Xaml Win32 Hosting – Dev Design

# Introduction

## Goals

This paper discusses a solution that allows Xaml UI to be hosted in classic Win32 applications.

The goal of this design is to allow Office to host modern Xaml UI in classic Win32 contexts. Another goal is to allow customers with existing investments in legacy UI frameworks (WinForms, WPF, etc.) to modernize those applications by mixing in modern Xaml UI.

Another goal is to deliver customer value now without compromising the long-term strategy for the Windows developer platform. The future architecture (and primary long-term investment) of Windows is WCOS, UWP apps, and upcoming interop/hosting technology such as Islands. But by investing a minor amount into this effort, we will unlock some short-term customer value now, and not subtract much effort from these longer-term investments.

## Principles

This design involves a very minimal hosting interface, which supports the high ROI goal. The hosting interface will not add surface area where an existing, available API provides equivalent functionality (examples of this principle are given later in this document – for example, see the [Layout](#_Layout) section).

As far as possible, the hosting interface will be interop technology agnostic. This supports the goal of being able to use this interface both today and in the future. In the first iteration the implementation will use CoreWindow as the basis of its interop. In the future, it will use Composition Islands as that basis instead. The same interface will be usable for both scenarios, and we will be able to switch the underlying interop technology without churning customers or creating technical debt.

## Constrast to XamlPresenter

There is an existing internal solution for Xaml Win32 hosting: XamlPresenter. XamlPresenter was developed and shipped in Windows 8.1 (for details see the Win8.1 era [design doc](https://microsoft.sharepoint.com/:f:/r/teams/osg_core_dep/UxP/Shared%20Documents/UWP%20in%20Win32/Background/(WinBlue)%20Xaml%20Hwnd%20Hosting%20Dev%20Design.docx)). It had the same primary goal: allow modern Xaml UI to be hosted in classic Win32 experiences. In Windows 8.1, it was used by the Search Pane, Search Results View, Settings Admin Flows, Lock Screen Camera UI, and the Contact Card UI. At present, it is used by the Lock Screen and CredUI.

The design philosophy behind XamlPresenter was to essentially bypass the modern app model. One aspect of this philosophy was that the “foundational pieces” of the modern app model – such as CoreWindow – were not made available. At that time this was not a huge limitation, but since then many more components have taken dependencies on these foundational pieces. This has introduced significant functional gaps in XamlPresenter scenarios: for examples, inking, MediaPlayer, and Reveal/Acrylic are all unavailable in XamlPresenter-based UI.

In contrast, this hosting model will embrace the modern app model. It will make foundational pieces – such as CoreWindow – available, eliminating many (but not all) functional gaps that XamlPresenter has. However, this should not be seen as a panacea – many modern features will require more than just enabling CoreWindow.

## Design Summary

This design will publish a new xaml hosting interface, using a working name XamlHost. This interface is intentionally minimal, and exposes only a small set of APIs needed for some simple hosting scenarios. It’s expected that over time this interface will be expanded. For details on what the interface looks like, see the [Startup and Shutdown](#_Startup_and_Shutdown) section.

Use of this hosting interface will create foundational app model components such as CoreWindow and CoreApplicationView. Having these foundational pieces available in a Win32 environment will cause many components – such as WUXC, InkPresenter, and MediaPlayer – to “just work” in Win32 without requiring special-case code. For more on this see the [Foundational App Model](#_Foundational_App_Model) section.

Input will be handled by existing mechanisms, very similar to how input in a UWP app works. Xaml will get its input through a mix of USER window messages and CoreWindow events. For more details see the [Input](#_Input) section.

Output will be initially handled by hwnd based interop. Once Composition Islands are available in a Win32 environment, output will be transitioned to use Islands and hwnds will only be used “at the edge” as required to integrate into existing Win32 UI. For more details see the [Output](#_Output) section.

**Late-breaking updates:**

There are several points that need to be revised in this paper. Until I have a chance to add more details about these points, here’s a summary:

* Multi-instancing will be supported by using a single CoreWindow and primary CoreApplicationView per thread. Instances will be represented by secondary CoreApplicationViews and Islands. The 3 dozen GetForCurrentView APIs will be split into functionality that is truly per-thread (and those APIs will remain as-is) and functionality that is per-instance of content (and those APIs will be deprecated and replaced by functionality on CoreApplicationView, Islands, CoreComponentInput, etc.).
* Some Win32 customers like WinForms are fundamentally hwnd based, and for those customers there will always be an hwnd “at the edge”. When we support multi-instancing, those customers will use HwndIslandSite. Other customers do not need to be fundamentally hwnd based – certainly Office, and even WPF could be made to host Xaml UI without any hwnds. For those customers, we will support multi-instancing using non-hwnd Island sites.

## Deliverables

This effort will produce three types of deliverables:

1. At the lowest layer is the new XAML hosting interface. This will very likely be initially shipped behind velocity to give us an opportunity for targeted customer feedback before shipping it publicly.
2. This effort will produce a WPF interop element that allows for the creation of hybrid WPF/Xaml applications. This WPF interop element will use the XamlHost interface and implement framework-to-framework bridging (layout, automation, etc.). It demonstrates the kind of logic that needs to be implemented at a framework-specific layer that’s higher than the base XamlHost interface.
3. Finally, there will be a set of sample apps that demonstrate Xaml UI in Win32 for various scenarios.

# Architectural Issues

## Architectural Goals

A primary goal of this design is that “leaf node components” such as Windows.UI.Xaml.Controls.dll, InkPresenter, MediaPlayer, etc. work without any changes – that is, without requiring special case support for Win32 within those components. These modern leaf nodes are where much of the new customer value is accruing – so this hosting technology must enable those components to “just work”.

A related goal is that this hosting technology will fully replace XamlPresenter, allowing us to remove XamlPresenter from the product, switch over all XamlPresenter uses to XamlHost, and remove much of the special case code inside the Xaml framework that supports running in a XamlPresenter-based environment. In other words, windows.ui.xaml.dll largely becomes just another leaf node component and doesn’t need to special-case Win32 as it currently does.

The primary way these goals will be realized is by making key foundational app model APIs available in a Win32 environment. In short, these APIs are CoreApplication, CoreApplicationView, CoreWindow, CoreDispatcher, and Xaml’s Application and Window classes. The way this works is detailed further below.

A secondary architectural goal is to begin work on unwinding some architectural choices that were made in Windows 8. For example, the Windows 8 app model set up a 1:1 relationship between windows and threads. This relationship has been modeled in dozens of APIs. As modern Windows matures, we continually find scenarios in which this relationship is problematic (for example, Tab Shell). We will need to modify this architecture over many releases so that it becomes more flexible, and this Win32 effort will serve as an initial step towards that. Much of what we learn about the existing limitations of the modern app model will directly feed into longer term efforts like Islands.

## Foundational App Model

### CoreWindow

CoreWindow will be supported in Win32, using the existing WindowServer implementation that is already based on USER window messages.

Windows.ui.dll already exposes a private DLL export for creating CoreWindows (PrivateCreateCoreWindow). This API takes a parameter that specifies the window type, and this design adds a new window type, used to specify that the window should be created “non-immersively”:

typedef enum \_WINDOW\_TYPE

{

IMMERSIVE\_BODY = 0,

IMMERSIVE\_DOCK = 1,

IMMERSIVE\_HOSTED = 2,

IMMERSIVE\_BODY\_ACTIVE = 4,

IMMERSIVE\_DOCK\_ACTIVE = 5,

NOT\_IMMERSIVE = 6,

} WINDOW\_TYPE;

This is implemented by creating a non-immersive CoreWindow on the desktop z-band and skipping certain shell code that is only needed for immersive windows. This design also allows the possibility of marking a non-immersive CoreWindow such that other code can behave differently for that window type (if such a need arises).

This ability to create non-immersive CoreWindows will be a private API. It will only be consumed by the Xaml hosting interface.

Note that Win32 CoreWindows have the following properties:

1. They are created as top-level windows, and can be modified after creation to be child windows
2. Internal code can get the underlying hwnd via the internal ICoreWindowInterop interface

These properties will be utilized by the xaml hosting interface to implement its contract (returning a child hwnd to its client). See the [Output](#_Output) section for details.

### CoreDispatcher and DispatcherQueue

CoreDispatcher has become critical for many modern Windows components like Windows.UI.Xaml.Controls, InkPresenter, and MediaPlayer. These components have a common need to marshal work back to a UI thread, and they assume CoreDispatcher exists and is the proper way to do that.

As of RS4, CoreDispatcher is effectively deprecated. A new component – DispatcherQueue – serves as its replacement. DispatcherQueue was designed to seamlessly integrate into a Win32 environment (for example, a DispatcherQueue is serviced by a canonical Get/Translate/DispatchMessage loop). However, it will be some time before all the modern Windows components switch their dependency from CoreDispatcher to DispatcherQueue, and during that time we need a Win32 solution for the CoreDispatcher dependency.

Unlike DispatcherQueue, a CoreDispatcher will not be serviced by a canonical Win32 message loop. Creating a CoreDispatcher in Win32 is not difficult, but servicing it is. When a CoreWindow is created, it creates a CoreDispatcher, and this will be true of non-immersive CoreWindows as well. However, servicing that CoreDispatcher is currently a responsibility of client code.

Typically, existing Win32 code will want to “own the message loop”. CoreDispatcher also wants to “own the message loop” (this is what its ProcessEvents(ProcessUntilQuit) API does). For example, the WPF framework has a complex message loop implementation, and it isn’t feasible to replace this implementation with CoreDispatcher’s message loop.

We have several options:

1. Do nothing – and do not support CoreDispatcher in Win32. Components that we need to work in Win32 scenarios will need to be modified to depend on DispatcherQueue instead of CoreDispatcher. (Examples of components that would need to update: Windows.UI.Xaml.Controls, InkPresenter, etc.).
2. Support CoreDispatcher in Win32 and require Win32 apps to replace their message loops with a call to CoreDispatcher’s ProcessEvents API. This is a feasible option for simple Win32 apps, but is infeasible for most apps that need to own the message loop and implement complex message processing. Examples of Win32 UI where this option is infeasible are WPF and Office.
3. CoreDispatcher exists and in "Win32 mode" is re-implemented entirely on top of DispatcherQueue. This is technically feasible, although not all of CoreDispatcher’s existing operations map to functionality on DispatcherQueue. In detail:
   1. ProcessEvents can continue to be supported and will run a canonical Win32 message loop as it does today. But as discussed, many customers will not be able to use this.
   2. Accelerator key APIs will not work, since these rely on hooks into the message loop that’s run by ProcessEvents, and Win32 customers have their own message loops. This shouldn’t be a practical problem, since the Accelerator key APIs are little used today, and existing Win32 apps have their own (unrelated) mechanisms for accelerator keys.
   3. Awaiting RunAsync can’t be mapped exactly since DispatcherQueue doesn’t offer the same facility for awaiting its operations. However, this is very little used functionality and it should be possible to map a portion of it.
4. CoreDispatcher exists and in “Win32 mode” has a different code path that uses CoreMessaging directly (similar to how DispatcherQueue works). This is similar to the third option, but it avoids the considerable overhead of an extra layer.

This design rejects the first option since many valuable components would be unavailable in Win32 scenarios. The second option is supported, but is not feasible for many customers as explained. The third option has been implemented as a proof of concept and will be detailed next. The fourth option is the option favored by the team that owns CoreDispatcher and will most likely be the long-term solution for CoreDispatcher in Win32.

Currently the design has implemented the third option:

CDispatcher (the implementation class for CoreDispatcher) has an internal Create function. This function now takes a flag to run in “dispatcher queue mode”:

static void Create(bool fDispatcherQueueMode, \_\_deref\_out CDispatcher\*\* ppDispatcher);

During CoreWindow creation, a CoreDispatcher is created. CoreWindow passes true for “dispatcher queue mode” if it is a NOT\_IMMERSIVE type window (as explained above):

CDispatcher::Create((NotImmersiveWindow == windowType) /\* fDispatcherQueueMode \*/, &\_spDispatcher);

CDispatcher::EnqueueAsyncWork is the implementation of the RunAsync/RunIdleAsync APIs. In “dispatcher queue mode”, EnqueueAsyncWork creates an IDispatcherQueueHandler that wraps the CoreDispatcher handler, and registers the callback with DispatcherQueue. It also maps CoreDispatcher priorities to DispatcherQueue priorities:

if (\_fDispatcherQueueMode)

{

ComPtr<IDispatcherQueueHandler> spDQHandler = Callback<Implements<RuntimeClassFlags<ClassicCom>, IDispatcherQueueHandler, FtmBase>>([priority, spHandler, spIdleHandler]() -> HRESULT

{

HRESULT hr;

if (priority == CoreDispatcherPriority\_Idle)

{

ComPtr<CIdleDispatchedHandlerArgs> spIdleDispatchedArgs;

hr = MakeAndInitialize<CIdleDispatchedHandlerArgs>(&spIdleDispatchedArgs);

if (SUCCEEDED(hr))

{

hr = spIdleHandler->Invoke(spIdleDispatchedArgs.Get());

spIdleDispatchedArgs->SetActiveState(false);

}

}

else

{

hr = spHandler->Invoke();

}

return hr;

});

boolean dqEnqueueResult;

DispatcherQueuePriority dqPriority = DispatcherQueuePriority\_Normal;

if (priority == CoreDispatcherPriority\_Idle || priority == CoreDispatcherPriority\_Low)

{

dqPriority = DispatcherQueuePriority\_Low;

}

else if (priority == CoreDispatcherPriority\_Normal)

{

dqPriority = DispatcherQueuePriority\_Normal;

}

else if (priority == CoreDispatcherPriority\_High)

{

dqPriority = DispatcherQueuePriority\_High;

}

else

{

hr = E\_UNEXPECTED;

}

if (SUCCEEDED(hr))

{

coreAsyncInfo->SetState(ICoreAsyncInfo::Created, ICoreAsyncInfo::Queued);

ComPtr<ICoreAsyncInfo> coreAsyncInfoPtr;

hr = coreAsyncInfo.As(&coreAsyncInfoPtr);

if (SUCCEEDED(hr))

{

hr = coreAsyncInfoPtr->StartOperation();

if (SUCCEEDED(hr))

{

hr = \_dispatcherQueue->TryEnqueueWithPriority(dqPriority, spDQHandler.Get(), &dqEnqueueResult);

if (!dqEnqueueResult)

{

hr = E\_FAIL;

}

if (SUCCEEDED(hr))

{

\*ppCoreAsyncInfo = coreAsyncInfoPtr.Detach();

}

else

{

coreAsyncInfoPtr->SetState(ICoreAsyncInfo::Queued, ICoreAsyncInfo::Error);

coreAsyncInfoPtr->SetError(hr);

}

}

}

}

}

Here's an example stack showing how a CoreDispatcher RunAsync callback is being serviced via this approach. In this stack, WPF’s message loop is dispatching to CoreMessaging, which is invoking a DQ callback, which is invoking a CD callback, which calls the original CD handler in Inking code.

        Windows.UI.Input.Inking.dll!Microsoft::WRL::Details::DelegateArgTraits<long (\_\_cdecl Windows::UI::Core::IDispatchedHandler::\*)(void) \_\_ptr64>::DelegateInvokeHelper<Microsoft::WRL::Implements<Microsoft::WRL::RuntimeClassFlags<2>,Windows::UI::Core::IDispatchedHandler,Microsoft::WRL::FtmBase>,<lambda\_86612196345b5da8d964f14874528f07>,-1>::Invoke() Line 244    C++     Symbols loaded.

       Windows.UI.dll!Windows::UI::Core::CDispatcher::EnqueueAsyncWork::\_\_l16::<lambda>() Line 909       C++        Symbols loaded.

       Windows.UI.dll!Microsoft::WRL::Details::DelegateArgTraits<long (\_\_cdecl Windows::System::IDispatcherQueueHandler::\*)(void) \_\_ptr64>::DelegateInvokeHelper<Microsoft::WRL::Implements<Microsoft::WRL::RuntimeClassFlags<2>,Windows::System::IDispatcherQueueHandler,Microsoft::WRL::FtmBase>,HRESULT <lambda>(void),-1>::Invoke() Line 245    C++     Symbols loaded.

       CoreMessaging.dll!Windows::System::DispatcherQueue::DeferInvokeCallback(void \* userInfo) Line 854 C++        Symbols loaded.

       CoreMessaging.dll!System\_\_Action$CallbackThunk(System::Delegate \* \_cn\_pDelegate) Line 461 C++     Symbols loaded.

       [Inline Frame] CoreMessaging.dll!System::Action::Invoke() Line 91 C++     Symbols loaded.

       CoreMessaging.dll!Microsoft::CoreUI::Dispatch::DeferredCall::Callback\_Dispatch() Line 79  C++     Symbols loaded.

       CoreMessaging.dll!Microsoft::CoreUI::Dispatch::DeferredCallDispatcher::Callback\_OnDispatch() Line 373     C++        Symbols loaded.

       [Inline Frame] CoreMessaging.dll!Microsoft::CoreUI::Dispatch::Dispatcher::Callback\_DispatchLoop(Microsoft::CoreUI::Dispatch::RunnablePriorityMask allowedPriorities) Line 582      C++     Symbols loaded.

     CoreMessaging.dll!Microsoft::CoreUI::Dispatch::EventLoop::Callback\_RunCoreLoop(Microsoft::CoreUI::Dispatch::RunMode mode, bool ignoreOuterLoop, bool & stopping) Line 841    C++     Symbols loaded.

       [Inline Frame] CoreMessaging.dll!Microsoft::CoreUI::Dispatch::UserAdapter::DrainCoreMessagingQueue(Microsoft::CoreUI::Dispatch::UserAdapter\_\_UserPriority) Line 794 C++     Symbols loaded.

       CoreMessaging.dll!Microsoft::CoreUI::Dispatch::UserAdapter::OnUserDispatch(bool handlingCompletion, Microsoft::CoreUI::Dispatch::UserAdapter\_\_UserPriority priority, System::IntPtr & sessionExportAdapter) Line 685   C++        Symbols loaded.

       [Inline Frame] CoreMessaging.dll!Microsoft::CoreUI::Dispatch::UserAdapter::OnUserDispatchRaw(System::IntPtr handle, Microsoft::CoreUI::Dispatch::UserAdapter\_\_UserPriority) Line 615  C++     Symbols loaded.

       CoreMessaging.dll!Microsoft::CoreUI::Dispatch::UserAdapter\_DoWork(Microsoft::CoreUI::Dispatch::UserData \* userData, Microsoft::CoreUI::Dispatch::UserAdapter\_\_UserPriority priority, bool handlingCompletion) Line 504      C++        Symbols loaded.

       [Inline Frame] CoreMessaging.dll!Microsoft::CoreUI::Dispatch::UserAdapter\_HandleDispatchNotifyMessage(HWND\_\_ \*) Line 566 C++     Symbols loaded.

       CoreMessaging.dll!Microsoft::CoreUI::Dispatch::UserAdapter\_WindowProc(HWND\_\_ \* hwnd, unsigned int message, unsigned \_\_int64 wParam, \_\_int64 lParam) Line 750 C++     Symbols loaded.

       user32.dll!UserCallWinProcCheckWow(\_ACTIVATION\_CONTEXT \* pActCtx, \_\_int64(\*)(tagWND \*, unsigned int, unsigned \_\_int64, \_\_int64) pfn, HWND\_\_ \* hwnd, \_WM\_VALUE msg, unsigned \_\_int64 wParam, \_\_int64 lParam, void \* fEnableLiteHooks, int) Line 259   C++     Symbols loaded.

       user32.dll!DispatchMessageWorker(tagMSG \* pmsg, int fAnsi) Line 3111      C++     Symbols loaded.

       WindowsBase.ni.dll!00007ffc6750bfc8()    Unknown No symbols loaded.

       [Managed to Native Transition]           Annotated Frame

       WindowsBase.dll!System.Windows.Threading.Dispatcher.PushFrameImpl(System.Windows.Threading.DispatcherFrame frame) Line 2326 C#      Symbols loaded.

       PresentationFramework.dll!System.Windows.Application.RunDispatcher(object ignore) Line 2762       C#        Symbols loaded.

       PresentationFramework.dll!System.Windows.Application.RunInternal(System.Windows.Window window) Line 1851  C#        Symbols loaded.

       WPFInteropSample.exe!WPFInteropSample.App.Main() Unknown Symbols loaded.

       [Native to Managed Transition]           Annotated Frame

       mscoreei.dll!\_CorExeMain() Line 6420     C++     Symbols loaded.

       [Inline Frame] mscoree.dll!\_CorExeMain() Line 6150       C++     Symbols loaded.

       [Inline Frame] mscoree.dll!ShellShim\_\_CorExeMain() Line 277      C++     Symbols loaded.

       mscoree.dll!\_CorExeMain\_Exported() Line 1222     C++     Symbols loaded.

       kernel32.dll!BaseThreadInitThunk(unsigned long RunProcessInit, long(\*)(void \*) StartAddress, void \* Argument) Line 64 C       Symbols loaded.

       ntdll.dll!RtlUserThreadStart(long(\*)(void \*) StartAddress, void \* Argument) Line 997      C       Symbols loaded.

### CoreApplicationView

CoreApplicationView has become an important part of the dependency set of many components. It’s not because of the APIs it exposes, but because there is an internal concept of a “StaticLifetimeStore” that is the implementation of many GetForCurrentView APIs. By creating and associating a CoreApplicationView with a Win32 UI thread, a large number of GetForCurrentView APIs start working in a Win32 scenario.

Enabling a CoreApplicationView to exist in this scenario is straightforward. There’s a set of CoreApplicationViews and each is associated with a thread, so in Win32 we can create a new view and associate it with a Win32 UI thread. One difference is that currently CAV is designed to create and run its thread, and in this design it’s attached to an existing thread.

To implement this, in this design CoreApplication will support a new private interface (final name TBD):

interface ICoreImmersiveApplication4 : IInspectable

{

HRESULT CreateNonImmersiveView([in] CoreWindow \*window, [out, retval] CoreApplicationView \*\*view);

}

This interface will create a new CoreApplicationView and associate it with the calling thread and the passed-in CoreWindow. It will be called by the Xaml hosting interface and not exposed publicly.

CreateNonImmersiveView is implemented by creating a CoreApplicationViewAgileContainer and passing a new window type to its RuntimeClassInitialize method. Normally this class creates a thread for the view. For a non-immersive view, the intent is to attach to the calling thread instead, so this code is added:

if (NotImmersiveWindow == type)

{

\_apartmentInitialized = TRUE;

RETURN\_IF\_FAILED(GetThreadInformation());

RETURN\_IF\_FAILED(MakeAndInitialize<CoreApplicationView>(\_viewLocal.ReleaseAndGetAddressOf(), this));

RETURN\_IF\_FAILED(InitializeWithCoreWindow(spWindow));

RETURN\_IF\_FAILED(\_globalizedView.Initialize(static\_cast<ICoreApplicationView\*>(\_viewLocal.Get())));

RETURN\_IF\_FAILED(\_app->AddViewForCurrentThread(this));

UpdateViewState(Initialized);

return S\_OK;

}

This performs some of the initialization that is otherwise done from the created thread, and uses the calling thread instead.

## Input

### Basic Input

In this design, basic input (mouse, touch, keyboard, and pen input) works the same way it does in UWP applications. Xaml subclasses the window procedure of the hwnd it’s targeting, and receives much of its input through that mechanism.

There is an important difference and benefit of this design relative to XamlPresenter. In this design, Xaml is initialized with a CoreWindow, just like when a UWP application is run. There is an ongoing effort to move more of Xaml’s input processing to be based on CoreWindow events instead of USER messages. This Win32 hosting design will be able to take advantage of that work, since there is a CoreWindow being used in Win32 scenarios.

### Mouse-In-Pointer

To summarize a complex issue, Xaml is written to receive mouse/touch input as pointer messages. Most legacy Win32 UI (including WPF) is written to receive mouse input as mouse messages, and touch input as legacy touch messages (if there is any support for touch input at all). The Windows input stack has a per-process setting (Mouse-In-Pointer) that decides whether mouse/touch input will be delivered as pointer messages or as mouse/touch messages. At first glance, this appears to present a conflict if we want to have Xaml UI and legacy Win32 UI coexisting in the same process.

Fortunately, the input stack has a design that solves the apparent conflict. A mechanism called promotion ensures that when an unhandled pointer message is passed to the default window procedure, it is promoted into a legacy message and dispatched again.

To allow modern and legacy UI to coexist, the Xaml hosting code will enable mouse-in-pointer for the process. Input that is directed at Xaml will be processed as pointer messages, and input that is directed at the legacy UI will be processed as promoted legacy messages.

### TSF1 vs. TSF3

The modern Windows Text Input stack has a distributed architecture with two major generations: Text Services Framework 1 (TSF1) and TSF3. The components that are versioned by these generations are IMEs which produce text input and documents which consume text input. For example, a particular scenario may involve a TSF1 IME producing text input which is consumed by a TSF3 document within an application.

There is a complex matrix of support where the dimensions are TSF1 vs. TSF3, IME vs. document, application type (classic or UWP), and SKU. Most combinations are supported, however, there is currently a gap in supporting TSF3 documents in Win32 apps. Until this gap is closed, Xaml in Win32 will need to use TSF1 documents.

New text features that are currently planned, such as Smart Words and Stickers, will only be supported in TSF3 documents. Until the Win32 gap is closed, these features will not be available for Xaml hosted in Win32 environments.

### Direct Manipulation

Need to investigate and add details here. XAML uses DManip and this document needs to detail how that will work in Win32.

### InteractionTracker

Need to investigate and add details here. XAML uses InteractionTracker and this document needs to detail how that will work in Win32.

### More Details

For more details on what works and doesn’t work from Input in this scenario, see this companion document written by Pravin:

<https://microsoft.sharepoint.com/:w:/r/teams/specstore/Developer%20Platform%20Team%20DEP/Redstone/XamlOnWin32STA.docx>

## Output

This design is hwnd based – to a degree. However, the nuances here are important.

There are two ways in which this design makes use of hwnds. The first is that in the short term, hwnds are the basis of interop: Xaml is rendering to an hwnd target, calling USER APIs to obtain the size of the output, and receiving much of its input through a USER window procedure.

This should be seen as a short term implementation detail. This detail should not leak through to app code, and as better interop technology becomes available (such as islands), this hosting mechanism will change to use it.

The second way this design makes use of hwnds is “at the edge”. Regardless of the underyling interop technology, legacy Win32 code needs an hwnd to bridge legacy UI and modern UI. All legacy frameworks are hwnd based to some degree, and all will expect hwnds to be the point of interop. So even in a future release when islands are used as the underlying interop, there will still be an HwndIslandSite that bridges islands into Win32 and presents an hwnd based interface to legacy code.

This “hwnds at the edge” concept is what enables this mechanism to transition between interop technologies without impacting app code. The Xaml hosting interface takes a parent hwnd and returns a child hwnd that the legacy UI framework can operate on. That returned hwnd could be created directly (CreateWindowEx), it could be the private interop hwnd corresponding to a CoreWindow, it could be the hwnd associated with a future HwndIslandSite, or something else entirely. The hosting interface guarantees an hwnd “at the edge” but it does not leak the detail of how that hwnd is created or exactly what type of window it is.

Returning to the concept of hwnd-based interop, it’s important to understand the limitations this will have. These are short-term limitations – since when better interop technology is available, we can make the switch. But until then, customers will need to know what these limitations are from a functional perspective.

The primary problem with hwnd based interop is that the hosted content is separately drawn over the host UI and is not integrated into the host’s rendering logic. This causes many functional gaps, for example:

* Scrolling is difficult. Most frameworks will implement scrolling by clipping a subset of content (the viewport) relative to the content’s overall bounds (the extent). With hwnd based interop, the only knob you can turn is the size of the hosted hwnd, which corresponds to the extent. There’s no good way to clip the content as well. In addition, the host must move the child window in sync with a scroll operation, which can lead to unwanted painting artifacts if these two actions are not well synchronized.
* Scenarios that require tight integration between the rendering of the host and hosted content are not possible. For example, a smooth “continuity animation” involving UI from both the host and hosted content will be difficult to impossible with hwnd interop.
* Transforms and other rendering manipulations set on the host’s content will not have any effect on the hosted UI.
* UI changes are generally not synchronized. This can lead to unwanted visual artifacts during transitions. If you stepped through the transitions frame-by-frame you would see undesired intermediate states where the host UI has updated by the hosted UI has not, etc.

The severity of these problems is highly scenario dependent. Some of them have workarounds – for example, the scrolling problem can often be solved by moving the scrolling to happen inside the hosted content and treating the hosted content as fixed size from the host’s perspective. Still, these can be severe limitations and only by closely examining a scenario can a determination be made.

For a much deeper look the kinds of problems that hwnd based interop has, see [this](https://blogs.msdn.microsoft.com/dwayneneed/2013/02/26/mitigating-airspace-issues-in-wpf-applications/) paper written by Dwayne Need, who investigated this issue several years ago on the WPF team.

## Instancing Model

The Windows 8 app model put a fundamental stake in the ground: that there would be a 1:1 relationship between a window and a thread. This assumption has since propagated into much of the architecture of modern components and as many as three dozen WinRT APIs (see an [appendix](#_Appendix_–_Current) to this document for a listing).

This modern instancing model is much simpler than the classic Win32 model, which allowed a one-to-many relationship between threads and windows. As a result, Win32 UI tends to be more decoupled as a unit – more of the Win32 API set tends to be explicitly about a specified piece of UI, and less about some implicit “current UI”.

Unwinding this assumption and re-introducing the flexibility (and complexity) of the Win32-like multi-instancing model will take many releases. However, we must do that work – as modern Windows evolves, many scenarios require it. For example, Tab Shell needs to produce and service many “units of UI” from a single thread, and as islands are being developed for Tab Shell, this functionality is provided through islands.

This Xaml in Win32 effort will begin this work of adding more instancing flexibility, but we won’t be able to fully solve the problem in one pass. As a result, there will be at least several releases where the support for multi-instancing is not ideal.

To put this in concrete terms: supporting a single instance/unit/chunk of Xaml UI in a legacy Win32 application is straightforward. Supporting multiple instances, interspersed throughout the legacy application, is more challenging. It’s likely that the initial version of this design will be extremely limited support for multiple instances and we will enhance that support over time.

Key aspects of this design (and plan) for multi-instancing support:

* The Xaml hosting interface will expose a Content property. In a single-instance scenario, this property appears to be redundant to the existing Xaml Window.Content property. In a multi-instance scenario, this property replaces the Window.Content property and provides a way to specify the content for each instance.
* Each existing WinRT API that assumes a 1:1 relationship between the calling thread and a single instance of UI will need to be revisited and possibly modified. An [appendix](#_Appendix_–_Current) to this document contains a list of these APIs – there are about three dozen. Some of the options are:
  + Deprecate the API and provide an overload that takes a parameter that scopes it to an instance
  + If the API provides functionality that can be reasonably shared by all instances of UI on a single thread, keep it as-is

The Xaml framework has internal support for multiple instances of UI that share a “xaml core” and a thread. This support is currently being used for several in-development projects inside Windows. We will likely be able to build on top of this support as the first step towards fully supporting multiple instances of content in Win32 hosting.

## Process and Threading Models

The main scenario being targeted by this Xaml in Win32 design is in-proc, same-thread, STA hosting of Xaml UI. Other designs are possible: for example, a customer may choose to host out-of-proc or initialize their UI thread as ASTA instead of STA. However, most legacy Win32 scenarios will involve mixing of UI on the same thread, and that thread will need to be STA for compatibility reasons.

For background on the ASTA threading model, see [this](https://microsoft.sharepoint.com/:p:/r/teams/osg_core_dep/UxP/Shared%20Documents/Framework/ASTA%20background/WinRT%20Threading%2004-04-12%20(DEVX%20PM%20Leads).pptx) slide deck. For an in-depth look at the differences between ASTA and STA, see [this](https://microsoft.sharepoint.com/:w:/r/teams/osg_core_dep/UxP/Shared%20Documents/Framework/ASTA%20background/Differences%20between%20STA%20and%20ASTA.docx) document. In short, the ASTA threading model prevents many kinds of reentrancy, providing safety guarantees to code running in an ASTA that the STA model doesn’t provide. While these safety guarantees present an easier developer model, they are incompatible with much legacy code that was written before the ASTA model was introduced.

To support interop between legacy Win32 code and modern Xaml UI, this design supports running modern Xaml UI and other modern components in a STA. This will require some platform work and investigation to find the places we need to apply changes to. Generally, the changes needed will take the following form:

* **Guidance**: Operations that were previously failed will now be possible, but will not be supported. We will need to provide guidance to Win32 developers about what they can’t do – for example, initiating a COM call from within a MeasureOverride callout will not be permitted since it will create a nested message loop that could cause reentrancy into Xaml’s Tick operation.
* **Reentrancy guards**: In many cases guidance will not be sufficient, and we will need to add explicit checks and failure points to the product. The architectural pattern here is moving the general restrictions of ASTA into more targeted, specific places where known reentrancy problems occur. For example, Xaml’s Tick operation could fail if there is another Tick already in progress.
* **Requeuing and retrying**: In other cases we may be able to support reentrancy by detecting it and requeuing/retrying operations. For instance, if Xaml gets a Tick while another Tick is in progress, we can ignore it and request a new Tick, which hopefully comes in after the stack has unwound.
* **Hardening**: It is possible, but extremely challenging, to write reentrancy-safe code. Typically, this involves protecting all state on the stack and re-verifying any assumptions made before a callout occurs. In some cases hardening product code paths to be reentrancy safe is the right options.
* **Asynchronous callouts**: Finally, asynchronous callouts to app code are usually reentrancy safe by design since they usually occur when other code isn’t on the stack. (This isn’t a full solution since multiple asynchronous callouts can trigger reentrancy). We may be able to switch some synchronous callouts to asynchronous to reduce the chance of reentrancy.

Unfortunately, detecting the places where product changes will be needed is difficult to do through reasoning alone. We will likely need to iterate on this once we have some real-world uses of Xaml-in-Win32 working. It’s likely that an initial version of this hosting technology will be more guidance-based to prevent STA problems, and as we enhance that version we will add more guards, hardening, etc.

## Scaling

The Xaml framework is high-DPI aware – it queries for an appropriate scale factor and scales its output accordingly. A lot of legacy Win32 UI not high-DPI aware and relies on the OS doing bitmap scaling to produce a somewhat usable output.

To bridge this gap, Xaml will need to operate in a mode in which it is not high-DPI aware. Effectively this means a mode in which the framework scales at 100% regardless the state of the system.

The hosting interface will need to provide an API to force 100% scaling. Until this API is provided, hosting Xaml UI will only be usable in Win32 applications that are high-DPI aware, or in scenarios where high-DPI displays are not part of the scenario.

## Resources

Here, resources refers to how assets (images, media, strings, compiled xaml files, etc.) are localized, packaged, and deployed along with an application.

There are three resource models possible for Win32 apps hosting Xaml content:

1. A loose file based strategy. Loose assets are deployed alongside the application binaries and are loaded directly from disk by components like Xaml, MediaPlayer, etc.
2. A legacy or custom strategy. Many UI frameworks and complex applications have their own resourcing model. For example, .NET apps have ResX localization, Office has Office-specific localization, there is a classic Win32 MUI mechanism, and so on.
3. Modern resourcing, using the MRT technology that UWP apps use.

All these strategies are possibilities for Xaml in Win32, and this design should allow for any of them, since different customers will likely want different things.

It’s important to note that these are not mutually exclusive choices. In particular, MRT can be used as a resource index, in which it supplies the logic to resolve a logical resource name to a physical resource (perhaps a file path). Other code can then load that file using any number of mechanisms. This means that the third option can be combined with the first two options with little change to an existing application. (For more on this technique, see [this](https://docs.microsoft.com/en-us/windows/uwp/app-resources/using-mrt-for-converted-desktop-apps-and-games) guidance).

The loose file strategy is easy to support since most modern components, including Xaml and MediaPlayer, already support loading loose files.

The MRT strategy is also easy to support, since MRT supports non-UWP use cases, and modern components like Xaml are already written to use MRT if available. An important aspect of this approach is that the design will likely need a way for the host to pass a MRT resource manager to the hosted content. This could be done via convention (i.e. a well-known resource index file name) or via an explicit API. The API is preferable because it allows for sharing a resource manager instance (benefits performance) and sharing any configuration done by the host (may be necessary for functionality).

Supporting legacy or custom resource strategies will be challenging. It’s common for modern components like Xaml to directly load resources (whether those resources are resolved via MRT or not). This “direct loading” code must either use MRT to load (embedded resources) or must make some assumptions about where and how to load resources (typically that they are loose files). With legacy or custom resource strategies, this “direct loading” must become extensible so that a Win32 application can customize how this is done.

Xaml already supports customization to some degree through mechanisms like CustomResourceLoader and markup extensions, which allow developers to plug custom functionality into extensibility points in xaml markup. This approach will work for customers like Office, but LOB customers will likely want something more turn-key. For example, a WinForms or WPF customer will expect XAML to be able to directly load compiled XAML files using the resource strategy most common to WinForms or WPF apps (ResX files), and this will require work in the platform to support.

As a result, this design will likely imply that platform components will need to do work to integrate more closely with legacy resourcing frameworks such as ResX and MUI. Alternatively, we could investigate an approach of having MRT be the central point for this integration. Components could use existing MRT APIs – such as those used to load PRI-embedded resources – and MRT could implement the logic to service those requests by loading resources from legacy mechanisms.

## Quirks

Although legacy UI frameworks didn’t make a lot of use of quirks, the Xaml framework uses them extensively. We will need a solution for legacy Win32 code to be able to specify a “max tested version” just like a UWP app can, so Xaml can apply the appropriate quirks.

Right now this is an open issue in the design. Possible solutions include either requiring Centennial (which provides an app manifest) or adding a quirk version parameter to the host initialization API.

# Hosting Domain

This section discusses a number of topics that arise when you host UI.

There are three levels of hosting:

1. Single-instanced UI with little to no integration into an outer UI. This is the scenario targeted by XamlPresenter. Many of the topics in this section are not interesting because the functionality all happens within the Xaml framework. For example, all layout is done within the Xaml framework.
2. Single-instanced UI, integrated into an outer UI – for example, a WPF app hosting Xaml UI. This introduces complexity because many of the topics in this section span the UI frameworks. For example, layout must now be negotiated between the hosted UI (Xaml) and the outer UI (WPF).
3. Multi-instanced UI, integrated into an outer UI. This introduces another layer of complexity because there can be multiple instances of hosted UI. This requires both architectural support from the hosted UI framework, as well as APIs that are contextual enough to be scoped to the correct chunk of UI.

## Startup and Shutdown

The host must be able to initialize an instance of hosted UI, create a tree of hosted content associated with that instance, and deinitialize the hosted UI when it’s no longer needed.

As discussed above, this design must eventually support multiple instances of hosted content. Therefore the hosting interface must expose a property that allows the content for that instance to be set and retrieved. The existing API for Xaml content – Window.Content – assumes that there is only a single instance of content per thread, which isn’t sufficient to support multi-instancing.

This design must support cleanly deinitializing any or all instances of hosted content and then later initializing new instances, as this will be a common pattern in many Win32 hosting scenarios.

There are some additional nuances related to initialization and deinitialization:

* Relying on activation to perform initialization is problematic. There are often pre-requisites to initialization that can’t be fit into constructor parameters, and often we discover or add these after initial versions of an API have been released. To allow for this very likely possibility, the hosting interface will expose an explicit Initialize API that must be called after activation but before the hosted UI tree is created and associated to the instance of the hosting interface.
* Relying on destruction to perform deinitialization is problematic. Managed clients do not have an explicit way to force release of a WinRT interface pointer, and often deinitialization is desired at a precise time. To allow for precise control over deinitialization, the hosting interface will implement IClosable (projected as IDisposable in managed apps).

The xaml host interface API related to startup and shutdown looks like this:

interface IXamlHost: IInspectable

{

[propget] HRESULT Content([out, retval] Windows.UI.Xaml.UIElement\*\* value);

[propput] HRESULT Content([in] Windows.UI.Xaml.UIElement\* value);

HRESULT InitializeHost();

}

interface IXamlHostStatics: IInspectable

{

HRESULT CreateForParentHwnd(

[in] INT parentHwnd,

[out] INT\* hostHwnd,

[out, retval] Windows.UI.Xaml.Hosting.XamlHost\*\* result);

}

[static(IXamlHostStatics, Windows.Foundation.UniversalApiContract, 6)]

[static(IXamlHostStatics, NTDDI\_WIN10\_RS3)]

runtimeclass XamlHost

{

[default] interface Windows.UI.Xaml.Hosting.IXamlHost;

[contract(Windows.Foundation.UniversalApiContract, 6)]

[version(NTDDI\_WIN10\_RS3)]

interface Windows.Foundation.IClosable;

}

This contract will require a host to perform the following steps:

1. **Set up** the Win32 environment as required for Xaml Hosting. This is a one-time step that the host must perform. For instance, as discussed above in the Input section, the host will need to enable Mouse-In-Pointer for the process.
2. **Create** an instance of XamlHost by calling a static Create API. The example above shows one such Create API, but others are possible, and all fit into this pattern. This is a medium-weight operation which will create a CoreWindow, CoreApplicationView, etc.
3. **Configure** the XamlHost as appropriate. The API above does not show any examples of this, but if we add configurable properties to the host interface, this is where the host would use them.
4. Call **InitializeHost**. This is a heavyweight operation that will create a xaml “core” object and perform other xaml-specific initialization.
5. Create Xaml **content** by using existing APIs and/or the Xaml parser and set that content as the Content property on XamlHost.
6. At this point, the host application continues to run and the xaml content is presented along with the host content.
7. Once the host is finished with the xaml content, the host should first set the Content property on XamlHost to null and then pump the message loop for a while. This step allows asynchronous Xaml events like Unloaded to be raised, which can be important in allowing application code to perform proper cleanup. Determining how long to pump messages at this stage is not easily defined, and is an area that we should add additional platform support for.
8. The host will then call **Close** on the XamlHost to deinitialize it. This is where the xaml core and other heavyweight objects will be released.
9. Finally, the host will release the XamlHost interface pointer.

Step 7 is optional but from past experience (XamlPresenter) many scenarios will require it. It’s also a fragile step that is hard to get right, so we will need some additional investments in this area to make it easier for hosts to do this correctly.

## Focus

There are four primary focus concerns that arise when hosting UI:

1. Tabbing in (directionally). When keyboard focus is on a control in the host, coordination is needed to move keyboard focus into the hosted content.
2. Tabbing out (directionally). When keyboard focus is in the hosted content and the last tab stop is reached, coordination is needed to move keyboard focus back to the host.
3. Tabbing within hosted content. While this sounds straightforward, many Win32 UI frameworks capture and filter keyboard input before it’s dispatched through normal means, and implement special handling of keys like tab. There is likely coordination needed to plug into this special processing and allow navigational input to be processed by the hosted content at the right times.
4. Obtaining keyboard focus. Regardless of where keyboard focus currently is, certain actions – such as a pointer down on a control – should move keyboard focus. Coordination is needed between host and hosted content to enable this.

The hosting interface does not need to expose any APIs related to focus. Instead, existing XAML APIs can be used by the host interop code to implement focus-related functionality.

The WPF interop element implements the first concern (tabbing in) with the following code. TabIntoCore is an override provided by the WPF HwndHost mechanism. This code uses Xaml’s FocusManager to find a focusable element (note that null is currently being passed as the search scope – once multi-instance is supported, this will change to be the content root for the correct instance). If a focusable element is found, the hosted hwnd is focused and Xaml’s Focus API is invoked on the element.

protected override bool TabIntoCore(TraversalRequest request)

{

Windows.UI.Xaml.DependencyObject elementToFocus;

if (request.FocusNavigationDirection == FocusNavigationDirection.First)

{

elementToFocus = Windows.UI.Xaml.Input.FocusManager.FindFirstFocusableElement(null);

}

else

{

elementToFocus = Windows.UI.Xaml.Input.FocusManager.FindLastFocusableElement(null);

}

if (elementToFocus is Windows.UI.Xaml.Controls.Control)

{

Win32Interop.FocusWindow(Handle);

((Windows.UI.Xaml.Controls.Control)elementToFocus).Focus(Windows.UI.Xaml.FocusState.Keyboard);

return true;

}

return false;

}

Tabbing out and tabbing within is implemented by the following code in the WPF interop element. This code overrides the TranslateAcceleratorCore function provided by WPF’s HwndHost mechanism, which supplies accelerator key input directly from the message loop owned by WPF. When a tab or shift tab is dispatched, Xaml’s FocusManager is used to get the currently focused element. Xaml’s FocusManager is then used to determine if that element is the first (for shift tab) or last (for tab) focusable element in the hosted content. If so, a WPF mechanism (KeyboardInputSite) is used to tell WPF that the hosted content has no more tab stops. If there are more tab stops, Xaml’s FocusManager is used to move focus to the next tab stop. One complication is that Xaml has no LoseFocus API, so when focus is moved, this code must cause the currently focused Xaml element to lose focus, which it does by temporarily making it not a tab stop and disabling it.

protected override bool TranslateAcceleratorCore(ref MSG msg, ModifierKeys modifiers)

{

if (msg.message == WindowMessages.WM\_KEYDOWN && msg.wParam.ToInt32() == VirtualKeyCodes.VK\_TAB)

{

object currentlyFocusedElement = Windows.UI.Xaml.Input.FocusManager.GetFocusedElement();

if (ModifierKeys.Shift == modifiers)

{

Windows.UI.Xaml.DependencyObject firstFocusableElement =

Windows.UI.Xaml.Input.FocusManager.FindFirstFocusableElement(null);

if (currentlyFocusedElement == firstFocusableElement)

{

bool focusDeparted = ((IKeyboardInputSink)this).KeyboardInputSite.OnNoMoreTabStops(

new TraversalRequest(FocusNavigationDirection.Previous));

if (focusDeparted)

{

// need xaml to lose focus

Windows.UI.Xaml.Controls.Control currentlyFocusedControl =

(Windows.UI.Xaml.Controls.Control)currentlyFocusedElement;

currentlyFocusedControl.IsTabStop = false;

currentlyFocusedControl.IsEnabled = false;

currentlyFocusedControl.IsTabStop = true;

currentlyFocusedControl.IsEnabled = true;

}

return focusDeparted;

}

return Windows.UI.Xaml.Input.FocusManager.TryMoveFocus(

Windows.UI.Xaml.Input.FocusNavigationDirection.Previous);

}

else

{

Windows.UI.Xaml.DependencyObject lastFocusableElement =

Windows.UI.Xaml.Input.FocusManager.FindLastFocusableElement(null);

if (currentlyFocusedElement == lastFocusableElement)

{

bool focusDeparted = ((IKeyboardInputSink)this).KeyboardInputSite.OnNoMoreTabStops(  
 new TraversalRequest(FocusNavigationDirection.Next));

if (focusDeparted)

{

// need xaml to lose focus

Windows.UI.Xaml.Controls.Control currentlyFocusedControl =

(Windows.UI.Xaml.Controls.Control)currentlyFocusedElement;

currentlyFocusedControl.IsTabStop = false;

currentlyFocusedControl.IsEnabled = false;

currentlyFocusedControl.IsTabStop = true;

currentlyFocusedControl.IsEnabled = true;

}

return focusDeparted;

}

return Windows.UI.Xaml.Input.FocusManager.TryMoveFocus(

Windows.UI.Xaml.Input.FocusNavigationDirection.Next);

}

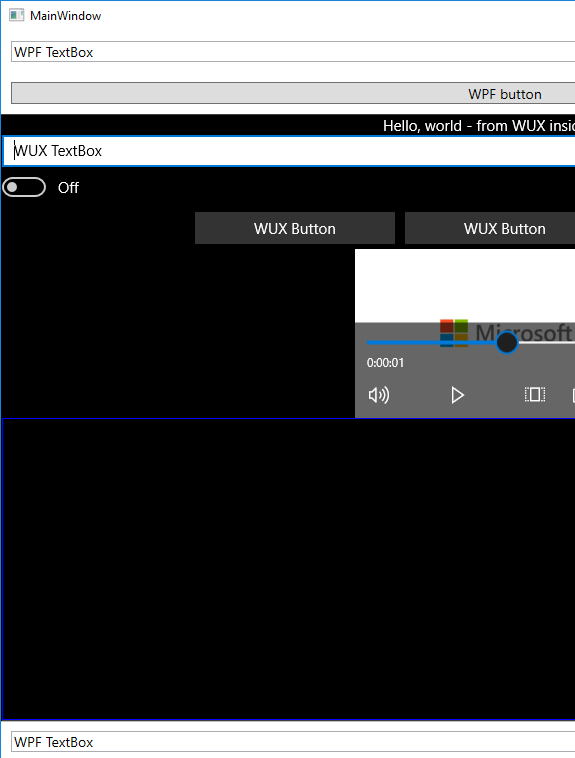
}

return false;

}

The fourth concern (obtaining keyboard focus) is currently handled inside CoreWindow’s implementation. When a pointer down event is processed, CoreWindow will set focus to its hwnd.

Because of all this, focus behaves as expected in a hybrid WPF/Xaml app. For example, in this sample application:



The keyboard (tab/shift tab) can be used to navigate through the application, going from host (WPF), to hosted content (Xaml), and back to the host. You can click or tap on any focusable element and focus moves as expected.

## Hotkeys (Accelerator Keys)

The modern developer stack has little support for hotkeys (also known as accelerator keys). There is an API on CoreDispatcher, but that API is little used, CoreDispatcher is effectively deprecated, and at least one option for supporting CoreDispatcher in Win32 will not able to enable that API.

Until hotkeys are better supported in modern development, there is little to say about issues involving hosting UI. Hotkeys should continue to work as intended in legacy UI, even if Xaml UI is being hosted in the process.

## Mnemonics (Access Keys)

Many Win32 UI frameworks, such as Dialogs and WinForms, provide support for mnemonics (also known as access keys). This feature allows specified keyboard input to move focus to a particular control.

Xaml also supports Access Keys. Some work will be needed to bridge frameworks and allow keyboard input to work correctly with Xaml Access Keys in Win32 apps. This has not yet been investigated.

## Scrolling

### Scrolling Hosted Content From a Host

As discussed above in the Output section, putting hosted content inside a scrolling control in a host application is currently problematic. This approach will likely have significant limitations imposed by the current interop technology (hwnd based). Once better interop technology is available (i.e. Composition Islands), this limitation should be able to be addressed.

For now, the guidance to application developers will be:

1. Avoid putting hosted Xaml content in a scrolling surface managed by the host application
2. If scrollable Xaml content is needed, use Xaml’s ScrollViewer inside the hosted UI if possible, as this works as expected
3. If hosted Xaml content must be used inside a host scrolling surface, extra work will be needed to size the hosted content to always fit within the viewport of the host’s scrolling surface

### Scroll Chaining

Scroll chaining is a feature that applies a single scrolling “gesture” (mouse wheel or pan) to multiple scrolling surfaces. The scrolling is initially implemented by an inner scrolling surface that the gesture is directed at. When that surface reaches the limit of its scrolling ability, if the gesture continues, an outer (containing) scrolling surface picks up and implements the next portion of the scroll. In theory this can continue through any amount of nested scrolling surfaces.

At this time there is no support for scroll chaining from hosted UI into the host. This should be possible, but investigation and likely platform work will be needed.

## Accessibility

Fully accessible hosted UI is a must-have requirement for many customers who will use Xaml in Win32. Xaml UI is accessible, and the work to bridge this support into a hosted scenario breaks down into the following pieces:

1. For UIA based hosts, bridging Xaml’s UIA accessibility tree into the host’s tree.
2. For MSAA based hosts, adapting Xaml’s UIA accessibility tree to MSAA and bridging that into the host’s tree.

Both of these work items need further exploration to be fully understood. The first work (UIA based scenarios) is easier. An initial version of this has been prototyped, which involves responding to WM\_GETOBJECT when sent to the hwnd of the hosted xaml content. This works, in the sense that Xaml’s accessibility tree appears to accessibility clients. The limitation is that the Xaml tree is not in the correct place within the host’s tree, so while the content is technically accessible, it’s not logically correct. Further work and investigation is needed to properly bridge the content so it appears in the correct logical place in the accessibility tree.

The second work item has not been investigated at all. UIA itself contains some adapter code to go between MSAA and UIA, so it’s possible that there is little or no work on the Xaml side.

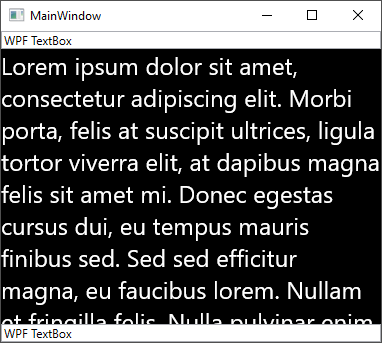
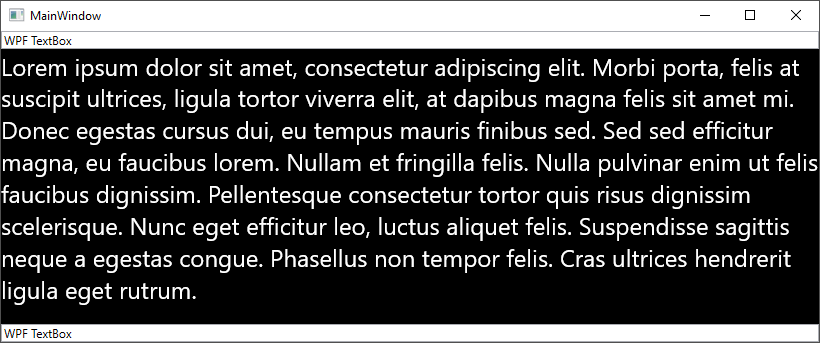
## Layout

There are four primary layout concerns that arise when hosting UI:

1. Desired size measurement. The outer UI queries the inner UI for what size it wants to be (possibly with some starting constraints).
2. Responding to available size changes. The outer UI decides what size the inner UI must be, and the inner UI performs its own layout to that size.
3. Invalidating layout when content changes. When the inner UI has a layout-related content change, it must communicate to the outer UI that a new layout is requested.
4. Synchronizing layout

The hosting interface does not need to expose any APIs related to layout. Instead, existing XAML element APIs on the content root element can be used to implement layout at the hosting layer.

For example, here are screenshots showing a WPF app hosting XAML content. The XAML content is fully integrated into the WPF layout algorithm. As the WPF window is resized, the XAML TextBlock wraps to accommodate the available size it’s given.



The way this is implemented in the WPF Xaml host element is as follows.

The first concern (desired size measurement) is implemented by forwarding Measure on to the hosted UI:

protected override Size MeasureOverride(Size availableSize)

{

Size desiredSize = new Size(0, 0);

Windows.UI.Xaml.UIElement rootXamlElement = Windows.UI.Xaml.Window.Current.Content;

rootXamlElement.Measure(new Windows.Foundation.Size(availableSize.Width, availableSize.Height));

desiredSize.Width = Math.Min(rootXamlElement.DesiredSize.Width, availableSize.Width);

desiredSize.Height = Math.Min(rootXamlElement.DesiredSize.Height, desiredSize.Height);

return desiredSize;

}

The second concern (available size changes) is implemented by having Xaml query for the size of the hwnd it’s presenting to during each tick, and pass that size to Xaml’s layout code. If the size has changed, a full layout pass is run.

The third concern (invaliding layout on content changes) is implemented by listening to the LayoutUpdated event on the root of the Xaml content, and invalidating WPF layout:

private void XamlContentLayoutUpdated(object sender, object e)

{

InvalidateMeasure();

}

The fourth concern (layout synchronization) is an open issue. Currently the design makes no attempt to synchronize layout. For one thing, layout synchronization isn’t very useful if there isn’t synchronization of the underlying rendering technologies. This lack of synchronization means that it’s possible for one frame of the hybrid app to show one framework’s content laid out to a new size, and the other framework still showing content laid out to the old size.

# Developer Experience

## Velocity Staging

To get early feedback from customers, there is a plan to stage this feature. The plan is to use Velocity to enable selected customers (both first- and third-party) to enable the feature, prototype and experiment, and give us feedback which we can iterate on.

The staging plan is:

1. The XamlHost API is added to the product. Its Velocity stage is DisabledByDefault (with a branch override of AlwaysEnabled for the active development branches).
2. The code RIs up, and first-party customers can selectively enable the feature by using the internal staging tool.
3. The code RIs up and eventually lands in the flighting branch (rs\_prerelease). A OS and SDK flight goes out with the API. It’s marked as experimental in the SDK and the runtime check fails (by default).
4. A third-party developer can opt-in a device through the Windows Device Portal. They can then experiment/prototype using the flighted SDK and build, and the API works.
5. Several more flights out of rs\_prerelease happen, and opted-in developers can continue to experiment/prototype on those flights.
6. Nearer to RTM, we create rs4\_release. A branch override for rs4\_release sets the Velocity stage of the API to AlwaysDisabled for the release branch.
7. First parties working out of rs4\_release will no longer be able to experiment with/prototype on the feature (because the feature is not present in rs4\_release builds). First parties who want to continue working with the feature will need to use builds out of rsmaster, etc.
8. The next flights come out of rs4\_release instead of rs\_prerelease. Third-party developers will no longer be able to experiment/prototype on the latest flight. (This is the “blackout” period). They will need to either stay on an older flight (problematic) or wait until the blackout is over.
9. Shortly after RTM, we start flighting out of rs\_prerelease again. Third-party developers can resume experimenting/prototyping.

## WinRT projections

The desktop CLR does include support for WinRT, and we have a demo application that demonstrates a WPF app (running on the full desktop CLR) hosting Xaml UI.

This support requires adding references to some WinRT-specific assemblies (System.Runtime.WindowsRuntime.dll and System.Runtime.WindowsRuntime.UI.Xaml.dll). With this support, all of the “xamlisms” (such as Windows.UI.Xaml.Data.INotifyPropertyChanged correctly projecting to System.ComponentModel.INotifyPropertyChanged) work as expected.

However, there are some subtle differences between the full desktop CLR and the version of the CLR that UWP apps use (CoreCLR), because the CLR codebases have diverged. These differences are not well understood at this stage.

## Tooling support

Most tooling support in VS can be provided by the existing UWP Class Library project type. This has support for the XAML designer and the XAML compiler.

Currently it’s not possible to directly reference a UWP Class Library from a WPF project. We will need to design that support with DevDiv. Until then, it is possible to manually add support for a solution layout where a WPF project uses XAML artifact from a UWP Class Library project. Details on how to set this up are in an [appendix](#_Appendix_–_Configuring) in this document.

## UI domain types

A UI framework introduces its own domain of types/values – for example, colors, sizes, etc. In the context of a hybrid app that mixes UI frameworks and app code that targets them, there are two issues:

1. Collisions. These are usually not true collisions due to namespaces or other scoping mechanisms, but logical collisions in the sense they produce developer confusion. For example, in a hybrid WPF/Xaml app, the developer will have access to two different Grid classes. They will be in separate namespaces but there may be confusion over which context each can be used in. This confusion could appear in the code editor, the xaml editor, etc.
2. Conversions. The developer will often need to convert values between the domains of each UI framework. For example, the struct that represents an ARGB color will be a different struct in WinForms and Xaml. The developer will need to write conversion code to move values between the two frameworks.

## Framework-to-framework interop

When developers are mixing UI frameworks, they may have some expectation of coordination of UI framework features – especially when the frameworks have similar features.

Some of this interop is “table stakes” and has already been discussed (input, output, layout, accessibility, etc.). In general any interop that involves something outside of the UI framework itself will be expected to be coordinated across the frameworks. However, some features are entirely within-framework features but may still be expected to interop.

For example, in a hybrid WPF/Xaml application, both UI frameworks support a very similar styling model. A developer might reasonably expect to be able to define a style in the WPF content and have that style apply to hosted Xaml content.

This design is not yet proposing any solution to this level of interop, but noting that the functional scenario exists and that if taken forward, customer feedback will be needed to help decide on a level of investment.

# Features, Open Issues, and Limitations

## Open Issues

The following issues require more design work before we’re ready to ship a functioning solution.

|  |  |  |
| --- | --- | --- |
| **Issue** | **Status** | **More Details** |
| Impact of STA support not fully understood | Supporting modern components to be used in a classic STA will create work in those components. We will need to allocate time to fix a tail of issues in affected components. | [Process and Threading Models](#_Process_and_Threading) |
| Quirks | Modern components – especially Xaml – make heavy use of quirks for app compat. This design needs to expose a way for legacy applications to describe their max tested OS version to allow the existing quirks mechanism to provide app compat for this scenario. | [Quirks](#_Quirks) |
| Supporting Xaml Access Keys | The work to support Xaml Access Keys (mnemonics) in a hosted UI scenario is not yet understood. | [Mnemonics (Access Keys)](#_Mnemonics_(Access_Keys)) |
| Full Accessibility Support | Currently the hosted content is accessible, but isn’t in the right logical location in the accessibility tree. More work is needed to understand how to fully bridge the Accessibility tree of the hosted content into a host’s tree. | [Accessibility](#_Accessibility) |

## Limitations

The following are the currently understood limitations of this design.

|  |  |  |  |
| --- | --- | --- | --- |
| **Issue** | **Implications** | **Status** | **More Details** |
| TSF3 documents not supported | As new text features are developed (for example Smart Words), they will only be available for TSF3 documents. | The input team will need to schedule work to support TSF3 documents in a Xaml-in-Win32 scenario. | [TSF1 vs. TSF3](#_TSF1_vs._TSF3) |
| Output interop is hwnd based | “Airspace” problems, such as inability to blend surfaces, properly clip hosted content, or scroll hosted content (from the outside). | When Composition Islands are available in Win32, this mechanism will switch to use Islands for interop, eliminating many airspace problems. | [Output](#_Output) |
| Only a single instance of hosted UI is supported | Developers will not be able to place multiple “islands” of hosted XAML content onto the same host thread. | Work needs to be done across a number of teams to support multiple instances of hosted content. | [Instancing Model](#_Instancing_Model) |
| Only high-DPI aware Win32 applications are supported | Legacy Win32 applications that are not high-DPI aware (that rely on system bitmap scaling) will not interop well with hosted xaml content in high-DPI scenarios. | The xaml hosting interface will need to expose a property to force 100% xaml scaling for such legacy applications. | [Scaling](#_Scaling) |
| Limited support for legacy resource models | Win32 applications that use legacy resource models (ResX, MUI, etc.) will have a less-than-smooth experience in integrating Xaml content. | Work will be required from Xaml or MRT to better integrate with legacy resource technologies. | [Resources](#_Resources) |
| Deinitialization of hosted Xaml content is fragile | In scenarios that involve transient Xaml UI, it could be difficult to avoid leaks and deterministically reduce the overhead of Xaml. | The current state is sufficient for prototyping, experimentation, and simple scenarios. More work will be required to provide a cleaner, more predictable shutdown/deinitialization experience, especially for complex UI. | [Startup and Shutdown](#_Startup_and_Shutdown) |
| Developer experience choppy | Things that developers expect to “just work” are not available – such as having a built-in project type for this kind of hybrid app in Visual Studio. | While it is possible to construct a fully functional developer experience, including support for the modern toolchain, it is rough around the edges. We will need to work with DevDiv to polish the developer experience for this scenario. | [Developer Experience](#_Developer_Experience_1) |
| CoreDispatcher support has perf overhead |  | Current design has implemented CoreDispatcher on top of DispatcherQueue in Win32 scnearios. This may suffice for a v1, but the long-term plan that will give better performance is to skip a layer and have CoreDispatcher go directly to CoreMessaging. | [CoreDispatcher and DispatcherQueue](#_CoreDispatcher_and_DispatcherQueue) |

# Appendix - Samples

Several sample apps are available that show Xaml UI hosted in Win32 scenarios.

The samples depend on some OS changes, so you need to run them on a build out of the rs\_onecore\_dep\_uxp\_dev4 branch. When running on a VM (not physical hardware) you’ll need to follow these instructions to enable Composition effects (which are disabled by default on VMs). From an admin command prompt, run these commands:

reg add HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\Dwm /v ForceEffectMode /t REG\_DWORD /d 2

reg add HKEY\_LOCAL\_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Windows\Dwm /v ForceEffectMode /t REG\_DWORD /d 2

Then reboot.

|  |  |
| --- | --- |
| **Demo #1: Standalone Win32 App**  This is a simple win32 app built in C++. It hosts XAML UI and demonstrates a few modern features working in Win32 like Reveal, MediaPlayer, and Inking.   * To install: copy all the files from [\\redmond\win\users\benpryor\Xaml-Win32\XamlOnWin32](file:///\\redmond\win\users\benpryor\Xaml-Win32\XamlOnWin32) to c:\temp (yes, this sample app has a hardcoded file path, so you need to run it from c:\temp). * To run: run c:\temp\XamlOnWin32.exe. This sample has no window chrome, so to close the window you need to use the taskbar. * The source for this demo is in the OS repo at %sdxroot%\windows\dxaml\xcp\scenarios\DirectUI\ABI\XamlOnWin32. | cid:image003.jpg@01D34F0A.779F1D20 |
| **Demo #2: WPF Interop Sample**  This demo is a WPF app that has both WPF UI and XAML UI on the same window. The sample shows modern features working in a WPF app via XAML (again Reveal, MediaPlayer, and Inking). It shows interop between the two frameworks (tabbing, layout, early support for accessibility, etc.). It’s also a good way to see a preview of what the dev experience could be (for more on that see the bottom of this mail).   * To install: Copy the WPFInteropSample folder from [\\redmond\win\Users\BENPRYOR\Xaml-Win32](file:///\\redmond\win\Users\BENPRYOR\Xaml-Win32) to anywhere on your test machine. * To run: run WPFInteropSample.exe from the folder you copied. Note that you can tab in/out of the XAML UI, resize the window to update layout, etc. * The source for this demo is available: [\\redmond\win\Users\BENPRYOR\Xaml-Win32\WPFInteropSample\_2017\_10\_27\_8\_45\_13.zip](file:///\\redmond\win\Users\BENPRYOR\Xaml-Win32\WPFInteropSample_2017_10_27_8_45_13.zip) | cid:image006.jpg@01D34F0A.779F1D20 |
| **Demo #3: SuperJupiter in WPF**  SuperJupiter is an internal sample app that the XAML team uses as a scenario testbed. It provides broad adhoc coverage of a number of XAML features. We have ported this sample into a Win32 environment by hosting it within a WPF app.   * To install: Copy the WPFSuperJupiter folder from [\\redmond\win\Users\BENPRYOR\Xaml-Win32](file:///\\redmond\win\Users\BENPRYOR\Xaml-Win32) to anywhere on your test machine. * To run: run WPFSuperJupiter.exe from the folder you copied. You’ll probably want to maximize the window. Each button on the home screen takes you to a scenario. * The source for this demo is available: [\\redmond\win\Users\BENPRYOR\Xaml-Win32\WPFSuperJupiter\_2017\_10\_26\_17\_32\_05.zip](file:///\\redmond\win\Users\BENPRYOR\Xaml-Win32\WPFSuperJupiter_2017_10_26_17_32_05.zip) | cid:image008.jpg@01D34F0A.779F1D20 |

# Appendix – Current View APIs in WinRT

As mentioned above, a large number of WinRT APIs assume that there’s a 1:1 mapping between a thread, a view/window, and a unit of UI. Each of these APIs will need to be considered in an effort to remove this assumption and provide more instancing flexibility. See the Instancing Model section of this document for more details.

Those APIs are:

|  |  |
| --- | --- |
| **DeclaringType** | **MemberName** |
| Windows.UI.Core.Preview.SystemNavigationManagerPreview | GetForCurrentView |
| Windows.UI.Core.SystemNavigationManager | GetForCurrentView |
| Windows.UI.Text.Core.CoreTextServicesManager | GetForCurrentView |
| Windows.UI.ViewManagement.Core.CoreInputView | GetForCurrentView |
| Windows.UI.ApplicationSettings.AccountsSettingsPane | GetForCurrentView |
| Windows.UI.ViewManagement.ApplicationView | GetForCurrentView |
| Windows.Graphics.Display.DisplayInformation | GetForCurrentView |
| Windows.ApplicationModel.DataTransfer.DragDrop.Core.CoreDragDropManager | GetForCurrentView |
| Windows.UI.Input.EdgeGesture | GetForCurrentView |
| Windows.UI.ViewManagement.InputPane | GetForCurrentView |
| Windows.ApplicationModel.DataTransfer.DataTransferManager | GetForCurrentView |
| Windows.UI.ViewManagement.UIViewSettings | GetForCurrentView |
| Windows.ApplicationModel.Core.CoreApplication | GetCurrentView |
| Windows.ApplicationModel.LockScreen.LockApplicationHost | GetForCurrentView |
| Windows.ApplicationModel.Resources.ResourceLoader | GetForCurrentView |
| Windows.ApplicationModel.Resources.ResourceLoader | GetForCurrentView |
| Windows.ApplicationModel.Resources.Core.ResourceContext | GetForCurrentView |
| Windows.Devices.Input.MouseDevice | GetForCurrentView |
| Windows.Graphics.Display.BrightnessOverride | GetForCurrentView |
| Windows.Graphics.Display.Core.HdmiDisplayInformation | GetForCurrentView |
| Windows.Graphics.Printing.PrintManager | GetForCurrentView |
| Windows.Media.Capture.AppCapture | GetForCurrentView |
| Windows.Media.PlayTo.PlayToManager | GetForCurrentView |
| Windows.Media.SystemMediaTransportControls | GetForCurrentView |
| Windows.ApplicationModel.Preview.Holographic.HolographicApplicationPreview | IsCurrentViewPresentedOnHolographicDisplay |
| Windows.UI.Composition.CompositionCapabilities | GetForCurrentView |
| Windows.UI.Core.CoreWindow | GetForCurrentThread |
| Windows.UI.Core.CoreWindowResizeManager | GetForCurrentView |
| Windows.UI.Input.KeyboardDeliveryInterceptor | GetForCurrentView |
| Windows.UI.Input.PointerVisualizationSettings | GetForCurrentView |
| Windows.UI.Input.RadialController | CreateForCurrentView |
| Windows.UI.Input.RadialControllerConfiguration | GetForCurrentView |
| Windows.UI.Input.Spatial.SpatialInteractionManager | GetForCurrentView |
| Windows.UI.Xaml.Window | Current |
| Windows.UI.Xaml.Media.Animation.ConnectedAnimationService | GetForCurrentView |

# Appendix – Configuring a WPF/Xaml Interop Solution in Visual Studio

This section describes how to add XAML hosting support to a WPF app that builds in Visual Studio.

The way this is set up is a solution that contains a WPF project and a UWP class library. The XAML content is authored and built in the UWP class library. This allows you to use the full modern XAML designer experience, as well as the full XAML toolchain (Xaml Compiler, XBF generator, etc.). The output is then automatically integrated into the WPF project, sources are recompiled against desktop .NET, and the binaries (XBF files) are deployed along with the WPF app.

What this means is that you can edit a XAML control in the Xaml Designer, hit F5, and see your changes automatically built and deployed into a WPF app. You can also share the UWP class library project between a UWP app and WPF app, allowing you to write modern XAML UI once and reuse it in both environments.

## Add WinRT support

1. Right click on your WPF project and select Add, then Reference. Click Browse, change the filter to All Files, and browse to:

C:\Program Files (x86)\Windows Kits\10\UnionMetadata\10.0.16299.0

(you may need to use a different version number depending on what SDKs you have installed)

1. Select the Windows.winmd file in that location and click OK
2. Right click on your WPF project and select Add, then Reference. Click Browse, and browse to:

C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETCore\v4.5

1. Select the following files:

System.Runtime.WindowsRuntime.dll

System.Runtime.WindowsRuntime.UI.Xaml.dll

1. click OK

 Add XAML Interop Code

1. Copy the Interop folder from here into your WPF project folder:

[\\redmond\win\users\benpryor\Xaml-Win32](file:///\\redmond\win\users\benpryor\Xaml-Win32)

1. Right click on the project, select Add, then New Folder. Name the new folder Interop.
2. Right click on the Interop folder, select Add, then Existing Item. Select all the files in the Interop folder you copied and click Add.

## Adding a XAML Class Library Project

To author the XAML UI, you need to add a XAML class library project to the solution containing the WPF project. This adds support for the XAML designer and XAML toolchain.

1. Add a "Class Library (Universal Windows)" project to the solution.
   1. Name it what you want - the name used in these instructions will be XamlClassLibrary, but you can name it anything.
   2. The Target/Minimum version doesn't matter much (yet) - for now, pick the latest version in the dropdown, which corresponds to the latest SDK you have installed.
2. Right-click on the Solution and choose Project Dependencies. Set up a dependency from the WPF project on the class library project so the class library project builds first.
   1. Double-check in the Build Order tab of the Project Dependencies dialog that you see the class library project listed first, then the WPF project.
3. Create a folder in the WPF project with the same name as the class library project (for example, "XamlClassLibrary")
4. Right-click on the class library project, Properties. Go to Build Events. Add the following lines as Post-build event command lines, changing the project names to match your solution:

copy "$(ProjectDir)\*.xaml.cs" "$(ProjectDir)..\WPFInteropSample\XamlClassLibrary"

copy "$(ProjectDir)obj\$(Configuration)\\*.g\*.cs" "$(ProjectDir)..\WPFInteropSample\XamlClassLibrary"

copy "$(ProjectDir)obj\$(Configuration)\\*.xbf" "$(ProjectDir)..\WPFInteropSample\XamlClassLibrary"

1. Add a xaml UserControl to the class library (right-click on the project, Add, New Item, UserControl. Author the UserControl as you normally would.
2. Rebuild the solution - everything should build clean at this point.
3. In the WPF project, right-click on the folder that has the name of the class library, Add, Existing Item, and change the filter to All Files. Select all the files that were copied over from the class library post-build step. For a given UserControl, there will be 4 files, such as:

TestUserControl.g.cs

TestUserControl.g.i.cs

TestUserControl.xaml.cs

TestUserControl.xbf

Also, make sure the XamlTypeInfo.g.cs is added.

1. Right-click on the .xbf file in the WPF project, select Properties. In the Properties pane, set Copy to Output Directory to Copy if newer.
2. Rebuild the solution again. This time the XAML code from the class library project will be compiled into the WPF project. Again everything should build cleanly.

Note: each time you add new items to the class library project, you'll to repeat the step of adding the copied items to the WPF project (steps 7 and 8).

## Adding the JupiterHost into the WPF element tree

1. In the WPF project, edit MainWindow.xaml.
2. On the root element (Window), add a XAML namespace definition for the Interop namespace: xmlns:interop="clr-namespace:Interop"
3. In the element tree, add an instance of the interop:JupiterHost element, and set the ControlType property to the fully qualified class name of the root element. For example:

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

<RowDefinition Height="Auto" />

</Grid.RowDefinitions>

<TextBox Text="WPF TextBox" Grid.Row="0" />

<interop:JupiterHost ControlType="SuperJupiterViews.TestUserControl" Grid.Row="1" />

<TextBox Text="WPF TextBox" Grid.Row="2" />

</Grid>

## Building

By default, the WPF binaries will target 32 bit. If you want them to target 64 bit (perhaps you're using private binaries that are 64 bit), open the project properties, go to the Build tab, and change the Platform target (not Platform) to x64.

## Deploying

To deploy, you'll need to set up a workflow to manually copy the binaries over to a test machine each time you rebuild. As far as I can tell, Visual Studio doesn't support automatically deploying built binaries for WPF projects. Manual steps:

1. On the source machine, ensure that the directory where the project is built is shared out
2. On the target machine, create a directory that's the same path as on the source machine: md "c:\projects\WPFInteropSample\WPFInteropSample\bin\Debug"
3. On the target machine, switch to that directory: cd "c:\projects\WPFInteropSample\WPFInteropSample\bin\Debug"
4. Copy the built binaries from the share: xcopy /EY [\\benpryor-dev\projects\WPFInteropSample\WPFInteropSample\bin\Debug](file:///\\benpryor-dev\projects\WPFInteropSample\WPFInteropSample\bin\Debug)

## Debugging

To debug remotely, VS requires that the binaries on the target machine are in the same location as the built binaries on the debugging machine. That's why in the previous step you deployed the binaries at the same location on the target machine.

To debug:

1. Ensure msvsmon is running on the target machine
2. Go to the WPF project properties in VS, Debug tab
   1. Check Use remote machine, and enter the target machine name.
   2. You may also want to check Enable native code debugging if needed for your scenario.

You should now be able to F5 in VS and remotely debug the WPF app.