundColor value under /Application/VisualElements will also be used as the background color of the SplashScreen.

For more information about the property values, see Default Tile.

Manifest schema

The following XSD is used to validate your VisualElementsManifest.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:simpleType name="st_nonemptystring">
        <xs:restriction base="xs:string">
            <xs:minLength value="1"/>
            <xs:maxLength value="32767"/>
            <xs:pattern value="[^\s]|([^\s].*[^\s])"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="st_shortdisplayname">
        <xs:restriction base="st_nonemptystring">
            <xs:pattern value="ms-resource:.{1,256}"/>
            <xs:pattern value=".{1,13}"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="st_displayname">
        <xs:restriction base="st_nonemptystring">
            <xs:pattern value="ms-resource:.{1,256}"/>
            <xs:pattern value=".{1,256}"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="st_filenamecharset">
        <xs:restriction base="st_nonemptystring">
            <xs:pattern value=&quot;[^&lt;&gt;&quot;:%\|\?\*]+&quot;/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="st_filename">
        <xs:restriction base="st_filenamecharset">
            <xs:pattern
value="([^/\\]*[^./\\]+)(\\[^/\\]*[^./\\]+)*"/>
            <xs:pattern value="([^/\\]*[^./\\]+)(/[^/\\]*[^./\\]+)*"/>
            <xs:maxLength value="256"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="st_imagefile">
        <xs:restriction base="st_filename">
            <xs:pattern value=".+\.((jpg)|(png)|(jpeg))"/>
```

```
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="st_color">
        <xs:restriction base="xs:string">
            <xs:pattern value="#[\da-fA-F]{6}"/>
            <xs:pattern
value="black|silver|gray|white|maroon|red|purple|fuchsia|green|lime|ol
ive|yellow|navy|blue|teal|aqua"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="st_foregroundtext">
        <xs:restriction base="xs:string">
            <xs:enumeration value="light"/>
            <xs:enumeration value="dark"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="ct_defaulttile">
        <xs:attribute name="ShortName" type="st_shortdisplayname"
use="optional"/>

    </xs:complexType>

    <xs:complexType name="ct_splashscreen">
        <xs:attribute name="Image" type="st_imagefile"
use="required"/>
    </xs:complexType>

    <xs:complexType name="ct_visualelements">
        <xs:all>
            <xs:element name="DefaultTile" type="ct_defaulttile"
minOccurs="0"/>
            <xs:element name="SplashScreen" type="ct_splashscreen"/>
        </xs:all>
        <xs:attribute name="DisplayName" type="st_displayname"
use="required"/>
        <xs:attribute name="Logo" type="st_imagefile" use="required"/>
        <xs:attribute name="SmallLogo" type="st_imagefile"
use="required"/>
        <xs:attribute name="ForegroundText" type="st_foregroundtext"
use="required"/>
        <xs:attribute name="BackgroundColor" type="st_color"
use="required"/>
    </xs:complexType>

    <xs:complexType name="ct_application">
        <xs:all>
            <xs:element name="VisualElements" type="ct_visualelements"
/>
        </xs:all>
    </xs:complexType>

    <xs:element name="Application" type="ct_application" />

</xs:schema>
```

Register an Application User Model ID (AppUserModelID)

A Metro style enabled desktop browser must register an explicit Application User Model ID (AppUserModelID) for itself. This ID must be provided under the progID registered with SUD. The AppUserModelID may not be longer than 64 characters. When running in the Metro style experience, this AppUserModelID is immutable by the running browser process.

The AppUserModelID should be specified in the shortcut properties for the app as created during installation. For more information about assigning an AppUserModeID, see http://msdn.microsoft.com/en-us/library/dd378459%28VS.85%29.aspx#where.

Note: Metro style enabled desktop browsers should not set an AppUserModelID at runtime when participating in the Metro style user experience (browsers may continue to do so when activating in the desktop). This is not supported in the new experience and can lead to unpredictable behavior.

Declare a square tile for the Start screen

During installation, a Metro style enabled desktop browser must communicate the details of its app tile. The tile assets that must be included are:

- Square tile logo
  - At 1x (.75x and 1.4x plateaus are optional but highly recommended)
- Small Logo
- Background Color
- Foreground Text Value
- Short Name (optional – use only if the display name is too long to fit on a the tile on the Start screen)

For more information, see Creating and managing tiles, toast, and Windows push notifications.

Declare a splash screen logo

During installation, a Metro style enabled desktop browser must communicate the details of its splash screen logo. This is a png, jpg, or jpeg and must be specified at .75x, 1x, and 1.4x plateaus. These images must occur within a Package Resource Index (PRI). The PRI may be created using free tools included with the Windows Software Development Kit (SDK) for Metro style apps.

For more information about PRI, review the documentation for the ResourceManager Class.

Becoming the default browser

Upon installation, Windows Consumer Preview presents the user with a system notification that enables selection of the browser as the system default. Dismissing or ignoring the notification and/or dismissing the flyout without making a selection results in the default browser remaining unchanged. The user may also change the default browser via other operating system mechanisms, including SUD.

Tiles

In Windows Consumer Preview, the start menu has been replaced by a Start screen of tiles.

If the default browser supports the Metro style user experience, the browser's app tile (and secondary tiles, if they exist) will display in Start in a manner similar to Metro style apps (as defined by properties that were specified in the .XML manifest that was copied during browser installation). Otherwise, the tile will appear as a desktop app. Verbs appropriate to the Metro style user experience and desktop mode of the app will appear on the tile. The tile will appear in search results like any other tile. Other tile actions (re-arrange, pin/unpin, etc.) are no different for browsers than they are for other app tiles.

The default Metro style enabled desktop browser always appears as a square tile. Any secondary tiles belonging to the default Metro style enabled desktop browser also appear with Metro style user experience tile visuals.

Pinning secondary tiles

Secondary tiles enable users to promote interesting content and deep links – for example, a reference to a specific location inside of the pinning app – from Metro style apps onto the Start screen. A typical use of secondary tiles by a Metro style enabled desktop browser might involve surfacing the ability to create a tile that represents a link to a particular website important to the user, e.g. a "favorite". For more information about using secondary tiles, see Creating and managing secondary tiles.

A Metro style enabled desktop browser may pin secondary tiles while running in the Metro style user experience. These tiles will be associated with the app that created them (the Metro style enabled desktop browser), the same as any other secondary tile.

While the secondary tiles' parent Metro style enabled desktop browser app is the current default browser, those secondary tiles launch in the Metro style user experience and have Metro style visuals. If their parent app is no longer the default browser, those secondary tiles are treated the same as any desktop tile.

- The system examines the Application User Model ID of the app to verify the calling app is the current default browser.
- The app supplied arguments will be stored in the shortcut and supplied in the Arguments field of the LaunchActivatedEvent.

Attempts to call the secondary tiles APIs from any browser running in desktop mode will fail.

Activation (launch)

Metro style enabled desktop browsers are capable of activating in both the desktop and the Metro style user experience (the latter only when configured as the default browser). To support desktop activation, no changes are necessary. To support Metro style user experience activation, changes are required.

The following rules govern app activation:

- Desktop shortcuts, pinned taskbar icons, and other “desktop artifacts” activate the Metro style enabled desktop browser in the desktop.
- Tiles in the Start screen activate the browser in the Metro style user experience when the browser is the default. When the browser is not the default, tiles activate the browser in the desktop. This same behavior also applies to a browser’s secondary tiles.
- Browsers that wish to activate in the Metro style user experience for file and protocol contracts must call IApplicationActivationManager::ActivateForFile or IApplicationActivationManager::ActivateForProtocol in the implementation of their verb handler. Verb handlers must be implemented as a DelegateExecute handler. For more information about verb handling, see the FileActivatedEventArgs.Verb property.
- It is strongly recommended that browsers implement “contextual launching”. This means taking the calling app’s presentation (desktop or Metro style user experience) into account when determining the browser’s presentation (desktop or Metro style user experience) for activations originating from other apps,such as when a user clicks an http:// link in their mail app.
- Search contract activations always activate in the Metro style user experience. These activations can only occur if the browser is the user’s default. For more information, please reference the Search contract section of this document.

Contextual launch

Users expect a seamless experience when activating a browser via a link; for example, launching a browser by clicking on an http:// link in a mail or messaging app. When following a link within a Metro style app, most users expect a Metro style presentation of the linked content. Conversely, when following a link from a desktop app, most users expect a desktop style presentation of the linked content.

When the browser is activated, use the extended information obtained from the IExecuteCommandHost::GetUIMode() method to obtain the current UI context of the app or system component where the browser’s activation originated. Use this information to provide the best browser presentation.

```
// IExecuteCommandHost
IFACEMETHODIMP GetUIMode(_Out_ EC_HOST_UI_MODE *pUIMode)
    {
        *pUIMode = ECHUIM_IMMERSIVE;
        return S_OK;
    }
```

The enumeration has three values:

- ECHUIM_DESKTOP – Desktop application launch
- ECHUIM_IMMERSIVE – Metro style application launch
- ECHUIM_SYSTEM_LAUNCHER – Start menu launch (includes Tile activation, typing a URL into the search box in Start, etc.)

It's recommended that browsers provide a configurable setting to give users control over their preferred presentation experience. For example, a setting named "Choose how you open links", with options a) Always launch Metro style, b) Always launch in the desktop, c) Let the browser decide, aka contextual (suggested default).

Follow these steps to implement contextual launch:

- Author a DelegateExecute verb handler that implements IExecuteCommand, IObjectWithSite, IInitializeCommand, IObjectWithSelection, and IExecuteCommandApplicationHostEnvironment.
- Register the handler in the registry under HKCR\{browser desktop AppID}\.exe
  - Any shortcut that contains a PKEY_AppUserModel_ID value will cause Windows to first look for a verb handler registered for the shortcut's target's type registered under the AppUserModelID. Browser shortcuts will point at the browser's executable file and thus must be registered for the '.exe' type under the AppUserModelID. When the verb is invoked, it will be provided with the shortcut's target and any arguments stored in the link (this includes arguments specified in secondary tiles).
- During launch, the implementation of IExecuteCommandApplicationHostEnvironment::GetValue is called. The browser should return which presentation to launch in, but it will likely need more information in order to make this determination.
  - Use IUnknown_QueryService(_pSite, SID_ExecuteCommandHost, …) to obtain the IExecuteCommandHost implementation.
  - Call IExecuteCommandHost::GetUIMode to discover the launch context. Use this information, along with the user configurable preference and information such as that provided to IInitializeCommand and IObjectWithSelection, to determine the presentation, either desktop or Metro-style. Return the value from the IExecuteCommandApplicationHostEnvironment::GetValue call.

In all launch cases except launch via the browser's tile(s) on the Start screen, the browser's IExecuteCommand::Execute handler will get called and the browser must fully handle activation. In the case of the tile on the Start screen, if IExecuteCommandApplicationHostEnvironment::GetValue returns AHE_IMMERSIVE, Windows finishes activation of the browser in order to provide a consistent launch animation. IExecuteCommand::Execute will not be called in this case.

Register the DelegateExecute handler.

The following is an example of the registry entry:

```
HKCR\Example.Browser\.exe\shell\
        (Default) = "open"
        open\command
                (Default) = ""
                DelegateExecute = "{1e1946d6-5e51-4548-b7e8-
        033c60abffaf}"
```

Launching vs. switching

In the desktop case, it's a common occurrence for the browser to already be running when a user attempts to activate it via the Start menu tile or file / protocol activation. Rather than creating a separate new instance of the browser for each activation, you should use the DelegateExecute handler to communicate with a running instance of the browser.

It's recommended that browsers support, at a minimum, the "open" verb. Also, when a user chooses "Open new window", the browser should open a new tab rather than a new instance of the browser. Browsers are suggested to implement a verb, e.g. "open new window", corresponding to that common user action.

For more information about setting a state or parameter related to a verb or invoking the verb, see IExecuteCommand interface.

Activating the Metro style enabled desktop browser for file and protocol contracts

Browsers should implement their file and protocol associations as Execute Command verbs (see the Execute Command Verb Sample for an example) in a manner similar to tile launch. When directing the verb invocation into the Metro style user experience, browsers call a method on IApplicationActivationManager to activate in the Metro style user experience for the File or Protocol contract. The Application Activation Manager is a COM component implemented in twinui.dll identified by CLSID_ApplicationActivationManager.

```
    interface IApplicationActivationManager : IUnknown
{
    …

HRESULT ActivateForFile(
        [in] LPCWSTR appUserModelId,
        [in] IShellItemArray *itemArray,
        [in, unique] LPCWSTR verb,
        [out] DWORD *processId);

HRESULT ActivateForProtocol(
        [in] LPCWSTR appUserModelId,
        [in] IShellItemArray *itemArray,
        [out] DWORD *processId);
 }
```

The ActivateApplicationForFile and ActivateApplicationForProtocol methods accept an IShellItemArray, which is the input to DelegateExecute verb implementations via IObjectWithSelection::SetSelection.

Note that IAplicationActivationManager API only works from medium integrity level processes (High IL will not work). Therefore, it is recommended that this object is CoCreated with CLSCTX_LOCAL_SERVER as this object is registered to run in the DLLHOST surrogate at Medium IL.

Invocation into desktop presentation

Metro style enabled desktop browsers must set an AppUserModelID at runtime when running in desktop mode by calling SetCurrentProcessExplicitAppUserModelId and

providing the value specified under PKEY_AppUserModel_ID in the browser's primary tile. This ensures that the AppUserModelID "link" between tiles and the Metro style enabled desktop browser is preserved.

Additionally, it should be noted that since secondary tiles are associated with the AppUserModelID of the browser's primary tile, the browser must have the same runtime AppUserModelID as its primary tile.

When activating into the Metro style presentation, SetCurrentProcessExplicitAppUserModelId has no effect and therefore should not be called.

Note: If the Metro style enabled desktop browser is not currently selected as the user's default browser, it can only launch in the desktop.

Launching other Apps via protocols

It is common for web browsers to support launching apps through the windows file and URI protocol scheme association system using the ShellExecuteEx() API. It is strongly recommended that browsers implement the AssocIsDangerous() API to safely handle protocol activation of other apps on the system. It is also strongly recommended that browsers provide a consistent, predictable user experience when users select protocol links that cause the activation of a different Windows app.

The AssocIsDangerous() API identifies file types that have potentially dangerous content, including the ability to execute code. Browsers should discourage users from launching programs that are identified as dangerous by this API or mitigate the threats using the attachment execution services (AES) API.

The following steps are best practices:

- Verify the source of the content via signatures
- Verify the reputation of the particular file
- Inform the user about what app will be launched to handle the file or protocol
- Require the user to confirm that the source of the launch is allowed to perform the action

ShellExecuteEx() has been extended to report information to the caller about the identity of the app that will be activated. The API provides the name of the process or CLSID of the handler that will be launched, the name of the app, and the publisher name and icon. This information should be used when constructing a user consent dialog, or when evaluating per application policy as managed by browser-specific settings.

Consistent with the guidance in the Contextual Launch section of this document, it's recommended that browsers provide a configurable setting to give users greater awareness of the presentation experience of the to-be-invoked app. For example, use the information gathered by the extended ShellExecuteEx() API to populate a contextually appropriate dialog. In that dialog, indicate to the user what app will be launched and whether that app uses Metro style or desktop presentation. The CLSID can be used to distinguish Metro style apps from desktop apps.

Windowing

The Windows 8 Consumer Preview Metro style user experience is designed around a number of principles related to immersion and consumption:

- Metro style apps run only in the Metro style user experience. Using the new windowing APIs, they cannot create windows in the desktop experience.
- The Metro style user experience is focused on a single app at a time, with multitasking, app switching, and notifications provided by the Metro style experience.
- Metro style apps are sized to fill available space in the Metro style user experience environment, with a static number of supported "views" (e.g. portrait, landscape, full screen).
- Ongoing activities in the desktop remain in the desktop environment.

A Metro style enabled desktop browser, configured as the user's default, can choose to participate either in the Metro style user experience or the desktop, but not both from the same running process.

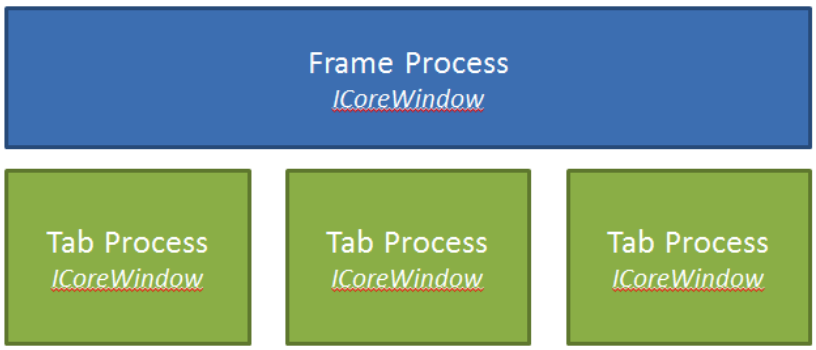
- The browser must first activate itself as a Metro style app; once this has been done and it is the user's default browser, it will be identified as such by the system:
  - A Metro style enabled desktop browser, when identified and activated as a Metro style enabled desktop browser, can create and manage windows in the Metro style user experience.
  - Attempting to create a window in the Metro style user experience when not identified as a Metro style enabled desktop browser will fail.
  - Attempting to create a desktop window when identified as a Metro style enabled desktop browser will succeed, but the Metro style enabled desktop browser will not be able to change focus (bring to foreground) nor manage any desktop windows. Any calls to SetForegroundWindow() or SetActiveWindow() will fail under these conditions. There is no automatic switching between the Metro style experience and the desktop.
  - A Metro style enabled desktop browser may not call LockSetForegroundWindow(); any calls to the API will fail.
  - Any calls to AllowSetForegroundWindow() may succeed if the Metro style enabled desktop browser is in the foreground, however this call will not facilitate any attempt to allow a desktop app or another Metro style app to move to the foreground.
  - Any calls to EnableWindow() from a Metro style enabled desktop browser will fail if the target hWnd is not a Metro style user experience window.
  - A Metro style enabled desktop browser may not attach input threads with any desktop app and vice-versa. Any calls to AttachThreadInput(*idAttach, idAttachTo*) where the *idAttach* and *IdAttachTo* parameters are neither both viewable in the Metro style user experience nor both desktop will fail.
  - A Metro style enabled desktop browser may not change the foreground (even to a Metro style user experience window) when a desktop app,

accessibility app or system component is in the foreground. Any calls to SetForegroundWindow() or SetActiveWindow() will fail under these conditions.

- Before activating as a Metro style app, it is advised that the browser destroy any visible desktop windows as these will be unmanageable once identified as a Metro style enabled desktop browser.
- Should a Metro style enabled desktop browser wish to launch a desktop app and have that app come to the foreground (for example, downloading and launching a file such as a PDF reader), it may do so by using ShellExecuteEx() and specifying the *SEE_MASK_FLAG_LOG_USAGE* flag in the *fMask* field. If this flag is not specified the desktop app will launch, but will not be able to come to the foreground. There is no automatic switching between the Metro style experience and the desktop.

Guidance for multi-process browsers

For browsers that have adopted a multi-process architecture—typically, a separation of the "frame" process from the "tabs" that display webpage content—create a frame window and host each tab in its own separate top-level ICoreWindow, as illustrated in the following example:



There are a few identified limitations with this method of window hosting, and users may experience the following behavior:

- Flickering of windows on tab switches
- Briefly showing the Start background on tab creation / switching
- Narrator and other accessibility tools having difficulty navigating multiple top-level windows

Note: Microsoft is investigating a revision to the recommended model: from multiple top-level ICoreWindows to a single top-level ICoreWindow held by the frame process. Should this happen, a move from the current guidance to a redesigned windowing architecture would likely be necessary to achieve the desired level of app fidelity and correct participation in app contracts.

App contracts

Contracts are like agreements between Windows and Metro style apps, including Metro style enabled desktop browsers. They support some kind of user interaction

and help users complete scenarios. Metro style enabled desktop browsers are encouraged to participate in app contracts, particularly Search, Sharing, PlayTo, and Print. Some browser-specific examples of the utility of contracts include playing music originating from a website on a connected (to the PC) stereo and/or sharing article content as viewed on a website with another Metro style app.

For more information, see Windows application contracts.

Search contract

Search is unchanged from Windows 7 in terms of defaults. Metro style enabled desktop browsers that are set as the user's default may choose to (and are encouraged to) participate in the Search contract of the Metro style user experience. When a Metro style enabled desktop browser is not set as the user's default browser, a tile entry for the browser will not appear in the search pane.

For more information, see Quickstart: Adding Search.

Share contract

Metro style enabled desktop browsers are encouraged to help users share website content with other Metro style apps on the system. A typical flow starts with the user selecting some content displayed in the browser – a video, a block of text, an image – and choosing the share charm to share the content with another app. When the user has selected an object and chosen to share, the browser should share that selection. In the absence of a currently selected object, the browser should share the URL to the webpage along with metadata such as the page title and description. Browsers should also consider implementing and sharing custom formats. For example, a differentiating feature (that would also enhance the user experience) might be to add extra entities to a user selection, such as a movie, and share that data as a custom format.

For more information, see Quickstart: Sharing Content.

Play To contract

The Play To contract provides the ability to stream HTML audio, video and images from websites to certified Play To devices. The typical scenarios are streaming a single video from a website, or streaming a playlist of audio/video elements. It's up to the browser to identify and specify/update the media element that should be set as the Play To source.

The algorithm to set the Play To source is up to the browser and could rely on a combination of user input/selection, site developer specifying the source using a media element attribute, and/or other disambiguation heuristic when multiple media elements are present on the same page. For example, Internet Explorer 10 enables webpage developers to use the *x-ms-xPlayToPrimary* attribute to indicate an audio, video, or image is the default media source. Also, because Play To is enabled by default in Internet Explorer 10, the *x-ms-xPlayToDisabled* attribute can be used by a webpage developer to disable the functionality on the page. Other browser vendors may choose to provide similar vendor-specific attributes.

The typical flow starts with the user swiping the charms and selecting Devices. At this point the browser is asked to provide a Play To source. If a source is specified, available Play To target devices are shown in the Device charm. To synchronize device events with the media elements, the browser must register for device events from the Play To Manager.

For more information, see Streaming Media to Devices using Play To and the related Quickstart: Using Play To in apps.

Print contract

Browsers are encouraged to implement the print contract, to support simple and predictable printing experiences within the Metro style user experience. For more information, see: Quickstart: Adding simple print capability.

Process Lifetime Management

Metro style enabled desktop browsers are subject to the same Process Lifetime Management (PLM) rules as other Metro style apps. When drawing desktop windows and managing background downloads, browsers are subject to the process lifetime standards of desktop apps.

Close gesture

Users may choose to close a Metro style enabled desktop browser via operating system features such as the keyboard combination ALT+F4 or by using the close gesture. Users invoke the close gesture by "dragging" a representation of the window from the top of the app to the bottom of the screen. When the Metro style enabled desktop browser is active (not suspended, not in the background), painting via ICoreWindow, it will be sent a registered window message with the string "DefaultBrowserClosing" as it moves off screen during window close. The browser should use this message as a signal to prepare for termination after suspension is complete.

For example, a browser may send events to the active webpage, such as BeforeUnload and Unload when the DefaultBrowserClosing message is received. Some websites rely on events such as these to trigger specific behaviors, such as sending data back to a server to mark the end of a user session.

Note: Open browser dialogs, such as those originating from webpages, should not prevent the user from closing the browser. For instance, confirmation dialogs that sometimes show when a page handles the BeforeUnload event should be suppressed and not allowed to interfere with closing the browser. For more information, see Managing the Application Lifecycle.

Completing downloads and uploads prior to suspend

Metro style enabled desktop browsers suspend when the user switches away from the browser and resume when the user switches back to it. There are a limited number of cases where the browser may postpone the act of suspension in order to complete a long running user task:

- Completing file uploads, such as an active upload via POST with a file attached or via XMLHttpRequest
- Completing file downloads

Browsers are strongly encouraged to leverage the background APIs in a manner consistent with Metro style apps. Doing so improves battery life by more efficiently consuming system resources. For more information, see Quickstart: Downloading and uploading files. Alternatively, Metro style enabled desktop browsers may use the PowerCreateRequest, PowerSetRequest, and PowerClearRequest functions as documented in the Power Availability Requests whitepaper. Note that these APIs are only callable from medium integrity level (or higher) processes. For more information, see Guidelines for managing app lifecycle.

Roaming browser application data

Users can easily keep their browser's application data in sync across multiple devices when you support roaming. Doing so benefits the user by:

- Reducing the amount of setup work that the user needs to do for your browser on their second device.
- Enabling users to continue a task, such as composing a list, right where they left off, even on a different device. Windows replicates roaming data to the cloud and synchronizes the data to other devices where the user has installed the browser.

Windows limits the size of the application data that each app may roam. For this reason, it is a best practice to use roaming data only for user preferences such as favorites and browser history. For more information, see Guidelines for roaming app data.

Metro style enabled desktop browsers can sync the following:

- Favorites / Shortcuts
- Typed URLs
- History
- Up to 5 other registry based settings

Register for roaming

The following registrations are required in order to roam data:

- HTTP and HTTPS protocol registration.
- A registry key HKLM\Software\RegisteredApplications with the browser name and the path to Capabilities Registry Key (of type REG_SZ For example: Software\Example Company\Example Browser\Capabilities).
- A Start shortcut with PKEY_AppUserModel_IsDualMode set to TRUE (VT_BOOL).
- A registry key under HKLM\Software\Microsoft\Windows\CurrentVersion\SettingsSync\BrowserSettings. The key name should be the same as the browser name. No sub keys and values are needed under this.

- The Capabilities Registry Key should contain a subkey named “Roaming”. This is the location where data stores are registered.

Sync behavior

Under normal circumstances, application data will be uploaded within a few minutes of receiving a change notification. For performance reasons, however, the sync framework may delay upload of new history entries, in some cases up to 24 hours. When a sync causes a conflict the “last writer wins”.

Syncing favorites / Internet shortcuts

In order to roam favorites, create a registry key with the name “Favorites” under the browser’s roaming capabilities registry key. For example: HKLM\Software\Example Company\Example Browser\Capabilities\Roaming\Favorites. Then, create a registry value with name “KnownFolderID” of type REG_SZ that specifies the GUID for the known folder where the user’s favorites are stored. Internet shortcuts within this folder and subfolder are roamed. The shortcuts must be created using the APIs as explained in Internet Shortcuts.

Optionally, you can register to roam the order of shortcuts. To accomplish this, create a registry value under “Favorites” capabilities with name “OrderRegPath” that specifies the root registry key under HKCU containing the order information for the top level “Favorites” folder and all sub folders. For example: Software\Example Company\Example Browser\FavoritesOrder.

For the order of shortcuts in the top level “Favorites” folder, the above registry key can contain a registry value with the name “Order” and type REG_BINARY with a value of the browser’s choosing. For the order of shortcuts in any subfolders, this registry key must contain corresponding sub keys mirroring the folder hierarchy (again, with the name “Order” and type REG_BINARY).

For example, if the favorites folder is %LOCALAPPDATA%\Example Company\Example Browser\Favorites\Sub Folder1\Sub Folder2, then the order information for the root favorites folder must be in a registry value under HKCU\Software\Example Company\Example Browser\FavoritesOrder, the order information for the first sub-folder (…\Favorites\Sub Folder1) must be in a registry value under HKCU\Software\Example Company\Example Browser\FavoritesOrder\Sub Folder1, and so on.

Watch for changes to the favorite’s folder before refreshing the browser UI representation of the user’s favorites. When a shortcut is roamed for the first time, an additional property {FMTID_Intshcut, PID_IS_ROAMED } of type VT_BOOL and value VARIANT_TRUE is added. Set the value to VARIANT_FALSE once the browser has processed the update.

Additionally, Windows sends an extended shell change notification SHCNE_EXTENDED_EVENT with the SHCNEE_ORDERCHANGED flag set and the PIDL of the folder in which the shortcut was changed. Browsers can listen for this notification to refresh the favorites order following roaming.

Syncing typed URLs

In order to sync typed URLs, create a registry key with the name “TypedURLs” under the browser’s roaming capabilities registry key. For example: HKLM\Software\Example Company\Example Browser\Capabilities\Roaming\TypedURLs. Then, create a registry value with the name “URLsRegPath” of type REG_SZ that specifies the path (within HKCU) that contains the URLs in key value format. For example:

HKCU\Software\Example Company\Example Browser\TypedURLs

Name	Type	Value
url1	REG_SZ	http://www.microsoft.com
url2	REG_SZ	http://msdn.microsoft.com
…		

Windows roams 50 of the user’s most recently accessed URLs. If a user has more recent entries, browsers may choose to store the time when each of the URLs was typed; Windows uses this information to ensure that the most recently typed URLs sync first. To do this, create a registry value with the name “URLsTimeRegPath” of type REG_SZ that specifies the path (within HKCU) that contains the URL’s typed time. For example:

HKCU\Software\Example Company\Example Browser\TypedURLsTime

Name	Type	Value
url1	REG_BINARY	FILETIME, in UTC, of the time when the URL was typed in the browser
url2	REG_BINARY	FILETIME, in UTC, of the time when the URL was typed in the browser
…		

Windows notifies the browser of an update to the list of typed URLs by sending a message to the app’s top level window with a LPARAM value “TypedUrlsRoamed”. Specify the window class Windows should send the message to by creating a registry value of name “WindowClassesToNotify” of type REG_SZ and value equal to the name of the window classes, separated by comma.

Syncing browser history

The following registrations are required in order to sync browser history:

- Implement a shell folder and register it as a KNOWNFOLDER. When the user’s browsing history changes, call SHChangeNotify to indicate that the folder contents have changed.
- Store URL history in a WinInet cache using a browser-specific prefix ending with a colon. For example, the Contoso browser might use the prefix “Contoso:” when storing URLs in a WinInet cache.

In order to sync browser history, create a registry key with the name “WinInet” under the browser’s roaming capabilities registry key. For example: HKLM\Software\Example Company\Example Browser\Capabilities\Roaming\WinInet. Then, create a registry value with the name “KnownFolderID” of type REG_SZ that specifies the shell folder GUID. Also, create a registry key value with the name

"WinInetPrefix" of type REG_SZ and value equal to the name of the browser-specific prefix, ending with a colon (:).

For more information about using WinInet caching APIs, see Caching.

Store the URL in the format "Prefix: username@URL" where "username" is the value returned by the GetUserName() function. For example,

Contoso: user@http://www.microsoft.com/en-us/default.aspx

The CommitUrlCacheEntry function allows a browser to store a list of URLs that have been accessed and associate each with a local copy of the data that was last received. It also allows the browser to store header information (in the lpHeaderInfo parameter). In order to roam history data, the lpHeadInfo parameter must contain a serialized property store instead of the header info. This is done by using PSCreateMemoryPropertyStore to create a store and saving the browser's properties into it. Then, use IPersistStream::Save to serialize the property store into a byte array in memory. Finally, call CommitUrlCacheEntryA and pass the byte array for the lpHeaderInfo field to store the property store for that URL.

Note: A browser can append arbitrary bytes to the end of the lpHeadInfoStream to store local data that should not be altered or roamed by Windows.

Retrieve binary history data by performing the inverse: use GetUrlCacheEntryInfoA, remove the prefix and username from the returned URL, and then use PSCreateMemoryPropertyStore and IPersistStream::Load to convert the lpHeaderInfo back into an IPropertyStore. The ANSI versions of other WinINet caching APIs such as FindFirstUrlCacheEntryA and FindNextUrlCacheEntryA can also be used to enumerate the cache in a similar manner.

Note: Roaming history requires use of the ANSI WinInet cache APIs to persist binary data. If a URL contains non-ANSI characters, encode the URL using well-known schemes such as percent-encoding.

Detect that new URLs have been added to the cache via roaming by checking the {FMTID_InternetSite, PID_INTSITE_ROAMED} property.

Value of PID_INTSITE_ROAMED	Description
Value not set or PIDISR_UP_TO_DATE	This cache entry has not been modified by roaming.
PIDISR_NEEDS_ADD	This cache entry was added to the cache by roaming. Set PIDISR_UP_TO_DATE once processing of the entry is complete.
PIDISR_NEEDS_UPDATE	This cache entry already existed on the local machine, but it was updated by roaming. Set PIDISR_UP_TO_DATE once processing of the entry is complete.
PIDISR_NEEDS_DELETE	Roaming detected that this cache entry should be deleted. For example, the user may have cleared his or her browser history. Delete the entry using DeleteUrlCacheEntry.

The following rules govern conflict resolution for roaming browser history:

- The property stores in the lpHeaderInfo stream are merged such that the most recently changed values are preserved.
- If the property store contains the public “visit count” property {FMTID_InternetSite, PID_INTSITE_VISITCOUNT}, the largest of the values is preserved.
- If there is additional data in the local cache entry’s stream after the serialized property store, the local data is preserved.

The following rules govern what history entries roam:

- Roam: URLs with http:// or https:// schemes; Cache entries of type “URLHISTORY_CACHE_ENTRY | NORMAL_CACHE_ENTRY”; The 1000 most recent updated history entries
- Not Roam: URLs to images; Cache entries that do not contain a serialized property store in their lpHeaderInfo; Properties in the serialized property store that would require deserialization (such as VT_UNKNOWN, VT_STREAM, or VT_STORAGE); History entries with more than 2KB in their lpHeaderInfo structure

When the browser has updated or added any history data, use the SHChangeNotifyAPI and send a change notification for the registered known folder. Conversely, when Windows has concluded roaming a user’s history, a timestamp of the most recent operation is written to the registry at HKCU\Software\Microsoft\Windows\CurrentVersion\SettingsSync\Namespace\BrowserSettings\WinInet-browsername!LastRoamed. Compare this value with the last known time that entries were detected to determine if history has changed. Alternatively, running instances of the browser will receive a window message “WM_SETTINGCHANGE” with a wParam value of “0” and an lParam value of “WinINetRoamed”. Specify the window class Windows should send the message to by creating a registry value of name “WindowClassesToNotify” of type REG_SZ and value equal to the name of the window classes, comma separated, in the \Capabilities\Roaming\WinInet registry key.

Roaming other registry settings

In addition to roaming user favorites, typed URLs, and browser history, browsers may also sync up to a maximum of five other registry keys. Create a registry key with the name of your choosing under the browser’s roaming capabilities registry key. For example: HKLM\Software\Example Company\Example Browser\Capabilities\Roaming\NewKey. Then, create a registry value named “RegistryRoot” of type REG_SZ that specifies the path to the key under HKCU that should be roamed.

Creating a registry value of name “FilterIn” of type REG_SZ and value containing a comma separated list of registry values lets you specify an inclusion list for roaming. Any values not specified that appear under the “RegistryRoot” will not be roamed. Conversely, creating a registry value of name “FilterOut” type REG_SZ and value containing a comma separated list of registry values to exclude lets you specify an exclusion list for roaming. All values under the “RegistryRoot” will be roamed except for those specified in the “FilterOut” value.

Each time a registry key is roamed, Windows sends a message to the browser's window with LPARAM value "<SettingName>Roamed". Specify the window class Windows should send the message to by creating a registry value of name "WindowClassesToNotify" of type REG_SZ and value equal to the name of the window classes, comma separated, in the \Capabilities\Roaming\<SettingName> registry key.

For more information, see Application Data.