



Silverlight for Windows Embedded Developer's Guide

Published: January 2012

Applies to: Windows Embedded Compact 7

Abstract

This paper is a comprehensive introduction to developing Silverlight for Windows Embedded applications. It describes the programming model and design workflow in Silverlight for Windows Embedded. It covers the following concepts:

- How to provide safe implicit-type conversion of generic objects
- How to work with visual hosts and visual trees
- How to handle events in Silverlight for Windows Embedded

This paper includes tutorials that show you how to create applications, create custom user controls, and implement hardware acceleration for graphics. These tutorials require Visual Studio 2008, Windows Embedded Compact 7, an OS image and a development device or virtual CEPC. To create XAML source files for your UI design, you also need Microsoft Expression Blend 3 or another XAML editor.

Contents

Introduction	4
Overview.....	4
Features and Benefits of Silverlight for Windows Embedded Development Framework.....	5
Supported Silverlight UI Features.....	6
Programming Model in Silverlight for Windows Embedded	10
Design Workflow in Silverlight for Windows Embedded	10
Win32 Control Compatibility	12
Concepts.....	13
Providing Safe Implicit-Type Conversion of Generic Object Types in Silverlight for Windows Embedded	13
Helper Template Overload Method Example	14
QueryInterface Example	15
Working with Visual Hosts and Visual Trees in Silverlight for Windows Embedded	16
Access the Host Window from the Visual Host	19
Handle Additional Window Messages in the Visual Host	19
Handling Events in Silverlight for Windows Embedded	21
Create an Event Handler.....	21
Retrieve Event Data.....	22
Modify Other UI Elements in Event Handling Code.....	23
Add an Event Handler to Parsed XAML Elements	24
Tutorials	25
Create an Application in Silverlight for Windows Embedded	25
Prerequisites.....	25
Step 1: Add Silverlight to Your OS Design.....	26
Step 2: Decide Whether to Write Application Code or Generate Template Code.....	26
Step 3: Create a Subproject for Your Application.....	26
Step 4: Create an Object That Has Event Handlers.....	26
Step 5: Prepare the Silverlight Visual Tree	27
Step 6: Add Variables to Enable Accessing UI Objects in an Event Handler	31
Step 7: Implement Your Application	32
Step 8: Write Shutdown Code	33
Step 9: Build the Application and Your OS Design.....	35
Step 10: Run Your Application	35
Create a Custom User Control in Silverlight for Windows Embedded.....	36
Prerequisites.....	36

Step 1: Define the GUI for the User Control in XAML.....	36
Step 2: Create a C++ Source File for the Custom User Control	37
Step 3: Implement the Custom User Control Class	38
Register.....	41
GetXamlSource	42
Step 4: Register the Custom User Control.....	43
Step 5: Add the Custom User Control to the Visual Tree	43
Step 6: Build the Application and Your OS Design.....	45
Implement Hardware Acceleration for Graphics in Silverlight for Windows Embedded	45
Prerequisites.....	45
Step 1: Add Support for Hardware Acceleration to the OS Design	46
Step 2: Implement Your Hardware Configuration and Graphics-Rendering Behavior	46
IRenderer	47
ICustomSurface.....	47
ICustomGraphicsDevice	48
Step 3: Customize DrawTriangleStrip.....	51
Step 4: Build Your Code.....	52
Step 5: Include the Security Model	53
Step 6: Build the OS Design into a Run-Time Image	53
Step 7: Use Hardware Acceleration in Your Application.....	53
Conclusion	54
Additional Resources	54

Introduction

By following the step-by-step guidelines that are provided in this article, you can learn how to develop applications that are based on Silverlight for Windows Embedded.

Key developer concepts, such as using safe implicit-type conversion for C++ objects in Silverlight, working with visual hosts and visual trees, and creating event handlers are covered in the section [Concepts](#).

Step-by-step guidelines for three developer scenarios—creating an application, creating a custom user control, and implementing hardware acceleration—are covered in the [Tutorials](#) section.

Silverlight for Windows Embedded is a native (C++) UI development framework for Windows Embedded Compact devices that is based on Microsoft Silverlight 3. You can use Silverlight to do the following:

- Define visual UIs for embedded applications in XAML.
- Collaborate with UI designers by using XAML projects.
- Separate C++ programming logic and UI design.
- Add, modify, and customize the UI at run time.
- Create interactive multimedia UIs for embedded devices.

Silverlight for Windows Embedded is compatible with Silverlight 3 XAML and provides a set of equivalent classes for supported XAML elements. For more information about Silverlight, see [Microsoft Silverlight](http://go.microsoft.com/fwlink/?LinkId=162150) (<http://go.microsoft.com/fwlink/?LinkId=162150>).

Overview

Silverlight for Windows Embedded is a UI development framework for embedded devices and is based on Microsoft Silverlight 3 for the desktop browser.

With Silverlight for Windows Embedded, you can create a UI that provides advanced visual effects for your Windows Embedded Compact device shell and applications. Silverlight makes this possible by supporting a subset of Silverlight 3 XAML elements and by supplying a set of C++ classes that provide access to these elements.

Silverlight for Windows Embedded parses a XAML UI and loads it into a C++ object tree. It then integrates the C++ objects with the Graphics, Windowing, and Events (GWES) subsystem to provide a UI for Windows Embedded Compact devices.

The following picture shows a sample home screen that was developed by using Silverlight for Windows Embedded.

Figure 1: Sample Home Screen with Default Theme for Large Size Screens

Features and Benefits of Silverlight for Windows Embedded Development Framework

The Silverlight for Windows Embedded development framework provides the following features and benefits:

- Silverlight C++ API and a programming model that integrates with the Windows Embedded Compact operating system. This interoperability helps shorten the learning curve for developers who are familiar with programming in C++ for Windows Embedded Compact.
- C++ classes that developers can use to create or customize visual appearance and the behavior of UI elements.
- Support for advanced UI features, including gradients, transformations, and animations, so that developers can create interactive Silverlight-based controls in UIs for embedded applications.
- Compatibility with Silverlight 3 XAML and a set of equivalent classes for supported XAML elements for developers and designers who are familiar with using Silverlight 3 XAML.
- Run-time support for displaying XAML UIs so that if they prefer, OEMs can design UIs for applications that are entirely in XAML.
- Ability to dynamically change the UI at run time by using C++.
- Separation of programming logic and UI design to encourage XAML designers to focus on designing user experiences; and to help developers to focus on integration, programming logic, and run-time behavior.

- Interoperability with Microsoft Expression Blend 3 XAML projects and Windows Embedded Silverlight Tools. You can use them together to generate template C++ project code that is based on an Expression Blend 3 XAML project.

Supported Silverlight UI Features

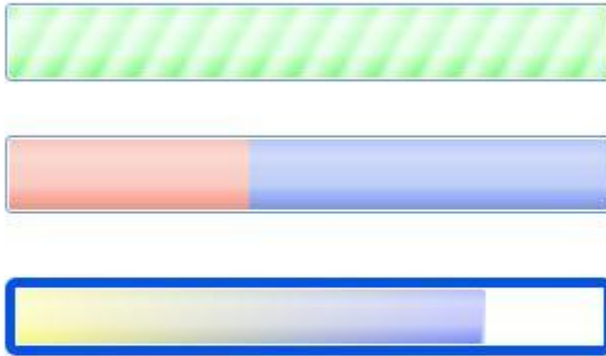
Silverlight for Windows Embedded introduces a subset of Silverlight 3 features to embedded devices, including the following UI features:

- **Advanced Graphics:** Graphics capabilities include brushes that produce gradient multicolor blend effects; image brushes that can paint the interior of UI elements; and **transforms**, which are used to rotate, scale, translate, or skew UI elements. Three-dimensional (3-D) graphics capabilities include 3-D transforms, which transform UI elements in three-dimensional space.
- **Layout System:** The layout system handles the tasks of drawing, resizing, and positioning the UI elements in the graphical window. This built-in functionality removes from the developer the responsibility of drawing and sizing elements on-screen by using C++ code. Silverlight supports layout container objects that work with the layout system to manage the arrangement of UI elements on-screen at run time.
- **Animation, Timers, and Storyboards:** With animation storyboards, UI elements can appear to dynamically change or move on the screen. This effect is achieved by quickly cycling through a series of images, or **key frames**, each slightly different from the previous one, over a specified duration of time. This visual effect can be achieved by defining each key-frame C++ object and implementing an animation storyboard.
- **Pixel Effects:** Pixel effects modify the appearance of a UI element, for example, by blurring the appearance of an element or adding a drop shadow. Pixel effects usually require hardware acceleration so that the graphics processing is done as fast as possible.
- **Text and Typography:** You can use both text controls and typography to display text in a Silverlight-based application. You can customize the display of text and provide unique customizations by changing visual properties or layout, or by applying transforms to the text. With built-in support for the XML Paper Specification (XPS), you can also use predefined glyphs in the Silverlight UI. A glyph is a rendered image that is a visual representation of a character in a font.
- **New Controls:** Silverlight for Windows Embedded introduces new Silverlight controls to Windows Embedded Compact developers, such as user controls, content controls, grids, canvases, paths, rectangles, and password boxes, which each inherit from the intermediate base class [IXRFrameworkElement](http://go.microsoft.com/fwlink/?LinkId=208849) (<http://go.microsoft.com/fwlink/?LinkId=208849>). Silverlight also provides intermediate base classes that are specifically intended for creating custom controls.
- **Styles and Templates:** Styles and templates collectively define the pieces that make up the appearance of a control and that provide the default visual behavior of the control. You can apply styles and templates to controls to define a consistent look for specific types of controls in your application.
- **Geometry:** Geometry objects can be used for rendering two-dimensional (2-D) graphic data on-screen.

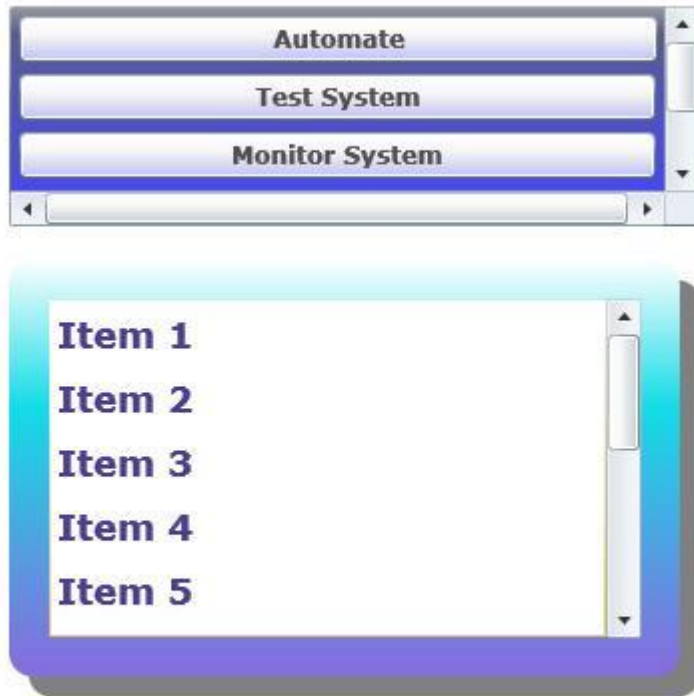
- **Triggers and Events:** Triggers let developers define a custom visual behavior that is demonstrated in response to the **Loaded** event. Additionally, storyboard animations that show visual behavior can be started from within event-handler code. Custom visual behaviors can include changing the color, shape, or size of an object, or playing a short animation storyboard.

The following figures show the types of new Silverlight controls with custom visual appearances that you can design and add to Windows Embedded Compact applications to replace the standard Win32 controls.

Progress Bars



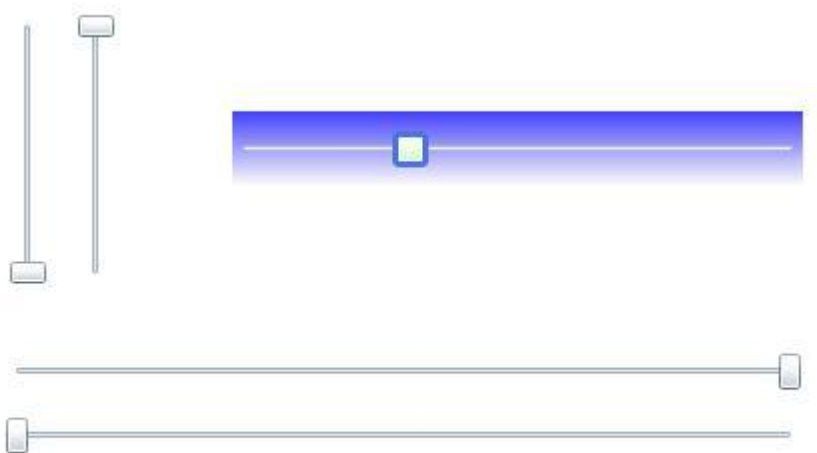
List Boxes



Scroll Viewers



Sliders



Programming Model in Silverlight for Windows Embedded

Silverlight for Windows Embedded offers a comprehensive C++ API that is interoperable with XML-based declarative markup and has no dependency on the .NET Framework.

With Silverlight for Windows Embedded, embedded OEMs can either completely predefine the visual appearance, effects, and behavior in source XAML, or use the C++ programming model to create or customize the UI appearance and functionality for the shell and applications.

Developers can use the Silverlight C++ API to load and display an existing XAML UI, implement event handling for the XAML elements, or customize the UI at run time by adding new elements or changing the visual appearance to respond to factors present at run time.

To enable interaction with the XAML UI at run time, Silverlight for Windows Embedded provides Win32 support for hosting the Silverlight visual tree. It also uses the same visual tree to store new objects that are added by C++ application code at run time.

Design Workflow in Silverlight for Windows Embedded

Silverlight for Windows Embedded provides Windows Embedded Compact 7 developers and XAML designers with the API and tools for creating advanced UIs. With Silverlight, OEMs can use the C++ Silverlight programming model to create UI elements, graphics, and animations that leverage blend effects, weights, shadows, and gradients. OEMs can also work closely with XAML designers who use Expression Blend 3 to develop customized device UIs for both the shell and applications.

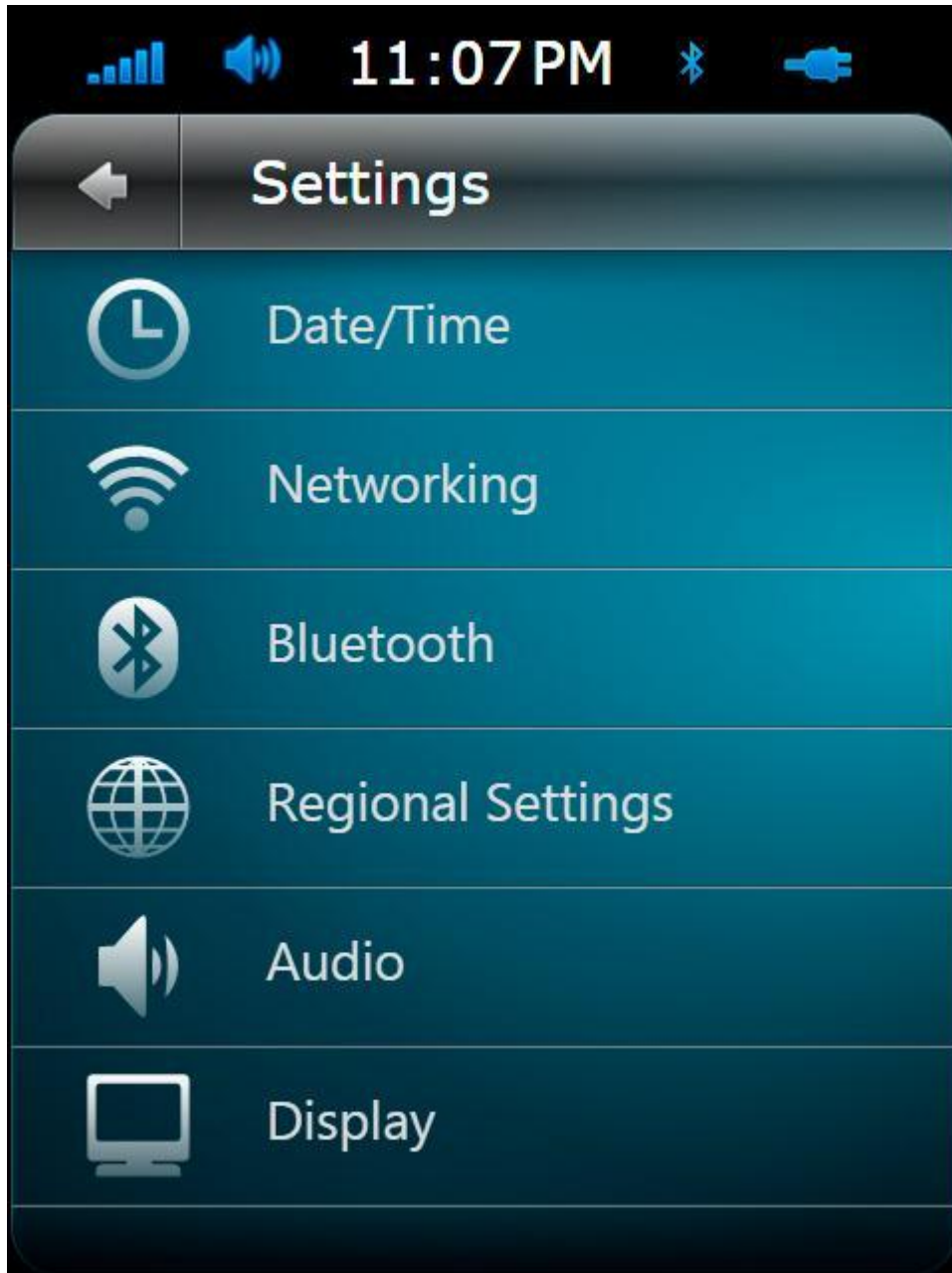
XAML designers can design UIs with XAML by using Expression Blend 3, and embedded developers can develop UI functionality in C++ by using Platform Builder.

The following figures show examples of themes that are designed by using Silverlight XAML for a Windows Embedded Compact-powered device.

Figure 6: Sample Home Screen with Theme 1 for Medium Size Screens



Figure 7: Sample Home Screen with Theme 2 for Small Size Screens



Win32 Control Compatibility

Silverlight for Windows Embedded is interoperable with the Win32 controls that were developed in previous versions of Windows Embedded Compact.

With Win32 control compatibility, you can add window controls, such as a combo box, button, or list box, to the Silverlight object tree at run time, without having to rewrite or change the source code for your window controls.

Concepts

To develop applications in Silverlight for Windows Embedded, you must be familiar with several key concepts in this development framework.

This section provides information about the following key development concepts:

- [Providing Safe Implicit-Type Conversion of Generic Object Types in Silverlight for Windows Embedded.](#)
- [Working with Visual Hosts and Visual Trees in Silverlight for Windows Embedded.](#)
- [Handling Events in Silverlight for Windows Embedded.](#)

Providing Safe Implicit-Type Conversion of Generic Object Types in Silverlight for Windows Embedded

The C++ API in Silverlight for Windows Embedded provides methods that return objects, such as **IXRApplication::CreateObject**, **IXRFrameworkElement::FindName**, and **IXRFrameworkElement::GetParent**. These methods typically return generic object types, such as **IXRDependencyObject**. However, to access or change the characteristics of the UI object that you retrieve, you usually must work with a specific object type.

To work with a specific type of object, you must use a derived interface type instead of **IXRDependencyObject**.

You can ensure that an object is a specific type in two ways:

- Use a helper template overload method to retrieve a type-safe object. A helper template overload method allows you to supply the specific object type to the method, and the method retrieves the object and converts it to the specified type so that you can use the object immediately. This approach simplifies your code and saves you time.



Note

Helper template overload methods are used for C++ programming in Silverlight for Windows Embedded and are not available in Silverlight 3 for web applications.

- Call **IUnknown::QueryInterface** on the object. Use this approach when the method does not support a helper template overload version.



Important

Using the **QueryInterface** approach can be error-prone; if you do not manually add it to your code in a consistent manner, type-safety issues can result.

Helper Template Overload Method Example

The following example shows the signature of a template helper version of the **IXRFrameworkElement::FindName** method, which is defined in XamlRuntime.h.

```
template<typename XRObj> HRESULT FindName(__in const WCHAR* pName, __out XRObj**
ppFoundObject)
{
    IXRDependencyObject* pTemp = NULL;

    HRESULT hr = FindName(pName, &pTemp);

    if (SUCCEEDED(hr) && pTemp)
    {
        hr = pTemp->QueryInterface(
            __uuidof(XRObj), reinterpret_cast<void**>(ppFoundObject));

        pTemp->Release();
    }

    return hr;
}
```

The following example shows how to use a helper template overload method to automatically convert a type from a generic interface to **IXRButton**.



Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
#include "XamlRuntime.h"

HRESULT hr;
IXRButton* pButton = NULL;

hr = pRoot->FindName(L"OKButton", &pButton);
if (SUCCEEDED(hr))
{
```

```

    // Do something with your button

    pButton->Release();
}

```

To make Silverlight application-development easier and to ensure correct reference counting, you can use smart pointers that manage the reference count and ownership lifetime. When you use smart pointers, you do not have to call **IUnknown::Release** to release an interface to an object. The following example shows how to automatically convert a type by using smart pointers with helper methods.

```

#include "XamlRuntime.h"
#include "XRPtr.h"

HRESULT hr;
IXRButtonPtr pButton;
hr = pRoot->FindName(L"OKButton", &pButton);
if (pButton)
{
    // Use the button
}

```

QueryInterface Example

The following code example shows how to retrieve an object by using the **QueryInterface** method and then obtain the derived interface type. We recommend that you use this method only when a helper template overload method is unavailable.



Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```

#include "XamlRuntime.h"

HRESULT hr;
IXRDependencyObject* pDO = NULL;
hr = pPanel->GetChildren(&pDO);

if (SUCCEEDED(hr))

```

```

{
    IXRUIElementCollection* pCollection = NULL;

    hr = pDO->QueryInterface(IID_IXRUIElementCollection, (void**)&pCollection);

    if(SUCCEEDED(hr))
    {
        // Do something with your collection

        pCollection->Release();
    }

    pDO->Release();
}

```

Working with Visual Hosts and Visual Trees in Silverlight for Windows Embedded

A **visual host** represents a Silverlight **visual tree** in a Win32 window. The concept of a **visual host** is unique to Silverlight for Windows Embedded. Silverlight for Windows Embedded is a framework that uses Win32 and C++ programming, unlike Silverlight 3, which is based on the .NET Framework and uses managed code. All visual elements that are parsed from XAML are stored in a visual tree, and only one visual tree can belong to one visual host. By loading XAML markup into a visual tree that belongs to a visual host, you can populate the on-screen content for the UI of a Silverlight for Windows Embedded application. Then a Silverlight for Windows Embedded application can search, modify, and add to the XAML elements in the visual tree with C++ code. For more information about **visual trees**, see [Silverlight Object Trees](http://go.microsoft.com/fwlink/?LinkId=162552) (<http://go.microsoft.com/fwlink/?LinkId=162552>) on MSDN.

To work with the visual tree, you must first obtain a handle to the visual host. You can obtain this handle when you create the visual host by calling **IXRApplication::CreateHostFromXaml** or **IXRApplication::CreateHostFromElementTree**.

Each visual host object contains an **IXRFrameworkElement** object that is a pointer to the root of its visual tree. You can obtain a pointer to the root by calling **IXRVisualHost::GetRootElement**. Then, to find an element, call **IXRFrameworkElement::FindName**. To find an **IXRDependencyObject**-derived object that matches a UI element, create an **XRPtr** smart pointer. You can then use the smart pointer as the following example shows.

 **Important**

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
#include "XamlRuntime.h"
#include "XRPtr.h"

IXRButtonPtr pButton;
IXRFrameworkElementPtr pRootElement;
pVisualHost->GetRootElement(&pRootElement);
pRootElement->FindName(L"MyIXRButton", pButton);
```

After you obtain a smart pointer to an object that is stored in the visual tree, you can work with that object. The following example code shows how to add items to a list box that was parsed from XAML and is stored in the visual tree.

 **Important**

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
#include "windows.h"
#include "XamlRuntime.h"
#include "XRPtr.h"
#include "XRDelegate.h"

void AddNewItemToListBox(IXRApplication* pApplication, IXRListBox* pListBox,
CustomObject* pObject)
{
    // Initialize variables

    IXRListBoxItemPtr pLocationBasedItem;
    IXRImageBrushPtr pImageBrush;
    IXRBitmapImagePtr pDinerBitmap;

    float itemHeight = 50;
    float itemWidth = 100;
    XRValue itemValue;
```

```

itemValue.vType = VTYPE_READONLY_STRING;
itemValue.pReadOnlyStringVal = "Ninth Avenue Diner";

// Create new XR objects

pApplication->CreateObject(&pLocationBasedItem);
pApplication->CreateObject(&pImageBrush);
pApplication->CreateObject(&pDinerBitmap);

// Set values for the image brush to paint the list-box item

pDinerBitmap->SetUriSource(L"Assets/ninthAve.png");
pImageBrush->SetImageSource((IXRImageSource*)&pDinerBitmap);

// Set values for the list-box item

pLocationBasedItem->AddMouseDownEventHandler(CreateDelegate(pObject,
&CustomObject::OnMouseDown));
pLocationBasedItem->AddMouseEnterEventHandler(CreateDelegate(pObject,
&CustomObject::OnMouseEnter));
pLocationBasedItem->AddMouseLeaveEventHandler(CreateDelegate(pObject,
&CustomObject::OnMouseLeave));
pLocationBasedItem->AddOnLoadedEventHandler(CreateDelegate(pObject,
&CustomObject::OnLoaded));
pLocationBasedItem->SetHeight(itemHeight);
pLocationBasedItem->SetWidth(itemWidth);
pLocationBasedItem->SetContent(&itemValue);
pLocationBasedItem->SetBackground((IXRBrush*)&pImageBrush);

// Add the new list-box item to the item collection

IXRItemCollectionPtr pItemCollection;
UINT index = 0;
XRValue xrValue;

```

```

xrValue.vType = VTYPE_OBJECT;
xrValue.pObjectVal = pLocationBasedItem;
pListBox->GetItems(&pItemCollection);
pItemCollection->Insert(index, &xrValue);
}

```

Access the Host Window from the Visual Host

To access the host Win32 window, first call [IXRVisualHost::GetContainerHWND](http://go.microsoft.com/fwlink/?LinkID=198439) (http://go.microsoft.com/fwlink/?LinkID=198439) to retrieve an HWND, and then call Windows Embedded Compact Win32 functions that take an HWND as a parameter, such as **UpdateWindow**, **IsChild**, or **SetParent**. Having access to the host window handle allows you to call Win32 functions directly in Silverlight for Windows Embedded, which is a feature that is not available in Silverlight 3.

Handle Additional Window Messages in the Visual Host

The visual host provides event handling at run time for objects that are stored in the Silverlight visual tree. For example, when the host window receives a **WM_PAINT** message, the visual host draws UI elements on the screen.

When the window receives a user-input event, such as a button click, the visual host must route it to the correct element in its visual tree. By default, window messages such as **WM_PAINT**, **WM_TIMER**, **WM_SETTINGCHANGE**, **WM_SIZE**, **WM_MOUSEMOVE**, **WM_SYSKEYUP**, **WM_CHAR**, **WM_INPUTLANGCHANGE**, **WM_KILLFOCUS**, **WM_MOVE**, and **WM_GESTURE** are processed in the internal **WndProc** of the visual host.

You can provide custom handling for these window messages or handle other window messages that the Silverlight for Windows Embedded **WndProc** does not automatically handle, for example, window messages that are developed for a specific feature, such as Bluetooth wireless technology. To do this, do one of the following:

- Implement a hook procedure for the window message by using the **XR_HOOKPROC** type and add the event-handling code directly to the **HostHookProc** window procedure.
- Create a subclass of the default **WndProc** for the host window. When the window message is received, you can provide message-handling code that calls **IXRUIElement::HitTest** to determine which Silverlight element that the message is for.

To create a hook procedure

1. Define a procedure that matches the signature that is provided in **XR_HOOKPROC**, and give it a custom name.
2. In your procedure, create a message loop that handles **WM_*** messages.

3. In your application startup code, do the following:
 - a. Create an **XRWindowCreateParams** structure, populate its values, and then pass the name of the hook procedure that you implemented into the *pHookProc* member of **XRWindowCreateParams**.
 - b. Pass **XRWindowCreateParams** into the call that creates the visual host, such as **IXRApplication::CreateHostFromXaml** or **IXRApplication::CreateHostFromElementTree**.

The following code shows an example hook procedure.



Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
#include "XamlRuntime.h"

BOOL CALLBACK CustomHookProc (
    VOID* pv,
    HWND hwnd,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam,
    LRESULT* pRetVal)
{
    switch (Msg)
    {
        // Add cases such as WM_CREATE, WM_COMMAND, WM_PAINT if you do not
        // want to pass these messages along for default processing.

        case WM_CAPTURECHANGED:
            // implement custom message-handling code
            return TRUE;

        case WM_MOUSEWHEEL:
            // implement custom message-handling code
            return TRUE;
    }
}
```

```

        return FALSE;
    }

```

Handling Events in Silverlight for Windows Embedded

The way that you use events and delegates in Silverlight for Windows Embedded is different from Silverlight 3 for web applications. In Silverlight for Windows Embedded, you can attach C++ delegates to UI objects through **Add*EventHandler** methods, in order to handle events.

In contrast, in Silverlight 3 you attach event handlers in XAML attributes or by using the common language runtime (CLR) event model in C#.

Silverlight for Windows Embedded defines a C++ template class, **IXRDelegate<ArgType,[SenderType]>**, which represents a delegate that resembles a pointer to an event handler. Then, when a UI action raises that event, Silverlight calls **IXRDelegate<ArgType,[SenderType]>::Invoke** on the event delegate to invoke your event handler.

Create an Event Handler

To create an event handler in Silverlight for Windows Embedded, you first define a class that contains all the event handlers for your application; you then define event handlers for the events you want to handle. An event handler in Silverlight for Windows Embedded must have a signature that matches the following signature.

```
HRESULT EventHandler(IXRDependencyObject* pSender, XREventArgs* pArgs);
```

In the signature, *pSender* must be of type [IXRDependencyObject](http://go.microsoft.com/fwlink/?LinkId=208838) (<http://go.microsoft.com/fwlink/?LinkId=208838>), and *pArgs* must be either **XREventArgs** or a type that is derived from **XREventArgs**.

In your event handler implementation, you can use **QueryInterface** to get a specific interface pointer to the **IXRDependencyObject** object sender (*pSender*), in order to call a method on the object that raised the event.

After you implement an event handler, you can create a delegate, and then attach the delegate to an event of a UI object.

The following example code shows an event handler in Silverlight for Windows Embedded.



Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
#include "XamlRuntime.h"
```

```

HRESULT OnMouseEnter(IXRDependencyObject* pSender, XRMouseEventArgs* pArgs)
{

    IXRUIElementPtr pUIElement;

    HRESULT hr;

    bool Captured = false;

    if((NULL == pSender) || (NULL == pArgs))
    {
        hr = E_INVALIDARG;
    }

    else
    {
        // Get XRMouseEventArgs.Position from the event data and
        // add code for additional coordinate position processing
        XRPoint Position = pArgs->Position;

        // Get the object sender and call IXRUIElement::CaptureMouse
        pSender->QueryInterface(IID_IXRUIElement, (void**)&pUIElement);
        pUIElement->CaptureMouse(&Captured);

        hr = S_OK;
    }

    return hr;
}

```

Retrieve Event Data

An event handler must accept an **XREventArgs** derived structure that contains event data. Silverlight for Windows Embedded provides a variety of such event structures that contain data for specific types of events. As long as it reflects the appropriate event type, you can use any **XREventArgs** derived structure in the method signature for your custom event handler. Then, you can extract data from the

structure to determine information about the event. For example, **XRMouseEventArgs** provides a **Position** member that describes the x-coordinate and y-coordinate for the position on the screen where the mouse event occurred.



Note

The **XREventArgs** derived structure type that you provide in the event handler's *pArgs* parameter must be the same structure type as the *ArgType* that you used in the *pDelegate* of the **Add*EventHandler** method. For more information, see [Add an Event Handler to Parsed XAML Elements](#) later in this topic.

Modify Other UI Elements in Event Handling Code

An event handler takes two input parameters: the object sender and an event data structure. To access or modify an object in the visual tree other than the object sender, you need a pointer to the visual host.

To access the visual host in event-handling code, create a **SetHost** custom method in your event class that sets the pointer for the visual host to a variable in the event class. You can then use that variable to access the visual host in any event handlers in your class.

Make sure that you call your **SetHost** method after you call **IXRApplication::CreateHostFromXaml** to set the **IXRVisualHost** pointer.

The following example shows an implementation of **SetHost**.



Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
#include "XamlRuntime.h"

class EventHandler
{
public:
    IXRVisualHostPtr g_pHost;

    HRESULT SetHost(IXRVisualHost* pHost);
    {
        HRESULT hr = S_OK;
        ASSERT(! g_pHost);
        if(NULL == pHost);
        {
            hr = S_FALSE;
```

```

        return hr;
    }

    g_pHost = pHost;

    return hr;
}

};

```

Add an Event Handler to Parsed XAML Elements

In Silverlight for Windows Embedded, you cannot add names of C# event handlers to XAML elements by using event handler syntax for an attribute of the element. Instead, you must find each named element at run time and add event handlers to them by using the **Add*EventHandler** methods. To do this, you must retrieve an object pointer to each named element from the visual tree. For more information about the visual tree, see [Working with Visual Hosts and Visual Trees in Silverlight for Windows Embedded](#) earlier in this article. For a list of the **Add*EventHandler** methods that are available in each object, see the Silverlight for Windows Embedded reference documentation for that object.

You must know the **x:Name** values for each element that raises events to add an event handler to a parsed element. The **x:Name** values are defined in the source XAML file.

The following code example shows how to attach a delegate to a **MouseEnter** event for a button in C++, after the source XAML is parsed into a visual tree.



Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```

#include "windows.h"

#include "XamlRuntime.h"

#include "XRDelegate.h"

#include "XRPtr.h"

void AddEventHandler(IXRVisualHost* pVisualHost, CustomObject* pObject)
{
    IXRButtonPtr pButton;

    IXRFrameworkElementPtr pRootElement;

```

```

pVisualHost->GetRootElement(&pRootElement);

pRootElement->FindName(L"Button1", &pButton);

if (pButton)
{
    pButton->AddMouseEnterEventHandler(CreateDelegate(pObject,
&CustomObject::OnMouseEnter));
}
}

```

Tutorials

As a way to learn how to use Silverlight for Windows Embedded, this section provides tutorials for the following fundamental development scenarios:

- [Create an Application in Silverlight for Windows Embedded.](#)
- [Create a Custom User Control in Silverlight for Windows Embedded.](#)
- [Implement Hardware Acceleration for Graphics in Silverlight for Windows Embedded.](#)

Create an Application in Silverlight for Windows Embedded

To create an application for a Windows Embedded Compact powered device that is based on Silverlight for Windows Embedded, you create the XAML file that defines the UI and create a subproject. Then, use the Silverlight C++ API to initialize Silverlight, parse the XAML file into an object tree, and add event handlers to the parsed elements.

To learn more about the classes that are available for creating a UI, see [Silverlight for Windows Embedded](#) (<http://go.microsoft.com/fwlink/?LinkId=209193>).

Then, you can add functionality for features to implement custom methods to call from inside event handlers, add window controls, implement custom hook procedures, or add new objects at run time in C++.

Prerequisites

Before you begin this tutorial to create an application, be sure you have satisfied these prerequisites:

- You have decided which type of application to build based on Silverlight. You have also identified the SYSGEN variables and .h files that include the additional APIs to use in your application.
- You have an OS design, a run-time image, and a connection from Platform Builder to your device.
- If you decide to use a source XAML file to define the UI design, you have already created that XAML file by using Silverlight 3 XAML, Expression Blend 3, or by writing directly in a XAML file by

using another XAML editor, such as XAMLPad or WordPad. For more information, see [XAML Overview](http://go.microsoft.com/fwlink/?LinkId=162291) (<http://go.microsoft.com/fwlink/?LinkId=162291>) at MSDN.

Step 1: Add Silverlight to Your OS Design

In Platform Builder, in your OS design project, add Silverlight support by including the Silverlight for Windows Embedded catalog item (SYSGEN_XAML_RUNTIME).

Step 2: Decide Whether to Write Application Code or Generate Template Code

To generate template application code for a Silverlight for Windows Embedded application by using an Expression Blend 3 project, see [A Sample Application Tutorial Using Windows Embedded Silverlight Tools](http://go.microsoft.com/fwlink/?LinkId=189508) (<http://go.microsoft.com/fwlink/?LinkId=189508>) on the web or at %Program Files%\Windows Embedded Compact 7\Documentation. Then skip to step 4.

To learn how to write your own application code for a Silverlight for Windows Embedded application, go to step 3.

Step 3: Create a Subproject for Your Application

To create a subproject in a target directory

1. In Platform Builder, open an existing OS design.
2. On the **Project** menu, click **Add New Subproject**.
3. In the **Subproject Wizard** dialog box, from the **Available templates** list, select the kind of subproject you want to create.
4. In the **Subproject name** field, enter a name for your project.
5. Depending on the kind of subproject that you chose, the wizard displays an additional set of options for your subproject that you can select from. After selecting the kind of subproject foundation that you want to create, click **Next**.
6. Select **Add to the current Dirs file**, and then click **Finish**. The wizard generates a set of source files into which you add code.
7. When you are ready to build your subproject, in Solution Explorer select the subproject, and then, from the shortcut menu, click **Build**.

If you want to define the UI in Silverlight 3 XAML instead of in C++, copy the existing XAML files to your subproject.

Step 4: Create an Object That Has Event Handlers

Create a custom object that includes method implementations for event handlers for all the interactive UI elements in your application, whether they are defined in the source XAML or created in C++.

You can create this object in *<ProjectName>.cpp*, before the **WinMain** routine.

The following example template code shows how to define this custom object.

Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
class CustomObject
{
public:
    HRESULT OnKeyDown( IXRDependencyObject* pSender, XRKeyEventArgs* pArgs )
    {
        //event handler implementation
    }
};
```

Step 5: Prepare the Silverlight Visual Tree

To display the application UI and modify UI elements, you must first prepare Silverlight to run in an application by parsing a source XAML file to populate the visual tree with UI elements.

To complete the task, you initialize the system, load the source XAML file, generate an object tree, and begin routing and processing messages for the application.

To prepare the Silverlight visual tree

1. In the C++ source code file, include the related header files. For example:

```
#include "XamlRuntime.h"
#include "XRDelegate.h"
#include "XRPtr.h"
```

2. In the **WinMain** routine, initialize the system by calling **XamlRuntimeInitialize**. For example:

```
BOOL IsXRInitialized = XamlRuntimeInitialize();
```

3. Obtain a singleton application object by calling **GetXRAApplicationInstance**. For example:

```
IXRApplicationPtr pApplication;
if (IsXRInitialized)
{
```

```

        GetXRApplicationInstance(&pApplication);
    }

```

4. (Optional) Add a resource module for Silverlight to use when it resolves image source Uniform Resource Identifiers (URIs) in the source XAML that it parses and loads into a visual tree. To do this, use one or both of the following methods:

- Call **IXRApplication::AddResourceModule** and supply the handle to the current instance of the application (HINSTANCE) from the **WinMain** function signature.

```
pApplication->AddResourceModule(hInstance);
```

- Create an **IXRResourceManager** object, and then call **IXRApplication::RegisterResourceManager**. For example:

```

IXRResourceManagerPtr pResource;

pApplication->RegisterResourceManager(&pResource);

```

5. (Optional) If the source XAML includes an App.xaml file for resources, call **IXRApplication::GetResourceDictionary** and **IXRApplication::LoadResourceDictionary** to parse that XAML file.

```

IXRResourceDictionaryPtr pResourceDictionary;
XRXmlSource Source;

Source.SetFile(RESOURCE_DIR L"App.xaml");

pApplication->LoadResourceDictionary(&Source, &pResourceDictionary);
pApplication->GetResourceDictionary(&pResourceDictionary);

```

6. Specify the XAML source by creating an **XRXmlSource** structure and populating it with information about the source of the XAML markup.

```

XRXmlSource SourceXaml;

SourceXaml.SetFile(L"AppScene.xaml");

// Also set the XAML resource if you called AddResourceModule
SourceXaml.SetResource(hInstance, L"XAML", MAKEINTRESOURCE(500));

```

7. (Optional) Set up the default window by creating an **XRWindowCreateParams** structure and populating it with window parameters.

```

XRWindowCreateParams WindowParameters;

ZeroMemory(&WindowParameters, sizeof(WindowParameters));

WindowParameters.Style                = WS_POPUP;
WindowParameters.pTitle               = L"Title Name";
WindowParameters.Left                 = 100;

```

```
WindowParameters.Top = 100;

WindowParameters.AllowsMultipleThreadAccess = true;
```

8. Parse the source XAML markup and generate the visual tree by calling **IXRApplication::CreateHostFromXaml**.

```
IXRVisualHostPtr pVisualHost;

pApplication->CreateHostFromXaml(&SourceXaml, &WindowParameters,
&pVisualHost);
```

9. (Optional) Obtain an **IXRFrameworkElement** pointer to the root of the visual tree so that you can add new UI objects to it. These objects can be integrated with the layout system by Silverlight and displayed in the graphical window.

```
IXRFrameworkElementPtr pRoot;

pVisualHost->GetRootElement(&pRoot);
```

10. Display the host window for your application by using one of the following options:

- For a modal dialog box, call **IXRVisualHost::StartDialog**.

```
UINT uiExitCode = 0;

if(pVisualHost != NULL)
{
    // save the exit code for WinMain
    pVisualHost->StartDialog(&uiExitCode);
}
```

- For a modeless dialog box, call **IXRVisualHost::ShowWindow**, and then call **IXRApplication::StartProcessing** to process messages for objects that were loaded into the visual tree.

```
pVisualHost->ShowWindow();
```

The following example code displays the host window for a modeless dialog box.

```
HWND hwnd = NULL;

UINT exitCode = 0;

pVisualHost->GetContainerHWND(&hwnd);

pVisualHost->ShowWindow();

pApplication->StartProcessing(&exitCode);

UpdateWindow(hwnd);
```

After you complete this task, the application is displayed on the screen and is ready for user interaction.

In the **WinMain** routine, after you parse the XAML into an object tree by calling **IXRApplication::CreateHostFromXaml**, insert code that adds the event handlers into each interactive UI object in the object tree.

You can also add an **OnKeyDown** event handler to the **IXRVisualRoot** object that is the default event handler for key events in the window.

► **To add an event handler to a UI object that is created in C++**

1. Add XRDelegate.h and XRPtr.h to the #include list in the Silverlight application.

```
#include "XRDelegate.h"
#include "XRPtr.h"
```

2. Initialize an object variable for a smart pointer, and then use the **IXRApplication::CreateObject** method to convert it into an object and return a reference to the new object.

```
IXRRectanglePtr pRect;
pApplication->CreateObject(&pRect);
```

3. Define the new object instance by using its methods.

```
pRect->SetRadiusX(50);
pRect->SetRadiusY(50);
```

4. Create a delegate for your custom event handler and attach it to the object. To do this, call the associated **Add*EventHandler** method, and call the **CreateDelegate(class,class::method)** inline function in the first argument of the method. Each class for an interactive UI object has one or more **Add*EventHandler** methods, such as **IXRUIElement::AddMouseEnterEventHandler**. To complete this step, you must have already implemented a custom object in step 4 with the required event handlers. For more information, see [Step 4: Create an Object That Has Event Handlers](#) earlier in this white paper.

```
pRect->AddKeyDownEventHandler(CreateDelegate(this,
&CustomObject::OnKeyDown));
```

To add event handlers to parsed XAML elements, you can create a helper function that finds all the interactive objects that are parsed from XAML and then attaches delegates to each object. You can then call this function after you parse the XAML in the **WinMain** routine.

► **To add event handlers to UI objects that are parsed from XAML**

1. Retrieve the **x:Name** values of the XAML elements that you parsed in your Silverlight application.

You can retrieve the values by requesting a list of **x:Name** values from the UI designer; or you can scan the XAML files yourself and extract the value of **x:Name** from each interactive element for which you want to handle events. The following example markup shows a XAML

element that has an **x:Name** value of "Button7."

```
<Button x:Name="Button7" Content="Remove" Width="80" Height="30" />
```

2. Add **XRDelegate.h** and **XRPtr.h** to the **#include** list in the Silverlight application.

```
#include "XRDelegate.h"
```

```
#include "XRPtr.h"
```

3. Obtain an **IXRFrameworkElement** pointer to the root of the visual tree by using the **IXRVisualHost** object that is returned by **IXRApplication::CreateHostFromXaml**.

```
IXRFrameworkElementPtr pRoot;
```

```
pVisualHost->GetRootElement(&pRoot);
```

4. Use **IXRFrameworkElement::FindName** to locate each interactive UI object by its **x:Name** value.

```
IXRButtonPtr pButton;
```

```
pRoot->FindName(L"Button7", &pButton);
```

5. Create a delegate for your custom event handler that you can attach to the UI object.

To do this, call the **CreateDelegate(class,class::method)** inline function in the first argument of the **Add*EventHandler** method. Each class for an interactive UI object has one or more **Add*EventHandler** methods, such as **IXRButtonBase::AddClickEventHandler**. To complete this step, you must have already implemented a custom object in step 4 with the required event handlers. For more information, see [Step 4: Create an Object That Has Event Handlers](#) earlier in this white paper.

```
pButton->AddClickEventHandler(CreateDelegate(this,
```

```
&CustomObject::OnClick));
```

Step 6: Add Variables to Enable Accessing UI Objects in an Event Handler

(Optional) If you want to access the visual tree from an event handler, add a global variable to represent the visual host in the event object. Then implement a method called **SetHost**, which sets the global variable to the visual host of the application, so that the code inside each event handler can access other UI objects in the visual host.

For example:



Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
#include "XamlRuntime.h"
```

```

#include "XRPtr.h"

class CustomObject
{
public:
    IXRVisualHostPtr g_pHost;

HRESULT SetHost(IXRVisualHost* pHost)
    {
        HRESULT hr;
        ASSERT(! g_pHost);
        if (NULL == pHost)
        {
            hr = S_FALSE;
            return hr;
        }

        m_pHost = pHost;

        hr = S_OK;
        return hr;
    }
};

```

Call the new method **SetHost** after you call **IXRApplication::CreateHostFromXaml**.

```

CustomObject events;
events.SetHost(pVisualHost);

```

Step 7: Implement Your Application

Write code to add the functionality you want to your application.

If you want to change the UI design at run time, you can use the Silverlight API to modify or add new UI objects to the object tree. Use **IXRFrameworkElement::FindName** to traverse the object tree and get pointers to objects that you want to customize or add new objects to.

For example, you can change the UI design by setting new values for brushes, images, coordinate positions, animation storyboard collections, and so on.

Step 8: Write Shutdown Code

To enable application users to close your Silverlight for Windows Embedded application, you can add code to an event handler for a **Close** button that shuts down the application.

When the user clicks the button, the corresponding event handler can shut down the application by calling the appropriate methods on the visual host. When you use smart pointers, object instances will automatically call **IUnknown::Release** when they go out of scope. If you did not use smart pointers by using **XRPtr**, call **SAFE_DELETE** on object instances, and call **IUnknown::Release** on the **IXRFrameworkElement** root element instance.

To close an application that runs as a modal dialog and that you created by calling **IXRVisualHost::StartDialog**, call **IXRVisualHost::EndDialog**, which sends a window message to the visual host.

To shut down a visual host window that you displayed by calling **ShowWindow**, call **IXRVisualHost::DestroyWindow**, which shuts down the compositor, frees the off-screen graphical frame buffer and graphics renderer, cleans up window resources, and removes the animation timer.

To hide an application's window when it is not in use yet keep the application running in the background, call **IXRVisualHost::HideWindow**.

The following example code calls **IXRVisualHost::EndDialog** to shut down an application. The example code assumes that the **IXRVisualHostPtr** object in `m_pVisualHost` was already created by the application.



Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
#include "XamlRuntime.h"

// IXRVisualHost object variable
static IXRVisualHostPtr      m_pVisualHost;

inline HRESULT App::GetVisualHost(IXRVisualHost ** ppHost)
{
    if (!ppHost)
        return E_INVALIDARG;

    if (m_pVisualHost)
```

```

    {
        *ppHost = m_pVisualHost;
        (*ppHost)->AddRef();
        return S_OK;
    }

    return E_FAIL;
}

HRESULT MainPage::OK_Click (IXRDependencyObject* pSender, XRMouseButtonEventArgs*
pArgs)
{
    HRESULT hr = E_NOTIMPL;

    if ((NULL == pSender) || (NULL == pArgs))
    {
        hr = E_INVALIDARG;
    }

    IXRVisualHostPtr pHost;
    UINT ExitCode = 1;
    hr = App::GetVisualHost(&pHost);
    if(hr == S_OK)
    {
        pHost->EndDialog(ExitCode);
    }
    return hr;
}

```

Step 9: Build the Application and Your OS Design

1. In **Solution Explorer**, expand **Subprojects**, right-click the subproject you created, and then click **Build (build)**.

The build progress is displayed in the **Output** tab of the **Output** window.

2. (Optional) If you defined the UI in Silverlight 3 XAML, add the XAML files to the subproject BIB file. In **Solution Explorer**, expand **Subprojects**, expand **Parameter files**, and then double-click **<ProjectName>.bib**. Create a **FILES** section and add entries for the XAML files. For more information, see [FILES Section](http://go.microsoft.com/fwlink/?LinkId=211195) (<http://go.microsoft.com/fwlink/?LinkId=211195>) in the Windows Embedded Compact Documentation.
3. Rebuild the run-time image, and download it to your device through the connection that you have already configured.

Step 10: Run Your Application

Run the application on the run-time image by doing one of the following tasks:

- In Platform Builder, on the **Target** menu, click **Run Programs**, click **<ProjectName>.exe**, and then click **OK**.

- In Platform Builder, on the **Target** menu, click **Target Control**. Then at the command prompt, type `s <ProjectName>`.

Create a Custom User Control in Silverlight for Windows Embedded

You can create your own user control that provides customized functionality and unique visual characteristics by using the **XRCustomUserControlImpl::<Base,IFace>** class in Silverlight for Windows Embedded. For example, you might need a control that is customized for a particular type of application, such as a control for navigating a map, a control for browsing a three-dimensional menu, or a control that shows progress or status by using advanced visual indicators instead of a simple progress bar.

You define the appearance of a custom user control in XAML, implement its functionality in C++ by using Platform Builder, and access it from the visual tree by using the **IXRFrameworkElement::FindName** method.

When you create a custom user control, you can reuse it across multiple Silverlight for Windows Embedded applications.



Important

If you used Windows Embedded Silverlight Tools to generate template code for your Silverlight application, do not use the steps in this tutorial to create a custom user control. Instead, refer to [A Sample Application Tutorial Using Windows Embedded Silverlight Tools](http://go.microsoft.com/fwlink/?LinkId=189508) (<http://go.microsoft.com/fwlink/?LinkId=189508>) at %Program Files%\Windows Embedded Compact 7\Documentation.

Prerequisites

Before you begin this tutorial to create a custom user control, be sure you have satisfied this prerequisite:

- A Silverlight application. For more information, see [Create an Application in Silverlight for Windows Embedded](#) earlier in this article.

Step 1: Define the GUI for the User Control in XAML

Use Expression Blend 3 or another XAML editor to create a new XAML file that contains the GUI definition and the namespace declaration for the control. To add the user control to other .xaml files, add its custom namespace declaration to the root element. For more information about XAML namespaces, see [Silverlight XAML Namespaces, and Mapping XAML Namespaces as Prefixes](http://go.microsoft.com/fwlink/?LinkId=149691) (<http://go.microsoft.com/fwlink/?LinkId=149691>) on MSDN. For more information about doing this with Expression Blend 3, see the [Microsoft Expression website](http://go.microsoft.com/fwlink/?LinkId=154604). (<http://go.microsoft.com/fwlink/?LinkId=154604>)

 **Note**

Silverlight for Windows Embedded does not support code-behind in C# for user controls that are defined in XAML. If you have code-behind written in C# for a Silverlight custom user control that you want to reuse, you can convert it to the C++ class that you create in this tutorial.

Step 2: Create a C++ Source File for the Custom User Control

In your application subproject in Platform Builder, create a C++ source file for your user control that includes `XamlRuntime.h`, `XRPtr.h`, and `XRCustomControl.h`.

If your application calls methods in the custom user control, you must create an interface in the source file that derives from [IXRCustomUserControl](http://go.microsoft.com/fwlink/?LinkId=208904) (<http://go.microsoft.com/fwlink/?LinkId=208904>) for the control. You then define any custom methods or fields on the interface.

The following example code provides a beginning code template for creating an interface for a Silverlight custom user control.

```
__interface __declspec(uuid("{F01D249B-89EC-4134-9CF7-822CB326D5A3}"))
class ICustomCtrl : public IXRCustomUserControl
{
public:
    virtual HRESULT SomeMethod() = 0;
};
```

As shown in the previous code, you must create a new unique identifier (ID) for the interface by using the GUIDGEN tool and the **__declspec** keyword. For more information, see [__declspec](http://go.microsoft.com/fwlink/?LinkId=163934) (<http://go.microsoft.com/fwlink/?LinkId=163934>) at MSDN.

 **To obtain a unique ID for the `__declspec` keyword**

1. In the Command Prompt window in Windows, change the directory to `C:\Program Files\Microsoft Visual Studio 9.0\Common7\Tools`, and then type `guidgen`.
2. In the **Create GUID** dialog box, select **4 Registry Format**, and then click **Copy**.
3. Open the new source file by double-clicking it in **Solution Explorer**, and then paste the GUID into either the **__declspec** keyword, or into the second parameter of the **DEFINE_XR_IID** macro.

 **Note**

You can also use the **DEFINE_XR_IID** macro to assign a unique ID to the interface. **DEFINE_XR_IID** is described in the Silverlight for Windows Embedded reference documentation.

Step 3: Implement the Custom User Control Class

Implement a custom user control class that inherits from the template wrapper class

XRCustomUserControlImpl<Base,IFace>.

If you created a custom interface in the previous step, the following requirements must be true:

- The optional **IFace** parameter of the template must be the name of that interface.
- The GUID you supply in the required **__declspec** keyword must be different from the one that you associated with the custom interface.

The following code shows a beginning template for implementing a Silverlight custom user control class.



Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
class __declspec(uuid("{91C2F5FF-8FCC-4626-AC67-850283620C09}")) CustomCtrl : public
XRCustomUserControlImpl<CustomCtrl, ICustomCtrl>
{
public:
    static HRESULT GetXamlSource(XRXamlSource* pXamlSource)
    {
        // add implementation
    }

    static HRESULT Register(HINSTANCE hInstance)
    {
        // add implementation
    }
};
```

If you derived the control class from the wrapper class **XRCustomUserControlImpl<Base,IFace>**, you must implement **Register** and **GetXamlSource**.

In addition to **Register** and **GetXamlSource**, you can also define and implement any additional methods, fields, or events that you want in your custom user control class implementation.

If you will register a dependency property or an attached property, you can implement the **Get*** and **Set*** methods that use **XRValue** objects for values. The following example code shows examples of **Get*** and **Set*** methods for a dependency property, and a method that registers the property.

 **Important**

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
static HRESULT RegisterDependencyProperties()
{
    HRESULT hr = E_FAIL;
    IXRApplicationPtr pApplication;
    XRDependencyPropertyMetaData metaData;

    // Get the XAML Runtime application instance.
    hr = GetXRApplicationInstance(&pApplication);

    // Set dependency property metadata.
    // Populate metadata with the name of a custom property change handler
and type converter.
    metaData.pfnPropertyChangeNotification = TextPropertyChanged;
    metaData.pfnTypeConverter = ConvertTextTypeConverter;

    // Register this property and set its metadata.
    hr = pApplication->RegisterDependencyProperty(L"CustomText", VTYPE_BSTR,
ControlID(), &metaData, &m_dpIdCustomText);

    return hr;
}

/ Example of custom Set method

HRESULT SetCustomText(WCHAR* pText)
{
    HRESULT hr = E_FAIL;

    // Wrap the value in an XRValue object.
    XRValue xrValue;
    xrValue.vType = VTYPE_READONLY_STRING;
```

```

        xrValue.pReadOnlyStringVal = pText;

        // SetPropertyValue is a function on the base class,
        IXRCustomUserControlBase.

        hr = SetPropertyValue(m_dpIdCustomText, &xrValue);

        return hr;
    }

// Example of custom Get method

HRESULT GetCustomText(BSTR* pbstrText)
{
    HRESULT hr = E_FAIL;

    // The value is returned in an XRValue object.
    XRValue xrValue;

    // GetPropertyValue is a function on the base class,
    IXRCustomUserControlBase.

    hr = GetPropertyValue(m_dpIdCustomText, &xrValue);

    if (SUCCEEDED(hr))
    {
        *pbstrText = xrValue.bstrStringVal;
    }

    return hr;
}

```

When you define a custom user control class that inherits from **XRCustomUserControlImpl<Base,IFace>**, you must also provide custom implementations of the **Register** and **GetXamlSource** methods.

Register

In the **WinMain** procedure of your application, after you retrieve the application instance and optionally, add a resource module, you call your custom **Register** method. You then create the visual host by calling **IXRApplication::ParseXaml**.

At minimum, you must include code that calls the **XRCustomUserControlImpl::Register** method in your implementation. This method registers a custom user control by associating a specific element name, which was defined in a specific XAML namespace with the **x:Class** attribute, with a specific interface ID (IID). **XRCustomUserControlImpl::Register** is a static method, and you can call it without creating an instance of **XRCustomUserControlImpl<Base,IFace>**.

If the custom user control has a dependency property or an attached property, your **Register** method implementation must also include code that registers the property. In this case, the **Register** method implementation must obtain an application instance by calling **GetXRApplicationInstance**, and then call either the **IXRApplication::RegisterAttachedProperty** or **IXRApplication::RegisterDependencyProperty** method, depending on the type of property included. Or it must call a custom implemented method that registers the property.

Typically, you also implement a **GetCustomProperty** and **SetCustomProperty** method in your control class that provides access to the value of the property.

The following example code shows an implementation of the **Register** method that is intended to be used in the declaration of your implementation of **CustomCtrl**.



Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
static HRESULT Register()
{
    HRESULT hr = S_OK;

    hr = XRCustomUserControlImpl::Register(__uuidof(CustomCtrl)),
    L"CustomUserControl", L"clr-namespace:CustomUserControlNamespace");

    if (FAILED(hr))
    {
        goto Exit;
    }

Exit:
    return hr;
}
```

GetXamlSource

Before you can parse and load the control into the visual tree, you have to call the **GetXamlSource** method to get the source XAML file that defines the GUI for the control. The **GetXamlSource** method retrieves the control-definition XAML file so that Silverlight can parse its XAML and load the control into the visual tree.

You implement the **GetXamlSource** method so that the **PFN_CREATE_CONTROL** callback function can call **GetXamlSource** internally when Silverlight parses XAML for the application. This callback function is provided by the wrapper class.

At minimum, your **GetXamlSource** implementation must include code that populates the **XRXamlSource** structure, which is supplied in the input parameter of this method, with information about the source XAML for your control. To add this code, do one of the following:

- Call the member function **XRXamlSource.SetResource**. This member function takes an application instance that is returned by an earlier call to **GetXRApplicationInstance**, a string that describes the XAML resource type, and the ID of the resource, which is a .xaml file. If you define IDs for XAML resources in a resource (.rc) file, you can convert the IDs into resource types by using the **MAKEINTRESOURCE** macro inline. The following code example shows how to use this member function in a **GetXamlSource** implementation.

Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
#define RT_XAML          L"XAML"

static HRESULT GetXamlSource (XRXamlSource* pXamlSource)
{
    pXamlSource->SetResource (s_hInstance, RT_XAML,
    MAKEINTRESOURCE (ID_XAML_2DLISTVIEW));
    return S_OK;
}
```

- Call the member function **XRXamlSource.SetFile**. Using this approach, you can set the source XAML file by providing the file name. The following code example shows how to use this member function in a **GetXamlSource** implementation.

Important

For readability, the following code example does not contain security checking or error handling. Do not use the following code in a production environment.

```
static HRESULT GetXamlSource (XRXamlSource* pXamlSource)
{
    pXamlSource->SetFile (RESOURCE_DIR L"CustomControlDefinition.xaml");
}
```

```

        return S_OK;
    }

```

Step 4: Register the Custom User Control

To register the custom user control in your application startup code, call the **Register** method that you implemented before the application startup code calls **CreateHostFromXaml**. The wrapper class default implementation of **PFN_CREATE_CONTROL** then calls **GetXamlSource** to provide the correct XAML file to parse and load.

Step 5: Add the Custom User Control to the Visual Tree

(Optional) If you want to reuse the custom user control from C++ code in Silverlight applications, after you prepare the Silverlight visual tree, add an instance of the custom user control to the visual tree at run time in C++. For more information, see Step 5 in the tutorial [Create an Application in Silverlight for Windows Embedded](#) earlier in this article.

► To create a custom user-control instance and add it to the visual tree

1. Initialize an object variable by using the class type of the custom user control.

```
XRPtr<ICustomCtrl> pControl;
```

2. Call **IXRApplication::CreateObject** to convert into an object the class that you registered by using the **Register** method and also to return a reference to the new object.

```
pApplication->CreateObject(&pControl);
```

3. (Optional) To locate this object by **Name** after you add it to the visual tree, call **IXRDependencyObject::SetName**.

4. Define the new object instance by using its methods.

```
XRThickness Margin = {25, 25, 100, 200};
```

```
pControl->SetMargin(&Margin);
```

```
pControl->SetCustomPropValue(100);
```

```
pControl->SetName(L"ControlInstance1");
```

5. Attach delegates to the object so that it can respond to events.
6. Obtain an **IXRFrameworkElement** smart pointer to the root of the visual tree. The custom user control is integrated with the layout system in Silverlight and displayed in the graphical window. When you use a smart pointer, you do not have to call **Release** when you are done with using the pointer.

```
IXRFrameworkElementPtr pRoot;
```

```
pVisualHost->GetRootElement(&pRoot);
```

7. Locate the name of the panel element on which UI elements are positioned, such as an **IXRPanel** derived object.
8. Find the element in the tree. On the root element, call **IXRFrameworkElement::FindName**, pass in the name of the chosen panel element, and also pass an object pointer that references the panel.

```
IXRCanvasPtr pCanvas;

pRoot->FindName(L"MainCanvas", &pCanvas);
```

9. Retrieve the panel **IXRUIElementCollection** collection by calling **IXRPanel::GetChildren**.

```
IXRUIElementCollectionPtr pCollection;

pCanvas->GetChildren(&pCollection);
```

10. Add the control instance to the collection by calling **IXRUIElementCollection::Add**.

```
pCollection->Add(pControl, NULL);
```

 **To retrieve a custom user control instance from the visual tree**

1. Obtain an **IXRFrameworkElement** smart pointer to the root of the visual tree. When you use a smart pointer, you do not have to call **Release** when you are done with it.

```
IXRFrameworkElementPtr pRoot;

pVisualHost->GetRootElement(&pRoot);
```

2. Initialize an object variable by using the class type of the custom user control.

```
XRPtr<CustomCtrl> pControl;
```

3. Find the control in the visual tree. On the root element, call **IXRFrameworkElement::FindName**, pass in the name of the chosen control, and also pass an object variable to receive the pointer.

```
pRoot->FindName(L"ControlInstance1", &pControl);
```

4. Execute the functionality of the custom user control or change its property values by using the object pointer to call the methods that are implemented by control class.

```
// set values for the custom user control

float Height = 250;

float Width = 250;

pControl->SetWidth(Width);

pControl->SetHeight(Height);
```

Step 6: Build the Application and Your OS Design

In **Solution Explorer**, expand **Subprojects**, right-click the subproject you created, and then click **Build (build)**.

The build progress is displayed in the **Output** tab of the **Output** window.

Rebuild the run-time image, and download it to the device through the connection that you already configured.

Implement Hardware Acceleration for Graphics in Silverlight for Windows Embedded

You can implement support for graphics hardware acceleration for bitmaps that are created with Silverlight for Windows Embedded, which lets you offload common graphics operations from the main microprocessor onto a display controller by blitting bitmaps first to an off-screen graphical frame buffer, or **surface**, before they are rendered on the screen. This process creates a faster and more stable graphics environment.

Graphics hardware acceleration in Silverlight for Windows Embedded is based on one of the following:

- **DirectDraw** Supports hardware-accelerated 2-D graphics. For more information, see [DirectDraw](http://go.microsoft.com/fwlink/?LinkId=211198) (<http://go.microsoft.com/fwlink/?LinkId=211198>) in the Windows Embedded Compact 7 Documentation.
- **OpenGL** Supports hardware-accelerated 2-D or 3-D graphics, skewing, and rotation. For more information, see [Introduction to OpenGL](http://go.microsoft.com/fwlink/?LinkId=160256) (<http://go.microsoft.com/fwlink/?LinkId=160256>) on MSDN.

If you do not implement hardware-accelerated graphics, Windows Embedded Compact uses Graphics Device Interface (GDI) to draw UI objects pixel-by-pixel onto the primary display surface in the order they were first loaded into the object tree during XAML parsing.

Hardware acceleration relies on a process called **cached composition**, which minimizes CPU usage during rasterization by using the graphics processing unit (GPU) and memory instead of the CPU. The disadvantage of this technique is that it requires additional video and system memory on the hardware board.



Note

If you implement hardware acceleration based on OpenGL, the XAML UI that is rendered on the screen has a maximum size limitation of 2,048 × 2,048 pixels.

Prerequisites

Before you begin this tutorial to implement hardware acceleration for graphics in Silverlight for Windows Embedded, be sure you satisfy these prerequisites:

- A Silverlight application. For more information, see [Create an Application in Silverlight for Windows Embedded](#) earlier in this article.

- Your hardware board meets the guidelines that are described in the [Silverlight for Windows Embedded Hardware Recommendations](http://go.microsoft.com/fwlink/?LinkId=205720) (<http://go.microsoft.com/fwlink/?LinkId=205720>) topic on MSDN or in the Windows Embedded Compact 7 Documentation.

Step 1: Add Support for Hardware Acceleration to the OS Design

To add the DirectDraw rendering plug-in, include the BSP flag `BSP_XRPLUGIN_DDRAW`. To add the OpenGL rendering plug-in, include the BSP flag `BSP_XRPLUGIN_OPENGL`.

To set a BSP flag at the command line:

1. In Platform Builder, on the **Build** menu, click **Open Release Directory in Build Window**.
2. At the command prompt, type `Set`, type a space, and then type the flag you want to set. To add the DirectDraw rendering plug-in to your OS design, type `Set BSP_XRPLUGIN_DDRAW=1`.

To add the OpenGL rendering plug-in to your OS design, type `Set BSP_XRPLUGIN_OPENGL=1`.

Step 2: Implement Your Hardware Configuration and Graphics-Rendering Behavior

Before you customize the graphics renderer, backup the source code files by using a source code control system. For more information, see [Adding an OS Design to a Source Code Control System](http://go.microsoft.com/fwlink/?LinkId=226881) (<http://go.microsoft.com/fwlink/?LinkId=226881>).

The default DirectDraw plug-in is implemented in:

- **IRenderer** To customize, implement your changes in `%_WINCEROOT%\Public\Common\Oak\Xamlrendererplugin\Ddraw\Ddrawrenderer.cpp`.
- **ICustomSurface** To customize, implement your changes in `%_WINCEROOT%\Public\Common\Oak\Xamlrendererplugin\Ddraw\Ddrawsurface.cpp`.
- **ICustomGraphicsDevice** To customize, implement your changes in `%_WINCEROOT%\Public\Common\Oak\Xamlrendererplugin\Ddraw\Ddrawdevice.cpp`.

The default OpenGL plug-in is implemented in:

- **IRenderer** To customize, implement your changes in `%_WINCEROOT%\Public\Common\Oak\Xamlrendererplugin\Opengl\Openglrenderer.cpp`.
- **ICustomSurface** To customize, implement your changes in `%_WINCEROOT%\Public\Common\Oak\Xamlrendererplugin\Opengl\Openglsurface.cpp`.
- **ICustomGraphicsDevice** To customize, implement your changes in `%_WINCEROOT%\Public\Common\Oak\Xamlrendererplugin\Opengl\Opengldevice.cpp`.



To customize the DirectDraw or OpenGL graphics renderer

1. In Platform Builder, open your OS design project.
2. In Solution Explorer, browse to **<OS Design Name>\C:\WINCE700\Public\Common\Oak\XamlRundererPlugin**.

3. Expand either **DDraw** or **OpenGL**, browse to the .cpp file to customize, and double-click the .cpp file.
4. Customize the code in the .cpp file, and save your changes.

To customize the source code for the interfaces, you can add code to the following default method implementations, which are called internally by Silverlight for Windows Embedded when it displays graphics.

IRenderer

Programming element	Description
CreateRenderer	Creates an IRenderer object for the host window. If the plug-in supports cached composition of bitmaps, also creates an ICustomGraphicsDevice object.
IRenderer::FreeResources	Frees resources that are associated with the off-screen surface and the ICustomGraphicsDevice object.
IRenderer::PreRender	For the DirectDraw renderer, creates an off-screen surface for blitting the graphics before they are displayed on the screen. You can add code inside this method to perform rendering tasks that are required before the UI can be displayed on the screen.
IRenderer::PostRender	Blits the off-screen surface to the primary surface. Adds code to complete all work that is needed after rendering a scene.
RenderPluginInitialize	Initializes the rendering plug-in.
RenderPluginCleanup	Cleans up allocated resources when Silverlight shuts down.

ICustomSurface

Programming element	Description
ICustomSurface::Lock	Obtains a pointer to the surface that is allocated

Programming element	Description
	in system memory, and retrieves the dimensions, in pixels, of the surface to be created.
<code>ICustomSurface::Unlock</code>	Unlocks the surface that was previously locked.
<code>ICustomSurface::Present</code>	<p>Dumps the content of the back buffer onto the screen.</p> <p>For DirectDraw, the content of the back buffer is blitted on to the primary surface that is associated with the client window.</p> <p>For OpenGL, you can use eglSwapBuffer() to post the content of the target buffer to the associated window.</p>
<code>ICustomSurface::GetWidth</code>	Retrieves the width, in pixels, of the surface.
<code>ICustomSurface::GetHeight</code>	Retrieves the height, in pixels, of the surface.
<code>ICustomSurface::GetPixelFormat</code>	Retrieves the color and pixel format of the surface.
<code>ICustomSurface::IsVideoSurface</code>	Indicates whether the surface contains video content.
<code>ICustomSurface::IsOpaque</code>	Returns 1 if the custom surface is opaque. Otherwise, returns 0 (zero).
<code>ICustomSurface::IsTransparent</code>	Returns 1 if the custom surface is transparent. Otherwise, returns 0 (zero).
<code>ICustomSurface::SetIsOpaque</code>	Sets whether the custom surface is opaque.
<code>ICustomSurface::SetIsTransparent</code>	Sets whether the custom surface is transparent.

ICustomGraphicsDevice

Programming element	Description
<code>ICustomGraphicsDevice::Clear(UINT uColor)</code>	<p>Clears the back buffer content by setting it to the color that is specified in <i>uColor</i>.</p> <p>For OpenGL, clears and fills its color buffer by calling glClearColor().</p>

Programming element	Description
	For the DirectDraw plug-in, fills the back buffer with the color in <i>uColor</i> by using the Blit function.
<code>ICustomGraphicsDevice::Initialize(HWND hWnd)</code>	Sets up the primary surface that is related to the client window.
<code>ICustomGraphicsDevice::Resize(UINT uWidth, UINT uHeight)</code>	<p>Updates all related structures or data after the client window size is changed. This element is called when a window size changes.</p> <p>For DirectDraw, a surface of the new size must be created.</p> <p>For OpenGL, the viewport and UniformMatrix are updated.</p>
<code>ICustomGraphicsDevice::CreateTexture(Int fRenderTarget, UINT nWidth, UINT nHeight, Int fKeepSystemMemory, ICustomSurface **ppSurface)</code>	<p>Provides an entry point for creating customized surfaces.</p> <p>The <i>fRenderTarget</i> parameter specifies whether the surface is the target that is used to buffer graphics before displaying them on the screen. The <i>fRenderTarget</i> parameter is currently only used in the DirectDraw-based renderer.</p> <p>The <i>fKeepSystemMemory</i> parameter specifies whether this surface must be kept as a copy in system memory. When this parameter is true, you must allocate a surface of the same size in system memory for performing rasterization and updates.</p> <p>For DirectDraw, the surface is a DirectDraw surface of type DDSCAPS_SYSTEMMEMORY.</p> <p>For OpenGL, allocates a section of memory on the heap.</p>
<code>ICustomGraphicsDevice::SetTexture(UINT uSampler, ICustomSurface *pTexture)</code>	<p>Sets the specified ICustomSurface object, <i>pTexture</i>, as the texture surface to be rendered. Call this method before calling DrawTriangleStrip().</p> <p>For DirectDraw, keeps this surface as the source surface to be drawn.</p> <p>For OpenGL, sets this texture surface to the</p>

Programming element	Description
	drawing context.
<code>ICustomGraphicsDevice::SetRenderTarget(ICustomSurface *pRenderTarget)</code>	Sets the surface in the <i>pRenderTarget</i> parameter as the target surface. Currently this method is only used in the DirectDraw based renderer.
<code>ICustomGraphicsDevice::DrawTriangleStrip(_ecount(cVertices) XRVertex *pVertices, UINT cVertice)</code>	<p>Draws a series of triangles by using the specified set of input vertices that are provided in an array of XRVertex structures.</p> <p>This is the worker function for cached composition. The XRVertex structure includes the destination surface coordinates, texture coordinates, and diffuse color. Each pixel is multiplied by the diffuse color, including the Alpha component. In this way, Silverlight can apply additional transparency or opacity to the bitmap.</p> <p>For OpenGL, defines the vertex attributes for vertex position, diffuse, and texture by calling glVertexAttribPointer() before it calls glDrawArrays(GL_TRIANGLE_STRIP) to draw the array of vertices.</p> <p>For DirectDraw, the plug-in calculates the destination and source rectangle according to the vertex coordinates and calls AlphaBlit() to draw the triangles. However, DrawTriangleStrips in DirectDraw does not support vertices of more than 4 nor does it support rotation and skewing. Instead, it falls back to software rendering.</p>
<code>ICustomGraphicsDevice::Present()</code>	<p>Presents the content of back buffer on the screen.</p> <p>For DirectDraw, blits the content in the target surface onto the primary surface.</p> <p>For OpenGL, uses eglSwapBuffer() to post the buffer to the associated window.</p>
<code>ICustomGraphicsDevice::GetTextureMemoryUsage</code>	Retrieves memory usage for the texture surface.

Programming element	Description
<code>ICustomGraphicsDevice::IsHardwareComposited</code>	Indicates whether the hardware meets the requirement of cache composition. If it does not, Silverlight does not follow the cached composition path, even if the source XAML has cached elements in it.

Step 3: Customize `DrawTriangleStrip`

`DrawTriangleStrip` draws a series of triangles by using a set of input vertices that are provided in an array of **`XRVertex`** structures.

For DirectDraw:

- The **`CDDrawDevice DrawTriangleStrip`** method is located in
`%_WINCEROOT%\Public\Common\Oak\Xamlrendererplugin\Ddraw\Ddrawdevice.cpp`.

For OpenGL:

- The **`COpenGLDevice DrawTriangleStrip`** method is located in
`%_WINCEROOT%\Public\Common\Oak\Xamlrendererplugin\Opengl\Opengldevice.cpp`.



Note

For DirectDraw, the element is rendered when the set of four vertices represent a rectangle.

The rectangle is drawn onto the surface in the form of two half-triangles within the boundaries of that rectangle. If *cVertices* is not equal to 4, the system uses default software rendering.

An **`XRVertex`** structure defines a vertex, which is the point at which two sides of an angle intersect. The following table lists the members of the **`XRVertex`** structure.

Member	Description
<code>XRVertex.x</code>	The x-coordinate of the vertex.
<code>XRVertex.y</code>	The y-coordinate of the vertex.
<code>XRVertex.z</code>	The z-coordinate of the vertex, which you can use only for 3-D rendering. This value should be the same for each vertex.
<code>XRVertex.u0</code>	The x texture-coordinate, which you use for texture-stage 0.
<code>XRVertex.v0</code>	The y texture-coordinate, which you use for texture-stage 0.

Member	Description
XRVertex.u1	The x texture-coordinate, which you use for texture-stage 1.
XRVertex.v1	The y texture-coordinate, which you use for texture-stage 1.
XRVertex.dwDiffuse	The opacity value.

You can assign texture coordinates directly to vertices by using the u0, v0, u1, and v1 members. A **texture** is a bitmap of pixel colors that give an object the appearance of texture. Texture can give visual depth to an object, such as applying the texture of sandpaper to a beige background to make it appear rough. By assigning texture coordinates directly to vertices, you can control which part of a texture is mapped onto a primitive.

A texture can be assigned to a stage so that you can define how that texture is rendered. For example, you can blend multiple textures together. With **XRVertex**, you can change how the textures appear in different areas by applying texture coordinates for two different texture stages.

Possible effects include changing texture filling, environment mapping, or adjusting the level of detail.

Step 4: Build Your Code

Before you customize the graphics renderer, backup the source code files by using a source code control system. For more information, see [Adding an OS Design to a Source Code Control System](http://go.microsoft.com/fwlink/?LinkId=226881) (<http://go.microsoft.com/fwlink/?LinkId=226881>).

► To recompile and build the graphics renderer

1. In Solution Explorer, browse to <OS Design Name>\C:\WINCE700\Public\Common\Oak\XamlRundererPlugin.
2. Right-click either **DDraw** or **OpenGL** and choose **Rebuild**.
3. In the **Output** window, verify that the build succeeded by looking for the following output message:

```
===== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped
=====
```

4. Create an updated run-time image that includes the rebuilt graphics renderer.
 - a. On the **Build** menu, click **Copy Files to Release Directory**.
 - b. On the **Build** menu, click **Make Run-Time Image**.
 - c. In the **Output** window, verify that the build succeeded.

If `BSP_XRPLUGIN_OPENGL` is set, Windows Embedded Compact renames `XRRendererOpengl.dll` to `XamlRenderPlugin.dll` during the build process. If `BSP_XRPLUGIN_DDRAW` is set, Windows Embedded Compact renames `XRRendererDDraw.dll` to `XamlRenderPlugin.dll` during the build process. If none is set, no `XamlRenderPlugin.dll` is generated; therefore Silverlight for Windows Embedded uses an internal GDI renderer. Customers may provide their own implementation of `XamlRenderPlugin.dll`.

In your DLL, you must export the following names:

- `RenderPluginInitialize`
- `RenderPluginCleanup`
- `CreateRenderer`

You must implement your DLL to return **S_OK** when Silverlight for Windows Embedded calls **RenderPluginInitialize**.

Step 5: Include the Security Model

To help ensure that access to the DLL is secure, include the trusted security model and implement load privilege for `XamlRenderPlugin.dll`. To do this, do the following:

1. Add the Security Loader (`SYSGEN_LVMOD`) to the OS image. You can set this Sysgen variable at the command-line during a build.
2. Add one or more certificates to the codesign store on the device.
3. Ensure that the graphics-rendering DLL is either stored in ROM, or signed with a certificate that is in the codesign store on the device.

This helps prevent a malicious user from installing an unauthentic `XamlRenderPlugin.dll` that has malicious software that compromises all Silverlight applications on the device.

Otherwise, do not allow non-trusted applications to be installed on the device.

Step 6: Build the OS Design into a Run-Time Image

If you already have a run-time image, you can copy the DLL to the release directory, which is located under `%_WINCEROOT%\OSDesigns\<OS design>\RelDir`.

Otherwise, build the run-time image, and download it to the device through the connection that you have already configured.

Step 7: Use Hardware Acceleration in Your Application

In your Silverlight application, to use hardware acceleration, you must set the cache mode to bitmap caching only for UI elements that use hardware acceleration at run time. You can set cache mode in either the source XAML file or in the C++ application.

We recommend that you set the cache mode for all UI elements for which you implement transformations or animations, both of which require additional GPU processing.

To set the cache mode for a UI element

Do one of the following:

- In your XAML file, in the element, add a **CacheMode** attribute and set its value to "BitmapCache."

```
<Rectangle Fill="#7FFF0000" Stroke="Black" Width="200" Height="100"
  CacheMode="BitmapCache"/>
```

- In your application, locate the UI element in the visual tree, create an **IXRBitmapCache**, and then pass the **IXRBitmapCache** into **IXRUIElement::SetCacheMode**.

```
IXRListBoxPtr pElement;

pRoot->FindName(L"FancyListBox", &pElement);

IXRBitmapCachePtr pCache;

pApplication->CreateObject(&pCache);

pElement->SetCacheMode(pCache);
```

You can also reduce graphics memory consumption during graphics rendering by changing the default z-order of UI elements and by positioning all cached elements higher in the z-order.

In the Silverlight C++ API, application developers can specify a different z-order value in the **Canvas.ZIndex** attached property by calling **IXRDependencyObject::SetAttachedProperty(const WCHAR*, int)** on a UI element.

In Silverlight 3 XAML, designers can specify z-order as the value of the **Canvas.ZIndex** attribute in a XAML element.

Conclusion

Silverlight for Windows Embedded is a native (C++) UI development framework for Windows Embedded Compact devices that is based on Silverlight 3 for the desktop browser. By using Silverlight for Windows Embedded, you can design a UI for the shell and applications on a Windows Embedded Compact device that supports many features that are also available in Silverlight 3, such as storyboard animations, transformations, the Silverlight layout system, and a Silverlight visual tree.

With Silverlight for Windows Embedded, you can create a Silverlight application for an embedded device, create and add custom user controls to your application, and support hardware-accelerated graphics in the OS.

Additional Resources

- [Windows Embedded Compact 7 Documentation](http://go.microsoft.com/fwlink/?LinkId=190787) (http://go.microsoft.com/fwlink/?LinkId=190787)
- [A Sample Application Tutorial Using Windows Embedded Silverlight Tools](http://go.microsoft.com/fwlink/?LinkId=189508) (http://go.microsoft.com/fwlink/?LinkId=189508)

- [Microsoft Silverlight](http://go.microsoft.com/fwlink/?LinkId=162150) (http://go.microsoft.com/fwlink/?LinkId=162150)
- [Microsoft Silverlight Documentation](http://go.microsoft.com/fwlink/?LinkId=190786) (http://go.microsoft.com/fwlink/?LinkId=190786)
- [Windows Embedded website](http://go.microsoft.com/fwlink/?LinkId=183524) (http://go.microsoft.com/fwlink/?LinkId=183524)

This document is provided “as-is.” Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2011 Microsoft. All rights reserved.