## PROFESSIONAL

### WINDOWS 7 DEVELOPMENT GUIDE

<table>
<thead>
<tr>
<th>INTRODUCTION .................................................. xxv</th>
</tr>
</thead>
<tbody>
<tr>
<td>❯ <strong>PART I</strong> INTRODUCING WINDOWS 7</td>
</tr>
<tr>
<td><strong>CHAPTER 1</strong> Windows 7 Improvements ........................ 3</td>
</tr>
<tr>
<td><strong>CHAPTER 2</strong> Developing a Windows 7 Strategy .............. 13</td>
</tr>
<tr>
<td><strong>CHAPTER 3</strong> Understanding .NET 4.0 ..................... 21</td>
</tr>
<tr>
<td>❯ <strong>PART II</strong> WORKING WITH THE WINDOWS 7 USER INTERFACE</td>
</tr>
<tr>
<td><strong>CHAPTER 4</strong> Interacting with the Taskbar ................ 37</td>
</tr>
<tr>
<td><strong>CHAPTER 5</strong> Advanced Taskbar Techniques .................. 57</td>
</tr>
<tr>
<td><strong>CHAPTER 6</strong> Working with the Ribbon Interface .......... 85</td>
</tr>
<tr>
<td><strong>CHAPTER 7</strong> Creating Custom Ribbon Interface Applications .. 105</td>
</tr>
<tr>
<td><strong>CHAPTER 8</strong> Programming for Aero Glass Functionality ... 139</td>
</tr>
<tr>
<td><strong>CHAPTER 9</strong> Working with the Multi-Touch User Interface .. 159</td>
</tr>
<tr>
<td>❯ <strong>PART III</strong> DEVELOPING SECURE APPLICATIONS</td>
</tr>
<tr>
<td><strong>CHAPTER 10</strong> Using Standard NT Security Features in Windows 7 .. 171</td>
</tr>
<tr>
<td><strong>CHAPTER 11</strong> Understanding the User Account Control .... 201</td>
</tr>
<tr>
<td><strong>CHAPTER 12</strong> Developing Applications with Enhanced Security .. 217</td>
</tr>
<tr>
<td><strong>CHAPTER 13</strong> Interacting with the Built-In Security Features .. 247</td>
</tr>
<tr>
<td>❯ <strong>PART IV</strong> ADVANCED WINDOWS 7 PROGRAMMING</td>
</tr>
<tr>
<td><strong>CHAPTER 14</strong> Working in the Background .................. 293</td>
</tr>
<tr>
<td><strong>CHAPTER 15</strong> Using the Windows 7 Libraries .............. 329</td>
</tr>
<tr>
<td><strong>CHAPTER 16</strong> Writing 64-Bit Applications for Windows 7 .. 347</td>
</tr>
<tr>
<td><strong>CHAPTER 17</strong> Using Parallel Programming in Windows 7 .... 361</td>
</tr>
<tr>
<td><strong>CHAPTER 18</strong> Using the Sensor and Location Platform ...... 373</td>
</tr>
<tr>
<td><strong>CHAPTER 19</strong> Using Windows XP Mode Effectively .......... 397</td>
</tr>
</tbody>
</table>
PROFESSIONAL

Windows 7 Development Guide

John Paul Mueller

Wiley Publishing, Inc.
Using the Sensor and Location Platform

WHAT’S IN THIS CHAPTER?

- Understanding the sensor types and location devices
- Getting a list of installed sensors
- Getting specific sensor information
- Working with other sensor and location devices

For many years now, specialty hardware designers have created special drivers and other software to make their hardware accessible from Windows. It’s possible to obtain special scientific sensors for Windows that rely on the RS422 and other standard interfaces. These sensors help scientists of all sorts to gather information and analyze it. Sensors also play a role in industrial automation. However, the fact that sensors are now in many homes is probably what prompted Microsoft to begin adding sensor support to Windows 7. Homes now have Windows-based security systems. You can obtain products to monitor the weather outside your home or to check on your children as they play. At some point, you’ll probably be able to turn on your stove and tell it to cook dinner before your arrival home. The refrigerator will eventually tell you that you’re out of milk. All these applications rely on sensors of various sorts, and they all require specialized device drivers and software.
It would be quite easy to write an entire book about the sensors that you can use with Windows and still not cover them all. In fact, it would be possible to write a rather larger book just about the various sorts of industrial sensors. Home security would require an entirely separate book because the number and variety of sensors have increased so greatly. For example, many people don’t realize it, but there’s a special sort of sensor that can detect the sound of breaking glass and report it to you. If someone threw a rock through your window, you’d know about it long before you arrived home. Consequently, this chapter gives you just a taste of the sorts of things you can do in Windows 7 with various kinds of sensors.

Although theoretically there’s no limit to the number of device types that Windows 7 can support, you can currently place the devices in three major groups: environmental, position, and light. The following sections provide an overview of these three major areas and demonstrate how you’d write code to work with external sensors. (If you have a specially equipped PC, you might find that some of these sensors are installed internally — the physical location doesn’t matter for the purpose of writing the code.) Of course, you must have the device in order to test the code.

In order to code for a specific device, you actually need to have the device attached to your system or network. Because devices differ in their interfaces, functionality, user interface, and other specifics, code written for one device won’t necessarily work for any other device. For example, writing code for one scanner won’t necessarily help you create code for a different scanner. The non-standard features of devices have been the bane of developers since someone decided to attach a printer to a computer. Manufacturers sell products based on differences, and consumers purchase products because they deem a particular set of features important. As a consequence, developers generally need to test code with each supported device to ensure it actually works. Because of these differences, the code in this chapter will likely require tuning to work with the specific device attached to your system — don’t assume any of the code will work without modification.
DEFINING THE SENSOR AND LOCATION DEVICES

Sensors and location devices are two separate kinds of input for a computer system. Sensors provide environmental information of some sort, while location devices tell you where you are. Combining these two elements is often the stuff of robotics, but this chapter won’t even broach that topic. The following sections describe the kinds of devices that Windows 7 can interact with and tell you a bit about them.

An Overview of the Sensor Categories

Sensors provide special functionality to computer systems. The term specifically applies to a device that mimics our own senses in some way. For example, sensors can detect light or sound. A special sensor may provide some type of tactile feedback. Of course, sensors can go well beyond human senses. A light detector can see into the infrared or ultraviolet ranges, neither of which is detectable by human eyes. So sensors can also augment human senses to let us detect things that are normally undetectable. No matter what the sensor does, however, it must convert what it detects into an electrical signal that the computer can interpret. The software you create assigns value to the electrical impulses and presents the resulting information in a form that the human viewer can understand.

Location devices are completely different from sensors. The vast majority of location devices today work on the Global Positioning System (GPS), which is an array of satellites that provides positional information in the form of radio waves. In sum, the location device is a special sort of radio receiver in most cases. When working with a GPS device, Windows receives actual satellite data, not electrical impulses. Your software accepts this data and formats it in a way that the human viewer can understand.

Another category of location device performs some sort of measurement and presents that measurement in the form of data or electrical impulses, depending on the characteristics of the device. For example, a location device could use a laser to measure the distance from the device to a wall. The output could be in the form of data (so many feet) or electrical impulse (a voltage level that falls within a minimum to a maximum value). The difference between the two categories of location devices is absolute versus relative position. A GPS device provides an absolute position, while the second location device category provides relative position.

The SensorCategories class provides a listing of sensor categories you can interact with. Table 18-1 provides a listing of these categories and describes them. The examples in this chapter will use a number of the categories.
### TABLE 18-1: Categories of Sensors Supported by Windows 7

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>GROUP</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>N/A</td>
<td>Accesses all the sensors, no matter what category each sensor is in. This is the category you use to perform tasks such as obtaining an inventory of the system.</td>
</tr>
<tr>
<td>BioMetric</td>
<td>Environmental</td>
<td>All biometric devices, such as fingerprint or retina readers. The biometric device need not be security-related. For example, biometric devices can interpret a person’s condition and locate potential health issues.</td>
</tr>
<tr>
<td>Electrical</td>
<td>Environmental</td>
<td>Any electrical device that doesn’t fit in some other category. This category would include devices actually used to measure electricity.</td>
</tr>
<tr>
<td>Environmental</td>
<td>Environmental</td>
<td>Any device used to measure environmental conditions, such as temperature and humidity. Some unusual devices might fit into this category, such as a device used to measure static electricity based on environmental conditions.</td>
</tr>
<tr>
<td>Light</td>
<td>Light</td>
<td>Any device used to measure light. The device could simply measure the amount of light, but there could also be devices used for detecting certain kinds of light.</td>
</tr>
<tr>
<td>Location</td>
<td>Position</td>
<td>Any GPS device used to measure an absolute position based on satellite information. Theoretically, this category could also include input from Long RAnge Navigation (LORAN) devices (see <a href="http://www.navcen.uscg.gov/?pageName=loranMain">http://www.navcen.uscg.gov/?pageName=loranMain</a>), but such devices are outdated and GPS is probably the only input device you’ll see in use.</td>
</tr>
<tr>
<td>Mechanical</td>
<td>Environmental</td>
<td>Either an output or an input device used to measure something mechanically. For example, a device of this type could measure how far a door is opened or the amount of force applied to a test frame.</td>
</tr>
<tr>
<td>Motion</td>
<td>Position</td>
<td>Any device used to measure direction and velocity. These devices normally employ accelerometers, but could also rely on other means of detecting motion. See <a href="http://www.dimensionengineering.com/accelerometers.htm">http://www.dimensionengineering.com/accelerometers.htm</a> for a description of how accelerometers work.</td>
</tr>
</tbody>
</table>
The sensors are also categorized by how they’re accessed. The `SensorConnectionType` enumeration tells you how the sensor is connected to the system. You’ll see one of the following options:

- **Invalid**: Essentially, the system doesn’t know how the device is connected. There might be something wrong with the device configuration. More likely, the device has been turned off so the system can’t see how it’s connected.

- **Integrated**: The device is built into the computer. Normally, this means that the device is part of the motherboard or is a peripheral directly attached to the motherboard.

- **Attached**: The device is attached to the computer indirectly. In most cases, the device attaches through a peripheral device or through an external port. The device may be connected through the USB port.

- **External**: The device uses an external connection. In many cases, this means that the device is attached through a network connection. The device may be on an external system and shared. The device could also be network-capable and have its own network connection.

---

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>GROUP</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orientation</td>
<td>Position</td>
<td>Any device used to measure angular rotation about a center of mass. The measurements are normally made in terms of roll, pitch, and yaw. Even though these terms are normally applied to ships and airplanes, they can apply to anything. See <a href="http://en.wikipedia.org/wiki/Flight_dynamics">http://en.wikipedia.org/wiki/Flight_dynamics</a> for a description of roll, pitch, and yaw.</td>
</tr>
<tr>
<td>Scanner</td>
<td>Light</td>
<td>Any device that captures an image of some type. These devices naturally include scanners, but cameras could also appear in this category, as could telescopes configured to connect to a computer. Generally, this category includes any device that contains a Charge-Coupled Device (CCD) or its equivalent.</td>
</tr>
</tbody>
</table>
Software Devices

You might not think that software can provide sensor capabilities, but it can. Windows 7 can work with software that performs analysis and creates sensor-like output. For example, you need not rely on a physical GPS to determine your position. It’s possible to obtain a reasonably precise location using Wireless Fidelity (WiFi) connections (IEEE 802.11b wireless networking). For that matter, if you’re willing to entertain a little less precision, you can use your IP connection as a means of determining location.

The example in the “Obtaining a List of Sensors” section of this chapter checks your system for any sort of sensor that Windows 7 detects. You might think that your system contains a wealth of sensors, but Windows 7 has shown itself extremely picky about what it considers a sensor. Only device drivers written as sensors actually appear as sensors when you check them. Consequently, your system filled with sensors might actually report that it has no sensors at all, as shown in Figure 18-1, when you run the example application.

Fortunately, you can add a usable sensor for experimentation and testing purposes that works just fine even on a desktop system. The Geosense for Windows driver is a free download from http://www.geosenseforwindows.com/. Download either the 32-bit or 64-bit driver as required for your machine. Follow these steps to install the software sensor on your system:

1. Double-click the Geosense_1.2_x86.MSI or Geosense_1.2_x64.MSI file. You’ll see a rather humorous licensing agreement and its equally silly condition.
2. Check I Authorize You to Slap Me if I Violate the Terms Above and click Install. After a few seconds, you’ll see the usual UAC warning.
3. Click Yes. The installation will proceed, and you’ll eventually see a success message.
4. Click Finish. At this point, you can run the example in the “Obtaining a List of Sensors” section of this chapter and see the sensor, but the sensor won’t do anything because it isn’t enabled. If your only goal is to determine how Windows installs sensors, you can stop now. However, if you want to see how sensors work, you’ll need to proceed to Step 5.
5. Open the Location and Other Sensors applet in the Control Panel. You’ll see a list of sensors installed on your system, as shown in Figure 18-2 (the Geosense for Windows sensor might be the only entry on the list).
6. Check Enabled and click Apply. The Geosense for Windows sensor is now enabled.

OBTAINING A LIST OF SENSORS

Before you can do anything with the sensors on a system, you need to know which sensors are installed. Fortunately, the SensorManager class makes getting the list relatively easy. The Get Sensors example described in the following sections shows how to use this class to query the sensors on a system.

The physical presence of a sensor on your system doesn’t mean that Windows will recognize it. As with any piece of hardware, your sensor must come with an accompanying driver. Unfortunately, many vendors haven’t released a version of their driver that works with the Windows 7 sensor functionality. For example, the Dell D series Latitude comes with a light sensor, but there isn’t a Windows 7 driver for it, and therefore Windows 7 doesn’t recognize the light sensor.

Configuring the Get Sensors Example

This example begins with a Windows Forms application. You’ll need to add a List button (btnList) to obtain a list of the sensors, and a list box (lstOutput) to output the results of the query. In addition, you’ll need to add references to Microsoft.WindowsAPICodePack.DLL, Microsoft.WindowsAPICodePack.Shell.DLL, and Microsoft.WindowsAPICodePack.Sensors.DLL. You must also provide the following using statements:

```csharp
```
Writing the Get Sensors Code

You can obtain a considerable amount of information about the sensors on your system. However, because the sensors vary quite a bit, writing generic code can be difficult. There are a number of hurdles to overcome, and sometimes you need to use generic types, such as `Object`, that don't provide the best information while writing the code. Of course, there are times when you'll need to take the generic approach because you need to inventory the devices that a system can access. Listing 18-1 shows one approach to handling this situation.

Listing 18-1: Displaying a list of installed sensors

```csharp
private void btnList_Click(object sender, EventArgs e)
{
    // Define a variable to hold the sensor list.
    SensorList<Sensor> AllSensors;

    // The system will raise an error if there aren't // any sensors to query.
    try
    {
        // Obtain a list of all the sensors.
        AllSensors = SensorManager.GetAllSensors();
    }
    catch (SensorPlatformException SensorErr)
    {
        MessageBox.Show(SensorErr.Message,
                         "Error Obtaining Sensors",
                         MessageBoxButtons.OK,
                         MessageBoxIcon.Error);
        return;
    }

    // Clear the previous entries.
    lstOutput.Items.Clear();

    // Display the total number of sensors.
    lstOutput.Items.Add("Number of Sensors: " + AllSensors.Count);

    // Display the sensor information.
    foreach (Sensor ThisSensor in AllSensors)
    {
        // Output common elements.
        lstOutput.Items.Add("Name: " + ThisSensor.FriendlyName);
        lstOutput.Items.Add("Category ID: " + ThisSensor.CategoryId);
        lstOutput.Items.Add("Type ID: " + ThisSensor.TypeId);
        lstOutput.Items.Add("Description: " + ThisSensor.Description);
        lstOutput.Items.Add("Device Path: " + ThisSensor.DevicePath);
        lstOutput.Items.Add("Manufacturer: " + ThisSensor.Manufacturer);
        lstOutput.Items.Add("Model: " + ThisSensor.Model);
    }
}
```
lstOutput.Items.Add("Serial Number: " + ThisSensor.SerialNumber);
lstOutput.Items.Add("Sensor State: " + ThisSensor.State);

// Check the connection type.
try
{
    // The device may not have a physical connection, so
    // you must place this code in a try block.
    lstOutput.Items.Add("Connection Type: " + ThisSensor.ConnectionType);
}
catch
{
    // The most common device lacking a connection is a software
    // device driver.
    lstOutput.Items.Add("Connection Type: Software Device");
}

// Determine whether the sensor provides a data report.
if (ThisSensor.AutoUpdateDataReport)
{
    // Output the common data.
    lstOutput.Items.Add("Automatic Data Update");
lstOutput.Items.Add("\tMinimum Report Interval: " +
                ThisSensor.MinimumReportInterval);
lstOutput.Items.Add("\tCurrent Report Interval: " +
                ThisSensor.ReportInterval);

    // Try updating the data.
    ThisSensor.TryUpdateData();

    // Process the data report if there is data available.
    try
    {
        lstOutput.Items.Add("\tData Report Source: " +
                        ThisSensor.DataReport.Source);
lstOutput.Items.Add("\tData Report Timestamp: " +
                ThisSensor.DataReportTimeStamp);
lstOutput.Items.Add("\tNumber of Report Values: " +
                ThisSensor.DataReport.Values.Count);

        // Enumerate the values.
        foreach (Guid Key in ThisSensor.DataReport.Values.Keys)
        {
            // Display the key.
            lstOutput.Items.Add("\t\t" + Key);

            // Obtain the values for the key.
            IList<Object> ValueEntries;

            if (ThisSensor.DataReport.Values.TryGetValue(
                Key, out ValueEntries))
            {
                // Display the values.
                foreach (object Value in ValueEntries)
                {
                    lstOutput.Items.Add("\t\t\t" + Value);
                }
            }
        }
    }
}
// Display the values associated with the key.
foreach (Object ValueEntry in ValueEntries)
    lstOutput.Items.Add("\t\t\t" + ValueEntry);
}
}
catch
{
    lstOutput.Items.Add("\tNo Data Available");
}
else
    lstOutput.Items.Add("No Automatic Data Update");

// Get the sensor properties.
IList<PropertyKey> SuppList = ThisSensor.GetSupportedProperties();

// Display the sensor properties.
lstOutput.Items.Add("Properties:");
foreach (PropertyKey PropKey in SuppList)
{
    try
    {
        // This call will error out on software devices in
        // most cases.
        object Property = ThisSensor.GetProperty(PropKey);

        // The actual output property type depends on the device.
        // You'll need to add special handling for each device type
        // you expect to see.
        lstOutput.Items.Add("\tKnown Property: " +
            "Property ID: " + PropKey.PropertyId +
            " Format ID: " + PropKey.FormatId);
    }
    catch
    {
        // Display the unknown property type information.
        lstOutput.Items.Add("\tUnknown Property: " +
            "Property ID: " + PropKey.PropertyId +
            " Format ID: " + PropKey.FormatId);
    }
}

// Add a space between items.
lstOutput.Items.Add("\t");

The application begins by retrieving a list of sensors on the system. You have to place the check within
a try...catch block because the code will create a SensorPlatformException exception if there
aren't any sensors installed on the system. The SensorManager.GetAllSensors() method retrieves
generic information about all the sensors on the system. If you want more specific information, you'll
need to obtain a list of particular sensors as described in the “Obtaining Specific Sensor Information” section of this chapter. On successful return from the `GetAllSensors()` method call, `AllSensors` will contain a list of sensors installed on the system. There are a number of failure modes that will return an error message similar to the one shown in Figure 18-1 at this point.

The code begins displaying information by telling you how many sensors the system found using `AllSensors.Count`. This value will always be greater than 0 when the `GetAllSensors()` method call succeeds. The number of sensors found on your system depends in part on the number of Windows 7 device drivers you’ve installed that provide sensor information. The `AllSensors.Count` value is unlikely to reflect the actual number of sensors installed on your system, since many hardware vendors haven’t updated their drivers to provide this information.

At this point, the code uses a `foreach` loop to begin processing each generic `Sensor` object in `AllSensors`. Even though this is a generic object, you can still obtain considerable information from it. The code begins by listing a number of elements common to all sensors, such as the sensor name, description, manufacturer, model, serial number, and so on. The `State` property tells you the status of the sensor. When a sensor is disabled, as shown in Figure 18-3, you’re limited as to how much information the system will actually provide. In this case, the sensor simply outputs `AccessDenied` as its status to show that it has been disabled.

It’s also possible to detect the connection type. This is one of the first tricky pieces of data you can obtain, because a sensor may not have a connection. The most common reason for a lack of a connection is that the sensor is software-based like the Geosense for Windows sensor. Trying to access the `ConnectionType` property for a sensor that doesn’t have a connection will raise an exception that you must catch. In this case, the code doesn’t perform a lot of analysis; it simply assumes that the sensor is software-based.

The `AutoUpdateDataReport` property tells you whether the sensor will automatically update its data for you. When working with a generic sensor, you don’t have many options for interacting with the sensor. If the `AutoUpdateDataReport` is `false`, the best you can do is say that there isn’t any data to process.

When the `AutoUpdateDataReport` property is `true`, you can obtain some general information about how the sensor collects data, such as the minimum allowable report interval and the current report interval. Some sensors have to have a minimum report interval to avoid data overruns, which result in lost data.
Because this is a generic query of the sensor and not a specific interaction with it, you'll probably want to see sample data, but you won't need specific data for processing. The code calls `ThisSensor.TryUpdateData()` to attempt to get a data update from the sensor. This call automatically overwrites any existing data that the sensor might possess, so you want to use this call carefully when you're actually processing data.

If the sensor has data to process, the code can retrieve the data source, the time that the data was collected, and the number of values that the sensor can provide. A call to any of these properties when there's no data present will cause an exception that you must catch. In this case, the exception simply indicates that there's no data to process.

The `DataReport` property includes three sub-properties of interest. The `Source` property contains the source of the report information. The `TimeStamp` property tells you when the data was collected. The `Values` property is actually a collection of key/value pairs. The `Key` member is a `Guid` object that uniquely identifies a particular set of values. The `Values` member is an `IList<object>` collection of values associated with the `Key`. For example, the Geospace for Windows sensor has a `Key` of `{055c74d8-ca6f-47d6-95c6-1ed3637a0ff4}` that represents the location information as shown in Figure 18-4. The associated `Values` member contains a list of entries that define the location, including longitude, latitude, city, state, and even ZIP code. However, you can't access the `Values` member directly — doing so raises an exception. Instead, the code uses the `TryGetValue()` method with the `Key` as the first argument and a buffer, `ValueEntries`, as the second argument. Notice that you must use the `out` keyword. The return from `TryGetValue()` is a `Boolean` that indicates success or failure. After the code obtains the values, it uses a `foreach` loop to display them.

In addition to the other information you can obtain about sensors, each sensor has a list of properties that it supports. The properties you obtain depend on the sensor, and with a software sensor the properties are going to be an unknown type for the most part unless you can get a matching list of Globally Unique IDentifiers (GUIDs) from the vendor.

The example shows how to obtain a list of properties using the `GetSupportedProperties()` method. The output of this method is an `IList<PropertyKey>` object, `SuppList`. The code relies on a `foreach` loop to process `SuppList`. Each entry is a `PropertyKey` object, `PropKey`. The code next tries to determine the kind of property by calling `GetProperty()` with `PropKey` as the argument.
If there’s a method to turn the PropKey into a property value, the call succeeds. Otherwise, the code throws an exception, and you must process it as an unknown property, as shown in the example. Theoretically, even if the system can’t find the PropKey, you should be able to provide custom processing of the property information using the vendor’s documentation.

Each kind of sensor has different property identifiers. You can see the list of these sensors at http://msdn.microsoft.com/library/dd318969.aspx. For example, when working with the Geosense for Windows sensor, you use the property identifiers found at http://msdn.microsoft.com/library/dd318981.aspx to discover the meaning of each property. Many of the GUIDs you see in the listings and output in this chapter appear in the Sensors.H file provided with the Windows 7 SDK in the \Program Files\Microsoft SDKs\Windows\v7.1\Include folder. Reviewing this header file can solve a number of the problems you’ll encounter trying to decipher some types of sensor data.

**OBTAINING SPECIFIC SENSOR INFORMATION**

An application may begin by performing an inventory of a system looking for particular sensors. However, in order to perform useful work, the application will eventually need to look for a specific sensor. For example, if you want to obtain location information, you’ll need to locate a Location sensor. The following sections move on to the next step by using the Geosense for Windows sensor, working directly with the sensor to obtain location information.

**Understanding the Geosense for Windows Data**

If you’re going to work with a specific sensor, it’s a good idea to know what type of information you’ll receive. This means knowing the values that you’ll obtain from the DataReport.Values property. The Geosense for Windows sensor outputs two keys. The first contains a timestamp, as shown in Figure 18-4. The second contains a list of 10 values, as shown in Table 18-2.

**TABLE 18-2: Geosense for Windows Sensor Data**

<table>
<thead>
<tr>
<th>ELEMENT</th>
<th>DATA TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Double</td>
<td>Latitude</td>
</tr>
<tr>
<td>1</td>
<td>Double</td>
<td>Longitude</td>
</tr>
<tr>
<td>2</td>
<td>Double</td>
<td>Altitude</td>
</tr>
<tr>
<td>3</td>
<td>Double</td>
<td>Accuracy (a value of 0 means that the accuracy level is unknown)</td>
</tr>
<tr>
<td>4</td>
<td>String</td>
<td>Street Number</td>
</tr>
</tbody>
</table>
CHAPTER 18 USING THE SENSOR AND LOCATION PLATFORM

TABLE 18-2 (continued)

<table>
<thead>
<tr>
<th>ELEMENT</th>
<th>DATA TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>String</td>
<td>Street</td>
</tr>
<tr>
<td>6</td>
<td>String</td>
<td>City</td>
</tr>
<tr>
<td>7</td>
<td>String</td>
<td>Region</td>
</tr>
<tr>
<td>8</td>
<td>String</td>
<td>State</td>
</tr>
<tr>
<td>9</td>
<td>String</td>
<td>Postal Code</td>
</tr>
</tbody>
</table>

Configuring the Get Location Example

This example begins with a Windows Forms application. You’ll need to add a series of textboxes to hold the data results, as shown in Figure 18-5, and a Get button (btnGet) to output the results of the query. Name the textboxes: txtLatitude, txtLongitude, txtAltitude, txtAccuracy, txtStreetNumber, txtStreet, txtCity, txtRegion, txtState, and txtZIP.

![Figure 18-5: Provide some textboxes to hold the output data, as shown here.](image)

In addition, you’ll need to add references to Microsoft.WindowsAPICodePack.DLL, Microsoft.WindowsAPICodePack.Shell.DLL, and Microsoft.WindowsAPICodePack.Sensors.DLL. You must also provide the following using statements:

**Initializing the Sensor**

Before you can use the sensor, you need to create an object to access it. The code in Listing 18-2 creates a global object used to access the Geosense for Windows sensor. It also performs some initialization tasks for working with the sensor.

**LISTING 18-2: Configure the sensor for use**

```csharp
// Storage for the location sensors.
SensorList<Sensor> LocationSensors;

// Storage for the Geosense for Windows sensor.
Sensor Geosense;

public frmMain()
{
    // Perform the standard initialization.
    InitializeComponent();

    // Obtain a list of location sensors.
    LocationSensors =
        SensorManager.GetSensorsByCategoryId(SensorCategories.Location);

    // Access the Geosense sensor.
    if (LocationSensors.Count == 1 &&
        LocationSensors[0].FriendlyName == "Geosense Sensor")
        Geosense = LocationSensors[0];
    else
        foreach (Sensor ThisSensor in LocationSensors)
            if (ThisSensor.FriendlyName == "Geosense Sensor")
                Geosense = ThisSensor;

    // Verify that we found the sensor.
    if (Geosense == null)
    {
        MessageBox.Show("Geosense for Windows sensor not installed!" +
            "\nClosing Application");
        Environment.Exit(1);
    }

    // Set an event handler for state changes.
    Geosense.StateChanged +=
        new StateChangedEventHandler(Geosense_StateChanged);

    // Set an event handler for data updates.
    Geosense.DataReportChanged +=
        new DataReportChangedEventHandler(Geosense_DataReportChanged);
}
```
The `SensorList<Sensor>` object, `LocationSensors`, contains a list of specific sensors. In this case, it contains those sensors used to obtain location data. The `Sensor` object, `Geosense`, contains the reference to the Geosense for Windows sensor.

The form's constructor begins with the usual call to `InitializeComponent()`. It then obtains a collection of location sensors by calling `SensorManager.GetSensorsByCategoryId()` with an argument of `SensorCategories.Location`. Notice how this call differs from the `SensorManager.GetAllSensors()` method call shown in Listing 8-1. The output is specific, and only the location sensors will appear in the output collection, `LocationSensors`.

The presence of a location sensor doesn't mean that you've found the Geosense for Windows sensor. The code performs two different checks for the sensor using an `if...else` structure. In the first case, there's only one sensor available, so the code checks its `FriendlyName` property value, which must be "Geosense Sensor". In the second case, the code uses a `foreach` statement to check all the sensors returned for a sensor that has a `FriendlyName` property value of "Geosense Sensor". In both cases, the result is to set `Geosense` to a reference of the sensor.

The code verifies that the Geosense for Windows sensor is present by checking `Geosense` for a `null` value. If the value is `null`, the code outputs an error message and exits using the `Environment.Exit()` method. This approach allows the example to return an error code of 1 in this case.

When the sensor does exist, the code creates event handlers for the two events that the sensor supports: `StateChanged` and `DataReportChanged`. Remember that you can use the IDE to help you create the event handlers. Press Tab after typing `+=` to create the remainder of the event handler statement. Press Tab a second time to create the event handler methods. This process is quick and easy.

### Creating and Handling Sensor Events

After you create a reference to the Geosense for Windows sensor, you can begin interacting with it. The first step is to check the sensor and then initiate a request, as shown in Listing 18-3.

#### Listing 18-3: Perform sensor updates as needed to check the position

```csharp
private void btnGet_Click(object sender, EventArgs e)
{
    // Verify that the Geosense sensor is on.
    if (Geosense.State != SensorState.Ready)
    {
        MessageBox.Show("Please enable the Geosense sensor.");
        return;
    }

    // Perform an update.
    Geosense.UpdateData();
}
```
The code begins by checking the Geosense.State property. If this property doesn’t have a value of SensorState.Ready, the code displays an error message and exits without initiating a request. Otherwise, it initiates a request by calling Geosense.UpdateData().

As previously mentioned, the Geosense for Windows sensor supports two events. Listing 18-4 shows the handlers for these events.

**LISTING 18-4: Use events to respond to sensor changes**

```csharp
void Geosense_DataReportChanged(Sensor sender, EventArgs e)
{
    Guid Location = new Guid("055c74d8-ca6f-47d6-95c6-1ed3637a0ff4");
    try
    {
        // Update the timestamp.

        // Update the location.
        txtStreet.Text = Geosense.DataReport.Values[Location][5].ToString();
        txtRegion.Text = Geosense.DataReport.Values[Location][7].ToString();
        txtZIP.Text = Geosense.DataReport.Values[Location][9].ToString();
    } catch
    {
    }
}

void Geosense_StateChanged(Sensor sender, EventArgs e)
{
    MessageBox.Show("Geosense state changed to: " + Geosense.State.ToString());
}
```
CHAPTER 18 USING THE SENSOR AND LOCATION PLATFORM

The Geosense_DataReportChanged() event handler is the more important of the two event handlers for this example. In this case, the code begins by creating a Guid object, Location, which uses the GUID shown in Figure 18-4 to access the location data values shown in Figure 18-4.

The output code begins by displaying the time of the data report using the TimeStamp property. It then displays all the data values listed in Table 18-2 on-screen by accessing each of the DataReport.Values[Location] entries in turn. Figure 18-6 shows typical output from this example. Of course, your display will show the data for your location.

The Geosense_StateChanged() event handler responds to changes in the Geosense for Windows sensor state. You can use this event handler to ensure that the user doesn’t turn off the sensor between calls. In this case, the event handler simply displays a message box with the new sensor status, as shown in Figure 18-7. The only two states available appear to be Ready and AccessDenied.

![FIGURE 18-6: Typical output data from Geosense for Windows](image)

![FIGURE 18-7: The application tells you when the sensor status changes.](image)

Configuring Geosense for Windows Security

It’s important to understand that not everyone on a system needs access to every sensor. You may, in fact, want to strictly limit access to some sensor types, depending on their function. For example, you’ll notice that the Geosense for Windows sensor comes with a privacy warning as described at http://www.google.com/privacy-lsf.html. In addition, Windows provides the privacy information shown in Figure 18-8 (click the How Is My Privacy Affected? link in the Location and Other Sensors applet to see it).

Everyone can automatically access a sensor after you install it. In order to control access to the sensor, you must manually change its security settings or locate the appropriate entry in the registry. To accomplish this task, click the Geosense Location Sensor link in the Location and Other Sensors applet. You’ll see the Sensor Properties window shown in Figure 18-9. This window provides access to all the sensor settings. In this case, you can change the sensor description, modify the security settings, or uninstall the sensor.
FIGURE 18-8: Sensors can incur privacy issues when you use them.

How does a sensor affect my privacy?

By default, when a sensor is enabled, all programs and users on your computer can access the information from that sensor. Windows notifies you when programs access information about your location by temporarily displaying the Location and Other Sensors icon in the notification area. Windows displays this icon the first time a program or a service accesses your computer's location from a sensor.

Some programs could potentially use personal information (such as your location) from sensors without permission. You can limit programs from accessing information from sensors by user account. For more information, see Change who can access information from a sensor. For information about enabling and disabling sensors, see Enable or disable a sensor.

FIGURE 18-9: All sensors provide properties that you can modify.
Locating the appropriate entry in the registry can be a bit tricky. Begin by polling every numbered subkey of the \HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\Root\SENSOR key. Check the HardwareID value of each of these numbered subkeys for the name of the sensor you want to change (the Geosense for Windows sensor appears as a multi-string entry of Sensors\GeosenseSensor). You must then check the Device Parameters\SensorPermissions subkey. For example, if the Geosense for Windows sensor is the only sensor installed on your system, you can see the security settings in the \HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\Root\SENSOR\0000\Device Parameters\SensorPermissions key. When everyone has access to the sensor, you’ll see a DWORD value entry for the S-1-1-0 well-known Security Identifier (SID) with a value of 1, which enables it (see the list of well-known SIDs at http://support.microsoft.com/kb/243330). When you clear the Everyone entry, this SID value entry will equal 0. Allowing services access to the sensor will add entries for the following SIDs: S-1-5-18, S-1-5-19, and S-1-5-20. Theoretically, you can provide finer-grained security control by modifying the registry entries for the particular sensor, but then you need to worry about potential side effects. The sensor may not work correctly if all services can’t access it, but you’ll find out only through experimentation.

Click the Change Who Can Access Sensors link and you’ll see the User Settings window shown in Figure 18-10. Make sure the correct sensor is selected (the Geosense Location Sensor in this case), then clear or check entries as needed to provide secure access to the sensor. The example shows that the Administrator, John, and three services accounts can access the Geosense for Windows sensor, but not the Guest account.

FIGURE 18-10: Configure the sensor to provide the level of security you feel is necessary.
An interesting thing will happen in the example application when you change access. You’ll actually see two messages. First, you’ll see a message box telling you that the sensor access is denied. Second, you’ll see another message box telling you that the sensor is ready. When you change the sensor access, Windows appears to remove permissions first and then add them back in. The example application will also receive an update event, even though you haven’t clicked Get. You can verify this update by starting the application and making the security changes without clicking Get first. The blanks will automatically fill with information after you clear the second message box telling you that the sensor is ready to use.

**Viewing Location Sensor Activity**

Privacy is a concern when using sensors, including the Geosense for Windows sensor. You may be curious as to who is using the sensor on the system. Fortunately, there’s a quick way to find out. Click the View Location Activity link in the Location and Other Sensors applet. Windows will display a message that it’s adding a snap-in for a few moments, and then you’ll see the Event Viewer applet displayed with the Location Activity entry highlighted, as shown in Figure 18-11.

**FIGURE 18-11:** Windows keeps track of location sensor activity for you.
The event viewer logs tell you everything you need to know about the location sensor access, including the name of the application performing the access and the account name used to run the application. Of course, you get all the normal event log entries, such as the time the access occurred. In addition, you can even determine details such as the Program IDentifi er (PID) and Thread IDentifi er (TID), which is helpful in deciding how often the application accessed the sensor during a single session. The event log entry won’t tell you precisely what kind of access was performed when working with sensors that have multiple functions, but at least you know about the sensor access.

DEVELOPING FOR OTHER SENSOR AND LOCATION DEVICES

The example in this chapter shows how to work with one type of sensor — a location sensor. However, you can develop applications to work with any sort of sensor that Windows 7 supports, and there’s a host of them. It would be nice if Microsoft provided a sensor emulator for developers, but such an emulator isn’t available at the time of this writing. You may find some third-party products online that will prove useful. For example, Validity provides a biometric sensor demo at http://www.validityinc.com/post.aspx?id=186.

The Code Pack comes with two additional sample sensor applications in the \CodePack\Samples\Sensors folder. The first shows how to work with accelerometers and the second shows how to work with ambient light sensors. These two examples are a little more complex than the one in this chapter — they provide a good next step in learning how to work with sensors. Of course, you must have the required sensors installed on your system, with the requisite sensor-aware device drivers. Otherwise, you won’t see anything happen when you run the example code.

As with all the other Code Pack examples, you need to convert the code for use with Visual Studio 2010. However, the code won’t compile in this particular case without error. You’ll see four errors appear on-screen that probably won’t make much sense. Fix the first two errors by opening the Settings1.Designer.CS file in the AccelerationMeasurement project. Highlight all the functional code in that file and click Comment Out the Selected Lines. Perform the same task for the Settings1.Designer.CS file in the AmbientLightMeasurement project. The solution will now compile as expected.

MOVING YOUR APPLICATIONS TO WINDOWS 7

This chapter has taken you a bit into the future. Windows 7 supports a host of external devices today, and you probably have at least a few of them, but the day of the talking refrigerator is still in the future for most people. The idea is an interesting one, though. If you take nothing else away from this chapter, it should be that Windows 7 is the first version of Windows to make it possible to attach a range of sensors and actually use them to develop practical applications. Just how you use this feature depends on your organization’s needs.

The reality is that you might be tasked today with updating applications to use at least a few sensors. A monitor that automatically adjusts to the ambient light is probably going to be a specialty item today, but it will eventually hit the mainstream. Like anything else, monitoring sensors requires system resources. The more sensors your system monitors, the slower it will
get. As part of your update planning, you need to consider where to obtain the resources used to
monitor sensors, how often to monitor the sensors, and what to do with the data once you obtain
it. Applications could become more efficient in some cases by partially relying on historical data
instead of constantly monitoring sensors. It pays to plan the use of sensors carefully. For example,
make sure you obtain spare sensors as future replacements, so that you don’t need to write new
code to accommodate replacements.

Chapter 19 moves in another direction, but it has something in common with this chapter — it also is
working on the fringe of existing programming needs. In Chapter 19 you'll discover how to work with
Windows XP Mode effectively. Windows XP Mode is a special virtual environment for applications
that just don’t work well with Windows 7, no matter how you configure them. Of course, Microsoft
prefers that you update the application to work with Windows 7, but this isn’t always practical.
Organizations often have a huge investment in existing code and can’t afford to simply throw that
investment away. If you have a reliable application, but don’t own the source code, you might find
that you absolutely must rely on Windows XP Mode — at least for the short term.