<table>
<thead>
<tr>
<th>Mixing Stylus and Touch Input on Windows* 8</th>
<th>August 1 2013</th>
</tr>
</thead>
</table>

This document covers the use of touch and stylus interactions in a user interface and briefly examines when to use one over the other. It demonstrates how to implement moving images onto a canvas using touch, mouse, and pen features. In addition, it discusses how to use the pen to capture ink and how to use touch to manipulate images. The examples are displayed in the context of a simple educational tool for manipulating images. The application is written in C# and is designed for Windows* 8 devices.

Documentation for Developers Interested in Mixing Touch and Stylus Interactions
# Contents

Introduction .............................................................................................................................................. 3

Choose the Right Tool for the Situation at Hand .................................................................................. 3
  Touch Interaction ..................................................................................................................................... 3
  Stylus Interaction ................................................................................................................................... 4

The Example Application .......................................................................................................................... 5
  Supported Actions ................................................................................................................................... 6
  Development Environment .................................................................................................................... 6

Using Touch, Mouse and Pen to Drop Images onto a Canvas ................................................................. 6
Using the Pen to Capture Ink ................................................................................................................... 9
Using Touch to Manipulate Images ......................................................................................................... 12
Avoiding Problems with Multiple Input Methods and Multiple Interface Items .............................. 16
Closing .................................................................................................................................................... 16
Introduction
Alternatives to mouse and keyboard interfaces have broadened the style and scope of user interactions with software applications. Instead of replacing the traditional mouse and keyboard, hardware like touch screens and stylus pens augment traditional devices, giving users more intuitive and efficient ways of interacting with applications. Unlike mice and keyboards, which were forced into a jack-of-all-trades type status, touch gestures and pen input can focus on their strengths, while improving user interaction—and satisfaction—with their computing devices.

While touch devices may be more common than stylus, pen input is showing up on more and more smartphones, tablets, and convertible devices. One may be tempted to dismiss using a stylus as a lesser substitute for using a finger on a touch screen; however, each has its own place in the user interaction world.

This paper covers the use of touch and stylus interaction in a user interface and offers some guidelines for choosing when to use one over the other. Using a sample application, we will demonstrate how to implement using touch, mouse, and pen to move images onto a canvas area, as well as how to use the pen to capture ink and how to use multi-touch gestures to move, rotate, and resize images. The goal is to help application designers and developers handle user interaction that mixes stylus- and finger-based touch events. Specifically, it is intended to demonstrate how to react to a user switching between touch and stylus.

Our examples are set in the context of a simple educational tool designed to let users choose images from a palette and place them on a canvas. Once on the canvas, users can use touch and stylus to manipulate the images and add annotations to the canvas. The application is written in C# and is designed for Windows 8 devices.

Choose the Right Tool for the Situation at Hand

Touch Interaction
With the proliferation of smartphones, tablets, and computers with touchscreens, touch interaction has become synonymous with today’s modern devices and for a good reason. For many of the day-to-day interactions users have with their devices, touch is hard to beat. Touch is easy to learn, convenient, natural, and its gestures can be very rich, enabling users to easily express intentions that would be very slow or awkward with any other interaction method.

Touch strengths:

- One of the biggest advantages of touch interactions is the ability to use multiple input points (fingertips) at the same time. Multi-touch enables a richer set of motions and gestures than the single point provided by a stylus.
- Touch supports direct and natural interactions with the objects in the user interface. Through gestures such as tapping, dragging, sliding, pinching, and rotating, users can manipulate objects on the screen similar to physical objects.
- Touch allows users to combine gestures to perform more than one action at a time (compound manipulations). For example, a user can rotate a picture while simultaneously moving it across the screen.
• Users can interact with the application without the need to first pick up a device, like a mouse or a stylus.
• Users rarely misplace their fingers or have their batteries go dead at the wrong time.

**Stylus Interaction**
Some people argue users have no need for a stylus; after all, most of us already come equipped with ten built-in touch devices and it is highly unlikely we will misplace any of them, as is often the case with a stylus. However, when it comes to precision and accuracy using a finger falls far short when compared to a stylus.

Stylus strengths:
• With a stylus you can select a single X/Y coordinate. The size of the contact area of a finger is too large to do this.
• The shape and size of the contact area does not change during movement, unlike the tip of a finger.
• It’s easier to keep the target object in one spot while holding the stylus (touch users naturally move their fingers even while trying to maintain a single location).
• With a stylus, the user’s hand travels a shorter physical distance than the cursor on the screen. This means it is easier to perform a straight-line motion than with a finger.
• Because the tip of the stylus does not obscure (occlude) the screen, the user interface can display a cursor to assist with targeting.
• Similarly, styli do not occlude the target spot, making it easier for users to see where they are placing items.
• A stylus can incorporate a 3-state model where it is on the screen, off of the screen, or near the screen (hovering). Hover can be used to display tooltips when the stylus passes over an item or to indicate which items can be selected. Touch does not have the concept of hover.
• An application can utilize stylus pressure to add another dimension to the interaction. For example, the amount of pressure can be used by the application to define the width of a line being drawn on the screen.
• Because of the smaller tip on a stylus, user interface controls can be placed in places that may be harder to reach than with a finger, such as close to the edge of the screen.

Situations where the precision of a stylus is an advantage over touch:
• Taking notes by hand (rather than using a keyboard)
• Creating mathematical or scientific notes containing formulas
• Drawing in a more natural way
• Marking up documents
• Recording signatures (digital ink)
• Selecting small, closely spaced items
• Precisely placing an item on an image board or screen
• Being able to see where an item will be placed (because the target is not obscured by the pointing device)
The Example Application
In this paper we demonstrate the touch and stylus concepts and how to implement them through a simple educational application. This example lets users capture handwritten annotations about images they have placed on the screen.

Imagine a bulletin board where users can place images anywhere on the board. The images can be moved, sized, and rotated. Using a stylus, users can make handwritten notes and diagrams on the board itself, outside of the pictures. For example, a user can write a caption below a picture or draw an arrow showing the relationship between two images.

The application features a main drawing area (the bulletin board) in the center with a palette of pre-defined images and line colors along the edges. Users use touch and drag to move a copy of an image from the palette to the drawing area or to a new location on the area. The application supports standard multi-touch manipulations to move, rotate, and size the images on the board. Anything “drawn” on the board with a stylus will automatically appear directly on the board, in the color selected in the color palette. A “Clear” button removes the images and drawings from the drawing area so the user can start afresh.

Figure 1: Sample application with mixed touch and stylus interaction
**Supported Actions**
The following table describes the user interactions for the sample application.

**Table 1: Supported Actions**

<table>
<thead>
<tr>
<th>Action</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Touch and drag image to drawing area</td>
<td>Place copy of the image on the drawing area</td>
</tr>
<tr>
<td>Finger drag on existing image in drawing area</td>
<td>Move image</td>
</tr>
<tr>
<td>Pinch on existing image in drawing area</td>
<td>Decrease size of image</td>
</tr>
<tr>
<td>Spread on existing image in drawing area</td>
<td>Increase size of image</td>
</tr>
<tr>
<td>Two-finger rotation on existing image in drawing area</td>
<td>Rotate image</td>
</tr>
<tr>
<td>Touch color on color palette</td>
<td>Select color to use when drawing a line</td>
</tr>
<tr>
<td>Stylus draw on drawing area</td>
<td>Draw line on drawing area using current color</td>
</tr>
<tr>
<td>Touch [Clear] button</td>
<td>Remove images and drawings from drawing area</td>
</tr>
</tbody>
</table>

**Development Environment**
This application is a Windows Store app, sporting the Windows 8 Modern Interface, suitable for any tablet, convertible, or Ultrabook™ device with touch and stylus support.

<table>
<thead>
<tr>
<th>Windows® 8</th>
<th>Language</th>
<th>.NET</th>
<th>IDE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C# and XAML</td>
<td>.Net for Windows Store apps</td>
<td>Visual Studio® 2012 for Windows 8</td>
</tr>
</tbody>
</table>

**Figure 2: Development environment**

**Using Touch, Mouse, and Pen to Drop Images onto a Canvas**
We start by populating our image picker using the AddImage method. This method simply adds the provided image instance to the image picker (a canvas) and defines its starting location. We also make sure to update our custom slider’s height when adding a new image. A custom slider is necessary because a ScrollView would interfere with being able to drag the image.

```csharp
private void AddImage(Image img)
{
    this.PickerStack.Children.Add(img);

    double left = 7.5;
    double top = 7.5 + (157.5 * m_images.Count);

    Canvas.SetLeft(img, left);
}```
The first thing we need in order to be able to drop images onto the central canvas is to provide image dragging functionality. To do this, we add handlers for the image object’s pointer events and feed the resulting event data into a GestureRecognizer instance configured to look for translation gestures. Note that we do not check the pointer device’s type as we decided to allow the user to perform picture dropping using any pointer device. However, you can check the device’s type if you want to restrict certain actions to specific device types.

```csharp
void Image_PointerPressed(object sender, PointerRoutedEventArgs e)
{
    if (m_activeImage != null || m_sliderActive) return;
    Image img = sender as Image;
    if (img != null )
    {
        m_activeImage = img;
        Canvas.SetZIndex(m_activeImage, 1);
        m_activePosition = m_imagePositions[m_activeImage];
        m_gestureRecognizer.ProcessDownEvent(e.GetCurrentPoint(img));
        e.Handled = true;
    }
}

void Image_PointerMoved(object sender, PointerRoutedEventArgs e) { ... }
```

In addition to feeding the pointer released event to the gesture recognizer, we also evaluate the pointer’s position relative to the destination canvas, as well as to the picker. If the release event took place outside
of the picker's bounds and inside of the target canvas, we raise the PictureDropped event to let the target implementation know it should add a new image instance at the given position.

```csharp
void Image_PointerReleased(object sender, PointerRoutedEventArgs e)
{
    if (m_activeImage == null || m_sliderActive) return;
    Image img = sender as Image;
    if (img != null)
    {
        PointerPoint imgPoint = e.GetCurrentPoint(img);
        m_gestureRecognizer.ProcessUpEvent(imgPoint);
        m_gestureRecognizer.CompleteGesture();
        e.Handled = true;

        if (m_droppingTarget != null
            && PictureDropped != null)
        {
            PointerPoint canvasPoint = e.GetCurrentPoint(m_droppingTarget);
            PointerPoint pickerPoint = e.GetCurrentPoint(this);
            Rect canvasRect = new Rect(0.0, 0.0,
                                         this.DropTarget.ActualWidth, this.DropTarget.ActualHeight);
            Rect pickerRect = new Rect(0.0, 0.0, this.ActualWidth,
                                        this.ActualHeight);

            if (ContainedIn(canvasPoint, canvasRect) &&
                !ContainedIn(pickerPoint, pickerRect))
            {
                Point imgPos = new Point(canvasPoint.Position.X -
                                           imgPoint.Position.X, canvasPoint.Position.Y -
                                           imgPoint.Position.Y);
                this.PictureDropped(this,
                                    new PictureDropEventArgs(img.Source, imgPos));
            }
        }
    }
    Canvas.SetZIndex(m_activeImage, 0);
    m_activeImage = null;
    m_activePosition = null;
}
```
Notice that when given pointer events occur, the GestureRecognizer instance translates them into *manipulation started*, *updated*, and *completed* events. We use those events to simply reposition the image using the static Canvas SetTop and SetLeft methods.

```csharp
void m_gestureRecognizer_ManipulationStarted(GestureRecognizer sender, ManipulationStartedEventArgs args)
{
    Point p = args.Cumulative.Translation;
    Canvas.SetLeft(m_activeImage, m_activePosition.X + p.X);
    Canvas.SetTop(m_activeImage, m_activePosition.Y + p.Y - m_itemOffset);
}

void m_gestureRecognizer_ManipulationUpdated(GestureRecognizer sender, ManipulationUpdatedEventArgs args)
{
    Point p = args.Cumulative.Translation;
    Canvas.SetLeft(m_activeImage, m_activePosition.X + p.X);
    Canvas.SetTop(m_activeImage, m_activePosition.Y + p.Y - m_itemOffset);
}
```

In contrast to the *manipulation started* and *updated* event handlers, the *manipulation completed* event handler just restores the image to its original position instead of using the event’s cumulative translation to reposition the image.

```csharp
ManipulationCompletedEventArgs args)
{
    Canvas.SetLeft(m_activeImage, m_activePosition.X);
    Canvas.SetTop(m_activeImage, m_activePosition.Y - m_itemOffset);
}
```

We intentionally omitted the code showing how we set up the picker’s target canvas and handle picture drops, as this is beyond the scope of this document. To find out more about the implementation details, please review the sample application code.

**Using the Pen to Capture Ink**

First thing to note while capturing ink input in a Windows Store app using C#, is that there is no dedicated class for ink rendering. You do however get an InkManager class that is capable of translating raw pointer input into ink strokes you can use for rendering. The sample application provides a basic implementation of an ink renderer that can be used to render and clear ink strokes. For a more complete sample please
With a working ink *renderer implementation* in place, all you have to do to capture ink input is handle the target's pointer pressed, moved, released, entered, and exited events. Note that we use the same code to handle pointer pressed and entered events. The same goes for released and exited events. The trick while reusing those events is to make sure that the pointer is in contact with the digitizer using the `IsInContact` property.

```csharp
private void inkCanvas_PointerPressed(object sender, PointerRoutedEventArgs e)
{
    if (m_activePointerId != 0) return;
    if (e.Pointer.PointerDeviceType == PointerDeviceType.Pen
        && e.Pointer.IsInContact)
    {
        PointerPoint pointerPoint = e.GetCurrentPoint(this.inkCanvas);
        m_renderer.StartRendering(pointerPoint, m_inkAttr);
        m_inkMan.Mode = InkManipulationMode.Inking;
        m_inkMan.ProcessPointerDown(pointerPoint);
        m_activePointerId = e.Pointer.PointerId;
        e.Handled = true;
    }
}
```

In the pointer pressed event handler we first check if we already have an active pointer device used to capture ink input. If not, we check the device type to make sure we are dealing with a pen device and extract the `PointerPoint` instance relative to our ink input canvas. We then make our *renderer implementation* live ink input and pass the pointer point to our `InkManager` instance for processing. We also make sure to store the source’s pointer device id for future reference and mark the event as handled, to prevent it from being propagated to other UI items.

```csharp
private void inkCanvas_PointerMoved(object sender, PointerRoutedEventArgs e)
{
    if (m_activePointerId != e.Pointer.PointerId) return;
    if (e.Pointer(PointerDeviceType == PointerDeviceType.Pen)
    {
        if (e.Pointer.IsInContact)
        {
            PointerPoint pointerPoint =
                e.GetCurrentPoint(this.inkCanvas);
```

In the pointer moved event handler we check to see if the source pointer device ID matches the ID of the pointer device, which we use to capture ink input. We also check the pointer device type just in case, although if the active ID matches the source’s device ID, we can be pretty sure that the device is indeed a pen and could just as well skip the additional if statement. Next, we check that the device is indeed in contact with its digitizer. If it is not, we treat the event as a released event to prevent the application from becoming caught with an inactive pointer device, in some rare situations.

With the device identified as a pen that is in contact with its digitizer, we simply pass in the device’s current position relative to our ink canvas to update the simplified live stroke. The thing to remember with pointer moved events, since we are not dealing with a real-time operating system, is that there is no guarantee that they will get fired for every single hardware pen position change. This does not mean however, that we cannot have a precise rendering of the user’s input. In turn, we simply use the event argument’s GetIntermediatePoints to get a collection of all the aggregated pointer position changes and push that into our InkManager for processing. Passing in all of the intermediate points will result in a much better representation of the actual strokes once we switch from live to permanent ink rendering.

```csharp
m_renderer.UpdateStroek(pointerPoint);

IList<PointerPoint> interPointerPoints =
e.GetIntermediatePoints(this.inkCanvas);
for (int i = interPointerPoints.Count - 1; i >= 0; --i)
    m_inkMan.ProcessPointerUpdate(interPointerPoints[i]);
e.Handled = true;
}
else
{
    HandlePenUp(e);
}
}

private void inkCanvas_PointerReleased(object sender,
    PointerRoutedEventArgs e)
{
    if (m_activePointerId != e.Pointer.PointerId) return;
    if (e.Pointer.PointerDeviceType == PointerDeviceType.Pen)
    {
        HandlePenUp(e);
    }
}
In the pointer released event handler, we again check if we are dealing with the device marked as the current active ink input device and check the device type just in case. The heavy lifting, however, has been moved to a private helper method to prevent any code duplication because, as you remember, the pointer moved event could also be interpreted as a released event in certain circumstances.

```csharp
private void HandlePenUp(PointerRoutedEventArgs e)
{
    PointerPoint pointerPoint = e.GetCurrentPoint(this.inkCanvas);
    m_inkMan.ProcessPointerUp(pointerPoint);
    m_renderer.FinishRendering(pointerPoint);
    IReadOnlyList<InkStroke> strokes = m_inkMan.GetStrokes();
    int lastStrokeIndex = strokes.Count - 1;
    if (lastStrokeIndex >= 0)
        m_renderer.AddPermaInk(strokes[lastStrokeIndex], m_inkAttr);
    m_activePointerId = 0;
    e.Handled = true;
}
```

In the HandlePenUp helper method we first extract the pointer up position relative to our ink canvas. We then pass the up pointer point for processing by our InkManager and make our ink renderer finish live ink rendering. We then get all the strokes handled by our InkManager instance and pass the last one to the ink renderer. The renderer uses this data to produce a detailed stroke rendering using Bezier curves.

**Using Touch to Manipulate Images**

As there are no convenience functions that would help us with generating a scale, rotation, and translation transformation matrix, the first thing we need when implementing image touch manipulations are some matrix helper functions. As you can see in the listings below the matrix helper functions are not very complicated and do a very good job at simplifying your manipulation code.

```csharp
private Matrix Rotation(double angle)
{
    double angleRad = Rad(angle);
    Matrix r = Matrix.Identity;
    r.M11 = Math.Cos(angleRad);
    r.M21 = -Math.Sin(angleRad);
    r.M12 = Math.Sin(angleRad);
    r.M22 = Math.Cos(angleRad);
    return r;
}

private Matrix Translation(double x, double y)
{
```
Matrix r = Matrix.Identity;
r.OffsetX = x;
r.OffsetY = y;
return r;
}

private Matrix Scale(double scale)
{
    Matrix r = Matrix.Identity;
    r.M11 = scale;
    r.M22 = scale;
    return r;
}

private double Rad(double angle)
{
    return (Math.PI * angle) / 180.0;
}

private Matrix MatMull(Matrix a, Matrix b)
{
    Matrix r = Matrix.Identity;
    r.M22 = (a.M21 * b.M12) + (a.M22 * b.M22);
    return r;
}

With the matrix helper code now in place we can proceed to passing in the pointer points into a properly set-up GestureRecognizer instance. The code is mostly identical to the one used for dragging pictures from the image picker, the only difference is we must check the device type this time as we only plan to support touch. In addition, the pointer released handler is much simpler here as we do not have to deal with restoring the image to its original position and firing any custom events.

private void m_image_PointerPressed(object sender, PointerRoutedEventArgs e)
{
if (e.Pointer.PointerDeviceType ==
h)
{
    e.Handled = true;
    m_gestureRecognizer.ProcessDownEvent(e.GetCurrentPoint(m_image));
}
else
{
    e.Handled = false;
}

private void m_image_PointerMoved(object sender, PointerRoutedEventArgs e)
{
    ...}

private void m_image_PointerReleased(object sender, PointerRoutedEventArgs e)
{
    if (e.Pointer.PointerDeviceType ==
h)
    {
        e.Handled = true;
        m_gestureRecognizer.ProcessUpEvent(e.GetCurrentPoint(m_image));
        m_gestureRecognizer.CompleteGesture();
    }
    else
    {
        e.Handled = false;
    }
}

With the pointer events handled it is now time to consume the gesture recognizer manipulation started,
updated, and completed events. Once we get a manipulation started event, we begin by setting the
images z index to 2, so it is rendered over any other image in the canvas and is first in line for pointer
events. We then make sure to store the item’s current transformation matrix and proceed to creating the
new scale, rotation, and translation matrices using the previously defined helper functions. Once we have
all of the individual matrices calculated, we combine them using the MatMull helper function and use the
resulting matrix to set up the image's transformation.

private void m_gestureRecognizer_ManipulationStarted(
    GestureRecognizer sender, ManipulationStartedEventArgs args)
{
Canvas.SetZIndex(m_image, 2);
m_inMatrix = (m_image.RenderTransform as MatrixTransform).Matrix;
Matrix scaleMatrix = Scale(args.Cumulative.Scale);
Matrix rotationMatrix = Rotation(args.Cumulative.Rotation);
Matrix translationMatrix = Translation(args.Cumulative.Translation.X,
    args.Cumulative.Translation.Y);

Matrix mat = MatMull(MatMull(MatMull(m_originToTranslation,
    scaleMatrix), translationMatrix), rotationMatrix),
    m_originFromTranslation);
(m_image.RenderTransform as MatrixTransform).Matrix =
MatMull(mat, m_inMatrix);
}

private void m_gestureRecognizer_ManipulationUpdated(  
    GestureRecognizer sender, ManipulationUpdatedEventArgs args)  
{
    ...  
}

private void m_gestureRecognizer_ManipulationCompleted(  
    GestureRecognizer sender, ManipulationCompletedEventArgs args)  
{
    Canvas.SetZIndex(m_image, 1);
    Matrix scaleMatrix = Scale(args.Cumulative.Scale);
    Matrix rotationMatrix = Rotation(args.Cumulative.Rotation);
    Matrix translationMatrix = Translation(args.Cumulative.Translation.X,
        args.Cumulative.Translation.Y);

    Matrix mat = MatMull(MatMull(MatMull(m_originToTranslation,
        scaleMatrix), translationMatrix), rotationMatrix),
        m_originFromTranslation);
    (m_image.RenderTransform as MatrixTransform).Matrix =
    MatMull(mat, m_inMatrix);
}

The *manipulation completed* event handler has the same flow as its *started* and *updated* counterparts, with the small exception that it resets the image’s z-index to stop the image from being rendered on top of all other items and possibly hijacking their pointer events.
Avoiding Problems with Multiple Input Methods and Multiple Interface Items

In summary, here are some hints on avoiding problems when handling multiple input devices across user interface items:

- Always remember to set the pointer event’s handled property to true if you do not want to propagate it to other items.
- Try to keep track of the pointer entered and exited events; some pointer devices, for example graphical tablets, get different pointer IDs when the stylus goes in and out of range.
- Remember to call CompleteGesture on your GestureRecognizer objects when feasible. Sometimes when working with multi-touch, pointers may abruptly leave the item’s scope, resulting in a dirty GestureRecognizer state. Calling CompleteGesture will help you restore the GestureRecognizer state to a clean condition and avoid future manipulation glitches.
- Remember that some stylus drivers block touch input while in use; this helps in cases where users rest their hands on the screen while using the stylus. In general, assume that stylus input takes priority.

Closing

Touch, pen, keyboard, and mouse inputs are all valid ways of interacting with an application, each has strengths and weaknesses. This paper focused on touch and stylus interactions, noting that touch offers a natural, easy to learn, direct manipulation style of interaction that enables users to combine gestures to express more than one command simultaneously. In contrast, stylus interactions work well when the application needs more accuracy than touch or when the user needs to write or draw.

As demonstrated, incorporating mixed touch and stylus interaction in an application is not difficult, providing a few simple guidelines are followed and you keep track of what component is handling each event. You can download the full source for the demo application at and then try it yourself, or use it as reference material to create your own touch/stylus app.

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.