1 Introduction

This paper explains how to write a Microsoft WPF application that displays on a dual screen configuration. Dual screen capability is useful when an application has content to be displayed on a secondary screen (TV or projector) that is controlled from the primary screen (laptop, Ultrabook™ system, Tablet or Phone). An example use case is displaying media files (pictures) on a TV from an Ultrabook device with Intel® WiDi technology. At times, this paper will refer to methods from the Intel® WiDi Extensions Library. To keep with the subject of dual screen capabilities in WPF, none of the details describing the necessary setup of code to use the Intel WiDi Extensions Library will be explained. The methods are only referenced to showcase the use of the Intel WiDi Extensions Library to develop a dual screen application.

2 Objectives

- Describe what is involved in creating a dual screen application.
- Show pitfalls to be aware of and how to address them.
- Demonstrate use of Intel WiDi Extensions Library to enhance the dual screen application user experience.
3 Table of Contents

1 Introduction ............................................................................................................................................. 1
2 Objectives .................................................................................................................................................. 1
3 Table of Contents ..................................................................................................................................... 2
4 Details ...................................................................................................................................................... 3
  4.1 Where to find screen information ........................................................................................................ 3
  4.2 How to position your WPF window on a secondary screen ................................................................. 4
  4.3 How to detect when a secondary screen becomes available ............................................................... 5
  4.4 Pit falls when maximizing windows and how to avoid them .............................................................. 6
  4.5 Using Intel® WiDi Extensions Library to create secondary screen in extended mode ...................... 8
  4.6 How to detect current display mode .................................................................................................... 9
  4.7 Using Intel® WiDi Extensions Library to change display mode ......................................................... 11
5 Summary ................................................................................................................................................. 11
4 Details

4.1 Where to find screen information

The Microsoft WPF framework does not have an API to access the screen information, but the older WinForms framework does. The `Screen` class in the System.Windows.Forms namespace provides information about the current available screens via the static `AllScreens` property. This property returns an array of `Screen` instances, one for each available screen. Note that the computer must be in the `Extended Desktop` mode for more than the primary screen to show up in the array returned by `AllScreens`. The `Primary` property on the `Screen` instances can be used to determine which is the primary screen and which is the secondary one. Below are a couple examples of how to use the `Screen` class to find the secondary screen.

Using LINQ query:

```csharp
Screen secondary = Screen.AllScreens.FirstOrDefault(screen => !screen.Primary);
```

Using standard for each loop:

```csharp
Screen secondary = null;
foreach (Screen screen in Screen.AllScreens)
{
    if (!screen.Primary)
    {
        secondary = screen;
        break;
    }
}
```

The examples above assume that the first non-primary screen is the target secondary screen. Some of the newer Ultrabook convertibles have two built-in monitors, which if configured as part of the extended desktop will complicate identifying the target external secondary screen. The `DeviceName Property` on the `Screen` instance does not give sufficient information to identify the target external secondary screen. To resolve this, the application may have to prompt the user to specify which screen to use as the target external secondary screen.
4.2 How to position your WPF window on a secondary screen

Once you have identified the target secondary screen, placing a WPF window on that screen is fairly simple. The Windows* extended desktop is one large virtual display with all the screens positioned within the virtual display coordinate system. Each Screen instance’s WorkingArea Property has the coordinates that define where the screen is located within the virtual display coordinate system. Placing a WPF window on the target secondary screen is accomplished by setting the Top and Left properties of the window to be located within the virtual display coordinate system where the target secondary screen resides. If your window is going to be maximized, simply set the Top and Left values for the window to the Top and Left values of the target secondary screen. Here is sample code showing this:

```
myWindow.Top = secondary.WorkingArea.Top;
myWindow.Left = secondary.WorkingArea.Left;
```

One place in the code to do this is the WPF window’s OnLoaded event of the Secondary Window.

```csharp
private void OnLoaded(object sender, RoutedEventArgs routedEventArgs)
{
    // Find the target secondary screen.
    Screen secondary = null;
    foreach (Screen screen in Screen.AllScreens)
    {
        if (!screen.Primary)
        {
            secondary = screen;
            break;
        }
    }
    // No target secondary screen found so abort opening the window.
    if (secondary == null)
    {
        MessageBox.Show("Unable to find a secondary screen to display window");
        this.Close();
        return;
    }
    // Set the location of the window so that it displays
    // on the target secondary screen.
    Top = secondary.WorkingArea.Top;
    Left = secondary.WorkingArea.Left;
    // Must wait to maximize window until after window is visible
    // so window maximizes on the correct screen and the WindowStyle
    // and ResizeMode setting have taken effect.
    WindowState = WindowState.Maximized;
}
```
4.3 How to detect when a secondary screen becomes available

When your application starts up the user may not have the system configured so that a secondary screen exists. If you want your application to auto detect when a secondary screen has become available, you can use the DisplaySettingsChanged event from Win32 as follows:

```csharp
using Microsoft.Win32;
using Forms = System.Windows.Forms;

public MainWindow()
{
    InitializeComponent();

    SystemEvents.DisplaySettingsChanged += SecondScreenCheck;
}

private void SecondScreenCheck(object sender, EventArgs eventArgs)
{
    // Make sure UI changes run on the main UI thread.
    this.Dispatcher.Invoke(() =>
    {
        if (Forms.Screen.AllScreens.Length > 1)
        {
            Do something to show dual screen capability available
        }
        else
        {
            Do something to show dual screen capability no longer available
        }
    });
}
```
4.4 Pitfalls when maximizing windows and how to avoid them

It is common styling for a dual screen application to maximize the window on the secondary screen without any boarder or frame. This is so the window isn’t seen and creates a distraction less viewing environment. The Windows OS has a couple of quirks when using this style, which are:

1. A small sliver of the window may still show at the bottom of the screen
2. The window maximized on the primary screen may bleed over onto the left edge of the secondary screen.
3. Setting `WindowState = WindowState.Maximized` in XAML alone does not allow WindowStyle and ResizeMode to be set prior to maximizing so these quirks may occur.

To avoid these quirks and maximize the window, you must set the `WindowStyle` and `ResizeMode` properties prior to maximizing the window. If the window is always going to be maximized, as is typically the case for the window on the secondary screen, these properties can be set in the windows XAML as follows:

```xml
<Window x:Class="DualScreenSampleApp.SecondaryWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    WindowStyle="None"
    ResizeMode="NoResize"
    Background="Black"
    Title="Secondary Window"/>
```

Or in the window’s constructor code behind as follows:

```csharp
public SecondaryWindow()
{
    InitializeComponent();

    // No window frame so window completely covers the screen when maximized
    WindowStyle = WindowStyle.None;

    // Removes all borders so no sliver shows at the bottom when maximized.
    ResizeMode = ResizeMode.NoResize;
}
```

Note that the `ResizeMode` property has to be set to `ResizeMode.NoResize` to avoid a sliver of the window shown at the bottom of the screen in some configurations. Also note that the window is not maximized at this point. This must wait until the window’s `OnLoaded` event, so it is properly placed on the target secondary screen and the `WindowStyle` and `ResizeMode` properties have been set; otherwise, it will be maximized on the primary screen and the above quirks may occur. See the `OnLoaded` event example in Section 4.2.
If your application is not maximized at startup and the user can double click the window frame or click the standard Maximize button, you must handle the `OnStateChanged` event and manually change `WindowState` back to `Normal`, set `WindowStyle` and `ResizeMode` as stated above, and then set `WindowState` to `Maximized` so the settings occur in the proper order to avoid the above quirks. Here is an example on handling the WPF window’s `OnStateChanged` event:

```csharp
private bool _manualMaximizing = false;

protected override void OnStateChanged(EventArgs e)
{
    if (_manualMaximizing)
    {
        _manualMaximizing = false;
        return;
    }

    if (WindowState != WindowState.Maximized)
        return;

    // Must, Must force the maximizing to go through our manual maximizing code,
    // so if user double clicks window frame to maximize, we must undo it and
    // call our manual method so all properties are set properly.
    WindowState = WindowState.Normal;

    // Remove all borders in maximized mode so app covers complete screen
    WindowStyle = WindowStyle.None;

    // On some monitors, windows leaves a sliver of the bottom border
    // showing when maximized. Setting ResizeMode to NoResize removes
    // the border completely.
    ResizeMode = ResizeMode.NoResize;

    // MUST, MUST set WindowStyle and ResizeMode before Maximizing window to
    // avoid frame bleeding over onto secondary screen and bottom sliver issues.
    _manualMaximizing = true;
    WindowState = WindowState.Maximized;
}
```

Note the use of the `_manualMaximizing` flag to know when the state change is caused by user versus the code.
4.5 Using Intel® WiDi Extensions Library to create the secondary screen in extended mode

When connecting to another monitor using WiDi, you can choose to connect in duplicate or extended mode. Typically dual screen applications will require the second screen to be in extended mode. The Intel WiDi Extensions Library offers a method to handle this. After creating a data member of type WiDiExtensions, call the StartConnectionToAdapter method on that data member, also passing in extended as a ScreenMode parameter. This method handles placing the WiDi connected monitor into extended mode. Here is an example of starting a connection to a receiver on the click of a button in the main window:

```csharp
private readonly IntelWiDiLib.WiDiExtensions widi;
private Window secondaryWindow;

public MainWindow()
{
    InitializeComponent();
}

// Event handler for clicking of the Widi Connect button
private void WidiConnectButton_OnClick(object sender, RoutedEventArgs e)
{
    try
    {
        // Connect to the designated receiver and
        // set the ScreenMode (SM) to extended when connection is made
        widi.StartConnectionToAdapter("ReceiverName", 0, 0, SM.Extended);
    }
    catch (Exception exception)
    {
        MessageBox.Show("There was an error connecting to the WiDi Receiver.");
    }
}

// This is a callback method and would be called
// after receiving the WiDiMessages.WM_WIDI_CONNECTED callback
private void AdapterConnected()
{
    foreach (var item in Screen.AllScreens)
    {
        if (!item.Primary)
        {
            secondaryWindow = new Window
            {
                // Set the location of the window so that it displays
                // on the target secondary screen.
                Top = item.WorkingArea.Top,
                Left = item.WorkingArea.Left,
            };
            break;
        }
    } // End foreach
```
The code above is fairly similar to the other code examples when dealing with the second screen. The only difference here is that when a call is made to connect you have to provide the name of the receiver, pass in zero value for both the source and target screen resolutions, and the ScreenMode that you want the monitor to display.

### 4.6 How to detect the current display mode

Another useful piece of information to know when developing a dual screen application is the display mode that the monitor is currently using. For example, knowing whether your secondary monitor is in clone or extended mode could help determine if the WiDi connected screen needs to be placed in extended mode or not (this topic will be visited in the next section). The following code uses P/Invoke to make a call to a User32 dll and display a message box to the user with the information acquired about the current display device. Here we can iterate through display devices and call the `EnumDisplayDevices` method with the correct parameters to get back information about each display device. The information that we are interested in is the devices’ `StateFlags` value.

```csharp
[DllImport("user32.dll")]
static extern bool EnumDisplayDevices(string lpDevice, uint iDevNum, ref DisplayDevice lpDisplayDevice, uint dwFlags);

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct DisplayDevice
{
    [MarshalAs(UnmanagedType.U4)]
    public int cb;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string DeviceName;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    public string DeviceString;
    [MarshalAs(UnmanagedType.U4)]
    public DisplayDeviceStateFlags StateFlags;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    public string DeviceID;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    public string DeviceKey;
}

[Flags]
public enum DisplayDeviceStateFlags
{
    AttachedToDesktop = 0x1,
    MultiDriver = 0x2,
    PrimaryDevice = 0x4,
    MirroringDriver = 0x8,
    VgaCompatible = 0x10,
}
```
// The device is removable; it cannot be the primary display.
Removable = 0x20,
// The device has more display modes than its output devices support.
ModesPruned = 0x8000000,
Remote = 0x4000000,
Disconnect = 0x2000000

public MainWindow()
{
    DisplayDevice device = new DisplayDevice();
device.cb = Marshal.SizeOf(device);

    try
    {
        for (uint id = 0; EnumDisplayDevices(null, id, ref device, 0); id++)
        {
            MessageBox.Show(String.Format("{0}\n, {1}\n, {2}\n", id,
                device.DeviceName,
                device.StateFlags));

            device.cb = Marshal.SizeOf(device);
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(String.Format("{0}", ex));
    }
}

If you have two monitors and the display mode is set to extended, the first device’s device.StateFlags
would output “AttachedToDesktop, PrimaryDevice, ModesPruned,” and the second device (the
secondary monitor) would display “AttachedToDesktop, ModesPruned.” On the other hand if your
display mode is clone/duplicate, then the first device’s device.StateFlags would output
“AttachedToDesktop, PrimaryDevice,” and the second would display “ModesPruned.” The difference
here is that if you are in extended mode, the second device will include a value of “AttachedToDesktop.”

For more information on using P/Invoke to call EnumDisplayDevices you can visit
http://msdn.microsoft.com/en-us/library/dd162609.aspx and
4.7 Using Intel® WiDi Extensions Library to change the display mode

One thing to take into consideration when building a dual screen app is the display mode of the secondary monitor. You may want to give the user a chance to clone the display for certain portions of your app and then change to extending the display or vice versa. The Intel WiDi Extensions Library offers another method that allows you to change the current screen mode. This method can be used with or without an active WiDi connection and can even be used without initializing the WiDi Extension. The following example will change, on the click of a button, the monitor’s screen mode from whatever the current mode is to extended mode.

```csharp
private readonly IntelWiDiLib.WiDiExtensions widi;

public MainWindow()
{
    InitializeComponent();
    widi = new WiDiExtensions();
    ...
}

// Event handler for clicking of Change ScreenMode button
private void ChangeScreenMode_OnClick(object sender, RoutedEventArgs e)
{
    try
    {
        widi.SetScreenMode(SM.Extended);
    }
    catch (Exception exception)
    {
        MessageBox.Show("There was an error while changing the screen mode.");
    }
}
```

Note: This code sample does not initialize the WiDi Extension Lib, which is required to connect to a WiDi receiver.

5 Summary

Dual screen applications can be developed using the techniques described above. This in conjunction with Intel WiDi creates a new compelling user experience. For a detailed sample of a dual screen application go to Reference Sample App source.

Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL
ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO
SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A
PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER
INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR
ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL
INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not
rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel
reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities
arising from future changes to them. The information here is subject to change without notice. Do not finalize a
design with this information.

The products described in this document 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.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your
product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature,
may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel
microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer
systems, components, software, operations, and functions. Any change to any of those factors may cause the
results to vary. You should consult other information and performance tests to assist you in fully evaluating your
contemplated purchases, including the performance of that product when combined with other products.

Any software source code reprinted in this document is furnished under a software license and may only be used
or copied in accordance with the terms of that license.

Intel, the Intel logo, and Ultrabook are trademarks of Intel Corporation in the U.S. and/or other countries.

Copyright © 2013 Intel Corporation. All rights reserved.

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