Using Accelerometer in Windows 8* Metro Style App and a Case Study of Tap Detection

Outlines

1. Introduction ............................................................................................................. 2
2. Accelerometer ......................................................................................................... 2
   2.1. Capabilities ...................................................................................................... 2
   2.2. Use cases ........................................................................................................ 3
3. Develop environment for accelerometer data access .............................................. 4
   3.1. Hardware ....................................................................................................... 4
   3.2. Software ....................................................................................................... 4
4. Case Study – Tap Detection .................................................................................. 8
   4.1. Experimental data .......................................................................................... 8
   4.2. State machine ............................................................................................... 11
   4.3. Detect a tap .................................................................................................... 16
   4.4. Detect left/right tap with slight device movement ......................................... 17
5. Conclusion ............................................................................................................ 18
6. About the Author .................................................................................................. 19
7. Notices ................................................................................................................ 19
1. Introduction

Accelerometers are hardware sensors embedded in many modern portable devices, like smartphones and tablets. They measure the acceleration of a device in weight per unit of mass with respect to the earth’s gravity at sea level. They can be widely used for motion tracking, fall detection, gesture detection, etc. This article covers the use of accelerometers in Windows 8* Metro style applications and a case study on how to use accelerometers to detect tapping on the edges of a tablet. In addition, the case study highlights how to detect the direction of taps that result in slight device movement.

2. Accelerometer

There are many types of accelerometers, for example optical accelerometer, magnetic induction accelerometer, laser accelerometer, etc. Typically, modern portable devices have general micro-electro mechanical system (MEMS)-based accelerometers. We focus on the typical MEMS-based accelerometers found in portable devices like tablets and smartphones. This kind of accelerometer has some basic and/or advanced capabilities depending on the model. In this section we introduce basic and advanced capabilities and classify several use cases where accelerometers have been used in mobile apps.

2.1. Capabilities

The capabilities we discuss here are the ones supported by the accelerometer device drivers, as opposed to capabilities provided by the operating system or user applications.

2.1.1. Basic Capability

The basic capability of a typical MEMS-based accelerometer is to measure the acceleration with reference to the earth’s gravity at the sea level. The unit is G-force (g), where 1 g is equal to 9.81 m/s². The key characteristics of the basic capability along with the associated terminologies are as follows:

- Number of axes: the number of acceleration axes in Euclidean space that the accelerometer outputs. For 2-dimensional positioning, two is enough and generally the accelerometer outputs x and y directional G-force values. For 3-dimensional positioning, three axes are needed so the accelerometer outputs x, y, and z directional G-force values.
- Maximum range: the range of the maximum and minimum measurable G-force values. Generally, it is ±2 g, ±4 g, ±6 g or even up to ±24 g.
- Sensitivity: how much change the accelerometer output signal will have for a given change in acceleration. Generally, the more sensitive, the better.
- Output data rate or bandwidth: how frequently we can take reliable readings. It can range from several Hz to several hundreds of Hz.
- Operating temperature range: the temperature range that the accelerometer can have reliable readings. The MEMS-based accelerometer on portable devices is normally from -40 degree to 85 degree.

### 2.1.2. Advanced Features

Some accelerometers have an embedded controller. The controller can read the accelerometer data in a high frequency (several hundreds of Hz) and analyze the data in order to provide advanced features like low-power mode, shake detection, tap detection, etc.

- **Low-power mode**: Some accelerometers provide a low-power mode feature for further reducing the power consumption of accelerometer. It allows the accelerometer to sleep internally longer and averages the data less frequently when no activity is detected.

- **Shake detection**: Some accelerometers can detect a fast shake motion. By analyzing the sequence of acceleration changes, they can determine whether a fast shake motion occurs.

- **Tap detection**: Some accelerometers can detect taps. The tap here is defined as not only the tapping on the touch screen, but tapping on the edge of the device. Whenever a user taps an edge of the device, the embedded controller in the accelerometer detects it and sends an interrupt.

Some accelerometers have more features, for example, freelfall detection. However, additional features make them more expensive. Accelerometers on typical mobile devices do not have many advanced features. For example, tap detection is commonly not available on most accelerometers on the market.

### 2.2. Use Cases

Accelerometer data can be used in many scenarios, and developers can find many creative uses for the data. We list several useful use cases here.

#### 2.2.1 Tracking Device Movement

Accelerometer data can be used to track device movement. This is usually done by sensor fusion with the gyroscope, compass, and other sensors together. The movement tracking is utilized in gesture detection, pedometer, drawing in space, remote controller, camera station ability, etc.

#### 2.2.2. Detection of Free Fall of Device
If a device is in free fall, the acceleration magnitude is below the threshold. This is because a free falling device is accelerating to the ground with earth’s gravity. As the accelerometer data is referencing earth’s gravity, the magnitude of the acceleration is around zero for a free fall.

2.2.3. Motion Detection

Accelerometers can be used for motion detection like shake, tap, and even press. This is because each motion triggers a specific accelerometer data change pattern. Motions are detected when a specific pattern of accelerometer data occurs. The motion detections are utilized in many application scenarios like adjusting voice volume by tapping, zooming in/out by double tapping, showing images by shaking, etc.

3. Development Environment for Accelerometer Data Access

3.1. Hardware

Our experiments were conducted on a typical tablet with an MEMS-based accelerometer made by STMicroelectronics. This accelerometer can provide three axis G-force values with respect to the x (lateral), y (longitudinal), and z (vertical) axes. The sampling rate is 50 Hz or higher.

3.2. Software

3.2.1. Operating System and Development Tool

Windows 8 release preview is the operating system installed on the target tablet. We use Visual Studio* RC 2012 to develop a Metro style C# app for the accelerometer case study.

3.2.2. Accelerometer Access

Windows Runtime API provides the access to the supported accelerometer via an accelerometer class in Windows.Devices.Sensors namespace for Metro style apps.

First, we declare an accelerometer instance with the Accelerometer class. Then we set up the minimum report interval for the accelerometer. Note that the minimum report interval is different from the available sampling rate of the accelerometer in hardware specification. The minimum report interval is the minimum reading interval allowed by Windows Runtime for the accelerometer. The allowed minimum report interval for the accelerometer in Windows Runtime is 16 milliseconds. This implies the maximum accelerometer reading frequency is 62.5 Hz. The frequency might be
far less than the typical sampling rate of the accelerometer hardware because too high a sampling frequency would overload the CPU.

We set the allowed minimum report interval as the default value (16 milliseconds) for the accelerometer instance. Then we add the event handler function for reacting to any accelerometer reading changes. The sample code is shown below.

```csharp
// Initialize accelerometer
_accelerometer = Accelerometer.GetDefault();

if (_accelerometer != null)
{
    // Initialize accelerometer stable values. Note that it will do automatical
    // calibration later.
    TapDetector.InitializeReading(0, 0, 0);

    // Establish the report interval for all scenarios.
    minReportInterval = _accelerometer.MinimumReportInterval;
    reportInterval = minReportInterval > 16 ? minReportInterval : 16;
    _accelerometer.ReportInterval = reportInterval;
}

if (_accelerometer != null)
{
    _accelerometer.ReadingChanged += new TypedEventHandler<Accelerometer, AccelerometerReadingChangedEventArgs>(ReadingChanged);
}
else
{
    doneCalibrate = 0;
    DisplayAppOutput("No accelerometer found");
}

Sample code 1: Initialize accelerometer instance and register accelerometer data change event handler function

3.2.3. Data Acquisition

In Windows Runtime, whenever the accelerometer data are changed, an accelerometer event occurs. The event is handled by the registered handler function, and the accelerometer data readings are passed as arguments to the function. In the code sample, ReadingChanged works as the event handler function and receives the arguments. The event handler argument includes the current G-force values with respect to the x (lateral), y (longitudinal), and z (vertical) axes and the accelerometer data change timestamp. Here we show the sample code for obtaining the three-axis accelerometer data.
3.2.4. Accelerometer Three Axis Data

What do accelerometer data look like? All the data are the G-force data with unit g in a pre-defined coordination system. The pre-defined coordinate system in Windows 8 Runtime is the same as the one in Silverlight*, SNA, or the Windows Phone operating system.

The coordinate system is shown in Figure 1. This is a Windows 8 tablet on a static surface, facing up with the camera at the top of the device. If the device has acceleration directed to the left along the x axis, the x value accessed through the accelerometer class is positive. Otherwise, it’s negative. If the device has acceleration directed to the bottom of the device, the y value from the accelerometer class is positive. Otherwise, it’s negative. If the device has acceleration directed to the center of the earth and the acceleration is higher than 1 g, the z value accessed through accelerometer class is positive. Otherwise, it is negative.

Sample code 2: Retrieve the accelerometer data reading and the timestamp **

```csharp
// Initialize accelerometer
private void ReadingChanged(object sender, AccelerometerReadingChangedEventArgs e)
{
    Dispatcher.RunAsync(CoreDispatcherPriority.High, () =>
    {
        AccelerometerReading reading = e.Reading;

        AcceleroData_X.Text = String.Format("{0,5:0.00}", reading.AccelerationX);
        AcceleroData_Y.Text = String.Format("{0,5:0.00}", reading.AccelerationY);
        AcceleroData_Z.Text = String.Format("{0,5:0.00}", reading.AccelerationZ);
        AcceleroData_TimeStamp = reading.TimeStamp;
    }
}
```
Ideally, when the tablet is placed in a stationary status on a flat surface, the x and y values are 0 and the z value is -1 g as the accelerometer is always referencing earth’s gravity at sea level.

We obtain the real accelerometer data from Windows Runtime APIs for a typical tablet in a stationary status on a flat surface and plot them in the graph, shown in Figure 2. As we can see, the x, y, z values are not the ideal values (0, 0, -1) as the surface is not 100% flat.

Figure 1: Accelerometer data coordinate system in Windows Runtime.
4. Case Study – Tap Detection

Windows Runtime provides access to the typical accelerometer data readings for Metro apps with the minimum report interval of 16 ms (the maximum report frequency is 62.5 Hz). With the report frequency at 62.5 Hz, it is challenging to find patterns for some motions if the motions trigger only slight accelerometer changes. However, it is still feasible to detect motions with consistent patterns. Our case study shows the feasibility of using the low-frequency accelerometer data to detect taps on the edge of the device and directional taps with slight device movement. Both trigger slight accelerometer data changes.

4.1. Experimental Data

We experimented with a static tablet on a flat surface facing up with the camera at the top of the device. As we tapped the edges of the device, we used the Windows Runtime API to collect the reported accelerometer data and plotted the data according to the time of accelerometer data reports.
4.1.1. Taps

Figure 3 plots the accelerometer data when we tap twice on each edge of the device (left, right, top, and bottom). We call the taps left, right, top, and bottom taps, respectively, and use different colors of oval cycling to depict when taps happen.

The x, y, and/or z axes data consistently change when a tap happens. If we use the accelerometer data when the device is static as a baseline and calculate the delta of the accelerometer data changes w.r.t. the baseline, we notice some pulses are triggered on the x, y, and/or z axis. The pulses could be used as the criteria to determine whether a tap occurred.

Figure 3: Accelerometer x, y, z axis data for two taps on each edge of the device (left, right, top, and bottom) without any device movement
4.1.2. Directional taps

We did experiments to find the consistent patterns for directional taps. However, the directional taps without any device movement show inconsistent changes of accelerometer data with the 62.5 Hz data reading frequency. To detect the directional taps without any device movement requires a sampling rate of more than 400 Hz. Therefore, we experimented with the feasibility of detecting directional taps when the tap resulted in a slight device movement.

In Figure 3, for the left and right taps, the accelerometer data changes mainly on the x axis as the device is facing up on a stationary flat surface. Figure 4 plots the accelerometer data when we tap on the left edge of the device (left taps). Figure 5 plots the accelerometer data when we tap on the right edge of the device (right taps). We ignore the z axis in both figures as there is little change on it.

As we can see, one left tap in Figure 4 triggers two consecutive pulses on the x axis. The first one is a negative pulse followed by a positive pulse. This is because a left tap slightly moves the device with acceleration to the right first and then decelerates because of friction.

Similarly, a right tap in Figure 5 triggers two consecutive pulses on the x axis. The first one is the positive pulse followed by a negative pulse. This is because a right tap makes the device move to the left and then decelerates due to friction. With the pulse patterns, we can determine whether a left or right tap happens.

![Accelerometer Reading Samples](image)

**Figure 4:** Accelerometer x and y axis data for two left taps on the left edge of the device with slight device movement.
The experimental data shows that it is feasible to detect left/right taps with slight device movement by identifying two consecutive pulses on the x axis and the order of positive/negative pulses. Next, we show that we can use a state machine to implement the detection process.

4.2. State Machine

We use a simple state machine to identify whether the current state is in a positive pulse or negative pulse or non-active state. We first define three states of the accelerometer data as NonActive, PositivePulse, NegativePulse. NonActive means the change is below a threshold (we define \( \text{deltax}_\text{threshold} \) in the sample code). The PositivePulse means the change exceeds the threshold and the x-axis G-force is larger than the one in NonActive state. The NegativePulse means the change exceeds the threshold and the x-axis G-force is less than the one in the NonActive state.

The state machine is shown in Figure 6. Let \( x \) be the G-force data on the x-axis reported by the accelerometer. We define \( \Delta x = |x - x_{\text{initial}}| \), where \( x_{\text{initial}} \) is the average G-force on the x axis when the device is static. We compare \( \Delta x \) with a pre-
defined threshold (Th) to identify whether the accelerometer data change is big enough to constitute a tap. The states are transitioned as in Figure 6.

![State machine for the accelerometer x-axis data when the device is placed on a flat surface.](image)

**Figure 6:** State machine for the accelerometer x-axis data when the device is placed on a flat surface.

As shown in the following sample code, we first define the `DetectTap` class with members.
To obtain the G-force values for the device in the NonActive state, we sample a small amount of accelerometer data and average them. We call this step calibration. Calibration is done by the user pressing a button when the device is placed on a flat surface. The calibration function is defined in the DetectTap class. The average G-force values for the NonActive state are used to calculate the change on each axis for tap detection.

Sample code 3: Definition of DetectTap class and members **

```java
public class DetectTap
{
    // Accelerometer data state
    enum PulseState
    {
        NonActive = 0,
        PositivePulse = 1,
        NegativePulse = 2,
        UnknownPulse = -1
    }

    // Threshold on x, y, z axes for identifying whether a tap triggers the
    // accelerometer data change exceeds a threshold or not.
    private static double deltax_threshold = 0.01;
    private static double deltay_threshold = 0.02;
    private static double deltaz_threshold = 0.03;

    // Declare the average x, y, z accelerometer data when device is static.
    public double x_initial;
    public double y_initial;
    public double z_initial;

    // Declare the number of samples to calibrate the x_initial, y_initial, z_initial.
    public int samplecounter_calibrate;
    // Declare the maximum number of samples for calibration.
    public int MaxCalibrateInterval = 10;
    // Declare the previous state, current state of x axis accelerometer data.
    private static PulseState PreviousState_X, CurrentState_X;

    // Ininitialization
    public DetectTap()
    {
        samplecounter_calibrate = 0;
    }
}
```
Further, the DetectTap class includes the DetectXPulse function to output the current state of x-axis data based on the changes in the x-axis. The current state output by the function is used for detecting left/right tap with slight device movement.

```java
public class DetectTap
{
    // Accelerometer calibration for the NonActive state.
    public int CalibrateInitialReading(double x, double y, double z)
    {
        int done = 0;

        // Initialize the variables.
        if (samplecounter_calibrate == 0)
        {
            x_initial = 0;
            y_initial = 0;
            z_initial = 0;
        }

        // Increment the sample number of calibration.
        samplecounter_calibrate++;

        // Skip the first 5 samples and then average the rest samplings of
        // accelerometer data. The skipping is to skip the accelerometer data
        // change due to the button press for calibration.
        if (samplecounter_calibrate > 5
                && samplecounter_calibrate <= MaxCalibrateInterval)
        {
            x_initial = (x_initial * (samplecounter_calibrate - 6) + x) /
                        (samplecounter_calibrate - 5);
            y_initial = (y_initial * (samplecounter_calibrate - 6) + y) /
                        (samplecounter_calibrate - 5);
            z_initial = (z_initial * (samplecounter_calibrate - 6) + z) /
                        (samplecounter_calibrate - 5);
        }

        if (samplecounter_calibrate >= MaxCalibrateInterval)
        {
            done = 1;
        }

        return done;
    }
}
```

Sample code 4: Definition of calibration function **
public class DetectTap
{

    // State machine to detect the pulse on x axis accelerometer data.
    public int DetectXPulse(double x)
    {
        double deltax;
        deltax = x - x_initial;

        if (Math.Abs(deltax) < deltax_threshold)
        {
            CurrentState_X = PulseState.NonActive;
            goto Exit;
        }

        if (Math.Abs(deltax) > Math.Abs(DeltaxPeak))
            DeltaxPeak = deltax;

        switch (PreviousState_X)
        {
            case PulseState.PositivePulse:
                if (deltax > 0)
                    CurrentState_X = PulseState.PositivePulse;
                else
                    CurrentState_X = PulseState.NegativePulse;
                break;

            case PulseState.NegativePulse:
                if (deltax > 0)
                    CurrentState_X = PulseState.PositivePulse;
                else
                    CurrentState_X = PulseState.NegativePulse;
                break;

            case PulseState.NonActive:
                if (deltax > 0)
                    CurrentState_X = PulseState.PositivePulse;
                else
                    CurrentState_X = PulseState.NegativePulse;
                break;
            default:
                break;
        }

    Exit:

        PreviousState_X = CurrentState_X;

        return (int)CurrentState_X;
    }
}
We insert the tap detection code into the accelerometer change event handler function ReadingChanged.

First, we use the current accelerometer data with the state machine function DetectXPulse to get the current x-axis state and put it into a first-in-first-out queue with limited size. The states in the queue will be the data for identifying the taps. The sample code for saving the current state into a queue is shown below.

```csharp
private void ReadingChanged(object sender, AccelerometerReadingChangedEventArgs e) {
    Dispatcher.RunAsync(CoreDispatcherPriority.High, () => {
        AccelerometerReading reading = e.Reading;

        int currentXState, currentYState, currentZState;

        if (doneCalibrate != 1) {
            // Do calibration
        } else {

            // Only keep small amount of history X, Y, Z data.
            if (TapXStateQueue.Count >= MaxQueueLength) TapXStateQueue.Dequeue();
            if (TapYStateQueue.Count >= MaxQueueLength) TapYStateQueue.Dequeue();
            if (TapZStateQueue.Count >= MaxQueueLength) TapZStateQueue.Dequeue();

            // Use the state machine to detect positive or negative pulse on x, y, z axes. // Put the current state in a first in first out queue.
            currentXState = TapDetector.DetectXPulse((double)(reading.AccelerationX));
            TapXStateQueue.Enqueue(currentXState);
            currentYState = TapDetector.DetectYPulse((double)(reading.AccelerationY));
            TapYStateQueue.Enqueue(currentYState);
            currentZState = TapDetector.DetectZPulse((double)(reading.AccelerationZ));
            TapZStateQueue.Enqueue(currentZState);

            TapInterval++;
            SingleXYZTapInterval++;
            ....
        }
    });
}
```

**Sample code 6: Save current state into a state queue for tap detection**

### 4.3. Detect a Tap

When a tap happens, a pulse is triggered in the x, y, or z axis. Thus, we use the current state of x, y, and z axis accelerometer data to identify a tap. If one state is PositivePulse or NegativePulse, we determine a tap is detected. Then we de-bounce the detection for a small time window. The sample code is as follows.
4.4. Detect Left/Right Tap with Slight Device Movement

When a left/right tap causes slight device movement on a static device on a flat surface, two consecutive pulses are triggered on x axis. We use the x-axis states saved in the queue to detect the left/right taps. The sample code is as follows.

```csharp
// … continue with Dispatcher.RunAsync in function ReadingChanged
// The below code detects single tap based on the x, y, z direction data
// pulse. If a pulse is detected in x, y, z (in this order),
// we identify a tap is detected. Then we debounce the tap for a time window
// MinSingleTapInterval.

// Debouncing condition
if (TapXStateQueue.Count >= MaxQueueLength && (SingleXYZTapInterval >= MinSingleTapInterval)) {
    // We identify x direction pulse by the X state from state
    // machine. The sequence only can be NegativePulse, PositivePulse.
    if (currentXState == 1 || currentXState == 2 ||
        currentYState == 1 || currentYState == 2 ||
        currentZState == 1 || currentZState == 2) {
        SingleXYZTapInterval = 0;
    }
    // Tap is detected based on the pulse detection on x, y, z direction.
    if (SingleXYZTapInterval == 0) {
        TapCount++;
        NonDTapEventOutput.Text = TapCount.ToString() + " Tap detected";
    }
}
```

Sample code 7: Detect a tap by identifying a pulse **
In this article, we introduce the capabilities and some use cases for MEMS-based accelerometers in modern portable devices. We also introduce the access method in

```csharp
// ... continue with Dispatcher.RunAsync in function ReadingChanged
// The below code detects directional tap based on the x direction data pulse
// sequence. If a pulse sequence is detected in x, we identify a directional tap is
// detected. Then we debounce the tap for a time window MinTapInterval.
if (TapXStateQueue.Count >= MaxQueueLength && (TapInterval >= MinTapInterval)) {
    // We identify x direction negative pulse by the X state.
    // The sequence for left tap is a negative pulse followed by a positive pulse.
    // So, the state sequence could be : 2,1 or 2,0,1, or 2,0,0,1.
    if (((TapXStateQueue.ElementAt(MaxQueueLength - 3) == 2 &&
        TapXStateQueue.ElementAt(MaxQueueLength - 2) == 0 &&
        TapXStateQueue.ElementAt(MaxQueueLength - 1) == 1)
        | (TapXStateQueue.ElementAt(MaxQueueLength - 4) == 2 &&
           TapXStateQueue.ElementAt(MaxQueueLength - 3) == 0 &&
           TapXStateQueue.ElementAt(MaxQueueLength - 2) == 0 &&
           TapXStateQueue.ElementAt(MaxQueueLength - 1) == 1)
        | (TapXStateQueue.ElementAt(MaxQueueLength - 2) == 2 &&)
           TapXStateQueue.ElementAt(MaxQueueLength - 1) == 1) {)
        LeftTapCount++;,
        TapEventOutput.Text = LeftTapCount.ToString() + " Left Tap";
        TapInterval = 0;
        DirectionalTap = 1;
}

    // We identify x direction positive pulse by the X state sequence from state
    // machine. The sequence for right tap is a positive pulse followed by a negative
    // pulse. So, the state sequence could be : 1,2 or 1,0,2, or 1,0,0,2.
    if (((TapXStateQueue.ElementAt(MaxQueueLength - 3) == 1 &&
        TapXStateQueue.ElementAt(MaxQueueLength - 2) == 0 &&
        TapXStateQueue.ElementAt(MaxQueueLength - 1) == 2)
        | (TapXStateQueue.ElementAt(MaxQueueLength - 4) == 1 &&
           TapXStateQueue.ElementAt(MaxQueueLength - 3) == 0 &&
           TapXStateQueue.ElementAt(MaxQueueLength - 2) == 0 &&
           TapXStateQueue.ElementAt(MaxQueueLength - 1) == 2)
        | (TapXStateQueue.ElementAt(MaxQueueLength - 2) == 1 &&
           TapXStateQueue.ElementAt(MaxQueueLength - 1) == 2)) {)
        RightTapCount++;,
        TapEventOutput.Text = RightTapCount.ToString() + " Right Tap";
        TapInterval = 0;
        DirectionalTap = 2;
}

    if (TapInterval == 0)
        Dispatcher.RunAsync(CoreDispatcherPriority.High, (DispatchedHandler)OnAnimation);
}
}} // End of function ReadingChanged
```

Sample code 8: Detect a left/right tap with slight device movement by identifying particular pulse sequence **

5. Conclusion

In this article, we introduce the capabilities and some use cases for MEMS-based accelerometers in modern portable devices. We also introduce the access method in
Windows Runtime for Metro style apps. We show a case study with sample codes showing that it is feasible to detect tapping on the edges of devices and the left/right direction when taps cause slight device movement.

About the Author

Sushu Zhang is a software engineer in Intel Corporation’s Software and Services Group. She is working on ISV scale enabling for Intel-based platforms with Android* and Windows OSs. She developed Intel® Power Monitoring Tool for Android Devices. Most recently, Sushu has been involved with several energy-efficiency projects at Intel. Prior to joining Intel, Sushu worked at Microsoft on Windows energy efficiency. Sushu earned a Ph.D in Computer Science at the Arizona State University. Her research area was system level power and thermal management.

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 and the Intel logo are trademarks of Intel Corporation in the US and/or other
countries.
Copyright © 2012 Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
**This sample source code is released under the Intel Sample Source Code License Agreement.