Contents

Introduction ............................................................................................................................... 1

Programming Choices for Windows 8 .................................................................................. 1

Sensors ..................................................................................................................................... 2

   Sensor Fusion ....................................................................................................................... 3

Identifying Sensors .................................................................................................................. 5

   Generating New GUIDs ...................................................................................................... 6

Using Sensor Manager Object ............................................................................................... 7

   Ask by Type ........................................................................................................................ 7

   Ask by Category .................................................................................................................. 8

   Ask by Category “All” ......................................................................................................... 9

Sensor Life Cycle – Enter and Leave Events ......................................................................... 10

   Enter Event Callback ........................................................................................................ 10

   Leave Event ......................................................................................................................... 11

Picking Sensors for an App ................................................................................................... 13

   Sensor Properties .............................................................................................................. 14

   Setting Sensor Sensitivity ................................................................................................. 16

   Requesting permissions for Sensors ............................................................................... 17

   Sensor Data Update ........................................................................................................... 18

Using Sensors in Windows Store apps ................................................................................ 21

Summary ................................................................................................................................. 21

Appendix .................................................................................................................................. 22

   Coordinate System for Different Form Factors ............................................................... 22

   Resources ............................................................................................................................ 23

   About the Authors .............................................................................................................. 23

Notices ...................................................................................................................................... 24
List of Tables
Table 1  Ultrabook Features Matrix for Windows 8 Developer Environments ........................................ 2
Table 2  Sensor types and categories ........................................................................................................ 5
Table 3  Example of Constants and unique Globally Unique IDs (GUIDs) ............................................. 6
Table 4  Data Field identifier constants .................................................................................................... 13
Table 5  Commonly used sensor Properties and PIDs ........................................................................ 15
Table 6  Sensor APIs for Metro Style Apps, pros and cons .................................................................... 21

List of Figures
Figure 1  Window Store Apps and Desktop Sensor frameworks in Windows 8 ................................. 1
Figure 2  Different sensors supported on Windows 8 ............................................................................ 3
Figure 3  Sensor fusion via combining output from multiple sensors .................................................. 4
Figure 4  Defining new GUIDs for value add sensors ......................................................................... 7
Introduction
This guide gives developers an overview of the Microsoft Windows 8 sensors application programming interfaces (APIs) for Windows 8 Desktop and Windows Store applications with a specific focus on the various sensor capabilities available in Windows 8 Desktop mode. This Development Guide summarizes the APIs that enable creating interactive applications by including some of the common sensors such as accelerometers, magnetometers, and gyroscopes with Windows 8.

Programming Choices for Windows 8
Developers have multiple API choices to program sensors on Windows 8. The new touch-friendly app environment, called “Windows Store apps” is shown on the left in Figure 1. The only API library that Windows Store apps can use is new with Windows 8 and called WinRT (Windows Run-Time). The WinRT sensor API represents a portion of the overall WinRT library. For more details, please refer to the MSDN Sensor API library: http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.sensors.aspx

Traditional Win Forms, or MFC-style apps, shown on the right, are now called “Desktop Apps” because they run in the Desktop Windows Manager environment. Desktop apps can either use the native Win32*/COM API, a .NET-style API or a subset of select WinRT APIs.

Figure 1 Windows Store Apps and Desktop Sensor frameworks in Windows 8*
The following is a list of WinRT APIs that can be accessed by Desktop apps:

- Windows.Device.Geolocation (GPS)
- Windows.UI.Notifications.ToastNotification
- Windows.Globalization
- Windows.Security.Authentication.OnlineId (including LiveID integration)
- Windows.Security.CryptographicBuffer (useful binary encoding/decoding functions)
- Windows.ApplicationModel.DataTransfer.Clipboard (access and monitor Windows 8 Clipboard)

In both cases, the APIs go through a Windows middleware component called the Windows Sensor Framework. The Windows Sensor Framework defines the Sensor Object Model. The different APIs “bind” to that object model in slightly different ways.

Differences in the Desktop and Windows Store app development will be discussed later in this document. For brevity, we will consider only Desktop app development. For Windows Store app development, please refer to [http://msdn.microsoft.com/library/windows/apps/br211369](http://msdn.microsoft.com/library/windows/apps/br211369).

### Sensors

There are many kinds of sensors, but we are interested in the ones required for Windows 8, namely accelerometers, gyroscopes, ambient light sensors, compass, and GPS. Windows 8 represents the physical sensors with object-oriented abstractions. To manipulate the sensors, programmers use APIs to interact with the objects. The following table provides information on how the sensors can be accessed from both the Windows 8 Desktop apps as well as from Windows Store apps.

<table>
<thead>
<tr>
<th>Feature/Toolset</th>
<th>Windows 8 Desktop Mode Apps</th>
<th>Windows Store Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C++</td>
<td>C#/VB</td>
</tr>
<tr>
<td><strong>Orientation Sensors</strong> (accelerometer, inclinometer, gyrometer)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Light Sensor</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>NFC</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>GPS</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Below, Figure 2 identifies that there are more objects than actual hardware. Windows defines some “logical sensor” objects by combining information from multiple physical sensors. This is called “Sensor Fusion.”

**Sensor Fusion**

Physical sensor chips have some inherent natural limitations. For example:

- Accelerometers measure linear acceleration, which is a measurement of the combined relative motion and the force of Earth’s gravity. If you want to know the computer’s tilt, you’ll have to do some mathematical calculations.

- Magnetometers measure the strength of magnetic fields, which indicate the location of the Earth’s Magnetic North Pole.

These measurements are subject to an inherent drift problem, which can be corrected by using raw data from the Gyro. Both measurements are (scaled) dependent upon the tilt of the computer from level with respect to the Earth’s surface. For example, to obtain the computer’s heading with respect to the Earth’s True North Pole (Magnetic North Pole is in a different position and moves over time), corrections must be applied.

Sensor Fusion (Figure 3) is defined by obtaining raw data from multiple physical sensors, especially the Accelerometer, Gyro, and Magnetometer, performing mathematical calculations to correct for natural sensor limitations, computing more human-usable data, and representing those as logical sensor abstractions. The application developer must implement the necessary transformations required to translate physical sensor data to the abstract sensor data. If the system design has a SensorHub, the fusion operations will take place inside the microcontroller firmware. If the system design does not have
a SensorHub, the fusion operations must be done inside one or more device drivers that the IHVs and/or OEMs provide.

**Figure 3** Sensor fusion via combining output from multiple sensors
Identifying Sensors
To manipulate a sensor, a system is needed to identify and refer to. The Windows Sensor Framework defines a number of categories that sensors are grouped into. It also defines a large number of specific sensor types. Table 2 lists some of the sensors available for Desktop applications.

Table 2 Sensor types and categories

<table>
<thead>
<tr>
<th>Biometric</th>
<th>Electrical</th>
<th>Environmental</th>
<th>Light</th>
<th>Location</th>
<th>Mechanical</th>
<th>Motion</th>
<th>Orientation</th>
<th>Scanner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human Presence</td>
<td>Capacitance</td>
<td>Atmospheric Pressure</td>
<td>Ambient Light</td>
<td>Broadcast</td>
<td>Boolean Switch</td>
<td>Accelerometer 1D</td>
<td>Compass 1D</td>
<td>Barcode</td>
</tr>
<tr>
<td>Human Proximity*</td>
<td>Current</td>
<td>Humidity</td>
<td>Gps</td>
<td>Boolean Switch Array</td>
<td>Accelerometer 2D</td>
<td>Compass 2D</td>
<td>Rfid</td>
<td></td>
</tr>
<tr>
<td>Touch</td>
<td>Electrical Power</td>
<td>Temperature</td>
<td>Static</td>
<td>Force</td>
<td>Accelerometer 3D</td>
<td>Compass 3D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inductance</td>
<td>Wind Direction</td>
<td>Multivalue Switch</td>
<td>Gyrometer 1D</td>
<td>Device Orientation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Potentiometer</td>
<td>Wind Speed</td>
<td>Pressure</td>
<td>Gyrometer 2D</td>
<td>Distance 1D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resistance</td>
<td></td>
<td>Strain</td>
<td>Gyrometer 3D</td>
<td>Distance 2D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voltage</td>
<td></td>
<td>Weight</td>
<td>Motion Detector</td>
<td>Distance 3D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Speedometer</td>
<td>Inclinometer 1D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inclinometer 2D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inclinometer 3D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The sensor types required by Windows 8 are shown in **bold** font:

- Accelerometer, Gyro, Compass, and Ambient Light are the required “real/physical” sensors
- Device Orientation and Inclinometer are the required “virtual/fusion” sensors (note that the Compass also includes fusion-enhanced/tilt-compensated data)
- GPS is a required sensor if a WWAN radio exists, otherwise GPS is optional
- Human Proximity is an oft-mentioned possible addition to the required list, but, for now, it’s not required.
All of these constants correspond to Globally Unique IDs (GUIDs.) Below, in Table 3, is a sample of some of the sensor categories and types, the names of the constants for Win32/COM and .NET, and their underlying GUID values.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Constant (Win32*/COM)</th>
<th>Constant (.NET)</th>
<th>GUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category &quot;All&quot;</td>
<td>SENSOR_CATEGORY_ALL</td>
<td>SensorCategories.SensorCategoryAll</td>
<td>{C317C286-C468-4288-9975-04D4A587CC42}</td>
</tr>
<tr>
<td>Category Biometric</td>
<td>SENSOR_CATEGORY_BIOMETRIC</td>
<td>SensorCategories.SensorCategoryBiometric</td>
<td>{CA19690F-A2C7-477D-A99E-99EC6E2B5648}</td>
</tr>
<tr>
<td>Category Electrical</td>
<td>SENSOR_CATEGORY_ELECTRICAL</td>
<td>SensorCategories.SensorCategoryElectrical</td>
<td>{FB73FCDB-FC4A-483C-AC58-27B691C6BEFF}</td>
</tr>
<tr>
<td>Category Environmental</td>
<td>SENSOR_CATEGORY_ENVIRONMENTAL</td>
<td>SensorCategories.SensorCategoryEnvironmental</td>
<td>{323439AA-7F66-492B-8A0C-73E9A9A65D55}</td>
</tr>
<tr>
<td>Category Light</td>
<td>SENSOR_CATEGORY_LIGHT</td>
<td>SensorCategories.SensorCategoryLight</td>
<td>{17A665CD-9063-4216-B202-5C7A255E18CE}</td>
</tr>
<tr>
<td>Category Location</td>
<td>SENSOR_CATEGORY_LOCATION</td>
<td>SensorCategories.SensorCategoryLocation</td>
<td>{BFA794E4-F964-4FD8-90F6-5105686E4B44}</td>
</tr>
<tr>
<td>Category Mechanical</td>
<td>SENSOR_CATEGORY_MECHANICAL</td>
<td>SensorCategories.SensorCategoryMechanical</td>
<td>{8D131DDB-8EF7-4656-8085-CCC0B9379C}</td>
</tr>
<tr>
<td>Category Motion</td>
<td>SENSOR_CATEGORY_MOTION</td>
<td>SensorCategories.SensorCategoryMotion</td>
<td>{CD09DAF1-382E-4C3D-B598-85EF593FD046}</td>
</tr>
<tr>
<td>Category Orientation</td>
<td>SENSOR_CATEGORY_ORIENTATION</td>
<td>SensorCategories.SensorCategoryOrientation</td>
<td>{9E6CD4B6-96FE-4954-8726-68882A473F6F}</td>
</tr>
<tr>
<td>Category Scanner</td>
<td>SENSOR_CATEGORY.Scanner</td>
<td>SensorCategories.SensorCategoryScanner</td>
<td>{B000E77E-F585-420F-8150-0270F726F270}</td>
</tr>
<tr>
<td>Type HumanProximity</td>
<td>SENSOR_TYPE.HUMAN_PROXIMITY</td>
<td>SensorTypes.SensorTypeHumanProximity</td>
<td>{5220DAE9-3179-4430-9F90-062662A34D0E}</td>
</tr>
<tr>
<td>Type AmbientLight</td>
<td>SENSOR_TYPE.AMBIENT_LIGHT</td>
<td>SensorTypes.SensorTypeAmbientLight</td>
<td>{9F115CF8-599A-4153-8894-D2D12B99918A}</td>
</tr>
<tr>
<td>Type Gps</td>
<td>SENSOR_TYPE.LOCATION_GPS</td>
<td>SensorTypes.SensorTypeLocationGps</td>
<td>{ED4CAS89-327A-4FF9-A560-91DA4B48275E}</td>
</tr>
<tr>
<td>Type Accelerometer3D</td>
<td>SENSOR_TYPE.ACCELEROMETER_3D</td>
<td>SensorTypes.SensorTypeAccelerometer3D</td>
<td>{C2FB0F5F-91DA-4B48-275E-B5E5FF93FD46}</td>
</tr>
<tr>
<td>Type Gyrometer3D</td>
<td>SENSOR_TYPE.GYROMETER_3D</td>
<td>SensorTypes.SensorTypeGyrometer3D</td>
<td>{D2D12899-918A-4B48-275E-B5E5FF93FD46}</td>
</tr>
<tr>
<td>Type Compass3D</td>
<td>SENSOR_TYPE.COMPASS_3D</td>
<td>SensorTypes.SensorTypeCompass3D</td>
<td>{76B5CE0D-17DD-414D-93A1-E17274080DF6}</td>
</tr>
<tr>
<td>Type DeviceOrientation</td>
<td>SENSOR_TYPE.DEVICE_ORIENTATION</td>
<td>SensorTypes.SensorTypeDeviceOrientation</td>
<td>{CDB5D8F7-3CFD-4C18-8542-CCCC622C5D6E}</td>
</tr>
<tr>
<td>Type Inclinometer3D</td>
<td>SENSOR_TYPE.INCLINOMETER_3D</td>
<td>SensorTypes.SensorTypeInclinometer3D</td>
<td>{884919FB-E885-4976-8444-6F6F5C6D31DB}</td>
</tr>
</tbody>
</table>

Above are the most commonly used GUIDs; there are many available. At first you might think that the GUIDs are silly and tedious, but there is one good reason for using them: extensibility. Since the APIs don’t care about the actual sensor names (they just pass GUIDs around), it is possible for vendors to invent new GUIDs for “value add” sensors.

**Generating New GUIDs**

Microsoft provides a tool in Visual Studio* for generating new GUIDs. Figure 4 shows a screenshot from Visual Studio for doing this. All the vendor has to do is publish them, and new functionality can be exposed without the need to change the Microsoft APIs or any operating system code at all.
Using Sensor Manager Object

In order for an app to use a sensor, Microsoft Sensor Framework needs a way to “bind” the object to actual hardware. It does this via Plug and Play, using a special object called the Sensor Manager Object.

Ask by Type

An app can ask for a specific type of sensor, such as Gyrometer3D. The Sensor Manager consults the list of sensor hardware present on the computer and returns a collection of matching objects bound to that hardware. Although the Sensor Collection may have 0, 1, or more objects, it usually has only one. Below is a C++ code sample illustrating the use of the Sensor Manager object’s GetSensorsByType method to search for 3-axis Gyros and return them in a Sensor Collection. Note that a ::CoCreateInstance() must be made for the Sensor Manager Object first.

```cpp
// Additional includes for sensors
#include <InitGuid.h>
#include <SensorsApi.h>
#include <Sensors.h>

// Create a COM interface to the SensorManager object.
ISensorManager* pSensorManager = NULL;
```
HRESULT hr = ::CoCreateInstance(CLSID_SensorManager, NULL, 
    CLSCTX_INPROC_SERVER,
    IID_PPV_ARGS(&pSensorManager));
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Unable to CoCreateInstance() the SensorManager."),
        _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}

// Get a collection of all 3-axis Gyros on the computer.
ISensorCollection* pSensorCollection = NULL;
hr = pSensorManager->GetSensorsByType(SENSOR_TYPE_GYROMETER_3D, 
    &pSensorCollection);
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Unable to find any Gyros on the computer."),
        _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}

Ask by Category
An app can request sensors by category, such as all motion sensors. The Sensor Manager consults the list of sensor hardware on the computer and returns a collection of motion objects bound to that hardware. The SensorCollection may have 0, 1, or more objects in it. On most computers, the collection will have two motion objects: Accelerometer3D and Gyrometer3D.

The C++ code sample below illustrates the use of the Sensor Manager object’s GetSensorsByCategory method to search for motion sensors and return them in a sensor collection.

// Additional includes for sensors
#include <InitGuid.h>
#include <SensorsApi.h>
#include <Sensors.h>
// Create a COM interface to the SensorManager object.
ISensorManager* pSensorManager = NULL;
HRESULT hr = ::CoCreateInstance(CLSID_SensorManager, NULL, 
    CLSCTX_INPROC_SERVER,
    IID_PPV_ARGS(&pSensorManager));
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Unable to CoCreateInstance() the SensorManager."),
        _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}

// Get a collection of all 3-axis Gyros on the computer.
ISensorCollection* pSensorCollection = NULL;
hr = pSensorManager->GetSensorsByCategory(SENSOR_CATEGORY_MOTION, &pSensorCollection);
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Unable to find any Motion sensors on the computer."),
                 _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}

**Ask by Category “All”**

In practice, it is most efficient for an app to request all of the sensors on the computer at once. The Sensor Manager consults the list of sensor hardware on the computer and returns a collection of all the objects bound to that hardware. The Sensor Collection may have 0, 1, or more objects in it. On most computers, the collection will have seven or more objects.

C++ does not have a `GetAllSensors` call, so you must use `GetSensorsByCategory(SENSOR_CATEGORY_ALL, ...)` instead as shown in the sample code below.

```
// Additional includes for sensors
#include <InitGuid.h>
#include < SensorsApi.h>
#include < Sensors.h>

// Create a COM interface to the SensorManager object.
ISensorManager* pSensorManager = NULL;
HRESULT hr = ::CoCreateInstance(CLSID_SensorManager, NULL, CLSCTX_INPROC_SERVER,
                               IID_PPV_ARGS(&pSensorManager));
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Unable to CoCreateInstance() the SensorManager."),
                 _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}

// Get a collection of all 3 sensors on the computer.
ISensorCollection* pSensorCollection = NULL;
h = pSensorManager->GetSensorsByCategory(SENSOR_CATEGORY_ALL, &pSensorCollection);
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Unable to find any sensors on the computer."),
                 _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}
Sensor Life Cycle – Enter and Leave Events

On Windows, as with most hardware devices, sensors are treated as Plug and Play devices. There are a few different scenarios where sensors can be connected/disconnected:

1. It is possible to have USB-based sensors external to the system and plugged into a USB port.

2. It is conceivable to have sensors that are attached by an unreliable wireless interface (such as Bluetooth*) or wired interface (such as Ethernet), where connects and disconnects are common.

3. If a Windows Update upgrades the device driver for the sensors, they will appear to disconnect and then reconnect.

4. When Windows shuts down (to S4 or S5), the sensors appear to disconnect.

In the context of sensors, a Plug and Play connect is called an **Enter** event, and disconnect is called a **Leave** event. Resilient apps need to be able to handle both.

Enter Event Callback

If the app is already running at the time a sensor is plugged in, the Sensor Manager reports the sensor **Enter** event; however, if the sensors are already plugged in when the app starts running, this action will not result in **Enter** events for those sensors. In C++/COM, you must use the **SetEventSink** method to hook the callback. The callback must be an entire class that inherits from **ISensorManagerEvents** and must implement **IUnknown**. Additionally, the **ISensorManagerEvents** interface must have callback function implementations for:

```cpp
STDMETHODIMP OnSensorEnter(ISensor *pSensor, SensorState state);
```

// Hook the SensorManager for any SensorEnter events.
pSensorManagerEventClass = new SensorManagerEventSink(); // create C++ class instance
// get the ISensorManagerEvents COM interface pointer
HRESULT hr = pSensorManagerEventClass->QueryInterface(IID_PPV_ARGS(&pSensorManagerEvents));
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Cannot query ISensorManagerEvents interface for our callback class."),
                 _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}
// hook COM interface of our class to SensorManager eventer
hr = pSensorManager->SetEventSink(pSensorManagerEvents);
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Cannot SetEventSink on SensorManager to our callback class."),
                 NULL, MB_OK);
}
```
Code: Hook Callback for Enter event

Below is the C++/COM equivalent of the Enter callback. All the initialization steps from the main loop would be performed in this function. In fact, it is more efficient to refactor the code so that the main loop merely calls `OnSensorEnter` to simulate an Enter event.

```cpp
STDMETHODIMP SensorManagerEventSink::OnSensorEnter(ISensor *pSensor, SensorState state)
{
    // Examine the SupportsDataField for SENSOR_DATA_TYPE_LIGHT_LEVEL_LUX.
    VARIANT_BOOL bSupported = VARIANT_FALSE;
    HRESULT hr = pSensor->SupportsDataField(SENSOR_DATA_TYPE_LIGHT_LEVEL_LUX, &bSupported);
    if (FAILED(hr))
    {
        ::MessageBox(NULL, _T("Cannot check SupportsDataField for SENSOR_DATA_TYPE_LIGHT_LEVEL_LUX."), _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
        return hr;
    }
    if (bSupported == VARIANT_FALSE)
    {
        // This is not the sensor we want.
        return -1;
    }
    ISensor *pAls = pSensor; // It looks like an ALS, memorize it.
    ::MessageBox(NULL, _T("Ambient Light Sensor has entered."), _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    // ... 
    return hr;
}

Code: Callback for Enter event

Leave Event
The individual sensor (not the Sensor Manager) reports when the **Leave** event happens. This code is the same as the previous hook callback for an **Enter** event.

// Hook the Sensor for any DataUpdated, Leave, orStateChanged events. SensorEventSink* pSensorEventClass = new SensorEventSink(); // create C++ class instance ISensorEvents* pSensorEvents = NULL; // get the ISensorEvents COM interface pointer HRESULT hr = pSensorEventClass->QueryInterface(IID_PPV_ARGS(&pSensorEvents));
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Cannot query ISensorEvents interface for our
callback class."),
        _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}
hr = pSensor->SetEventSink(pSensorEvents); // hook COM interface of our class
to Sensor eventer
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Cannot SetEventSink on the Sensor to our callback
class."),
        _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}

**Code: Hook Callback for Leave event**

The *OnLeave* event handler receives the ID of the leaving sensor as an argument.

STDMETHODIMP SensorEventSink::OnLeave(REFSENSOR_ID sensorID)
{
    HRESULT hr = S_OK;
    ::MessageBox(NULL, _T("Ambient Light Sensor has left."),
        _T("Sensor C++ Sample"), MB_OK | MB_ICONINFORMATION);
    // Perform any housekeeping tasks for the sensor that is leaving.
    // For example, if you have maintained a reference to the sensor,
    // release it now and set the pointer to NULL.
    return hr;
}

**Code: Callback for Leave event**
Picking Sensors for an App

Different types of sensors report different information. Microsoft calls these pieces of information **Data Fields**, and they are grouped together in a SensorDataReport. A computer may (potentially) have more than one type of sensor that an app can use. The app won’t care which sensor the information came from, so long as it is available.

Table 3 shows the constant names for the most commonly used Data Fields for Win32/COM and .NET. Just like sensor identifiers, these constants are just human-readable names for their associated GUIDs. This method of association provides for extensibility of Data Fields beyond those “well known” fields that Microsoft has pre-defined.

**Table 4 Data Field identifier constants**

<table>
<thead>
<tr>
<th>Constant (Win32*/COM)</th>
<th>Constant (.NET)</th>
<th>PROPERTYKEY (GUID,PID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENSOR_DATA_TYPE_TIMESTAMP</td>
<td>SensorDataTypeTimestamp</td>
<td>{DB5E0C2F-CF1D-4C18-B46C-DB6011D62150},2</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_LIGHT_LEVEL_LUX</td>
<td>SensorDataTypeLightLevelLux</td>
<td>{E4CC7CE2-DCB7-46E9-8439-4FEC54B833A6},2</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_ACCELERATION_X_G</td>
<td>SensorDataTypeAccelerationXG</td>
<td>{3F8A69A2-07C5-4E48-A965-CD797A8B56D5},2</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_ACCELERATION_Y_G</td>
<td>SensorDataTypeAccelerationYG</td>
<td>{3F8A69A2-07C5-4E48-A965-CD797A8B56D5},3</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_ACCELERATION_Z_G</td>
<td>SensorDataTypeAccelerationZG</td>
<td>{3F8A69A2-07C5-4E48-A965-CD797A8B56D5},4</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_ANGULAR_VELOCITY_X_DEGREES_PER_SECOND</td>
<td>SensorDataTypeAngularVelocityXDegreesPerSecond</td>
<td>{3F8A69A2-07C5-4E48-A965-CD797A8B56D5},10</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_ANGULAR_VELOCITY_Y_DEGREES_PER_SECOND</td>
<td>SensorDataTypeAngularVelocityYDegreesPerSecond</td>
<td>{3F8A69A2-07C5-4E48-A965-CD797A8B56D5},11</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_ANGULAR_VELOCITY_Z_DEGREES_PER_SECOND</td>
<td>SensorDataTypeAngularVelocityZDegreesPerSecond</td>
<td>{3F8A69A2-07C5-4E48-A965-CD797A8B56D5},12</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_TILT_X_DEGREES</td>
<td>SensorDataTypeTiltXDegrees</td>
<td>{1637DBA2-4248-4275-865D-558DE84AEDFD},2</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_TILT_Y_DEGREES</td>
<td>SensorDataTypeTiltYDegrees</td>
<td>{1637DBA2-4248-4275-865D-558DE84AEDFD},3</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_TILT_Z_DEGREES</td>
<td>SensorDataTypeTiltZDegrees</td>
<td>{1637DBA2-4248-4275-865D-558DE84AEDFD},4</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_MAGNETIC_HEADING_COMPENSATED_TRUE_NORTH_DEGREES</td>
<td>SensorDataTypeMagneticHeadingCompensatedTrueNorthDegrees</td>
<td>{1637DBA2-4248-4275-865D-558DE84AEDFD},11</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_MAGNETIC_FIELD_STRENGTH_X_MILLIGAUSSE</td>
<td>SensorDataTypeMagneticFieldStrengthXMilligauss</td>
<td>{1637DBA2-4248-4275-865D-558DE84AEDFD},19</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_MAGNETIC_FIELD_STRENGTH_Y_MILLIGAUSSE</td>
<td>SensorDataTypeMagneticFieldStrengthYMilligauss</td>
<td>{1637DBA2-4248-4275-865D-558DE84AEDFD},20</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_MAGNETIC_FIELD_STRENGTH_Z_MILLIGAUSSE</td>
<td>SensorDataTypeMagneticFieldStrengthZMilligauss</td>
<td>{1637DBA2-4248-4275-865D-558DE84AEDFD},21</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_QUATERNION</td>
<td>SensorDataTypeQuaternion</td>
<td>{1637DBA2-4248-4275-865D-558DE84AEDFD},17</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_ROTATION_MATRIX</td>
<td>SensorDataTypeRotationMatrix</td>
<td>{1637DBA2-4248-4275-865D-558DE84AEDFD},16</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_LATITUDE_DEGREES</td>
<td>SensorDataTypeLatitudeDegrees</td>
<td>{055C74D8-CA6F-47D6-95C6-1ED3637A0FF4},2</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_LONGITUDE_DEGREES</td>
<td>SensorDataTypeLongitudeDegrees</td>
<td>{055C74D8-CA6F-47D6-95C6-1ED3637A0FF4},3</td>
</tr>
<tr>
<td>SENSOR_DATA_TYPE_ALTITUDE_ELLIPSOID_METERS</td>
<td>SensorDataTypeAltitudeEllipsoidMeters</td>
<td>{055C74D8-CA6F-47D6-95C6-1ED3637A0FF4},4</td>
</tr>
</tbody>
</table>
One thing that makes Data Field identifiers different from sensor IDs is the use of a data type called \textit{PROPERTYKEY}. A \textit{PROPERTYKEY} consists of a GUID (similar to what sensors have), plus an extra number called a “PID” (property ID). You might notice that the GUID part of a \textit{PROPERTYKEY} is common for sensors that are in the same category. Data Fields have a native data type for all of their values, such as Boolean, unsigned char, int, float, double, etc.

In Win32/COM, the value of a Data Field is stored in a polymorphic data type called \textit{PROPVARIANT}. In .NET, there is a CLR (Common Language Runtime) data type called “object” that does the same thing. The polymorphic data type will need to be queried and/or typecast to the “expected”/“documented” data type.

The \textit{SupportsDataField()} method of the sensor should be used to check the sensors for the Data Fields of interest. This is the most common programming idiom that is used to select sensors. Depending on the usage model of the app, only a subset of the Data Field may be required. Sensors that support the desired Data Fields should be selected. Type casting will be required to assign the sub-classed member variables from the base class sensor.

```c
ISensor* m_pAls;
ISensor* m_pAccel;
ISensor* m_pTilt;
// Cycle through the collection looking for sensors we care about.
ULONG ulCount = 0;
HRESULT hr = pSensorCollection->GetCount(&ulCount);
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Unable to get count of sensors on the computer."), _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}
for (int i = 0; i < (int)ulCount; i++)
{
    hr = pSensorCollection->GetAt(i, &pSensor);
    if (SUCCEEDED(hr))
    {
        VARIANT_BOOL bSupported = VARIANT_FALSE;
        hr = pSensor->SupportsDataField(SENSOR_DATA_TYPE_LIGHT_LEVEL_LUX, &bSupported);
        if (SUCCEEDED(hr) && (bSupported == VARIANT_TRUE)) m_pAls = pSensor;
        hr = pSensor->SupportsDataField(SENSOR_DATA_TYPE_ACCELERATION_Z_G, &bSupported);
        if (SUCCEEDED(hr) && (bSupported == VARIANT_TRUE)) m_pAccel = pSensor;
        hr = pSensor->SupportsDataField(SENSOR_DATA_TYPE_TILT_Z_DEGREES, &bSupported);
        if (SUCCEEDED(hr) && (bSupported == VARIANT_TRUE)) m_pTilt = pSensor;
    }
}
```

\textbf{Code: Use of the} \textit{SupportsDataField()} \textbf{method of the sensor to check for supported data field}

\textbf{Sensor Properties}

In addition to Data Fields, sensors have \textbf{Properties} that can be used for identification and configuration. Table 4 shows the most commonly-used Properties. Just like Data Fields, Properties have constant
names used by Win32/COM and .NET, and those constants are really PROPERTYKEY numbers underneath. Properties are extensible by vendors and also have PROPVARIANT polymorphic data types. Unlike Data Fields that are read-only, Properties have the ability to Read/Write. It is up to the individual sensor’s discretion as to whether or not it rejects Write attempts. Because no exception is thrown when a write attempt fails, a write-read-verification will need to be performed.

Table 5 Commonly used sensor Properties and PIDs

<table>
<thead>
<tr>
<th>Identification (Win32*/COM)</th>
<th>Identification (.NET)</th>
<th>PROPERTYKEY (GUID,PID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENSOR_PROPERTY_PERSISTENT_UNIQUE_ID</td>
<td>SensorID</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},5</td>
</tr>
<tr>
<td>WPD_FUNCTIONAL_OBJECT_CATEGORY</td>
<td>CategoryID</td>
<td>{8F052D93-ABCA-4FC5-A5AC-B01DF4DBE598},2</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_TYPE</td>
<td>TypeID</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},2</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_STATE</td>
<td>State</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},3</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_MANUFACTURER</td>
<td>SensorManufacturer</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},6</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_MODEL</td>
<td>SensorModel</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},7</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_SERIAL_NUMBER</td>
<td>SensorSerialNumber</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},8</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_FRIENDLY_NAME</td>
<td>FriendlyName</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},9</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_DESCRIPTION</td>
<td>SensorDescription</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},10</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_MIN_REPORT_INTERVAL</td>
<td>MinReportInterval</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},12</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_CONNECTION_TYPE</td>
<td>SensorConnectionType</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},11</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_DEVICE_ID</td>
<td>SensorDevicePath</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},15</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_RANGE_MAXIMUM</td>
<td>SensorRangeMaximum</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},21</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_RANGE_MINIMUM</td>
<td>SensorRangeMinimum</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},20</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_ACCURACY</td>
<td>SensorAccuracy</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},17</td>
</tr>
<tr>
<td>SENSOR_PROPERTY_RESOLUTION</td>
<td>SensorResolution</td>
<td>{7F8383EC-D3EC-495C-A8CF-B8BBE85C2920},18</td>
</tr>
</tbody>
</table>
### Setting Sensor Sensitivity

The sensitivity setting is a very useful Property of a sensor. It can be used to assign a threshold that controls or filters the number of SensorDataReports sent to the host computer. In this way, traffic can be reduced: only send up those DataUpdated events that are truly worthy of bothering the host CPU.

The way Microsoft has defined the data type of this Sensitivity property as a container type called IPortableDeviceValues in Win32/COM and SensorPortableDeviceValues in .NET. This container holds a collection of tuples, each of which is a Data Field PROPERTYKEY followed by the sensitivity value for that Data Field. The sensitivity always uses the same units and data type as the matching Data Field.

```cpp
// Configure sensitivity
// create an IPortableDeviceValues container for holding the <Data Field, Sensitivity> tuples.
IPortableDeviceValues* pInSensitivityValues;
hr = ::CoCreateInstance(CLSID_PortableDeviceValues, NULL, CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pInSensitivityValues));
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Unable to CoCreateInstance() a PortableDeviceValues collection."), _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}
// fill in IPortableDeviceValues container contents here: 0.1 G sensitivity in each of X, Y, and Z axes.
PROPVARIANT pv;
PropVariantInit(&pv);
pv.vt = VT_R8; // COM type for (double)
pv.dblVal = (double)0.1;
pInSensitivityValues->SetValue(SENSOR_DATA_TYPE_ACCELERATION_X_G, &pv);
pInSensitivityValues->SetValue(SENSOR_DATA_TYPE_ACCELERATION_Y_G, &pv);
pInSensitivityValues->SetValue(SENSOR_DATA_TYPE_ACCELERATION_Z_G, &pv);
// create an IPortableDeviceValues container for holding the <SENSOR_PROPERTY_CHANGE_SENSITIVITY, pInSensitivityValues> tuple.
IPortableDeviceValues* pInValues;
hr = ::CoCreateInstance(CLSID_PortableDeviceValues, NULL, CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pInValues));
if (FAILED(hr))
{
    // ...
::MessageBox(NULL, _T("Unable to CoCreateInstance() a PortableDeviceValues collection.")), _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}
// fill it in
pInValues->SetIPortableDeviceValuesValue(SENSOR_PROPERTY_CHANGE_SENSITIVITY, pInSensitivityValues);
// now actually set the sensitivity
IPortableDeviceValues* pOutValues;
hr = pAls->SetProperties(pInValues, &pOutValues);
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Unable to SetProperties() for Sensitivity.")), _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}
// check to see if any of the setting requests failed
DWORD dwCount = 0;
hr = pOutValues->GetCount(&dwCount);
if (FAILED(hr) || (dwCount > 0))
{
    ::MessageBox(NULL, _T("Failed to set one-or-more Sensitivity values.")), _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}
PropVariantClear(&pv);

Requesting permissions for Sensors

Some information provided by sensors may be considered sensitive, i.e., Personally Identifiable Information (PII). Data Fields such as the computer’s location (e.g., latitude and longitude), could be used to track the user. For this reason, Windows forces apps to get end-user permission to access the sensor. The State property of the sensor and the RequestPermissions() method of the SensorManager can be used as needed.

The RequestPermissions() method takes an array of sensors as an argument, so an app can request permission for more than one sensor at a time. The C++/COM code is shown below. Note that (ISensorCollection *) must be provided as an argument to RequestPermissions().

    // Get the sensor's state
    SensorState state = SENSOR_STATE_ERROR;
    HRESULT hr = pSensor->GetState(&state);
    if (FAILED(hr))
    {
        ::MessageBox(NULL, _T("Unable to get sensor state.")), _T("Sensor C++ Sample"),
        MB_OK | MB_ICONERROR);
        return -1;
    }
    // Check for access permissions, request permission if necessary.
    if (state == SENSOR_STATE_ACCESS_DENIED)
{  // Make a SensorCollection with only the sensors we want to get permission to  access.
    ISensorCollection *pSensorCollection = NULL;
    hr = ::CoCreateInstance(CLSID_SensorCollection, NULL, CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&pSensorCollection));
    if (FAILED(hr))
    {
        ::MessageBox(NULL, _T("Unable to CoCreateInstance() a SensorCollection."),
            _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
        return -1;
    }
    pSensorCollection->Clear();
    pSensorCollection->Add(pAls);  // add 1 or more sensors to request permission
    for...
    // Have the SensorManager prompt the end-user for permission.
    hr = m_pSensorManager->RequestPermissions(NULL, pSensorCollection, TRUE);
    if (FAILED(hr))
    {
        ::MessageBox(NULL, _T("No permission to access sensors that we care about."),
            _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
        return -1;
    }
}

**Sensor Data Update**
Sensors report data by throwing an event called a DataUpdated event. The actual Data Fields are packaged inside a SensorDataReport, which is passed to any attached DataUpdated event handlers. An app can obtain the SensorDataReport by hooking a callback handler to the sensor’s DataUpdated event. The event occurs in a Windows Sensor Framework thread, which is a different thread than the message-pump thread used to update the app’s GUI. Therefore, a “hand-off” of the SensorDataReport from the event handler (Als_DataUpdate) to a separate handler (Als_UpdateGUI) that can execute on the context of the GUI thread is required. In .NET, such a handler is called a delegate function.

The example below shows preparation of the delegate function. In C++/COM, the SetEventSink method must be used to hook the callback. The callback cannot simply be a function; it must be an entire class that inherits from ISensorEvents and also implements IUnknown. The ISensorEvents interface must have callback function implementations for:

```cpp
STDMETHODIMP OnEvent(ISensor *pSensor, REFGUID eventID, IPortabl...```
// Hook the Sensor for any DataUpdated, Leave, or StateChanged events.
SensorEventSink* pSensorEventClass = new SensorEventSink(); // create C++ class instance
ISensorEvents* pSensorEvents = NULL;
// get the ISensorEvents COM interface pointer
HRESULT hr = pSensorEventClass->QueryInterface(IID_PPV_ARGS(&pSensorEvents));
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Cannot query ISensorEvents interface for our callback class."),
        _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}
hr = pSensor->SetEventSink(pSensorEvents); // hook COM interface of our class to Sensor eventer
if (FAILED(hr))
{
    ::MessageBox(NULL, _T("Cannot SetEventSink on the Sensor to our callback class."),
        _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
    return -1;
}

**Code: Set a COM Event Sink for the sensor**

The DataUpdated event handler receives the SensorDataReport (and the sensor that initiated the event) as arguments. It calls the Invoke() method of the form to post those items to the delegate function. The GUI thread runs the delegate function posted to its Invoke queue and passes the arguments to it. The delegate function casts the data type of the SensorDataReport to the expected subclass, gaining access to its Data Fields. The Data Fields are extracted using the GetDataField() method of the SensorDataReport object. Each of the Data Fields has to be typecast to their “expected”/”documented” data types (from the generic/polymorphic data type returned by the GetDataField() method). The app can then format and display the data in the GUI.

The OnDataUpdated event handler receives the SensorDataReport (and the sensor that initiated the event) as arguments. The Data Fields are extracted using the GetSensorValue() method of the SensorDataReport object. Each of the Data Fields needs to have their PROPVARIANT checked for their “expected”/”documented” data types. The app can then format and display the data in the GUI. It is not necessary to use the equivalent of a C# delegate. This is because all C++ GUI functions (such as ::SetWindowText() shown here) use Windows message-passing to post the GUI update to the GUI thread / message-loop (the WndProc of your main window or dialog box).

STDMETHODIMP SensorEventSink::OnDataUpdated(ISensor *pSensor,
SensorDataReport *pNewData)
{
HRESULT hr = S_OK;
if ((NULL == pNewData) || (NULL == pSensor)) return E_INVALIDARG;
float fLux = 0.0f;
PROPVARIANT pv = \{
hr = pNewData->GetSensorValue(SENSOR_DATA_TYPE_LIGHT_LEVEL_LUX, &pv);
if (SUCCEEDED(hr))
{
    if (pv.vt == VT_R4) // make sure the PROPVARIANT holds a float as we
        // expect
    {
        // Get the lux value.
        fLux = pv.fltVal;
        // Update the GUI
        wchar_t *pwszLabelText = (wchar_t *)malloc(64 * sizeof(wchar_t));
        swprintf_s(pwszLabelText, 64, L"Illuminance Lux: %.1f", fLux);
        BOOL bSuccess = ::SetWindowText(m_hwndLabel, (LPCWSTR)pwszLabelText);
        if (bSuccess == FALSE)
        {
            ::MessageBox(NULL, _T("Cannot SetWindowText on label
control."), _T("Sensor C++ Sample"), MB_OK | MB_ICONERROR);
        }
        free(pwszLabelText);
    }
    PropVariantClear(&pv);
    return hr;
}]

Properties of the SensorDataReport object can be referenced to extract Data Fields from the
SensorDataReport. This only works for the .NET API and for “well known” or “expected” Data Fields of
that particular SensorDataReport subclass. For the Win32/COM API, the GetDataField method must be
used. It is possible to use “Dynamic Data Fields” for the underlying driver/firmware to “piggyback” any
“extended/unexpected” Data Fields inside SensorDataReports. The GetDataField method is used to
extract those.
Using Sensors in Windows Store apps

Unlike the Desktop mode, the WinRT Sensor API follows a common template for each of the sensors:

• There is usually a single event called *ReadingChanged* that calls the callback with an xxxReadingChangedEventArgs containing a Reading object holding the actual data. The accelerometer is an exception; it also has a *Shaken* event.

• The hardware-bound instance of the sensor class is retrieved using the *GetDefault()* method.

• Polling can be done with the *GetCurrentReading()* method.

Windows Store apps are often written either in JavaScript* or in C#. There are different language-bindings to the API, which result in a slightly different capitalization appearance in the API names and a slightly different way that events are handled. The simplified API is easier to use, and the pros and cons are listed in Table 6.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| SensorManager        | There is no SensorManager to deal with. Apps use the GetDefault() method to get an instance of the sensor class. | • It is not possible to search for arbitrary sensor instances. If more than one of a particular sensor type exists on a computer, you will only see the “first” one.  
• It is not possible to search for arbitrary sensor types or categories by GUID. Vendor value-add extensions are inaccessible. |
| Events               | Apps only worry about the DataUpdated event.                         | • Apps have no access to Enter, Leave, StatusChanged, or arbitrary event types. Vendor value-add extensions are inaccessible. |
| Sensor properties    | Apps only worry about the ReportInterval property.                   | • Apps have no access to the other properties, including the most useful one: Sensitivity.  
• Other than manipulating the ReportInterval property, there is no way for Windows Store apps to tune or control the flow rate of Data Reports.  
• Apps cannot access arbitrary Properties by PROPERTYKEY. Vendor value-add extensions are inaccessible. |
| Data Report properties| Apps only worry about a few, pre-defined Data Fields unique to each sensor. | • Apps have no access to other Data Fields. If sensors “piggy-back” additional well-known Data Fields in a Data Report beyond what Windows Store apps expect, the Data Fields are inaccessible.  
• Apps cannot access arbitrary Data Fields by PROPERTYKEY. Vendor value-add extensions are inaccessible.  
• Apps have no way to query at run-time what Data Fields a sensor supports. It can only assume what the API predefines. |

Summary

Windows 8 APIs provide developers an opportunity to take advantage of sensors available on different platforms under both the traditional Desktop mode and the new Windows Store app interface. In this document, an overview was presented of the sensor APIs available to developers creating Windows 8 applications, focusing on the APIs and code samples for Desktop apps.
Appendix

Coordinate System for Different Form Factors

The Windows API reports X, Y, and Z axes in a manner that is compatible with the HTML5 standard (and Android*). It is also called the “ENU” system because X faces virtual “East”, Y faces virtual “North”, and Z faces “Up.”

To figure out the direction of rotation, use the “Right Hand Rule”:

* Point the thumb of your right hand in the direction of one of the axes.

* Positive angle rotation around that axis will follow the curve of your fingers.

These are the X, Y, and Z axes for a tablet form-factor PC, or phone (left) and for a clamshell PC (right). For more esoteric form factors (for example, a clamshell that is convertible into a tablet), the “standard” orientation is when it is in the TABLET state.

To develop a navigation application (e.g., 3D space game), a conversion from “ENU” systems in your program is required. This can be done by using matrix multiplication. Graphics libraries such as Direct3D* and OpenGL* have APIs for handling this.
Resources


About the Authors

Gael is a Software Engineer in the Developer Relations Division at Intel working with Business Client Technologies. Gael holds a BS in Math and an MBA, both from the University of New Mexico. Gael enjoys hiking, biking, and photography.

Deepak Vembar, PhD

Deepak Vembar is a Research Scientist in the Interaction and Experience Research (IXR) group at Intel Labs. His research interests are at the intersection of computer graphics and human computer interaction including areas of real-time graphics, virtual reality, haptics, eye-tracking, and user interaction. Prior to joining Intel Labs, Deepak was a Software Engineer in Software and Services Group (SSG) at Intel, where he worked with PC game developers to optimize their games for Intel platforms, delivered courses and tutorials on heterogeneous platform optimization, and created undergraduate coursework using game demos as an instructional medium for use in school curriculum.
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.

OpenGL is a registered trademark of Silicon Graphics, Inc. in the United States and/or other countries worldwide.

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

For more complete information about performance and benchmark results, visit www.intel.com/benchmarks

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804