Intel Unite®

Unified Communication

Reference Design for

HubView Factories

Guide

August 2019

Revision 1.0

Intel Confidential

Revision History

|  |  |  |
| --- | --- | --- |
| Revision | Description | Date |
| 1.0 | Initial release | August 2019 |
|  |  |  |
|  |  |  |
|  |  |  |

**Legal Disclaimers and Copyrights**

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel, the Intel logo, Intel® Core™ vPro™ processor family are trademarks of Intel Corporation in the United States and/or other countries.

\*Other names and brands may be claimed as the property of others

Contents

[1 Terminology 4](#_Toc17386572)

[2 UC Requirements 5](#_Toc17386573)

[2.1 Create DeepCopy class 5](#_Toc17386574)

[2.2 Create IHubView 6](#_Toc17386575)

[2.3 Implement IHubView on ViewBase 7](#_Toc17386576)

[2.4 Implement ViewBase on HubViewBase 10](#_Toc17386577)

[2.5 Create HubViewEventArgs 13](#_Toc17386578)

[2.6 Create QuickAccessIcon 14](#_Toc17386579)

[2.7 Create QuickAccessIconViewModel 16](#_Toc17386574)

[2.8 Create HubView 17](#_Toc17386575)

[2.9 Create HubViewFactory 18](#_Toc17386576)

[2.10 Create QuickAccesIconFactory 19](#_Toc17386577)

Figures

[Figure 1 Implement Abstract Class 20](#_Toc17386580)

Code

[Code 1 Create DeepCopy 5](#_Toc17386581)

[Code 2 Create IHubView 6](#_Toc17386582)

[Code 3 Create ViewBase 7](#_Toc17386583)

[Code 4 Create HubViewBase 10](#_Toc17386584)

[Code 5 Create HubViewEventArgs 13](#_Toc17386585)

[Code 6 Create QuickAccessIcon 14](#_Toc17386585)

[Code 7 Create QuickAccessIconViewModel 15](#_Toc17386585)

[Code 8 Create HubView 17](#_Toc17386585)

[Code 9 Create Factory 18](#_Toc17386585)

[Code 10 Create QuickAccessIconFactory 19](#_Toc17386585)

# Terminology

Table 1. Terminology

| Term | Description |
| --- | --- |
| UI | User Interface |
| WPF | Windows\* Presentation Foundation |
| Hub | Intel® Core™ vPro™ processor family-based PC running the Intel Unite Hub application |
| Client | Device used to connect to the hub |
| Server | Device that manages access and configuration of Hub and Client Devices |
| UC | Unified Communications |
| UC User | Unified Communication User |
| Factory | The Factory method is a creational design pattern which provides an interface for creating objects without specifying their concrete classes. |

## Implementation

Using factory design pattern simplifies creating new views for Intel Unite to display.

This UC Reference Design targets the following feature:

* As a UC Plugin, I want to easily create views, so that maintaining and developing HubViews is simplified.

### Create DeepCopy Class

Due to how Intel Unite handles memory between the Unite Core and the plug-in module, the keyword “new“ should not be used during the AllocateCallBack method. The DeepCopy method copies memory between processes ensuring that the transparent proxy puts the values in the correct location.

The DisplayView Class is needed for updating and maintain the view on the Intel Unite Hub. The Class is generated updated during the Callback from the DisplayManager method calls by Intel Unite Core process.

Code 1 Create DeepCopy

|  |
| --- |
| public static class DeepCopy  {  public static void CopyPhysicalDisplay(PhysicalDisplay toObj, PhysicalDisplay fromObj)  {  toObj.FriendlyName = fromObj.FriendlyName;  toObj.Id = fromObj.Id;  toObj.IsPrimary = fromObj.IsPrimary;  toObj.IsVirtualExtendedDisplay = fromObj.IsVirtualExtendedDisplay;  toObj.Name = fromObj.Name;  toObj.NumberOfPresentations = fromObj.NumberOfPresentations;  toObj.Size = fromObj.Size;  }  public static void CopyHubAllocationInfo(HubAllocationInfo toObj, HubAllocationInfo fromObj)  {  toObj.FriendlyName = fromObj.FriendlyName;  toObj.Id = fromObj.Id;  toObj.HubInfo = fromObj.HubInfo;  toObj.ModuleOwnerId = fromObj.ModuleOwnerId;  CopyPhysicalDisplay(toObj.PhysicalDisplay, fromObj.PhysicalDisplay);  toObj.ReuseControl = fromObj.ReuseControl;  toObj.Tag = fromObj.Tag;  toObj.UserInfo = fromObj.UserInfo;  toObj.ViewType = fromObj.ViewType;  }  public static void CopyDisplayView(DisplayView toObj, DisplayView fromObj)  {  toObj.AllowRemoteAnnotations = fromObj.AllowRemoteAnnotations;  toObj.BackgroundColor = fromObj.BackgroundColor;  toObj.GetScreenShotDelegate = fromObj.GetScreenShotDelegate;  toObj.Id = fromObj.Id;  CopyHubAllocationInfo(toObj.HubAllocationInfo, fromObj.HubAllocationInfo);  }  } |

### Create IHubView

An interface to all the views which implements methods used to manage the UI on the Hub.

* Allocated is used to place the view on the hub. During this call the DisplayView values are set.
* Show is only necessary for the HubDisplayViewType.QuickAccessAppView as it is only visible after show is called.
* Deallocate will not work on HubDisplayViewType.QuickAccessAppView and HubDisplayViewType.QuickAccessAppIconView

Code 2 Create IHubView

|  |
| --- |
| public interface IHubView  {  Guid ViewGuid { get; }  DisplayView DisplayView { get; set; }  HubAllocationInfo HubAllocationInfo { get; }  void Allocate();  bool Show();  void DeAllocate();  bool IsAllocated { get; }  } |

### Implement IHubView on ViewBase

The base class of a hubview which implements the IHubView Methods. These methods enact the display of views on the Intel Unite® Hub.

Common Errors:

* All classes that are passed between processes need to be decorated with the attribute Serializable.
* If a class is not serializable, then a private field needs to be created and decorated with the attribute “field: NonSerialized”. Note that Views, Dispatchers, and most events are not serializable.
* Accessing views must be done in a delegate passed to the UI Dispatcher.
* All the callback methods and methods called in them need to be set to public

In order to call allocate the following variables must be present

* runtimeContext – contains the Display manager
* display – the display which the view will be allocated to
* currentUiDispatcher – need so that only one UI thread runs at a time
* createContract – used to marshal the view

The ViewGuid is used by to identify this specific view on the plugin side.

The DisplayView value is set by the unite core when the view is allocated

The ViewMutex is used to allow testing of the views when run in parallel. Multiple views should never be allocated at once when run normally.

The HubView, HubViewModel, and HubDisplayViewType will all be implemented by the class which inherits this one.

Code 3 Create ViewBase

|  |
| --- |
| [Serializable]  public class ViewBase : IHubView  {  #region Fields  private readonly IHubModuleRuntimeContext \_runtimeContext;  [field: NonSerialized]  protected Dispatcher \_dispatcher { get; }  [field: NonSerialized]  Private Func<FrameworkElement, MarshalNativeHandleContract> \_createContext;  #endregion  #region Properties  protected Dispatcher CurrentUiDispatcher { get => \_dispatcher; set => \_dispatcher = value; }  [field: NonSerialized]  protected static readonly Mutex ViewMutex = new Mutex();  public Guid ViewGuid { get; set; }  public Func<FrameworkElement, MarshalNativeHandleContract> CreateContract { get => \_createContext; set => \_createContext = value; }  private bool IsAllocated => this.DisplayView.Id != new Guid();  private DisplayView DisplayView { get; set; } = new DisplayView { HubAllocationInfo = new HubAllocationInfo { PhysicalDisplay = new PhysicalDisplay() } };  public HubAllocationInfo HubAllocationInfo { get; set; }  protected virtual HubDisplayViewType HubDisplayViewType { get; set; }  protected IHubModuleRuntimeContext RuntimeContext { get; set; }  #endregion  public ViewBase(IHubModuleRuntimeContext runtimeContext, PhysicalDisplay display, Dispatcher currentUiDispatcher, Func<FrameworkElement, MarshalNativeHandleContract> createContract)  {  lock (this)  {  ViewGuid = Guid.NewGuid();  \_runtimeContext = runtimeContext;  CurrentUiDispatcher = currentUiDispatcher;  HubAllocationInfo = GetNewHubAllocationInfo(display);  CreateContract = createContract;  }  }  public abstract void Allocate();  public abstract void DeAllocate();  public abstract void Show();  public abstract void AllocatedCallBack(HubAllocationResult hubAllocationResult);  public abstract void AllocatedFail();  public abstract void AllocatedSuccess(DisplayView allocatedView);  public abstract void DeallocateCallBack(HubAllocationResult hubAllocationResult);  private HubAllocationInfo GetNewHubAllocationInfo(PhysicalDisplay display)  {  return new HubAllocationInfo  {  FriendlyName = GetType().Name,  ModuleOwnerId = ModuleConstants.ModuleInfo.Id,  PhysicalDisplay = display,  ViewType = HubDisplayViewType,  Tag = ViewGuid,  };  }  } |

### Implement ViewBase on HubViewBase

The base class of a hubview which implements the ViewBase abstract class’ Methods. These methods enact the display of views on the Intel Unite® Hub.

Common Errors:

* All classes that are passed between processes need to be decorated with the attribute Serializable.
* If a class is not serializable, then a private field needs to be created and decorated with the attribute “field: NonSerialized”. Note that Views, Dispatchers, and most events are not serializable.
* Accessing views must be done in a delegate passed to the UI Dispatcher.
* All the callback methods and methods called in them need to be set to public

In order to call allocate the following variables must be present

* runtimeContext – contains the Display manager
* display – the display which the view will be allocated to
* currentUiDispatcher – need so that only one UI thread runs at a time
* createContract – used to marshal the view

The ViewGuid is used by to identify this specific view on the plugin side.

The DisplayView value is set by the unite core when the view is allocated

The ViewMutex is used to allow testing of the views when run in parallel. Multiple views should never be allocated at once when run normally.

The HubView, HubViewModel, and HubDisplayViewType will all be implemented by the class which inherits this one.

Code 4 Create HubViewBase

|  |
| --- |
| [Serializable]  public class HubViewBase : ViewBase  {  [field: NonSerialized]  private UserControl \_hubView;  protected virtual UserControl HubView { get => \_hubView; set => \_hubView = value; }  protected virtual HubViewModel HubViewModel { get; set; }  protected HubViewBase(IHubModuleRuntimeContext runtimeContext, PhysicalDisplay display, Dispatcher currentUiDispatcher, Func<FrameworkElement, MarshalNativeHandleContract> createContract) :  base(runtimeContext, display, currentUiDispatcher, createContract)  { }  public override void Allocate()  {  ViewMutex.WaitOne();  var contract = CreateContract(HubView);  ViewMutex.ReleaseMutex();  RuntimeContext.DisplayManager.AllocateUiInHubDisplayAsync(  contract,  HubAllocationInfo,  AllocatedCallBack  );  }  public override void DeAllocate()  {  RuntimeContext.DisplayManager.DeallocateUiFromHubDisplayAsync(  DisplayView,  DeallocateCallBack  );  }  public override bool Show()  {  if (!IsAllocated) Allocate();  CurrentUiDispatcher.BeginInvoke(DispatcherPriority.ApplicationIdle, new Action(() => { })).Wait();  return IsAllocated && RuntimeContext.DisplayManager.ShowAllocatedUi(DisplayView);  }  // All CallBacks must be public  public override void AllocatedCallBack(HubAllocationResult hubAllocationResult)  {  if (hubAllocationResult.Success)  {  AllocatedSuccess(hubAllocationResult.AllocatedView);  }  else  {  AllocatedFail();  throw new Exception(hubAllocationResult.ResultType.ToString());  }  }  // All CallBacks must be public  public override bool AllocatedFail()  {  lock(this)  {  }  return true;  }  Public override void AllocatedSuccess(DisplayView allocatedView)  {  lock (this)  {  DeepCopy.CopyDisplayView(DisplayView, allocatedView);  }  }  // All Callbacks must be public  public void DeallocateCallBack(HubAllocationResult hubAllocationResult)  {  if (hubAllocationResult.Success)  {  lock (this)  {  SetDeallocate();  }  }  else  {  throw new Exception(hubAllocationResult.ResultType.ToString());  }  }    private void SetDeallocate()  {  DisplayView.Id = new Guid();  }  } |

### Create HubViewEventArgs

HubViewEventArgs is concrete class derived from EventArgs and used when a HubView is created.

1. In the folder **Model/EventArguments**, create a file name HubViewEventArgs.cs
2. Enter the contents of section Code 6 into that file.

In order to call allocate the following variables must be present

* IsOnAllDisplays – used when Allocating and toggles if 1 or All DisplayViews
* SenderControlIdentifier – the identifier
* HubViewType – the HubViewType enum value to target the action
* HubViewMethod – the action to enact on the HubView, usually “Allocate”, “DeAllocate” or “Show”

Code 5 Create HubViewEventArgs

|  |
| --- |
| namespace UnitePlugin.Model.EventArguments  {  [Serializable]  public class HubViewEventArgs : EventArgs  {  public bool IsOnAllDisplays { get; set; }  public Guid SenderControlIdentifier { get; set; }  public UI.HubView.Type HubViewType { get; set; }  public String HubViewMethod { get; set; }  }  } |

### Create QuickAccessIcon

The concrete class that binds the specific View, ViewModel, and Hub information together.

The enum type HubDisplayViewType.QuickAccessAppIconView is a unique identifier which determines where and how the view is displayed on the hub.

Note: Views are Not Serializable and need the attribute “field: NonSerialized”

1. In the folder **UI**, create a file name QuickAccessIcon.cs
2. Enter the contents of section Code 6 into that file.

Code 6 Create QuickAccessIcon

|  |
| --- |
| namespace UnitePlugin.UI  {  [Serializable]  public class QuickAccessIcon : HubViewBase  {  [field: NonSerialized]  private readonly HubDisplayViewType \_HubDisplayViewType = HubDisplayViewType.QuickAccessAppIconView;  [field: NonSerialized]  private QuickAccessIconView \_QuickAccessIconView;  private QuickAccessIconViewModel \_QuickAccessIconViewModel;  protected override UserControl HubView => \_QuickAccessIconView;  protected override HubViewModel HubViewModel => \_QuickAccessIconViewModel;  protected override HubDisplayViewType HubDisplayViewType => \_HubDisplayViewType;  public QuickAccessIconViewModel QuickAccessIconViewModel => \_QuickAccessIconViewModel;  public QuickAccessIcon(IHubModuleRuntimeContext runtimeContext, Func<FrameworkElement, MarshalNativeHandleContract> createContract, PhysicalDisplay display, Dispatcher currentUiDispatcher, EventHandler<HubViewEventArgs> eventCommandInvoker)  : base(runtimeContext, display, currentUiDispatcher, createContract)  {  SetQuickAccessIconView(eventCommandInvoker);  }  private void SetCommandEvents(EventHandler<HubViewEventArgs> eventCommandInvoker)  {  \_QuickAccessIconViewModel.ShowQuickAccessControl += eventCommandInvoker;  }  private void SetQuickAccessIconView(EventHandler<HubViewEventArgs> eventCommandInvoker)  {  CurrentUiDispatcher.Invoke(delegate  {  ViewMutex.WaitOne();  \_QuickAccessIconView = new QuickAccessIconView();  ViewMutex.ReleaseMutex();  \_QuickAccessIconViewModel = \_QuickAccessIconView.DataContext as QuickAccessIconViewModel;  \_QuickAccessIconViewModel.ControlIdentifier = ViewGuid;  SetCommandEvents(eventCommandInvoker);  string assemblyPath = ModuleConstants.EntryPoint;  if (!File.Exists(assemblyPath))  assemblyPath = @"C:\ProgramData\Intel\Intel Unite\Hub\current\FeatureModules\" +  ModuleConstants.ModuleInfo.Id + "\\" + ModuleConstants.EntryPoint;  \_QuickAccessIconViewModel.Image = Intel.Unite.Common.Utils.BytesHelper.SetImageFromResource(Guid.NewGuid(),  UniteImageType.Png, "UCPlugin.Images.UC-Plugin-icon.png", Assembly.LoadFrom(assemblyPath));  });  }  }  } |

## 

### Create QuickAccessIconViewModel

The logic for the QuickAccessIcon concrete class.

1. In the folder **ViewModel**, create a file name QuickAccessIconViewModel.cs
2. Enter the contents of section Code 7 into that file.

Code 7 Create QuickAccessIconViewModel.cs

|  |
| --- |
| namespace UnitePlugin.ViewModel  {  [Serializable]  public class QuickAccessIconViewModel : HubViewModel  {  #region Fields  [field: NonSerialized]  private EventHandler<HubViewEventArgs> \_showQuickAccessControl;  private ICommand \_quickAccessButton\_ClickCommand;  #endregion  [field: NonSerialized]  private UniteImage \_image;  public UniteImage Image { get => \_image; set => \_image = value; }  #region Properties  public event EventHandler<HubViewEventArgs> ShowQuickAccessControl  {  add { \_showQuickAccessControl += value; }  remove { \_showQuickAccessControl -= value; }  }  #endregion  public QuickAccessIconViewModel() : base()  { }  public void QuickAccessButton\_Click(object sender, HubViewEventArgs e)  {  \_showQuickAccessControl?.Invoke(this, e);  }  }  } |

### Create HubView

Plugins must have a HubView object that initiates the create of the views.

The property Type is very important and used by the HubViewFactory to create instances. You MUST add your new views to this enum or they will never be created.

1. In the folder **UI**, create a file name HubView.cs
2. Enter the contents of section Code 8 into that file.

Code 8 Create HubView

|  |
| --- |
| namespace UnitePlugin.UI  {  public class HubView  {  private readonly Dictionary<Type, HubViewFactory> \_factories;  public enum Type  {  QuickAccessIcon,  }  public HubView()  {  \_factories = new Dictionary<Type, HubViewFactory>();  foreach (Type hubViewType in Enum.GetValues(typeof(Type)))  {  var factory = (HubViewFactory)Activator.CreateInstance(System.Type.GetType("UnitePlugin.UI.Factory." + Enum.GetName(typeof(Type), hubViewType) + "Factory") ?? throw new InvalidOperationException());  \_factories.Add(hubViewType, factory);  }  }  public IHubView ExecuteCreation(  Type hubViewType,  IHubModuleRuntimeContext runtimeContext,  Func<FrameworkElement, MarshalNativeHandleContract> createContract,  PhysicalDisplay display,  Dispatcher currentUiDispatcher,  EventHandler<HubViewEventArgs> eventCommandInvoker) =>  \_factories[hubViewType].Create(runtimeContext, createContract, display, currentUiDispatcher, eventCommandInvoker);  }  } |

### Create HubViewFactory

Creates an instance of the HubView

1. In the folder **UI/Factory**, create a file name HubViewFactory.cs
2. Enter the contents of section Code 9 into that file.

Code 9 Create Factory

|  |
| --- |
| namespace UnitePlugin.UI.Factory  {  public abstract class HubViewFactory  {  public abstract IHubView Create(IHubModuleRuntimeContext runtimeContext, Func<FrameworkElement, MarshalNativeHandleContract> createContract, PhysicalDisplay display, Dispatcher currentUiDispatcher, EventHandler<HubViewEventArgs> eventCommandInvoker);  }  } |

### Create QuickAccessIconFactory

Creates an instance of the QuickAccessIconFactory

Open the Solution Explorer

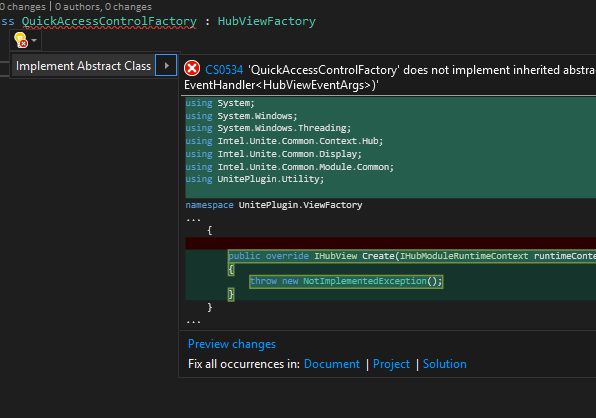
1. In the folder **UI/Factory**, create a file name QuickAccessIconFactory.cs
2. Enter the contents of section Code 10 into that file.

Code 10 Create QuickAccessIconFactory

|  |
| --- |
| namespace UnitePlugin.UI.Factory  {  public class QuickAccessIconFactory : HubViewFactory  {  public override IHubView Create(IHubModuleRuntimeContext runtimeContext, Func<FrameworkElement, MarshalNativeHandleContract> createContract,  PhysicalDisplay display, Dispatcher currentUiDispatcher, EventHandler<HubViewEventArgs> eventCommandInvoker)  {  runtimeContext.LogManager.LogMessage(  ModuleConstants.ModuleInfo.Id,  Intel.Unite.Common.Logging.LogLevel.Trace,  this.GetType().Name,  MethodBase.GetCurrentMethod() + Environment.NewLine + this.GetHashCode());  return new QuickAccessIcon(runtimeContext, createContract, display, currentUiDispatcher, eventCommandInvoker);  }  }  } |

1. Right click on the class name QuickAccessIconFactory
2. Select Implement Abstract Class as seen below.

Figure 1 Implement Abstract Class



8) The resulting Create method should be replaced with the following code

Code 6 Add Create Method

|  |
| --- |
| public override IHubView Create(IHubModuleRuntimeContext runtimeContext, Func<FrameworkElement, MarshalNativeHandleContract> createContract, PhysicalDisplay display, Dispatcher currentUiDispatcher, EventHandler<HubViewEventArgs> eventCommandEnvoker) =>  new QuickAccessControl(runtimeContext, createContract, display, currentUiDispatcher, eventCommandEnvoker); |

### Add Enum value to HubView

The list of different HubViews. This is used to create the dictionary of factories as well as all the HubViews.

In the HubView folder,

1. Open the file HubView.cs
2. Scroll down the file and find the Type enum

Code 8 Add value to Enum

|  |
| --- |
| public enum Type  {  QuickAccessIcon,  QuickAccessControl, // Add Your new View name here  } |