

Developing Windows* 8 Desktop Touch Apps with Windows* Presentation Foundation

By Bruno Sonnino

The launch of the Windows 8 operating system made touch a first-class citizen, and device manufacturers started to introduce new devices with touch-enabled displays. Such devices are now becoming more common and less expensive. In addition, manufacturers have introduced a new category of devices, 2 in 1 Ultrabook™ devices, which are lightweight and have enough power to be used as a traditional notebook with a keyboard and mouse or as a tablet using touch or a stylus.

These machines open new opportunities for developers. You can enable your apps for touch and make them easier to use and more friendly. When you create Windows Store apps for Windows 8, this functionality comes free. But what about desktop apps, the bread and butter of non-Web developers? You haven't been forgotten.

In fact, desktop apps are already touch enabled, and Windows Presentation Foundation (WPF) has had built-in support for touch since version 4.0, as you'll see in this article.

Design for Touch Apps

Microsoft has categorized touch for desktop apps according to three scenarios: good, better, and best.

Good Apps

With Windows 8, every desktop app has built-in touch support. All touch input is translated into mouse clicks, and you don't need to change your app for it. After you have created a desktop app, it will work with touch. Users can click buttons, select list items or text boxes with a finger or stylus, and can even use a virtual keyboard to input text if no physical keyboard is available. You can see this behavior with File Manager, the Calculator app, Microsoft Notepad, or any of your desktop apps.

Better Apps

The built-in touch behavior in Windows 8 is good and doesn't need much effort to develop, but it's not enough. You can go a step further by adding gesture support for your app. Gestures are one- or

two-finger actions that perform some predefined action—for example, tap to select, drag to move, flick to select the next or previous item, or pinch to zoom and rotate. See Figure 1.

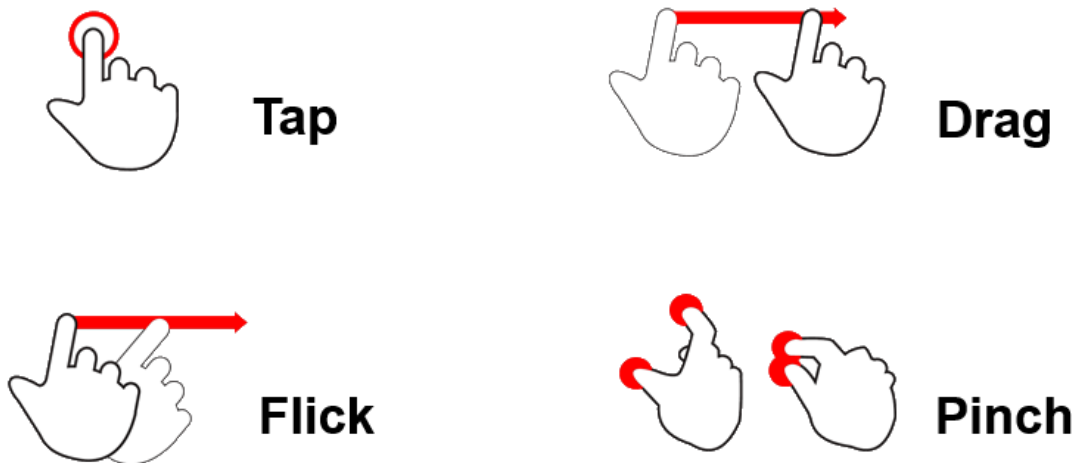


Figure 1. Common gestures

The operating system translates these gestures into the `WM_GESTURE` message. You can develop a program to handle this message and process the gestures, which will give your apps a bonus because you can support actions exclusive to touch-enabled devices.

Best Apps

At the pinnacle of Microsoft’s rating scheme, you can develop the best app for touch by designing it to support full touch functionality. Now, you may ask, “Why should I design for touch? Don’t my apps work well enough for touch?” The answer, most of the time, is no.

Apps designed for touch are different from conventional apps in several ways:

- ❑ The finger isn’t a mouse. It does not have the precision a mouse has and so the UI requires some redesign. Buttons, check boxes, and list items should be large enough that users can touch inside them with minimal error.
- ❑ Touch apps may not have a keyboard available. Although users can use a virtual keyboard, it’s not the same as the real thing. Rethinking the user interface (UI) to minimize keyboard input for touch apps can make the apps easier to use.
- ❑ Many simultaneous contacts can occur at the same time. With a mouse, the program has a single input point, but with touch apps, there may be more than one input. Depending on the device, the program could accept 40 or 50 simultaneous inputs (imagine a touch table with five or six players).
- ❑ Users can run the app in different orientations. Although traditional apps run in landscape, this is not true with touch apps. Users can rotate devices, and in some cases, there may be more than one user, such as one on either side of the device.

- ❑ Users don't have easy access to the whole device area. If a user is holding a tablet in his or her hands, it may be difficult to access the center of the device, because the user will have to hold it with one hand while touching it with the other one.

A “best” touch app must handle all of these issues and not abandon traditional data-entry methods with mouse and keyboard, or the app won't work on devices that don't have touch.

Touch Support in WPF

With WPF, you can add full touch support for your apps. You can add gestures or even full touch support with manipulations and inertia.

Adding Gestures to Your App

One way to add gestures to your apps is to process the WM_GESTURE message. The MTGestures sample in the Windows* 7 software development kit (SDK) shows how to do it. Just install the Windows 7 SDK and go to the samples directory (for the link, see the “For More Information” section at the end). Listing 1 shows the code.

Listing 1. Message processing in the MTGesture SDK sample

```
[PermissionSet(SecurityAction.Demand, Name = "FullTrust")]
protected override void WndProc(ref Message m)
{
    bool handled;
    handled = false;

    switch (m.Msg)
    {
        case WM_GESTURENOTIFY:
        {
            // This is the right place to define the list of gestures
            // that this application will support. By populating
            // GESTURECONFIG structure and calling SetGestureConfig
            // function. We can choose gestures that we want to
            // handle in our application. In this app we decide to
            // handle all gestures.
            GESTURECONFIG gc = new GESTURECONFIG();
            gc.dwID = 0; // gesture ID
            gc.dwWant = GC_ALLGESTURES; // settings related to gesture
            // ID that are to be turned on
            gc.dwBlock = 0; // settings related to gesture ID that are
            // to be

            // We must p/invoke into user32 [winuser.h]
            bool bResult = SetGestureConfig(
                Handle, // window for which configuration is specified
                0, // reserved, must be 0
                1, // count of GESTURECONFIG structures
                ref gc, // array of GESTURECONFIG structures, dwIDs
                // will be processed in the order specified
            );
        }
    }
}
```

```

        // and repeated occurrences will overwrite
        // previous ones
        _gestureConfigSize // sizeof(GESTURECONFIG)
    );

    if (!bResult)
    {
        throw new Exception("Error in execution of SetGestureConfig");
    }
}
handled = true;
break;

case WM_GESTURE:
    // The gesture processing code is implemented in
    // the DecodeGesture method
    handled = DecodeGesture(ref m);
    break;

default:
    handled = false;
    break;
}

// Filter message back up to parents.
base.WndProc(ref m);

if (handled)
{
    // Acknowledge event if handled.
    try
    {
        m.Result = new System.IntPtr(1);
    }
    catch (Exception excep)
    {
        Debug.Print("Could not allocate result ptr");
        Debug.Print(excep.ToString());
    }
}
}
}

```

You must override the window procedure, configure what kind of gestures you want when you receive the WM_GESTURENOTIFY message, and process the WM_GESTURE message.

As you can see, adding gestures to a C# app isn't a simple task. Fortunately, there are better ways to do it in WPF. WPF has support for the stylus and raises the StylusSystemGesture event when the system detects a touch gesture. Let's create a photo album that shows all photos in the Pictures folder and allows us to move between images by flicking to the right or left.

Create a new WPF app and add to the window three columns, two buttons, and an Image control. Listing 2 shows the code.

Listing 2. XAML markup for the new WPF app

```
<Grid>
```

```

<Grid.ColumnDefinitions>
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="40" />
</Grid.ColumnDefinitions>
<Button Grid.Column="0" Width="30" Height="30" Content="&lt;" />
<Button Grid.Column="2" Width="30" Height="30" Content="&gt;" />
<Image x:Name="MainImage" Grid.Column="1" />
</Grid>

```

Now, create a field named `_filesList` and another named `_currentFile`. See Listing 3.

Listing 3. Creating the `_filesList` and `_currentFile` fields

```

private List<string> _filesList;
private int _currentFile;

```

In the constructor of the main window, initialize `FilesList` with the list of files in the My Pictures folder. See Listing 4.

Listing 4. Main window constructor

```

public MainWindow()
{
    InitializeComponent();
    _filesList = Directory.GetFiles(Environment.GetFolderPath(
        Environment.SpecialFolder.MyPictures)).ToList();
    _currentFile = 0;
    UpdateImage();
}

```

`UpdateImage` updates the image with the current image, as shown in Listing 5.

Listing 5. Updating the image

```

private void UpdateImage()
{
    MainImage.Source = new BitmapImage(new Uri(_filesList[_currentFile]));
}

```

Then, you must create two functions to show the next and previous images. Listing 6 shows the code.

Listing 6. Functions to show the next and previous images

```

private void NextFile()
{
    _currentFile = _currentFile + 1 == _filesList.Count ? 0 : _currentFile + 1;
    UpdateImage();
}

private void PreviousFile()
{
    _currentFile = _currentFile == 0 ? _filesList.Count-1 : _currentFile - 1;
    UpdateImage();
}

```

The next step is to create the handlers for the Click event for the two buttons that call these functions.

In `MainWindow.xaml`, type the code in Listing 7.

Listing 7. Declaring the Click event handlers in `MainWindow.xaml`

```
<Button Grid.Column="0" Width="30" Height="30" Content="&lt;" Click="PrevClick"/>
<Button Grid.Column="2" Width="30" Height="30" Content="&gt;" Click="NextClick"/>
```

In `MainWindow.xaml.cs`, type the code in Listing 8.

Listing 8. Creating the Click event handlers in `MainWindow.xaml.cs`

```
private void PrevClick(object sender, RoutedEventArgs e)
{
    PreviousFile();
}

private void NextClick(object sender, RoutedEventArgs e)
{
    NextFile();
}
```

When you run the program, you will see that it shows the My Pictures images. Clicking the buttons allows you to cycle through the images. Now, you must add gesture support, which is simple. Just add the handler for the `StylusSystemGesture` event in the grid:

Listing 9. Declaring the `StylusSystemGesture` event handler in `MainWindow.xaml`

```
<Grid Background="Transparent" StylusSystemGesture="GridGesture" />
```

Note that I have added a background to the grid. If you don't do that, the grid won't receive the stylus events. The code of the handler is shown in Listing 10.

Listing 10. The grid handler

```
private void GridGesture(object sender, StylusSystemGestureEventArgs e)
{
    if (e.SystemGesture == SystemGesture.Drag)
        NextFile();
}
```

If you are following along with this article and performing the steps, you will notice that there is a `SystemGesture.Flick` that I didn't use. This gesture works only in Windows Vista*. Later Windows versions show the Drag gesture. You will also notice that I am not differentiating a forward flick from a backward one (or even horizontal from vertical). That's because there is no built-in support to do it, but we will take care of that next. Run the program and see that a flick in any direction brings up the next image.

To handle the direction of the flick, you must check its starting and end points. If the distance is larger in the horizontal direction, treat it as a horizontal flick. The sign of the difference between

the end and starting points shows the direction. Declare the handler for the `StylusDown` event for the grid in the `.xaml` file, as shown in Listing 11.

Listing 11. Declaring the `StylusDown` event for the grid

```
<Grid Background="Transparent"
      StylusSystemGesture="GridGesture"
      StylusDown="GridStylusDown">
```

The code for this handler is shown in Listing 12.

Listing 12. Creating the handler

```
private void GridStylusDown(object sender, StylusDownEventArgs e)
{
    _downPoints = e.GetStylusPoints(MainImage);
}
```

When the stylus is down, we store the contact points in the `_downPoints` array. You must modify the `StylusSystemGesture` event to get the direction for the flick. See Listing 13.

Listing 13. Modifying the `StylusSystemGesture` event

```
private void GridGesture(object sender, StylusSystemGestureEventArgs e)
{
    if (e.SystemGesture != SystemGesture.Drag)
        return;
    var newPoints = e.GetStylusPoints(MainImage);
    bool isReverse = false;
    if (newPoints.Count > 0 && _downPoints.Count > 0)
    {
        var distX = newPoints[0].X - _downPoints[0].X;
        var distY = newPoints[0].Y - _downPoints[0].Y;
        if (Math.Abs(distX) > Math.Abs(distY))
        {
            isReverse = distX < 0; // Horizontal
        }
        else
        {
            return; // Vertical
        }
    }
    if (isReverse)
        PreviousFile();
    else
        NextFile();
}
```

When the Drag gesture is detected, the program creates the new points and verifies the largest distance to determine whether it's horizontal or vertical. If it's vertical, the program doesn't do anything. If the distance is negative, then the direction is backwards. That way, the program can determine the kind of flick and its direction, going to the next or the previous image depending on the direction. The app now works for touch and the mouse.

Adding Touch Manipulation to a WPF App

Adding gestures to your app is a step in the right direction, but it's not enough. Users may want to perform complex manipulations, use more than one or two fingers, or want a physical behavior that mimics the real world. For that, WPF offers touch manipulations. Let's create a WPF touch app to see how it works.

In Microsoft Visual Studio*, create a new WPF app and change the window width and height to 800 and 600, respectively. Change the root component to a **Canvas**. You should have code similar to Listing 14 in `MainWindow.xaml`.

Listing 14. The new WPF app in Visual Studio

```
<Window x:Class="ImageTouch.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="600" Width="800">
    <Canvas x:Name="LayoutRoot">

    </Canvas>
</Window>
```

Go to the Solution Explorer and add an image to the project (right-click the project, click **Add/Existing Item**, and select an image from your disk). Add an image component to the main Canvas, assigning the `Source` property to the added image:

Listing 15. Image added to the main canvas

```
<Image x:Name="MainImage" Source="seattle.bmp" Width="400" />
```

If you run this program, you will see that it is already touch enabled. You can resize and move the window, and touch input is automatically converted to mouse input. However, that's not what you want. You want to use touch to move, rotate, and resize the image.

For that, you must use the `IsManipulationEnabled` property. When you set this property to true, the control receives touch events. The `ManipulationDelta` event is fired every time a manipulation in the control occurs. You must handle it and set the new properties of the image. In the .xaml file, set the property `IsManipulationEnabled` to true and declare a `ManipulationDelta` event, as shown in Listing 16.

Listing 16. Enabling touch manipulation

```
<Image x:Name="MainImage" Source="seattle.bmp" Width="400"
        IsManipulationEnabled="True"
        ManipulationDelta="MainImageManipulationDelta">
    <Image.RenderTransform>
        <MatrixTransform />
    </Image.RenderTransform>
</Image>
```


I have also added a `MatrixTransform` to the `RenderTransform` property. You change this transform when the user manipulates the image. The event handler should be similar to Listing 17.

Listing 17. Adding an event handler for image manipulation

```
private void MainImageManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
    FrameworkElement element = sender as FrameworkElement;
    if (element != null)
    {
        var transformMatrix = element.RenderTransform
            as MatrixTransform;
        var matrix = transformMatrix.Matrix;
        matrix.Translate(e.DeltaManipulation.Translation.X,
            e.DeltaManipulation.Translation.Y);
        ((MatrixTransform)element.RenderTransform).Matrix = matrix;
        e.Handled = true;
    }
}
```

Initially, you get the current `RenderTransform` of the image, use the `Translate` method to move it to the new position that the manipulation gives, and then assign it as the matrix for the `RenderTransform` of the image. At the end, you set the `Handled` property to `true` to tell WPF that this method has handled the touch event and WPF should not pass it on to other controls. This should allow the image to move when a user touches it.

If you run the app and try to move the image, you will see that it works but not as expected—the image flickers while moving. All manipulations are calculated relative to the image, but because this image is moving, you may have recursive recalculations. To change this behavior, you must tell WPF that all delta manipulations should be relative to the main window. You do so by using the `ManipulationStarting` event and setting the `ManipulationContainer` property of the event arguments to the `Canvas`.

In `MainWindow.xaml`, enter the code in Listing 18.

Listing 18. Correcting image movement in `MainWindow.xaml`

```
<Image x:Name="MainImage" Source="seattle.bmp" Width="400"
    IsManipulationEnabled="True"
    ManipulationDelta="MainImageManipulationDelta"
    ManipulationStarting="MainImageManipulationStarting">
```

In `MainWindow.xaml.cs`, enter the code in Listing 19.

Listing 19. Correcting image movement in `MainWindow.xaml.cs`

```
private void MainImageManipulationStarting(object sender, ManipulationStartingEventArgs e)
{
    e.ManipulationContainer = LayoutRoot;
}
```

Now, when you run the app and move the image, it moves with no flicker.

Adding Scaling and Rotation

To enable resizing and rotation, you must use the `Scale` and `Rotation` properties of the `DeltaManipulation`. These manipulations need a fixed center point. For example, if you fix the center point at the top left of the image, elements will be scaled and rotated around this point. To get a correct translation and rotation, you must set this point to the origin of the manipulation. You can set the correct scaling and rotation in code similar to Listing 20.

Listing 20. Setting scaling and rotation

```
private void MainImageManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
    FrameworkElement element = sender as FrameworkElement;
    if (element != null)
    {
        var transformMatrix = element.RenderTransform
            as MatrixTransform;
        var matrix = transformMatrix.Matrix;
        matrix.Translate(e.DeltaManipulation.Translation.X,
            e.DeltaManipulation.Translation.Y);
        var centerPoint = LayoutRoot.TranslatePoint(
            e.ManipulationOrigin, element);
        centerPoint = matrix.Transform(centerPoint);
        matrix.RotateAt(e.DeltaManipulation.Rotation,
            centerPoint.X, centerPoint.Y);
        matrix.ScaleAt(e.DeltaManipulation.Scale.X, e.DeltaManipulation.Scale.Y,
            centerPoint.X, centerPoint.Y);
        ((MatrixTransform)element.RenderTransform).Matrix = matrix;
        e.Handled = true;
    }
}
```

Adding Inertia

When you run the app, you will see that the image moves, scales, and rotates fine, but as soon as you stop moving the image, it stops. This is not the desired behavior. You want the same behavior you have when you move an image on a smooth table. It should continue moving slower and slower until it stops completely. You can achieve this effect by using the `ManipulationInertiaStarting` event. In this event, you state the desired deceleration in pixels (or degrees) per millisecond squared. If you set a smaller value, it will take longer for the element to stop (like on an icy table); if you set deceleration to a larger value, the object takes less time to stop (like on a rough table). Set this value to 0.005.

In `MainWindow.xaml`, enter the code in Listing 21.

Listing 21. Setting deceleration in `MainWindow.xaml`

```
<Image x:Name="MainImage" Source="seattle.bmp" Width="400"
```

```

IsManipulationEnabled="True"
ManipulationDelta="MainImageManipulationDelta"
ManipulationStarting="MainImageManipulationStarting"
ManipulationInertiaStarting="MainImageManipulationInertiaStarting"/>

```

In `MainWindow.xaml.cs`, enter the code in Listing 22.

Listing 22. Setting deceleration in `MainWindow.xaml.cs`

```

private void MainImageManipulationInertiaStarting(object sender,
    ManipulationInertiaStartingEventArgs e)
{
    e.RotationBehavior.DesiredDeceleration = 0.005; // degrees/ms^2
    e.TranslationBehavior.DesiredDeceleration = 0.005; // pixels/ms^2
}

```

Limiting the Inertial Movement

Now, when you run the app, you will see that the manipulations seem close to the physical behavior. But if you give the object a good flick, it goes out of the window, and you have to restart the program. To limit the inertial movement, you must determine whether the delta manipulation is inertial (the user has already lifted his or her finger) and stop it if it reaches the border. You do this with the code in the `ManipulationDelta` event handler, shown in Listing 23.

Listing 23. Limiting inertial movement

```

private void MainImageManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
    FrameworkElement element = sender as FrameworkElement;
    if (element != null)
    {
        Matrix matrix = new Matrix();
        MatrixTransform transformMatrix = element.RenderTransform
            as MatrixTransform;
        if (transformMatrix != null)
        {
            matrix = transformMatrix.Matrix;
        }
        matrix.Translate(e.DeltaManipulation.Translation.X,
            e.DeltaManipulation.Translation.Y);
        var centerPoint = new Point(element.ActualWidth / 2,
            element.ActualHeight / 2);
        matrix.RotateAt(e.DeltaManipulation.Rotation,
            centerPoint.X, centerPoint.Y);
        matrix.ScaleAt(e.DeltaManipulation.Scale.X, e.DeltaManipulation.Scale.Y,
            centerPoint.X, centerPoint.Y);
        element.RenderTransform = new MatrixTransform(matrix);

        var containerRect = new Rect(LayoutRoot.RenderSize);
        var elementRect = element.RenderTransform.TransformBounds(
            VisualTreeHelper.GetDrawing(element).Bounds);
        if (e.IsInertial && !containerRect.Contains(elementRect))
            e.Complete();
        e.Handled = true;
    }
}

```

```
}  
}
```

Now, determine whether the transformed image rectangle is in the container rectangle. If it isn't and the movement is inertial, stop the manipulation. That way, the movement stops, and the image doesn't go out of the window.

Conclusion

As you can see, adding touch manipulations to a WPF application is fairly easy. You can start with the default behavior and add gestures or full touch support with a few changes to the code. One important thing to do in any touch-enabled app is to rethink the UI so that users feel comfortable using it. You can also use different styles for the controls on a touch device, so the buttons are larger and the lists are more widely spaced only when using touch. With touch devices becoming increasingly common, optimizing your apps for touch will make them easier to use, thus pleasing your existing users and attracting new ones.

For More Information

- ❑ Windows 7 Training—Windows 7 Multi-Touch presentation:
<http://download.microsoft.com/download/8/C/D/8CD015BB-081B-49C5-A506-9C9B570B8DD2/Multitouch.pptx>
- ❑ Microsoft WinHEC 2008—Multi-Touch in Windows 7 presentation:
http://download.microsoft.com/download/8/6/A/86A97515-8AF4-4CA6-B516-CDA1139B8D93/MBL-T787_en.pptx
- ❑ Windows 7 Overview presentation:
<http://download.microsoft.com/download/5/2/2/522182D9-81B6-4D73-A58F-1092686CBC9E/Windows%207%20Overview.pptx>
- ❑ Windows 7 for Developers presentation:
http://download.microsoft.com/download/B/8/8/B8813050-D289-4A78-9A27-AA4ED49BCC65/2009-03-11_TechDays_MSDN_018.pptx
- ❑ Microsoft Windows SDK for Windows 7 and .NET Framework 3.5 SP1:
<http://www.microsoft.com/en-us/download/details.aspx?id=3138>

About the Author

Bruno Sonnino is a Microsoft Most Valuable Professional (MVP) located in Brazil. He is a developer, consultant, and author having written five Delphi books, published in Portuguese by Pearson Education Brazil, and many articles for Brazilian and American magazines and websites.

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.