# 2
# Getting Started

The Kinect for Windows SDK is a toolkit for developing applications for Kinect devices. Developing applications using Kinect SDK is fairly easy and straightforward. The SDK provides an interface to interact with Kinect via system drivers. The SDK includes drivers for the Kinect sensor, which interact with the device, and the OS and APIs interact with the device through program. Overall, the SDK provides an opportunity to the developers to build an application using either managed code (C# and VB.NET) or unmanaged code (C++) using Visual Studio 2010 or higher versions, running on Windows 7 or Windows 8.

Kinect for Windows Developer Toolkit is an additional installer that comes with a set of extended components, such as Face Tracking SDK, which helps to track human faces, and Kinect Studio to record and playback the depth and color stream data. The Developer Toolkit also contains samples and documentation to give you a quick hands-on reference.

While the application development with Kinect SDK is fascinating and straightforward, there are certain things that need to be taken care of during the SDK installation, configuration, and setting up of your development environment. The following is a quick overview of various aspects we'll be discussing in this chapter:

- Understanding the system requirements
- The evolutionary journey of Kinect for Windows SDK
- Installing and verifying the installed components
- Troubleshooting tips and tricks
- Exploring the installed components of SDK
- A quick lap around different features of Kinect for Windows SDK
- The Coding4fun toolkit

By the end of this chapter, you will have everything set up to start development with the Kinect sensor.

# System requirements for the Kinect for Windows SDK

While developing applications for any device using an SDK, compatibility plays a pivotal role. It is really important that your development environment must fulfill the following set of requirements before starting to work with the Kinect for Windows SDK.

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

## Supported operating systems

The Kinect for Windows SDK, as its name suggests, runs only on the Windows operating system. The following are the supported operating systems for development:

- Windows 7
- Windows Embedded 7
- Windows 8

The Kinect for Windows sensor will also work on Windows operating systems running in a virtual machine such as Microsoft HyperV, VMWare, and Parallels.

## System configuration

The hardware requirements are not as stringent as the software requirements. It can be run on most of the hardware available in the market. The following are the minimum configurations required for development with Kinect for Windows:

- A 32- (x86) or 64-bit (x64) processor
- Dual core 2.66 GHz or faster processor
- Dedicated USB 2.0 bus
- 2 GB RAM

# The Kinect sensor

It goes without saying, you need a Kinect sensor for your development. You can use the Kinect for Windows or the Kinect for Xbox sensor for your development.

Before choosing a sensor for your development, make sure you are clear about the limitations of the Kinect for Xbox sensor over the Kinect for Windows sensor, in terms of features, API supports, and licensing mechanisms. We have already discussed this in the *Kinect for Windows versus Kinect for Xbox* section in *Chapter 1*, *Understanding the Kinect Device*.

## The Kinect for Windows sensor

By now, you are already familiar with the Kinect for Windows sensor and its different components. The Kinect for Windows sensor comes with an external power supply, which supplies the additional power, and a USB adapter to connect with the system. For the latest updates and availability of the Kinect for Windows sensor, you can refer to `http://www.microsoft.com/en-us/kinectforwindows/site`.

## The Kinect for Xbox sensor

If you already have a Kinect sensor with your Xbox gaming console, you may use it for development. Similar to the Kinect for Windows sensor, you will require a separate power supply for the device so that it can power up the motor, camera, IR sensor, and so on.

If you have bought a Kinect sensor with an Xbox as a bundle, you will need to buy the adapter / power supply separately. You can check out the external power supply adapter at `http://www.microsoftstore.com`. If you have bought only the Kinect for Xbox sensor, you will have everything that is required for a connection with a PC and external power cable.

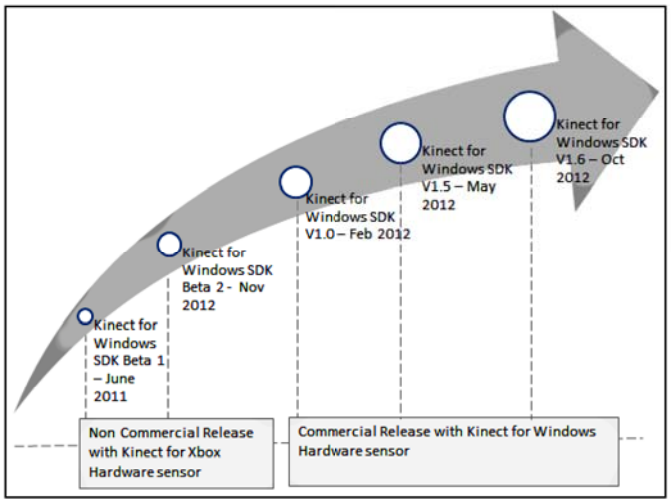
# Development tools and software

The following are the software that are required for development with Kinect SDK:

- Microsoft Visual Studio 2010 Express or higher editions of Visual Studio
- Microsoft .NET Framework 4.0 or higher
- Kinect for Windows SDK

Kinect for Windows SDK uses the underlying speech capability of a Windows operating system to interact with the Kinect audio system. This will require Microsoft Speech Platform – Server Runtime, the Microsoft Speech Platform SDK, and a language pack to be installed in the system, and these will be installed along with the Kinect for Windows SDK. The system requirements for SDK may change with upcoming releases. Refer to `http://www.microsoft.com/en-us/kinectforwindows/` for the latest system requirements.

# Evaluation of the Kinect for Windows SDK

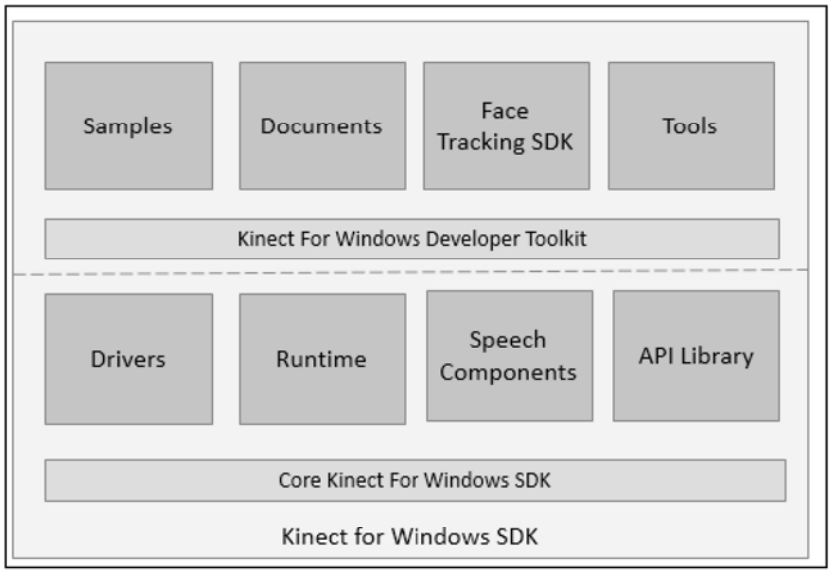



Though the Kinect for Xbox sensor has been in the market for quite some time, Kinect for Windows SDK is still fairly new in the developer paradigm, and it's evolving. The book is written on **Kinect for Windows SDK v1.6**. The Kinect for Windows SDK was first launched as a Beta 1 version in June 2011, and after a thunderous response from the developer community, the updated version of Kinect for Windows SDK Beta 2 version was launched in November 2011. Initially, both the SDK versions were a non-commercial release and were meant only for hobbyists. The first commercial version of Kinect for Windows SDK (v1.0) was launched in February 2012 along with a separate commercial hardware device. SDK v1.5 was released on May 2012 with bunches of new features, and the current version of Kinect for Windows SDK (v1.6) was launched in October 2012. The hardware hasn't changed since its first release. It was initially limited to only 12 countries across the globe. Now the new Kinect for Windows sensor is available in more than 40 countries. The current version of SDK also has the support of speech recognition for multiple languages.

# Downloading the SDK and the Developer Toolkit

The Kinect SDK and the Developer Toolkit are available for free and can be downloaded from `http://www.microsoft.com/en-us/kinectforwindows/`.

The installer will automatically install the 64- or 32-bit version of SDK depending on your operating system. The Kinect for Windows Developer Toolkit is an additional installer that includes samples, tools, and other development extensions. The following diagram shows these components:



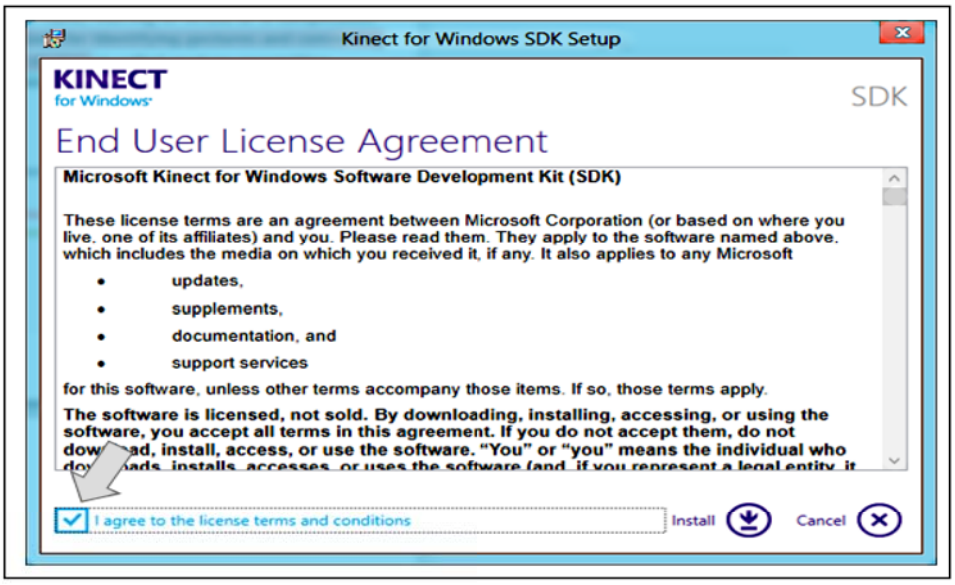

The main reason behind keeping SDK and Developer Toolkit in two different installers is to update the Developer Toolkit independently from the SDK. This will help to keep the toolkit and samples updated and distributed to the community without changing or updating the actual SDK version. The version of Kinect for Windows SDK and that for the Kinect for Windows Developer Toolkit might not be the same.

# Installing Kinect for Windows SDK

Before running the installation, make sure of the following:

- You have uninstalled all the previous versions of Kinect for Windows SDK
- The Kinect sensor is not plugged into the USB port on the computer
- There are no Visual Studio instances currently running

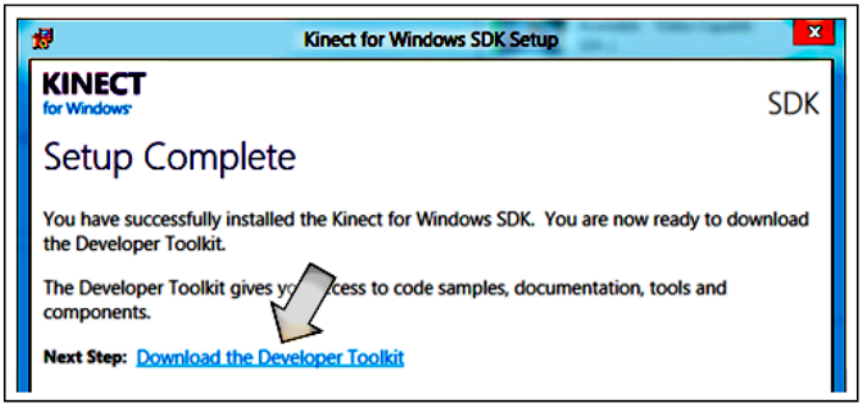
Start the installer, which will display the start screen as **End User License Agreement**. You need to read and accept this agreement to proceed with the installation. The following screenshot shows the license agreement:




Accept the agreement by selecting the checkbox and clicking on the **Install** option, which will do the rest of the job automatically.

Before the installation, your computer may pop out the **User Access Control** (**UAC**) dialog, to get a confirmation from you that you are authorizing the installer to make changes in your computer.

Once the installation is over, you will be notified along with an option for installing the Developer Toolkit, as shown in the next screenshot:




**Is it mandatory to uninstall the previous version of SDK before we install the new one?**

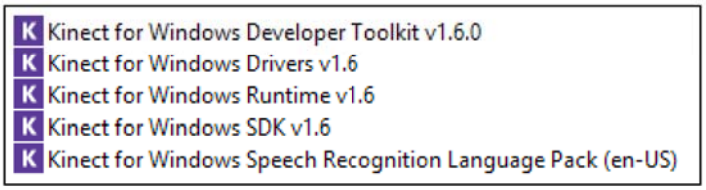
The upgrade will happen without any hassles if your current version is a non-Beta version. As a standard procedure, it is always recommended to uninstall the older SDK prior to installing the newer one, if your current version is a Beta version.

# Installing the Developer Toolkit

If you didn't downloaded the Developer Toolkit installer earlier, you can click on the **Download the Developer Toolkit** option of the SDK setup wizard (refer to the previous screenshot); this will first download and then install the Developer Toolkit setup. If you have already downloaded the setup, you can close the current window and execute the standalone Toolkit installer. The installation process for Developer Toolkit is similar to the process for the SDK installer.

# Components installed by the SDK and the Developer Toolkit

The Kinect for Windows SDK and Kinect for Windows Developer Toolkit install the drivers, assemblies, samples, and the documentation. To check which components are installed, you can navigate to the **Install and Uninstall Programs** section of **Control Panel** and search for `Kinect`. The following screenshot shows the list of components that are installed with the SDK and Toolkit installer:




The default location for the SDK and Toolkit installation is `%ProgramFiles%/Microsoft SDKs/Kinect`.

## Kinect management service

The Kinect for Windows SDK also installs **Kinect Management**, which is a Windows service that runs in the background while your PC communicates with the device. This service is responsible for the following tasks:

- Listening to the Kinect device for any status changes
- Interacting with the COM Server for any native support
- Managing the Kinect audio components by interacting with Windows audio drivers

You can view this service by launching **Services** by navigating to **Control Panel** | **Administrative Tools**, or by typing `Services.msc` in the **Run** command.

**Is it necessary to install the Kinect SDK to end users' systems?**

The answer is *No*. When you install the Kinect for Windows SDK, it creates a `Redist` directory containing an installer that is designed to be deployed with Kinect applications, which install the runtime and drivers. This is the path where you can find the setup file after the SDK is installed:

```
%ProgramFiles%/\Microsoft SDKs\Kinect\v1.6\Redist\
KinectRuntime-v1.6-Setup.exe
```

This can be used with your application deployment package, which will install only the runtime and necessary drivers.

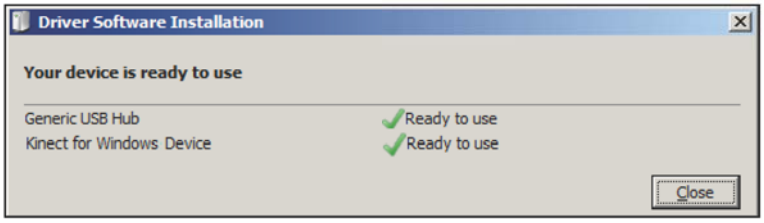
# Connecting the sensor with the system

Now that we have installed the SDK, we can plug the Kinect device into your PC. The very first time you plug the device into your system, you will notice the LED indicator of the Kinect sensor turning solid red and the system will start installing the drivers automatically.

The default location of the driver is `%Program Files%\Microsoft Kinect Drivers\Drivers`.

The drivers will be loaded only after the installation of SDK is complete and it's a one-time job. This process also checks for the latest Windows updates on USB Drivers, so it is good to be connected to the Internet if you don't have the latest updates of Windows.

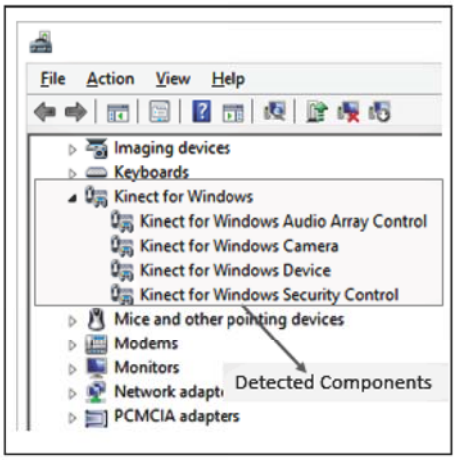
The check marks in the dialog box shown in the next screenshot indicate successful driver software installation:




When the drivers have finished loading and are loaded properly, the LED light on your Kinect sensor will turn solid green. This indicates that the device is functioning properly and can communicate with the PC as well.

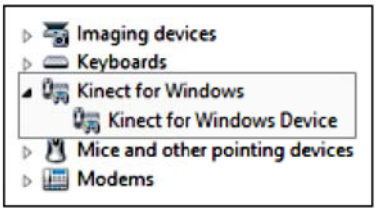
## Verifying the installed drivers

This is typically a troubleshooting procedure in case you encounter any problems. Also, the verification procedure will help you to understand how the device drivers are installed within your system. In order to verify that the drivers are installed correctly, open **Control Panel** and select **Device Manager**; then look for the **Kinect for Windows** node. You will find the **Kinect for Windows Device** option listed as shown in the next screenshot:



## Not able to view all the device components

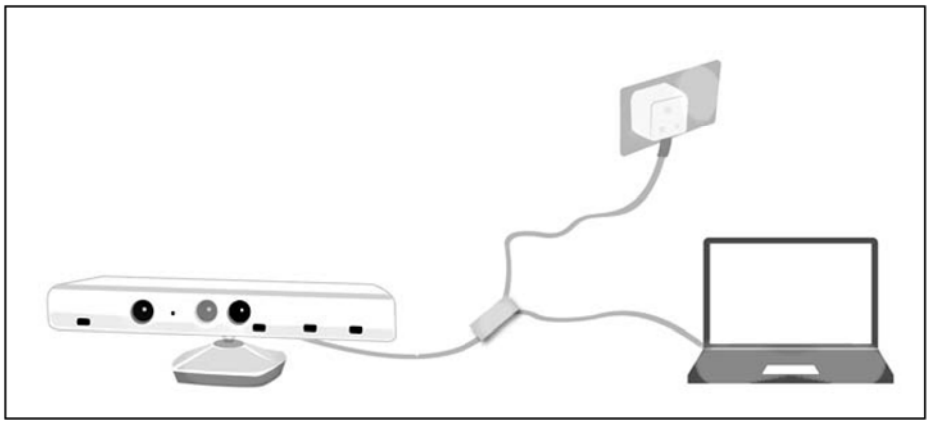
At some point of time, it may happen that you are able to view only the **Kinect for Windows Device** node (refer to the following screenshot). At this point of time, it looks as if the device is ready. However, a careful examination reveals a small hitch. Let's see whether you can figure it out or not! The Kinect device LED is on and **Device Manager** has also detected the device, which is absolutely fine, but we are still missing something here. The device is connected to the PC using the USB port, and the system prompt shows the device installed successfully—then where is the problem?




The default USB port that is plugged into the system doesn't have the power capabilities required by the camera, sensor, and motor. At this point, if you plug it into an external power supplier and turn the power on, you will find all the driver nodes in **Device Manager** loaded automatically.

This is one of the most common mistakes made by the developers. While working with Kinect SDK, make sure your Kinect device is connected with the computer using the USB port, and the external power adapter is plugged in and turned on.

The next picture shows the Kinect sensor with USB connector and power adapter, and how they have been used:



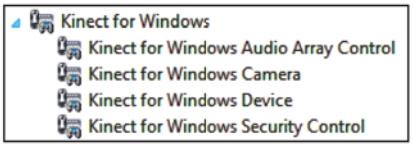
With the aid of the external power supply, the system will start searching for Windows updates for the USB components. Once everything is installed properly, the system will prompt you as shown in the next screenshot:


All the check marks in the screenshot indicate that the corresponding components are ready to be used and the same components are also reflected in **Device Manager**.

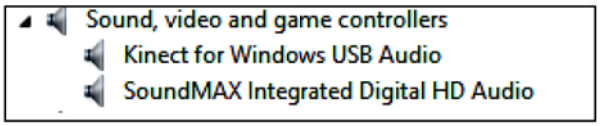
The messages prompting for the loading of drivers, and the prompts for the installation displaying during the loading of drivers, may vary depending upon the operating system you are using. You might also not receive any of them if the drivers are being loaded in the background.

## Detecting the loaded drivers in Device Manager

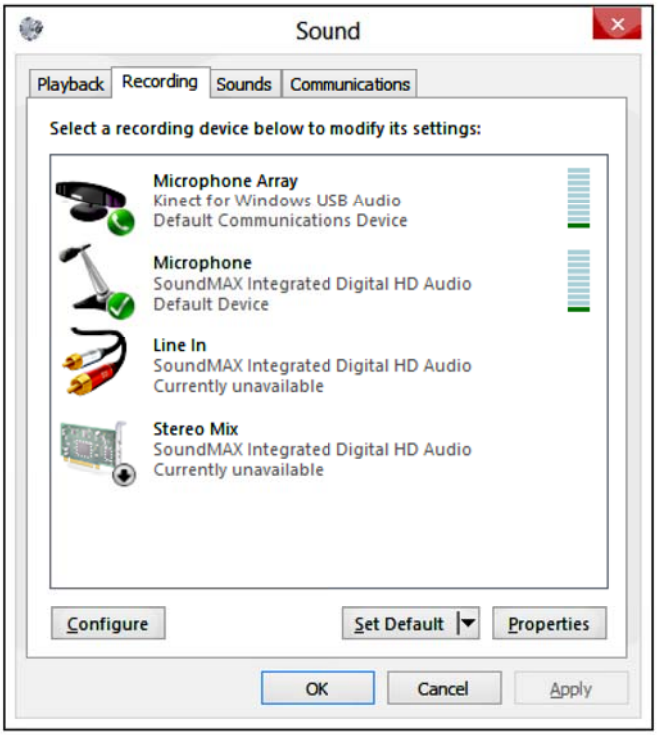
Navigate to **Control Panel** | **Device Manager**, look for the **Kinect for Windows** node, and you will find the list of components detected. Refer to the next screenshot:




The **Kinect for Windows Audio Array Control** option indicates the driver for the Kinect audio system whereas the **Kinect for Windows Camera** option controls the camera sensor. The **Kinect for Windows Security Control** option is used to check whether the device being used is a genuine Microsoft Kinect for Windows or not. In addition to appearing under the **Kinect for Windows** node, the **Kinect for Windows USB Audio** option should also appear under the **Sound, Video and Game Controllers** node, as shown in the next screenshot:




Once the Kinect sensor is connected, you can identify the Kinect microphone like any other microphone connected to your PC in the **Audio Device Manager** section. Look at the next screenshot:

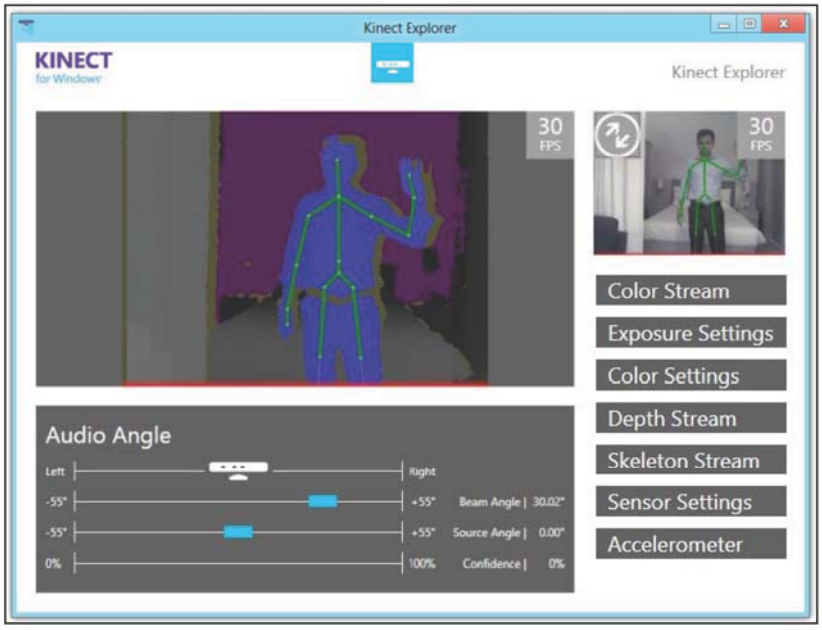



# Testing your device

Once the SDK installation is complete, you are ready to start the development. But let's start with a few bits of testing.

# Testing Kinect sensors

The Developer Toolkit has a set of sample applications; you can choose any of them to test the device. To quickly check it out, you can run the Kinect Explorer application from the Developer Toolkit. The Kinect Explorer demonstrates the basic features of the Kinect for Windows SDK, which retrieves color, depth, and skeleton data and displays them on the UI. The next screenshot shows the UI reference of the application:

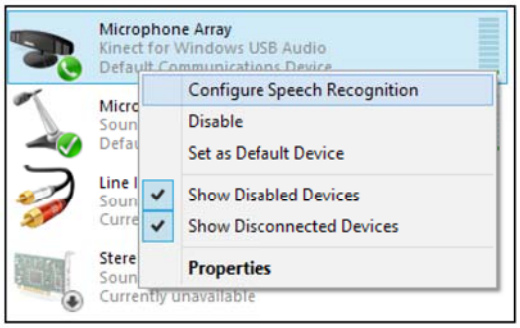



If you are also able to view a similar-looking output, where you can see the Kinect sensor returning the depth, color, and skeleton data, you can be sure that your device has been installed properly.

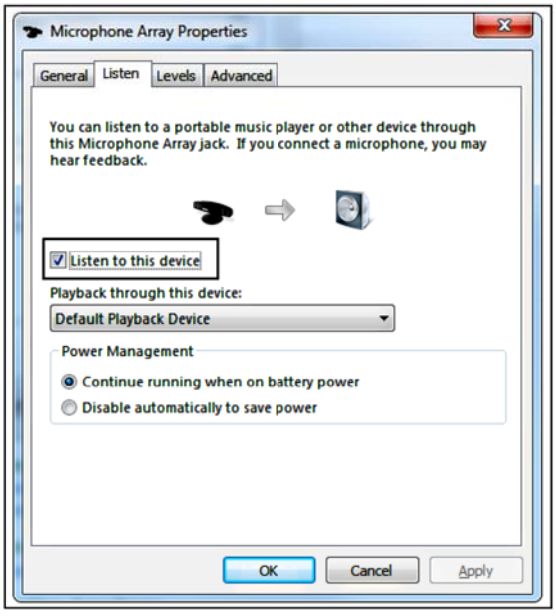
# Testing the Kinect microphone array

You can use Kinect as a microphone. Navigate to the **Sound** section in **Control Panel** and select the **Recording** tab to see Kinect's **Microphone Array**. This is shown in the following screenshot. You can also see the bar along with the microphone array, which indicates the level of sound. You can do everything that a PC's audio tool is capable of doing.

To ensure that the Kinect microphone array is capturing the sound clearly and passing it to your system, you can use the **Listen To this device** option. To enable this, right-click on **Microphone Array** and select **Properties**. This is shown in the next screenshot:




This will launch the **Microphone Array Property** window. Move to the **Listen** tab, select the **Listen to this device** checkbox, and click on **Apply**. This is shown in the next screenshot:




Now, if your system's speaker is turned on and you speak in front of the Kinect device, you should get to hear the same voice via your system's speaker. This ensures that your Kinect audio device is also configured properly.

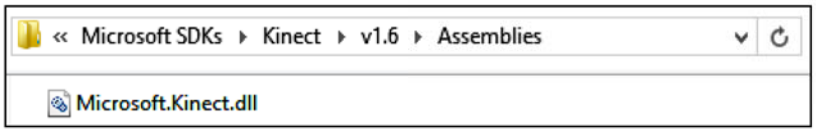
If you want to use multiple Kinect sensors for your application, you can see the detailed procedure given in *Chapter 10, Developing Application Using Multiple Kinects*.

# Looking inside the Kinect SDK

The Kinect SDK provides both managed and unmanaged libraries. If you are developing an application using either C# or VB.NET, you can directly invoke the .NET Kinect Runtime APIs; and for C++ applications, you have to interact with the Native Kinect Runtime APIs. Both the types of APIs can talk to the Kinect drivers that are installed as a part of SDK installation.

The unmanaged and managed libraries provide access to the same set of Kinect sensor features.

For managed code, the Kinect for Windows SDK provides **Dynamic Link Library** (**DLL**) as an assembly (**Microsoft.Kinect.dll**), which can be added to any application that wants to use the Kinect device. You can find this assembly in the SDK installation directory, as shown in the next screenshot:

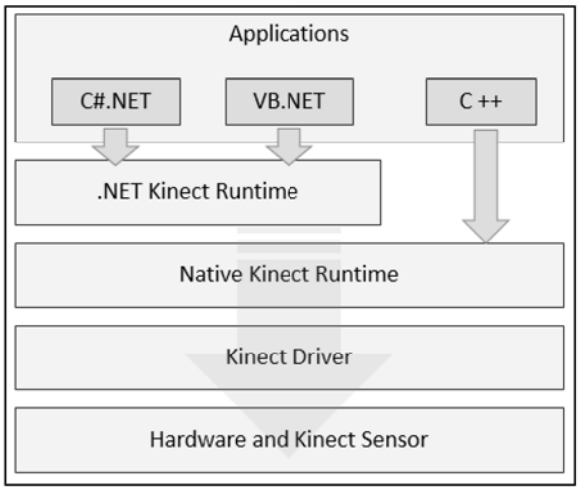



The Kinect driver can control the camera, depth sensor, audio microphone array, and the motor. Data passes between the sensor and the application in the form of data streams of the following types:

- Color data stream
- Depth data stream
- Audio data stream

The Kinect for Windows SDK is capable of capturing Infrared data stream as a part of color data stream channel as well as can read the sensor accelerometer data.

The next diagram illustrates the overall layered components for the Kinect SDK, and it shows how an application interacts with different layers of components:




# Features of the Kinect for Windows SDK

Well, as of now we have discussed the components of the Kinect SDK, system requirements, installation of SDK, and setting up of devices. Now it's time to have a quick look at the top-level features of Kinect for Windows SDK.

The Kinect SDK provides a library to directly interact with the camera sensors, the microphone array, and the motor. We can even extend an application for gesture recognition using our body motion, and also enable an application with the capability of speech recognition. The following is the list of operations that you can perform with Kinect SDK. We will be discussing each of them in subsequent chapters.

- Capturing and processing the color image data stream
- Processing the depth image data stream
- Capturing the infrared stream
- Tracking human skeleton and joint movements
- Human gesture recognition
- Capturing the audio stream
- Enabling speech recognition
- Adjusting the Kinect sensor angle
- Getting data from the accelerometer
- Controlling the infrared emitter

# Capturing the color image data stream

The color camera returns 32-bit RGB images at a resolution ranging from 640 x 480 pixels to 1280 x 960 pixels. The Kinect for Windows sensor supports up to 30 FPS in the case of a 640 x 480 resolution, and 10 FPS for a 1280 x 960 resolution. The SDK also supports retrieving of YUV images with a resolution of 640 x 480 at 15 FPS.

Using the SDK, you can capture the live image data stream at different resolutions. While we are referring to color data as an image stream, technically it's like a succession of color image frames sent by the sensor. The SDK is also capable of sending an image frame on demand from the sensor.

*Chapter 4, Getting the Most Out of Kinect Camera,* talks in depth about capturing color streams.

# Processing the depth image data stream

The Kinect sensor returns 16-bit raw depth data. Each of the pixels within the data represents the distance between the object and the sensor. Kinect SDK APIs support depth data streams at resolutions of 640 x 480, 320 x 240, and 80 x 60 pixels.

## Near Mode

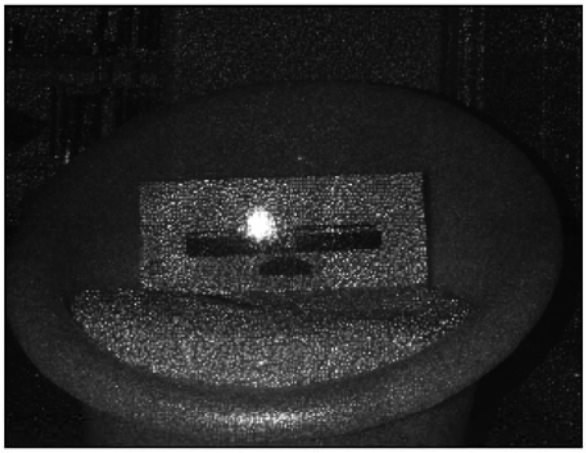
The **Near Mode** feature helps us track a human body within a very close range (of approximately 40 centimeters). We can control the mode of sensors using our application; however, the core part of this feature is built in the firmware of the Kinect sensor.

This feature is limited to the Kinect for Windows sensor only. If you are using the Xbox sensor, you won't be able to work with Near Mode.

We will talk about depth data processing and Near Mode in *Chapter 5, The Depth Data – Making Things Happen.*

# Capturing the infrared stream

You can also capture images in low light conditions, by reading the infrared stream from the Kinect sensor. The Kinect sensor returns 16 bits per pixel infrared data with a resolution of 640 x 480 as an image format, and it supports up to 30 FPS.
The following is an image captured from an infrared stream:



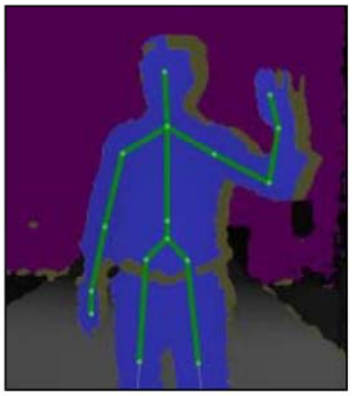
You cannot read color and infrared streams simultaneously, but you can read depth and infrared data simultaneously. The reason behind this is that an infrared stream is captured as a part of a color image format.

# Tracking human skeleton and joint movements

One of the most interesting parts of the Kinect SDK is its support for tracking the human skeleton. You can detect the movement of the human skeleton standing in front of a Kinect device. Kinect for Windows can track up to 20 joints in a single skeleton. It can track up to six skeletons, which means it can detect up to six people standing in front of a sensor, but it can return the details of the full skeleton (joint positions) for only two of the tracked skeletons.

The SDK also supports tracking the skeleton of a human body that is seated. The Kinect device can track your joints even if you are seated, but up to 10 joint points only (upper body part).

The next image shows the tracked skeleton of a standing person, which is based on depth data:



The details on tracking the skeletons of standing and seated humans, its uses, and the development of an application using skeletal tracking, is covered in *Chapter 6, Human Skeleton Tracking*.

# Capturing the audio stream

Kinect has four microphones in a linear configuration. The SDK provides high-quality audio processing capabilities by using its own internal audio processing pipeline. The SDK allows you not only to capture raw audio data, but also high-quality audio processing by enabling the noise suppression and echo cancellation features. You can also control the direction of the beam of the microphone array with the help of the SDK.

We have covered the details of audio APIs of Kinect SDK in *Chapter 7, Using Kinect's Microphone Array*.

# Speech recognition

You can take advantage of the Kinect microphone array and Windows Speech Recognition APIs to recognize your voice and develop relevant applications. You can build your own vocabulary and pass it to the speech engine, and design your own set of voice commands to control the application. If a user says something with some gestures, say while moving a hand as shown in the following picture, an application can be developed to perform some work to be done depending on the user's gestures and speech.


In *Chapter 8, Speech Recognition*, we will discuss the APIs and build some sample applications by leveraging the speech recognition capability of the Kinect for Windows SDK.

# Human gesture recognition

A **gesture** is nothing but an action intended to communicate feelings or intentions to the device. Gesture recognition has been a prime research area for a long time. However, in the last decade, a phenomenal amount of time, effort, and resources have been devoted to this field in the wake of the development of devices. Gesture recognition allows people to interface with a device and interact naturally with body motion, as with the person in the following picture, without any device attached to the human body.


In the Kinect for Windows SDK, there is no direct support for an API to recognize and deal with human gestures; however, by using skeleton tracking and depth data processing, you can build your own gesture API, which can interact with your application.

*Chapter 9, Building Gesture-controlled Applications*, has a detailed discussion about building gesture-controlled applications using the Kinect for Windows SDK.

# Tilting the Kinect sensor

The SDK provides direct access to controlling the motor of the sensor. By changing the elevation angles of the sensors, you can set the viewing angle for the Kinect sensor as per your needs. The maximum and minimum value of elevation angle is limited to +27 degrees and -27 degrees, in SDK. If you try to change the sensor angle more or less than these specified ranges, your application will throw an invalid operation exception.

The tilting is allowed only for the vertical direction. There is no horizontal tilting with Kinect sensors.

We will cover the details of tilting motors and the required APIs in *Chapter 4, Getting the Most Out of Kinect Camera*.

# Getting data from the accelerometer of the sensor

Kinect treats the elevation angle as being relative to the gravity and not its base, as it uses its accelerometers to control the rotation. The Kinect SDK exposes the APIs to read the accelerometer data directly from the sensor. You can detect the sensor orientation by reading the data from accelerometer of the sensor.

# Controlling the infrared emitter

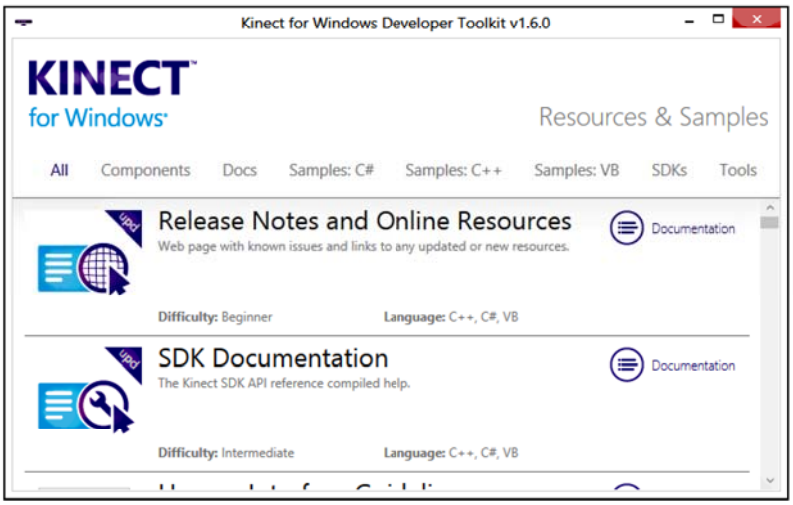
Controlling the infrared emitter is a very small but very useful feature of the Kinect SDK, where you can forcefully turn the infrared emitter off. This is required while dealing with the data from multiple sensors, and when you want to capture data from specific sensors by turning off the IR emitters of other sensors.

This feature is limited only to the Kinect for Windows sensor. If you are using the Xbox sensor, you will get `InvalidOperationException` with the **The feature is not supported by this version of the hardware** message.

# The Kinect for Windows Developer Toolkit

Kinect for Windows Developer Toolkit is an additional set of components that helps you to build sophisticated applications easily by providing access to more tools and APIs. This toolkit has a number of samples, documentation for SDK API libraries, the **Kinect Studio** tool (a tool that can help you record and play Kinect and data during debugging), as well as the **Face Tracking SDK**.

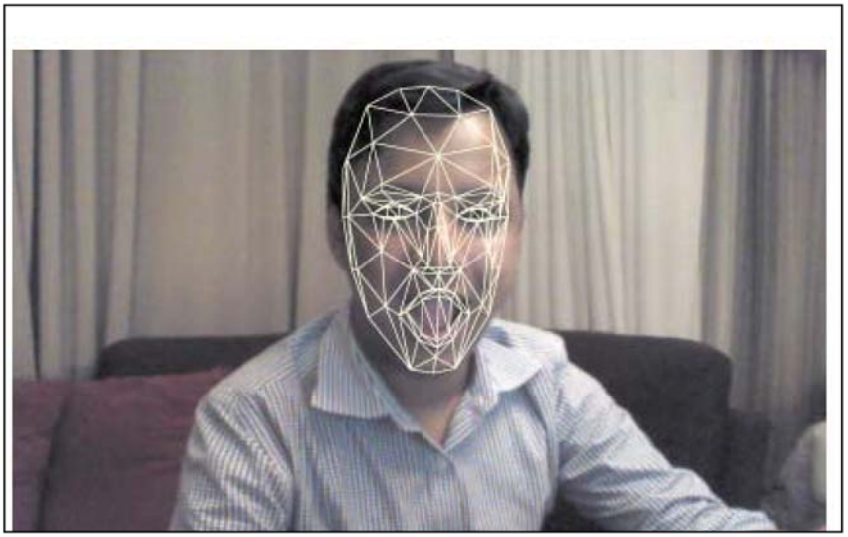
After the installation of Developer Toolkit, you will get a standalone executable within the toolkit that is installed in the directory. Run the application; it will display the screen as shown in the following screenshot. You can navigate through it for resources and samples.




# The Face Tracking SDK

The **Face Tracking SDK** is a part of the Kinect for Windows Developer Toolkit. It contains a few sets of APIs that you can use to track a human face, by taking advantages of Kinect SDK APIs. The SDK detects and tracks the positions and orientations of faces, and it can also animate eye brow positions and the shape of mouth in real time. The Face Tracking SDK can be used in several places, such as recognizing facial expressions, NUI interaction with the face, and tasks that are related to the face.

The next image shows a basic face tracking instance using the Kinect for Windows and Face Tracking SDKs.

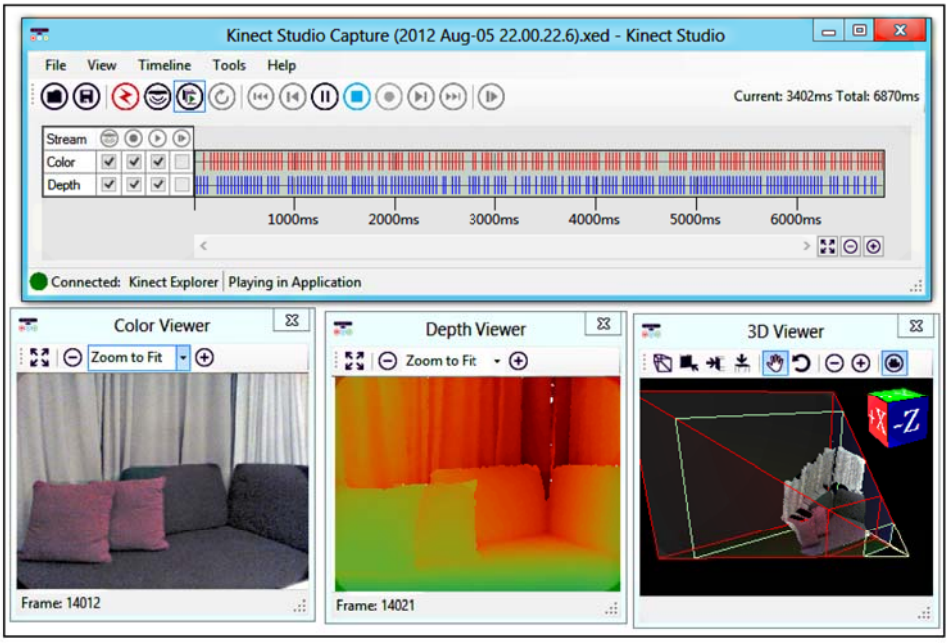


# Kinect Studio

**Kinect Studio** can record and playback the sensor's data stream. It's a very handy and useful tool for developers during testing and while dealing with debugging of Kinect applications. The Kinect data stream can be recorded and saved in a `.xed` file format for future use. What does that mean? How does that help? Well, let's say you are developing an application based on gestures, and you need to perform that gesture every time to test or debug your application; in this case, using the Kinect Studio you can record your action once and just play the recording again and again.

The permission level for both Kinect Studio and the application that is used by Kinect Studio, has to be the same.

The next screenshot shows a quick view of Kinect Studio, which is displaying the color view, depth view, and 3D view of the data captured by Kinect:




I have covered more on the Face Tracking SDK and Kinect Studio in the latter part of this book.

# Making your development setup ready

While the device and driver setup look good, you need to ensure your development environment is ready as well.

The basic software you require for setting up the development environment are Visual Studio 2010 Express Edition or any higher edition along with .NET 4.0 Framework, which we have already discussed as a part of system requirement.

If you are already familiar with development using Visual Studio, the basic steps for implementing an application using a Kinect device should be straightforward. You simply have to perform the following operations:

1. Launch a new instance of Visual Studio.
2. Create a new project.
3. Refer to the `Microsoft.Kinect.dll` file.
4. Declare the appropriate namespaces for the added assembly.
5. Start using the Kinect SDK API library.

We will be discussing the details about development in the next chapter.

# The Coding4Fun Kinect Toolkit

The **Coding4Fun Kinect Toolkit** provides several necessary extension methods to make developing of application using the Kinect for Windows SDK faster and easier. You can download the Coding4Fun Kinect Toolkit from `http://c4fkinect.codeplex.com/`.

The Coding4Fun Kinect Toolkit is also available as a NuGet package. You can install it using the NuGet Package Manager console within Visual Studio.

We will explore some of the extension methods and the installation procedure of Coding4Fun Kinect Toolkit in the upcoming chapters.

# Summary

In this chapter we have covered the prerequisites of an SDK installation, that is, the hardware and software requirements for the installation. A brief step-by-step guide for the installation, loading of drivers, and for setting up the development environment is also discussed. We have seen a quick overview of the Kinect SDK features and also the new Kinect SDK Developer Toolkit. We have also seen how the SDK provides an opportunity to developers to build applications using different languages such as C#, VB.NET, or C++. The knowledge gained from this chapter will help you fully grasp the subjects discussed in the subsequent chapters.

In the next chapter you will get started with development with the Kinect for Windows SDK.

# 3
# Starting to Build Kinect Applications

Let's begin our journey towards developing our first application with Kinect. For the development of every Kinect application, there are certain common operations we need to perform, listed as follows:

- The application must detect the connected Kinect device and needs to start it.
- Once the sensor is started, the application has to initialize and subscribe the type of data required from the sensor.
- During the overall execution cycle of an application, a sensor can change its state. The application must monitor the changes in the state for the connected device and handle them appropriately.
- When the application quits/ends, it should shut down the device properly.

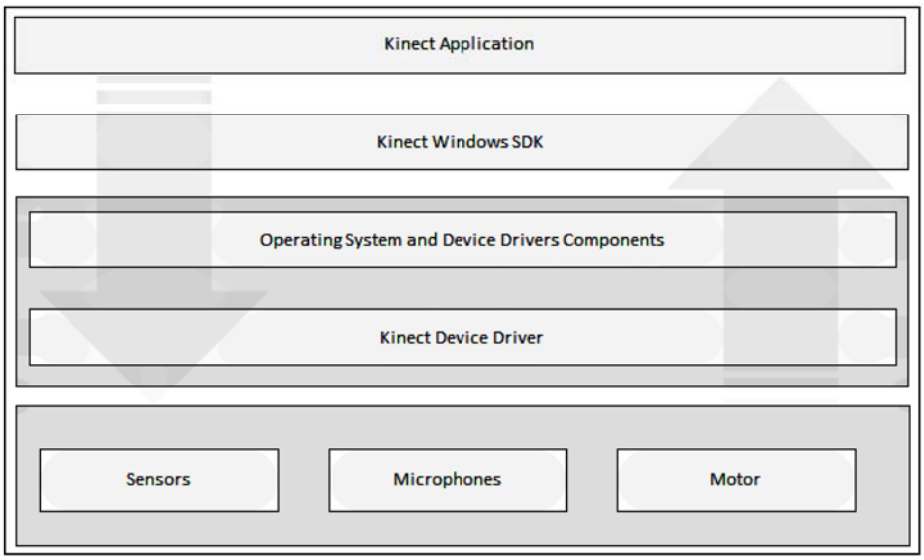
This chapter will cover the basic understanding of the Kinect SDK APIs and the development of applications with the Kinect SDK. We will discuss in a step-by-step manner the development of the applications in this chapter so that it helps you in upcoming chapters. The following is an overview of various aspects we'll be covering in this chapter:

- Getting familiar with the application's interaction with the Kinect device
- Understanding the classification of APIs based on the Kinect SDK libraries used
- Building a Kinect Info Box application by reading sensor information
- Using Kinect libraries in your application
- Exploring SDK libraries that read device information

- Different ways to examine whether Kinect devices are correctly connected and installed and are in working condition
- Building a KinectStatusNotifier application to notify the sensor state change in the system tray

# How applications interact with the Kinect sensor

The Kinect for Windows SDK works as an interface between the Kinect device and your application. When you need to access the sensor, the application sends an API call to the driver. The Kinect driver controls access to sensor data. To take a granular look inside the application interfacing with the sensor, refer to the following diagram:




The installed drivers for the sensors sit with the components of system device drivers and can talk to each other. The drivers help to stream the video and audio data from the sensors and return it to the application. These drivers help to detect the Kinect microphone array as a default audio device and also help the array to interact with the Windows default speech recognition engine. Another part of the Kinect device driver controls the USB hubs on the connected sensor as well.

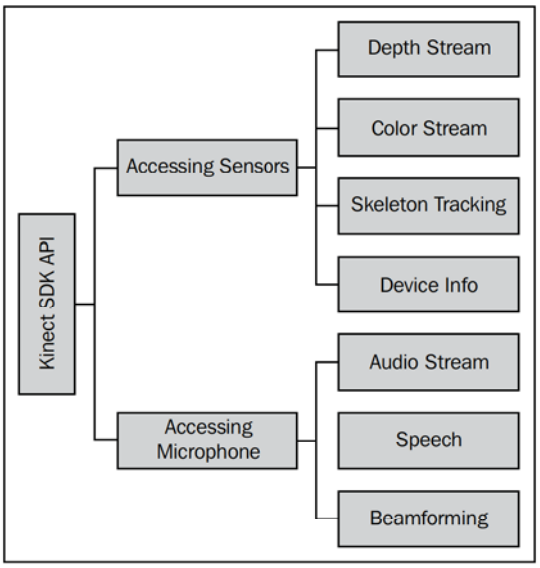
# Understanding the classification of SDK APIs

To understand the functionality of different APIs and to know their use, it is always good to have a clear view of the way they work. We can classify the SDK libraries into the two following categories:

- Those controlling and accessing Kinect sensors
- Those accessing microphones and controlling audio

The first category deals with the sensors by capturing the color stream, infrared data stream, and depth stream, by tracking human skeletons and taking control of sensor initialization. A set of APIs in this category directly talks to the sensor hardware, whereas a few APIs on processing apply the data that is captured from the sensor.

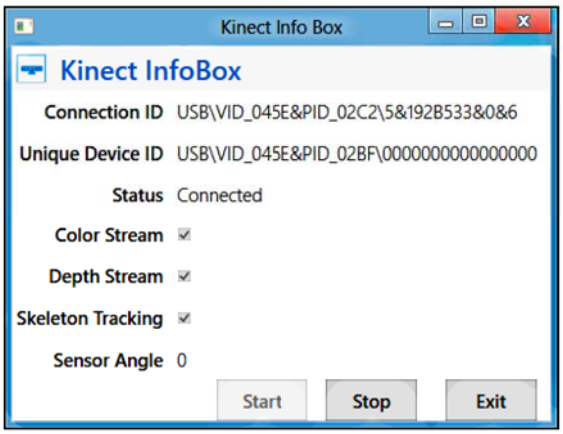
On the other hand, the audio APIs control the Kinect microphone array and help to capture the audio stream from the sensors, controlling the sound source and enabling speech recognition, and so on. The following diagram shows a top-level API classification based on the type of work the API performs:




We can also define the SDK API as a **Natural User Interfaces** (**NUI**) API, which retrieves the data from the depth sensor and color camera and captures the audio data stream. There are several APIs that are written on top of the NUI APIs, such as those for retrieving sensor information just by reading sensor details and for tracking human skeletons based on the depth data stream returned from the sensor.

# Kinect Info Box – your first Kinect application

Let's start developing our first application. We will call this application **Kinect Info Box**. To start development, the first thing we are going to build is an application that reads the device information from a Kinect sensor. The Info Box application is self-explanatory. The following screenshot shows a running state of the application, which shows the basic device information such as **Connection ID**, **Device ID**, and **Status**. The Info Box also shows the currently active stream channel and the sensor angle. You can also start or stop the sensor using the buttons at the bottom of the window.

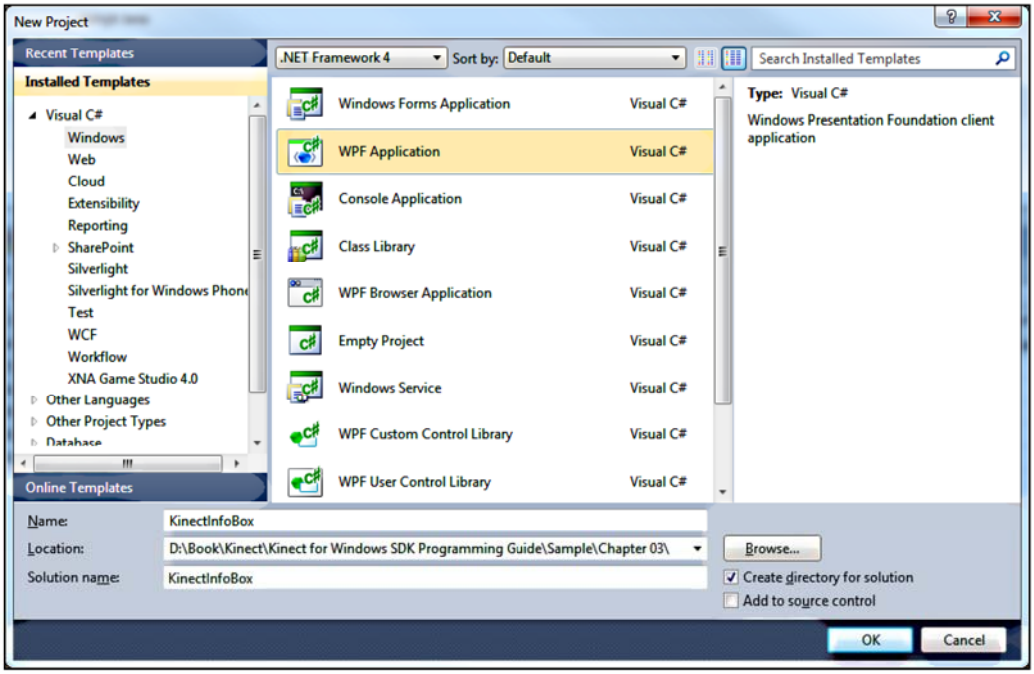



We will build this application in a step-by-step manner and explore the different APIs used along with the basic error-handling mechanisms that need to be taken care of while building a Kinect application.

# Creating a new Visual Studio project

1. Start a new instance of Visual Studio.
2. Create a new project by navigating to **File** | **New Project**. This will open the **New Project** window.

3. Choose **Visual C#** from the installed templates and select the **WPF Application**, as shown in the following screenshot:



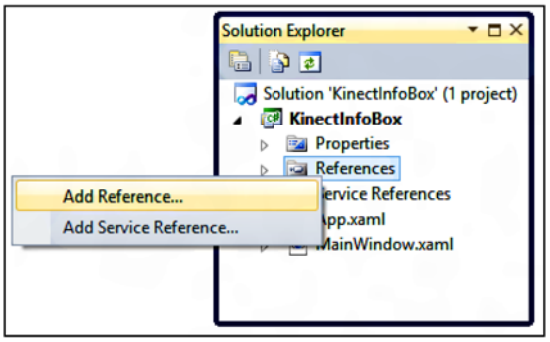

You can select the **Windows Forms Application** template instead of **WPF Application template**. In this book, all the sample applications will be developed using WPF. The underlying Kinect SDK API is the same for both **Windows Forms Application** and **WPF Application**.

4. Give it the name `KinectInfoBox`, and then click on **OK** to create a new Visual studio project.

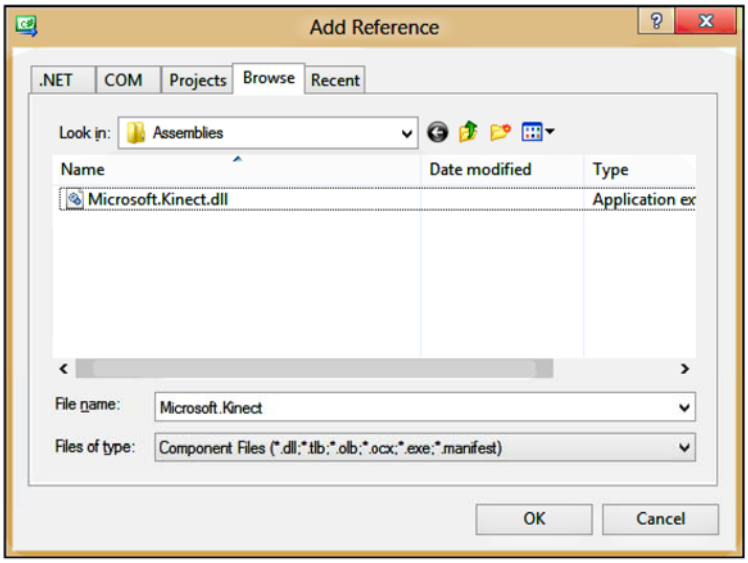
## Adding the Kinect libraries

The next thing you need to do is add the Kinect libraries to the Visual Studio project using the following steps:

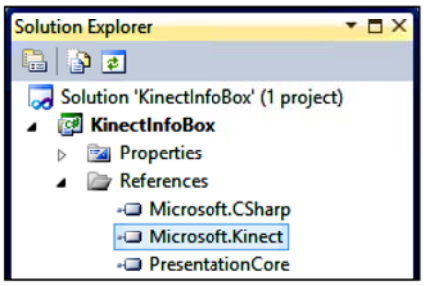
1. From the **Solution Explorer** window, right-click on the **References** folder and select **Add Reference...**, as shown in the following screenshot:




2. This will launch the **Add Reference** window. Then, search for the `Microsoft.Kinect.dll` file within the Kinect SDK folder. Select `Microsoft.Kinect.dll` and click on **OK**, as shown in the following screenshot:



3. This will add `Microsoft.Kinect.dll` as a reference assembly into your project, which you can see within the **References** folder, as shown in the following screenshot:




We now have a default project ready with the Kinect library added. The next thing to do is to access the library APIs for our application.

# Getting the Kinect sensor

The `KinectSensor` class is provided as part of the SDK libraries, which are responsible for most of the operation with the Kinect sensor. You need to create an instance of the `KinectSensor` class and then use this to control the Kinect sensor and read the sensor information.

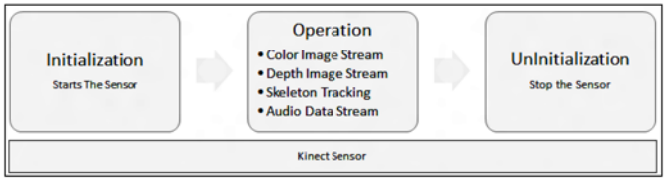
While writing code, the first thing you need to do is to add the `using` directives, which will enable the program to use the Kinect SDK reference. Open the `MainWindow.xaml.cs` file from **Solution Explorer**, and then add the following line of code at the top of your program with the other `using` statement:

```
using Microsoft.Kinect;
```

## The Kinect sensor

In a Kinect application, each Kinect device represents an instance of the `Microsoft.Kinect.KinectSensor` class. This represents the complete runtime pipeline for the sensor during the life span of the application.

The following diagram illustrates the usage of Kinect sensors over a life span of an application:




## Defining the Kinect sensor

Defining the sensor objects is as simple as defining other class objects. The defined object will come into action only when you initialize for a specific Kinect operation, such as color image streaming and depth image streaming. We can define a Kinect sensor object using the following code snippet:

```
public partial class MainWindow : Window
{
    KinectSensor sensor;
    // remaining code goes here
}
```

The `sensor` objects need a reference to the Kinect device that is connected with the system and can be used by your application. You can't instantiate the `KinectSensor` object as it does not have a `public` constructor. Instead, the SDK creates `KinectSensor` objects when it detects a Kinect device attached to your system.

## The collection of sensors

The `KinectSensor` class has a static property of the `KinectSensorCollection` type, named `KinectSensors`, which consists of the collection of sensors that are connected with your system. The `KinectSensor.KinectSensors` collection returns the collection of Kinect devices connected with your system. `KinectSensorCollection` is a read-only collection of the `KinectSensor` type. Each `KinectSensorCollection` class consists of an indexer of the `KinectSensor` object and an event named `StatusChanged`. The following code block shows the definition of the `KinectSensorCollection` class:

```
public sealed class KinectSensorCollection : ReadOnlyCollection<Kinect
Sensor>, IDisposable
    {
        public KinectSensor this[string instanceId] { get; }
        public event EventHandler<StatusChangedEventArgs>
        StatusChanged;
        public void Dispose();
    }
```

As this returns a collection of Kinect devices, you can use any index of that collection to get the reference of a particular sensor. As an example, if you have two devices connected with your system, `KinectSensor.KinectSensors[0]` will represent the first Kinect device and `KinectSensor.KinectSensors[1]` will represent the second Kinect device.

Consider that you have connected a device, so you will get a reference of this connected sensor as shown in this code:

```
this.sensor = KinectSensor.KinectSensors[0];
```

Once you have the sensor object for the Kinect device, invoke the `KinectSensor.Start()` method to start the sensor.

**Check whether any device is connected before you start the sensor**

It is always good practice to first check whether there is any sensor connected with the system before carrying out any operation with the `KinectSensor` objects. `KinectSensors` holds the reference to all connected sensors, and as this is a collection, it has a `Count` property. You can use `KinectSensors.Count` to check the number of devices.

```
int deviceCount = KinectSensor.KinectSensors.Count;
if (deviceCount > 0)
{
    this.sensor = KinectSensor.KinectSensors[0];
    // Rest operation here
}
else
{
    // No sensor connected. Take appropriate action
}
```

# Starting up Kinect

Starting up Kinect means initializing the Kinect sensors with different data streams. The sensor has to first start before reading from itself. You can write a method, such as the following one, that handles the sensor start:

```
private void StartSensor()
{
    if (this.sensor != null && !this.sensor.IsRunning)
    {
        this.sensor.Start();
    }
}
```

The `KinectSensor` class has a property named `IsRunning`, which returns `true` if the sensor is running. You can take advantage of this property to check whether the sensor is running or not. When you call the `sensor.Start()` method, the SDK checks the current sensor status internally before it actually starts the sensor. So, if you forgot to check `sensor.IsRunning`, the SDK will take care of this automatically; however, checking it well in advance will save an unnecessary call. Similarly, calling the `Start()` method multiple times won't cause any problems as the SDK will start the sensor only if it is not running.

In this application, we are instantiating the sensor in the `Window_Loaded` event, as shown in the following code snippet. This will start the sensor when the application starts; however, you can take the connected sensor as a reference and can start it anywhere in your application, based on your requirements.

```
protected void MainWindow_Loaded(object sender, RoutedEventArgs e)
{
    if (KinectSensor.KinectSensors.Count > 0)
    {
        this.sensor = KinectSensor.KinectSensors[0];
        this.StartSensor();
    }
    else
    {
        MessageBox.Show("No device is connected with system!");
        this.Close();
    }
}
```

In the preceding code block, you can see that, first of all, we check the count of the connected sensors and proceed if the number of the connected devices is greater than `0`, otherwise we prompt a message to the user and close the application. This is always a good practice. Once we have connected the sensor, we take the sensor as a reference and start the sensor by calling the `StartSensor` method that we defined earlier.

## Inside the sensor.Start() method

Before taking any further action, the `sensor.Start()` method first checks for the status of the sensor(with the `Status` term). The initialization of the sensor happens only if the sensor is connected. If the sensor is not connected and you are trying to start the sensor, it will throw an `InvalidOperationException` object with the `KinectNotReady` message.

If the sensor is connected, the `Start()` method initializes the sensor and then tries to open the `color`, `depth`, and `skeleton` data stream channels with the default values if they are set to `Enable`.

Initialization of different stream channels does not mean to start sending data from the sensor. You have to explicitly enable the channel and register an appropriate event to feed data from the sensor.

The initialization of the sensor happens with a set of enumeration flags, which provides the type of channel that needs to be initialized. The following table lists the types of initialization options and their descriptions:

| Initialization option | Description |
|---|---|
| `None` | This is the default option, so the sensor will just initialize the channels but it won't open any of them unless we explicitly mention the type of initialization we need. |
| `UseDepthAndPlayerIndex` | This option is used to capture depth stream data as well as the player index from the skeleton tracking engine. |
| `UseColor` | This option enables the color image stream. |
| `UseSkeletonTracking` | This option opens the channel for reading the positions of the skeleton joints from the sensor. |
| `UseDepth` | This option is used to enable the depth data stream. |
| `UseAudio` | To start the audio source, the sensor uses the `UseAudio` initialization option. |

The initialization of the stream channel is achieved by setting the options internally. From a developer's perspective, you just need to call the `Start()` method, and the rest will be taken care of by the SDK.

## Enabling the data streams

The `KinectSensor` class provides specific events that help to enable and to subscribe image stream, depth stream, and skeleton stream data. We will be dealing with details of each and every data stream in subsequent chapters. As of now, we will learn how to enable stream data in our application.

Within the Kinect for Windows SDK, the color, depth, and skeleton data streams are represented by the types of `ColorImageStream`, `DepthImageStream`, and `SkeletonStream` methods respectively. Each of them has an `Enable` method that opens up the stream pipeline. For example, to enable the `ColorStream` type, you need to write the following line of code:

```
this.sensor.ColorStream.Enable();
```

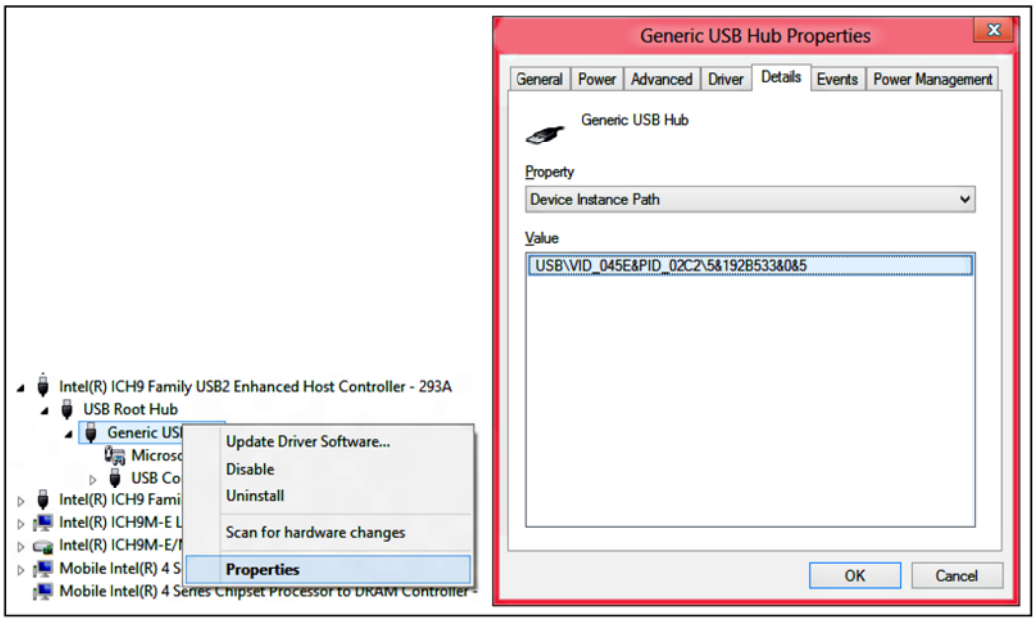
You can explicitly enable the stream only after the `StartSensor()` method, as shown in the following code snippet. In our Info Box application, we did the same by enabling the color, depth, and skeleton data streams after starting the sensor.

```
if (KinectSensor.KinectSensors.Count > 0)
{
    this.sensor = KinectSensor.KinectSensors[0];
    this.StartSensor();
    this.sensor.ColorStream.Enable();
    this.sensor.DepthStream.Enable();
    this.sensor.SkeletonStream.Enable();
}
```

## Identifying the Kinect sensor

Each Kinect sensor can be identified by the `DeviceConnectionId` property of the `KinectSensor` object. The connection ID of this device returns the Device Instance Path of the USB port on which it is connected.

To have a look at it, open **Control Panel** and navigate to **Device Manager**. Then, change the view of Device Manager to **Device by Connection**. Select **Generic USB Hub** for the **Kinect for Windows Device** node and open the **Properties** menu. There you will find the same device ID as you have seen previously. See the following screenshot:




# Initializing the sensor using device connection ID

If you know the device connection ID for your Kinect sensor, you can use the same for instantiating the senor instead of using an index. This will make sure that you are initializing the correct device if there are multiple devices and if you are not sure about the device index. In the following code snippet we have used the previously received unique instance ID:

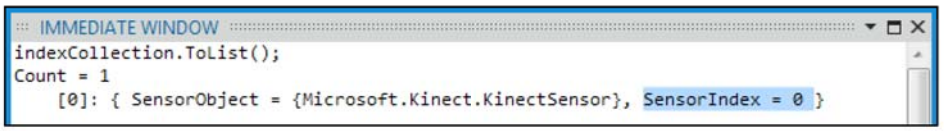
```
KinectSensor  sensor = KinectSensor.KinectSensors [@" USB\VID_045E&PID
_02C2\5&192B533&0&5"];
This.sensor.Start()
```

As this Connection ID returns the USB hub device instance path ID, it will be changed once you plug the device into a different USB hub.

If you are aware of your device ID, you can always refer to the device using the ID as stated in the following code block:

```
KinectSensor sensor = KinectSensor.KinectSensors[@"USB\
VID_045E&PID_02AE\A00362A01385118A"];
int position = 0;
var collection = KinectSensor.KinectSensors.Where(item => item.
DeviceConnectionId == sensor.DeviceConnectionId);
var indexCollection = from item in collection
let row - position++
select new { SensorObject = item, SensorIndex = row };
```

If it's a single sensor, the index should be `0`, but this code block will return the actual position from the list of sensors as well. Currently, we have tested the code with one sensor, so we have the sensor index value `0` as shown in the following screenshot:




The `KinectSensor` object has another property, named `UniqueKinectId`, that returns a unique ID for the Kinect sensor. You can use this ID to identify the sensor uniquely. Also you can use the same to map the index of the sensor.

# Stopping the Kinect sensor

You should call the `sensor.Stop()` method of the `KinectSensor` class when the sensor finishes its work. This will shut down the instance of the Kinect sensor. You can write a method such as the following that deals with stopping the sensor.

```
private void StopSensor()
{
    if (this.sensor != null && this.sensor.IsRunning)
    {
        this.sensor.Stop();
    }
}
```

Like `sensor.Start()`, the `Stop()` method also internally checks for the sensor's running state. The call goes to stop the sensor only if it is running.

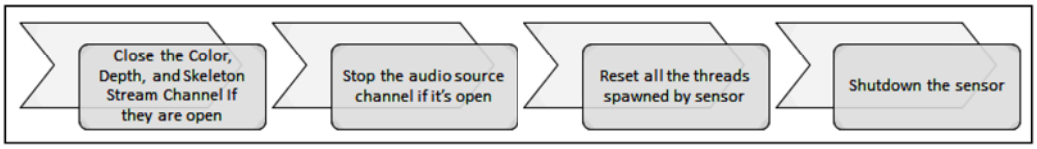
# The Stop() method does the clean-up operation

It's good practice to call `Stop()` once you are done with the Kinect sensor. This is because the `Stop()` method does some clean-up work internally before it actually shuts down the device. It completes the following tasks:

- It stops the depth, color, and skeleton data streams individually by calling the `Close` method if they are open
- It checks for the open Kinect audio source and stops it if it's running

- It kills all the threads that were spawned by events generated by the Kinect device
- It shuts down the device and sets the sensor initialization option to `None`

The following diagram shows the actual flow of the `Stop()` method of a Kinect device:




The Kinect sensor internally uses unmanaged resources to manage the sensor data streams. It sends the stream to the managed application and the `KinectSensor` class uses it. When we close the application, it cannot dispose of the unmanaged stream automatically if we don't forcefully send a termination request by calling the `Stop()` method. The managed `Dispose` method does this as well. So to turn off the device, the application programmer needs to call the `Stop()` method to clear unmanaged resources before the managed application gets closed.

For instance, if you are running the application while debugging, your application will be directly hosted in your `vshost` file and the Visual Studio debugger allows you to run your code line by line. If you close the application directly from Visual Studio, it will ensure that the process gets stopped without any code getting executed. To experience this situation, you can stop the application directly from Visual Studio without calling the `Stop()` method, and you will see the IR light of the device is still on.

**Turning off the IR light forcefully**

You can turn off the IR emitter using the `KinectSensor.ForceInfraredEmitterOff` property. By default, this property is set to `false`. To turn the IR light off, set the property to `true`.

You can test this functionality easily by performing the following steps:

- Once the sensor is started, you will find a red light is turned on in the IR emitter.
- Set `ForceInfraredEmitterOff` to `true`, which will stop the IR emitter; you will find that the IR light is also stopped.

Again, set the `ForceInfraredEmitterOff` property to `false` to turn on the emitter.

# Displaying information in the Kinect Info Box

So far, you have seen how you can use Kinect libraries in your application and how to identify, stop, and start it. In short, you are almost done with the major part of the application. Now, it's time to look at how to display information in the UI.

## Designing the Info Box UI

This application displays information using the `System.Windows.Controls.TextBlock` class inside a `System.Windows.Controls.Grid` class. That is, each cell on the grid contains a `Textblock` component. The following excerpt from the `MainWindow.xaml` file shows how this is accomplished in XAML:

```
<TextBlock Text="Connection ID" Grid.Row="1" Grid.Column="0"
Style="{StaticResource BasicTextStyle}"  />
<TextBlock Text="{Binding ConnectionID}" Grid.Row="1" Grid.Column="1"
Style="{StaticResource BasicContentStyle}" />
```

As we are going to display the information in text format, we will be splitting the window into a number of columns and rows, where each of the individual rows is responsible for showing information for one single sensor. As you can see in the preceding code, we have two `TextBlock` controls. One of them shows the label and another is bound to a property that shows the actual data.

Similar to this, we have several `TextBlock` controls that display the data for different information types. Apart from the text controls, we have button controls to start and stop the sensor.

## Binding the data

Data binding in WPF can be done very easily using the property notification API built into WPF applications. `INotifyPropertyChanged` is a powerful interface in the `System.Component` namespace and provides a standard way to notify binding to UI on a property change.

Implementing the `INotifyPropertyChanged` interface is not mandatory for data binding. You can use direct binding of data by just assigning data into control. Implementing `INotifyPropertyChanged` will allow changes to property values to be reflected in the UI and will help keep your code clean and get the most out of the complex work done by implementing data binding in the UI. As the standard process of data binding, in this section we will give you a step-by-step look into the application as we will be following the same approach across the book for all demo applications.

## A quick look at INotifyPropertyChanged

The `INotifyPropertyChanged` interface itself is a simple. It has no properties and no methods. It has just one event called `PropertyChanged` with two parameters. Refer to the following code snippet; the first parameter is the sender, and the second parameter is `PropertyChangedEventArgs`, which has a property named `PropertyName`:

```
this.PropertyChanged.Invoke(this, new PropertyChangedEventArgs(proper
tyName));
```

## Using INotifyPropertyChanged for data binding

Within our Kinect Info Box project, add a new class named `MainWindowViewModel.cs` and implement the `INotifyPropertyChanged` interface.

The very first thing we are going to do here is wrap the `PropertyChanged` event within a generic method so that we can call the same method for every property that needs to send the notifications.

As shown in the following code block, we have a method named `OnNotifyPropertyChange`, which accepts the `propertyName` variable as a parameter and passes it within `PropertyChangedEventArgs`:

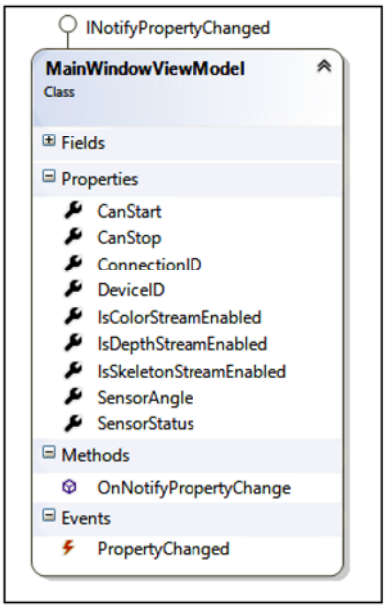
```
public void OnNotifyPropertyChange(string propertyName)
{
    if (this.PropertyChanged != null)
    {
        this.PropertyChanged.Invoke(this, new PropertyChangedEventArgs
        (propertyName));
    }
}
```

The class also contains the list of properties with `PropertyChanged` in the setter block to see if a value is changing. Any changes in the values will automatically notify the UI. This is all very simple and useful. Implementation of any property that needs a notification looks like the following code snippet:

```
private string connectionIDValue;
public string ConnectionID
{
    get
    {
        return this.connectionIDValue;
    }
    set
    {
```

```
            if (this.connectionIDValue != value)
            {
                this.connectionIDValue = value;
                this.OnNotifyPropertyChange("ConnectionID");
            }
        }
    }
```

We need all the properties to be defined in the same way for our `MainWindowViewModel` class. The following diagram is the class diagram for the **MainWindowViewModel** class:




## Setting the DataContext

Binding of the data occurs when the **PropertyChanged** event of the `MainWindowViewModel` class is raised. The `DataContext` property of the `MainWindow` class is set when the class is first initialized in the constructor of the `MainWindow` class:

```
private MainWindowViewModel viewModel;
public MainWindow()
{
    this.InitializeComponent();
    this.Loaded += this.MainWindow_Loaded;
    this.viewModel = new MainWindowViewModel();
    this.DataContext = this.viewModel;
}
```

### Setting up the information

The last thing we need to do is to fill up the `MainWindowViewModel` class instance with the values from the sensor object. The `SetKinectInfo` method does the same job in our Kinect Info Box application. Refer to the following code snippet; we have assigned the `DeviceConnectionId` value of the sensor object, which is nothing but the currently running sensor, to the `connectionID` property of the `ViewModel` object.

```
private void SetKinectInfo()
{
    if (this.sensor != null)
    {
        this.viewModel.ConnectionID = this.sensor.DeviceConnectionId;
        // Set other property values
    }
}
```

Whenever the `SetKinectInfo` method is called, the value of `DeviceConnectionId` is assigned to `ViewModel.connectionID`, and it immediately raises the `OnNotifyPropertyChange` notification, which notifies the UI about the changes and updates the value accordingly.

## That's all!

You are done! You have just finished building your first application. Run the application to see the information about the attached Kinect sensor. While starting the application it will automatically start the sensor; however, you can start and stop the sensor by clicking on the **Start** and **Stop** buttons. In the code behind these two buttons that were just mentioned, are the `SensorStart()` and `SensorStop()` methods, respectively.

# Dealing with the Kinect status

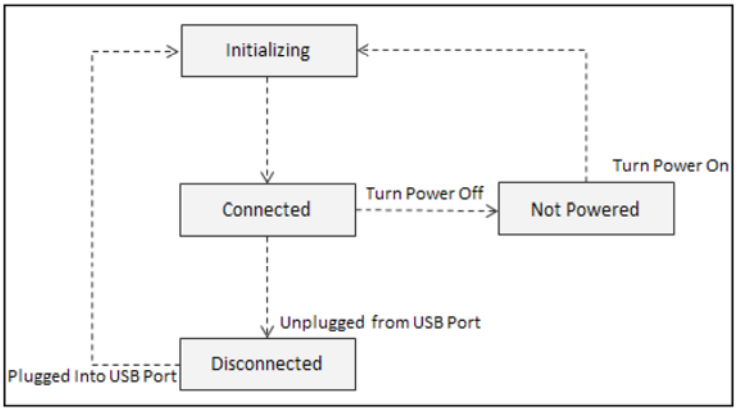
The Kinect sensor needs an external power supply to get the camera, IR sensor, and motor to work properly. Even after following all the standard practices and measures, your system might not detect any of the components of Kinect if there is any problem with your device. Other scenarios that may occur, and that are indeterministic in nature, could be the power suddenly going off while your application is running, , some error occurring in the device, or the device getting unplugged. All the earlier cases can cause your application to crash or throw an unknown exception.

It is of paramount importance to track the device status while your sensor is used by the application. The `KinectSensor` object has a property named `Status`, which indicates the current state of the devices. The property type is of the `KinectStatus` enumeration.

The following table has listed the different status values with their descriptions:

| Status | Description |
|---|---|
| `Connected` | This indicates that the device is connected properly with the system and that it can be used for receiving data using an application. |
| `Error` | The SDK will return this status if the system fails to detect the device properly or there is some internal error. |
| `Disconnected` | This is the status if the device gets disconnected at any point in time while the application is running, which could happen in different scenarios such as a power cut, unplugging the external power source, or even unplugging the USB device from system. |
| `NotReady` | This is the status returned when the device is detected but not loaded properly.<br>Generally, the Kinect SDK detects the device once you have plugged it into the system, but the system loads actual drivers once external power is supplied. This is the intermediate status when the device is not ready to be used. |
| `NotPowered` | This is the status if the device is connected to the system using a USB port, but the external power supplier is not plugged in or is turned off. |
| `Initializing` | This is the status when the device is getting connected or initialized. Generally, this occurs during reconnecting after disconnection or shutdown. |
| `DeviceNotSupported` | This is the status if the device is not supported by the SDK. This is not applicable to the Kinect for Xbox device. |
| `DeviceNotGenuine` | This status can be used to make your application check that you are using only Kinect for Windows devices. This is not applicable to the Kinect for Xbox device. |
| `InsufficicentBandwith` | This is the status when the USB hub is not able to process the complete data sent by the sensor. |
| `Undefined` | This is the status raised for any kind of unhandled issues that can cause the sensor to be undefined or erroneous. |

To understand the flow of the Kinect status and the scenarios in which it can occur can be explained in a simpler way refer to the following diagram. Once the device is connected and the power is turned off, it will show the `NotPowered` status. Similarly, unplugging the device from USB port will return the `Disconnected` status. If you plug it back in or turn the power on, it will first show the `Initializing` status before changing to the `Connected` status.




# Monitoring the change in sensor status

The `KinectSensorCollection` object has only an event named `StatusChanged`, which can be registered as follows:

```
KinectSensor.KinectSensors.StatusChanged += KinectSensors_
StatusChanged;
```

Once the event is registered, it will fire automatically if there are any changes in the device status, internally or externally.

The `StatusChanged` event is registered at the start of your application during the initialization of the sensor.

The `StatusChanged` event fires up with a `StatusChangedEventArgs` class, which holds the `KinectStatus` property and the instance of the sensor by which this event has been raised.

## Properties of the StatusChangedEventArgs class

The following table shows the properties of the `StatusChangedEventArgs` class:

| Name | Description |
|---|---|
| `Sensor` | This property refers to the Kinect sensor that raised this event. This information is useful if you are handling any operation on any status change. If there are multiple Kinect devices, you will get the reference of individual sensors using the property itself. |
| `Status` | This property provides the current status of the Kinect device that raised the events. `KinectStatus` is a flag enumeration defined in Microsoft. Kinect namespaces. |

In the `StatusChanged` event handler, you can check for the status that is returned by the `KinectStatus` enumeration and display the proper message to end users. The uses of different statuses with the `StatusChanged` event handler are shown in the following code snippet:

```
void Kinects_StatusChanged(object sender, StatusChangedEventArgs e)
{
    switch (e.Status)
    {
        case KinectStatus.Connected:
            // Device Connected;
            break;
        case KinectStatus.DisConnected:
            // Device DisConnected;
            break;
    }
}
```

So, you must have noticed that the Kinect SDK is flexible enough to detect the device status this well. This will really help avoid unnecessary exceptions and application crashes.

The `StausChanged` events are attached to all the elements of `KinectSensorCollection`. So, you can track the status change of each and every Kinect device if there is more than one device connected. When the `StatusChanged` event is fired, it invokes the event handler with `StatusChangedEventArgs`, which has associated with the sensor. The following image shows the sensor property of the Kinect `StatusChangedEventArgs` class within the event handler that is raised by the `StatusChanged` event:

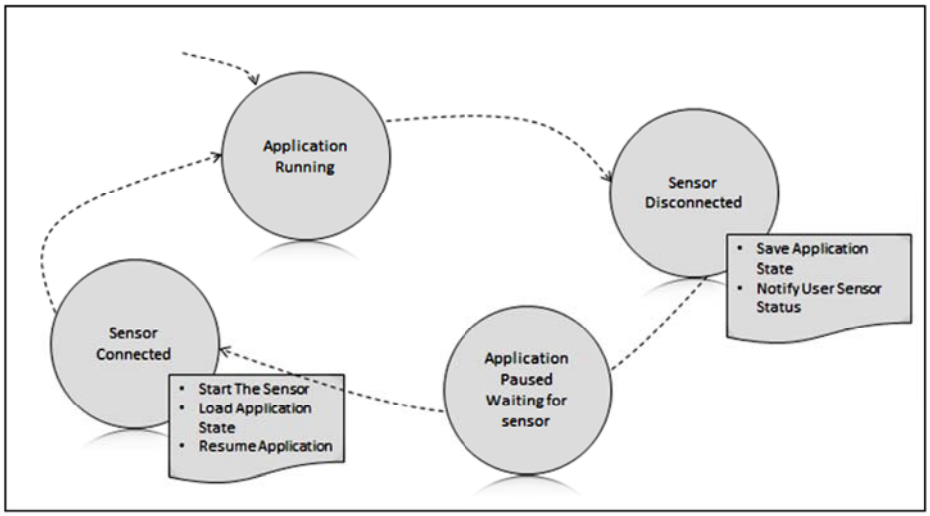
```
void KinectSensors_StatusChanged(object sender, StatusChangedEventArgs e)
{
    e.
}
```

Equals
GetHashCode
GetType
Sensor
Status
ToString

KinectSensor StatusChangedEventArgs.Sensor
Gets the sensor that has changed status.

# Resuming your application automatically

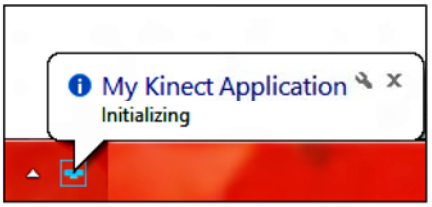
You have also seen that, during the lifespan of a Kinect application the status of the sensor can change. You can start the sensor only when it's in the connected state and you need to call the `Start()` method explicitly to start it. We can take advantage of the `StatusChanged` event to start the sensor and resume our application automatically when it is connected. You can save the state of your application when the status is `Disconnected` or `NotPowered` and can resume it automatically once it is connected by starting the sensor and reloading your application state. This is shown in the following diagram:



# Building KinectStatusNotifier

In this section, we are going to learn how to build a notification application named KinectStatusNotifier that uses the Kinect sensor and shows the sensor status in the system tray. KinectStatusNotifier will pop up a notification icon in the system tray whenever there is a change in the sensor status (refer to the following screenshot).


If you want to show some custom messages with the status change, you can also explicitly call KinectStatusNotifier to notify of a status change in the system tray, as shown in the following screenshot:

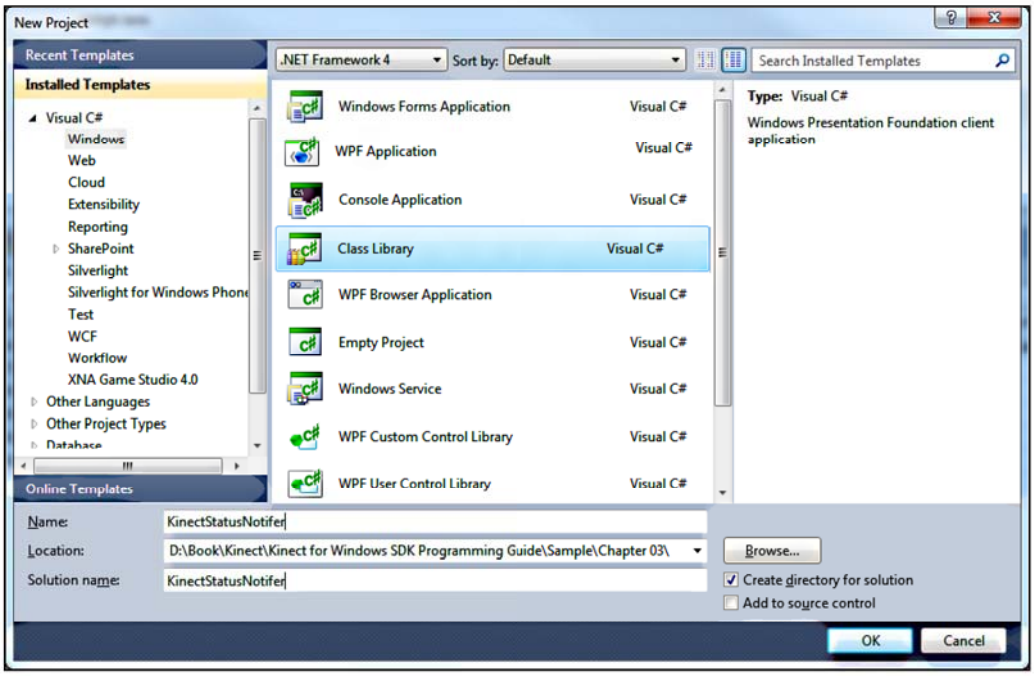



The KinectStatusNotifier application can be useful for any Kinect application, so we will implement it as a general class library so that we can inject it into any Kinect application.

## Setting up an application

We will start this application from scratch with a new `ClassLibrary` project, as follows:

1. Start a new instance of Visual Studio.
2. Create a new project by navigating to **File** | **New Project**.
3. Select the **Visual C#** template and pick the **Class Library** option from the template options.
4. Name the library `KinectStatusNotifier`, as shown in the following image. Click on **OK** to create the project.




`NotifyIcon` is a class in the `System.Windows.Forms` namespace and can be used to invoke the default notification from the system tray. By default, the WPF application does not have `NotifyIcon`, so we are going to create a wrapper around `System.Windows.Forms.NotifyIcon` so that we can easily invoke it from any application.

Perform the following steps to set up our projects:

5. Remove the exiting classes from the **KinectStatusNotifier** project and add a new class by right-clicking on **Project** | **Add New Item** | **Class**. Give it the name `StatusNotifier` and click on **OK**.
6. Add a reference to `System.Windows.Forms` and `System.Drawing` to the `KinectStatusNotifier` project.

## How it works

Once we have the project set up, the first thing we need to do is to create an instance of `NotifyIcon`, as follows:

```
private NotifyIcon kinectNotifier = new NotifyIcon();.
```

`KinectNotifier` now holds the reference to `NotifyIcon` and can be invoked by a status change of the sensor. Hence, we need the reference to `KinectSensorCollection` in the KinectStatusNotifier project.

Add a property of type `KinectSensorCollection`, as follows:

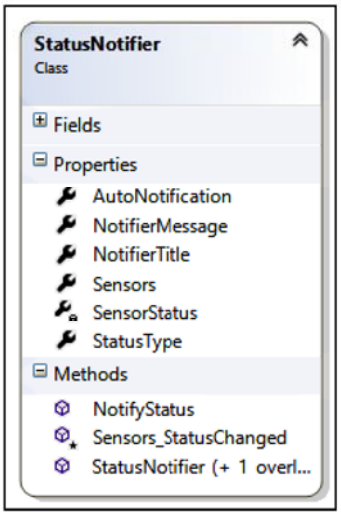
```
private KinectSensorCollection sensorsValue;
public KinectSensorCollection Sensors
{
    get
    {
        return this.sensorsValue;
    }
    set
    {
        this.sensorsValue = value;
        this.AutoNotification = true;
    }
}
```

`Sensors` is a public property of the `StatusNotifier` class that holds the reference to `KinectSensorCollection` that is passed from the calling application. If you have noticed, we have an additional `AutoNotification` property, which is by default set to `true`; however, if you look inside the definition of this property, you will find this:

```
private bool autoNotificationValue;
public bool AutoNotification
{
    get
    {
        return this.autoNotificationValue;
    }
    set
    {
        this.autoNotificationValue = value;
        if (value)
        {
            this.Sensors.StatusChanged += this.Sensors_StatusChanged;
        }
        else
        {
```

```
                this.sensors.StatusChanged -= this.Sensors_StatusChanged;
            }
        }
    }
```

We are subscribing to the `StatusChanged` event handler only when `AutoNotification` is set to `true`. This will give you a choice between using the automatic notification with status change and not using it, as shown in the following screenshot:




The `StatusNotifer` class has a few more properties for the notification title, message, and sensor status, as shown in the preceding class diagram. The `StatusNotifer` class has a defined enumeration called `StatusType`, which is either the information or a warning. The `NotifierMessage` and `NotifierTitle` properties are set in the `Sensor_StatusChanged` event handler, which was registered from the `AutoNotification` property as follows:

```
protected void Sensors_StatusChanged(object sender,
StatusChangedEventArgs e)
{
    this.SensorStatus = e.Status;
    this.NotifierTitle = System.Reflection.Assembly.
    GetExecutingAssembly().GetName().Name;
    this.NotifierMessage = string.Format("{0}\n{1}", this.
    SensorStatus.ToString(), e.Sensor.DeviceConnectionId);
    this.StatusType = StatusType.Information;
    this.NotifyStatus();
}
```

As you can see in the preceding code, the `NotifierTitle` property is set to the name of the application, and the `NotifierMessage` property is set to `SensorStatus` and `DeviceConnectionId`. Finally, the call to the `NotifyStatus()` method sets the `StatusNotifier` property to the `kinectNotifier` instance of the `NotifyIcon` class and invokes the `ShowBallonTip()` method to notify an icon on the system tray. The `NotifyStatus` class is shown in the following code snippet:

```
public void NotifyStatus()
{
    this.kinectNotifier.Icon = new Icon(this.GetIcon());
    this.kinectNotifier.Text = string.Format("Device Status : {0}",
    this.SensorStatus.ToString());
    this.kinectNotifier.Visible = true;
    this.kinectNotifier.ShowBalloonTip(3000, this.NotifierTitle,
    this.NotifierMessage, this.StatusType == StatusType.Information ?
    ToolTipIcon.Info : ToolTipIcon.Warning);
}
```

# Using KinectStatusNotifier

`KinectStatusNotifier` is not a self-executable; it generates a `KinectStatusNotifier.dll` assembly that can be used with a Kinect application. Let's integrate this to our previously built Kinect Info Box application and see how it works. This can be done simply by performing the following steps:

1. Add the `KinectStatusNotifier.dll` assembly as a reference assembly to the Kinect Info Box application from the **Add References** window.
2. Add the following namespace in the application:

   ```
   using KinectStatusNotifier;
   ```

3. Instantiate a new object for the `StatusNotifier` class, as follows:

   ```
   private StatusNotifier notifier = new StatusNotifier();
   ```

4. Assign the `KinectSensor.KinectSensors` collection as a reference to `notifer.Sensors`, as follows:

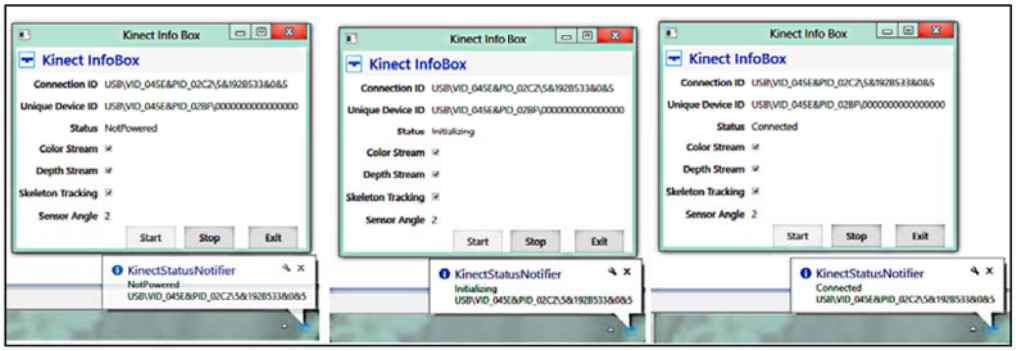
   ```
   this.notifier.Sensors = KinectSensor.KinectSensors;
   ```

That's all! The `StatusNotifer` class will take care of the rest. Whenever there is a change in the status of the sensor, you can see a notification with the current status in the System Tray icon.

You can set the value of `AutoNotification` to `false`, which will stop the automatic notification in the system tray at the `StatusNotifer` class level and invoke the `NotifyIcon` class explicitly when there is a status change. It will do this by handling the `StatusChanged` event handler in your application itself. You can also handle it from both places, while you can change the status in the tray icon from a single place.

## Test it out

To test the application out and see how the application and the sensor work together, first run the Kinect Info Box application and then switch off the power to the sensor and switch it back on. As shown in the following screenshot, you will able to see exactly three different changes in sensor status in the system tray notification:




# Summary

In this chapter we discussed the fundamentals of building Kinect applications. As the sensor is prevalent in many other platforms and devices, the Kinect applications also make substantial use of it in device identification, initialization, and disposing. Thus we have learned a comprehensive view of these approaches in this chapter. We have explored several APIs that help to start and stop the sensor, as well as identifying them and enabling different types of streams by building a small utility. Tracking the Kinect status will safeguard the application from failures and crashes, and we have also discussed a way to do so by notifying the state change in the system tray. With the knowledge gained from this chapter, we will be building a few more complex applications by directly consuming Kinect camera information in the next chapter.