



Essential Studio 2013 Volume 4 - v.11.4.0.26

Essential XlsIO



Contents

1	Overview	7
1.1	Introduction to Essential XlsIO	7
1.2	Prerequisites and Compatibility	9
1.3	Documentation	10
1.4	Advantages of Using Excel Reports	12
1.5	Problems of Using MS Excel to Generate Reports	12
2	Installation and Deployment	14
2.1	Installation.....	14
2.2	Samples Location	14
2.3	Deployment Requirements	25
3	Getting Started	27
3.1	Class Diagram	27
3.2	Creating a Platform Application	28
3.3	Deploying Essential XlsIO	33
3.3.1	Windows	33
3.3.2	ASP.NET	38
3.3.3	WPF.....	42
3.3.4	Silverlight	46
3.3.5	ASP.NET MVC	54
3.4	Web Application Deployment	59
3.5	Feature Summary.....	60
3.6	Spreadsheet	65
3.6.1	Excel Engine.....	65
3.6.2	Workbook	69
3.6.3	Worksheet	74
3.7	Saving a Workbook	82
3.7.1	Excel97to2003.....	83
3.7.2	XLSX.....	84
3.7.2.1	Reducing size of Excel 2007 & Excel 2010 files	92
3.7.3	SpreadsheetML	93

3.7.4	CSV Format.....	96
3.8	Using Templates.....	97
3.9	Improving Performance	100
3.10	Supported Elements	102
3.10.1	Windows, ASP.NET, WPF, ASP.NET MVC	102
3.10.2	Silverlight	106
4	Concepts and Features	110
4.1	Home	110
4.1.1	Formatting	110
4.1.1.1	Font Settings	110
4.1.1.2	Alignment Settings.....	114
4.1.1.3	Number Formatting.....	119
4.1.1.4	Border Settings.....	128
4.1.1.5	Fill Settings	132
4.1.1.6	Styles 136	
4.1.1.6.1	Cell Styles.....	137
4.1.1.6.2	Conditional Formatting	142
4.1.2	Editing.....	161
4.1.2.1	Range Manipulation.....	162
4.1.2.2	Find and Replace	167
4.1.3	Cells.....	176
4.1.3.1	Insert 176	
4.1.3.2	Delete 178	
4.1.3.3	Formats	181
4.1.3.3.1	Changing Cell Size.....	181
4.1.3.3.2	Cell Visibility	189
4.1.3.3.3	Sheet Organization.....	196
4.1.3.3.4	Protection	204
4.1.4	Clipboard	206
4.2	Insert.....	207
4.2.1	Illustrations	207
4.2.2	Charts	210
4.2.2.1	Embedded Chart	210
4.2.2.2	Chart Worksheet.....	218

4.2.2.3	Edit Charts.....	221
4.2.2.3.1	Resizing and Positioning of Chart Elements	224
4.2.2.3.2	Rich-Text Formatting for Chart Elements.....	226
4.2.2.3.3	3-D Chart Wall Settings.....	229
4.2.2.3.4	Filtering Chart Series and Categories	235
4.2.2.4	Sparklines.....	243
4.2.3	Links	248
4.2.4	Header/Footer	251
4.2.5	Tables.....	256
4.2.6	Pivot Tables.....	258
4.2.6.1	Excel 2003.....	258
4.2.6.2	Excel 2007.....	259
4.2.7	PivotCharts	282
4.2.8	Form Controls.....	288
4.2.8.1	Text Box	288
4.2.8.2	Check Box	289
4.2.8.3	Combo Box.....	291
4.2.8.4	Option Button.....	293
4.2.9	OLE Objects	295
4.3	Page Layout	305
4.3.1	Page Setup.....	306
4.3.1.1	Margins306	
4.3.1.2	Orientation	308
4.3.1.3	Paper Size	310
4.3.1.4	Breaks 311	
4.3.1.5	Background	314
4.3.2	Scale To Fit	316
4.3.3	Sheet Options	317
4.3.3.1	Print settings.....	318
4.4	Formulas.....	321
4.4.1	Function Library.....	321
4.4.1.1	Array Formula.....	333
4.4.1.2	External Formula	334
4.4.2	Calculation	335
4.4.2.1	Calculation Options	335

4.4.2.2	Calculation Engine.....	338
4.4.2.2.1	Adding Calculation Engine to an Application	338
4.4.3	Defined Names.....	342
4.4.4	Formula Auditing	347
4.5	Data	350
4.5.1	Filter.....	350
4.5.2	Data Validation	352
4.5.3	Data Sorting for a Given Range of Cells in the Worksheet	357
4.5.3.1	Properties, Methods and Events tables	357
4.5.3.2	Sorting Data by Cell Values	359
4.5.3.3	Sorting by Font Color.....	360
4.5.3.4	Sorting by Cell Color.....	362
4.5.4	Import/Export.....	363
4.5.4.1	Binding Data Objects to Template Markers	372
4.5.4.1.1	Use Case Scenario	372
4.5.4.1.2	Feature Summary.....	372
4.5.4.1.3	Attributes	373
4.5.4.1.4	Binding Business Objects to Template Markers	373
4.5.5	Template Markers.....	378
4.5.6	Outlines	386
4.6	Review.....	391
4.6.1	Comments	391
4.6.2	Changes	395
4.6.2.1	Workbook Protection	395
4.6.2.2	Worksheet Protection	399
4.6.2.3	Edit Range	402
4.7	View	402
4.7.1	Window	402
4.7.1.1	FreezePane	402
4.7.1.2	Split Pane	404
4.7.2	Zooming.....	406
4.7.3	Show/Hide Worksheet Elements.....	407
4.7.4	Macros	410
4.8	Prepare	412
4.8.1	Document Properties.....	412

4.8.2	Encryption and Decryption	414
4.8.2.1	Encryption and Decryption for Excel 2010	419
4.9	Add-Ins	420
5	Frequently Asked Questions	423
5.1	Common	423
5.1.1	How to change the grid line color of the Excel sheet?	423
5.1.2	How to copy and paste the values of the cells that contain only formulas?	423
5.1.3	How to copy a range from one workbook to another?	424
5.1.4	How to ignore the green error marker in worksheets?	425
5.1.5	How to merge several Excel files to a single file?	426
5.1.6	How to open an Excel file from Stream?	426
5.1.7	How to open an existing Xlsx workbook and save it as Xlsx?	427
5.1.8	How to protect certain cells in a spreadsheet?	427
5.1.9	How to save a file to stream?	428
5.1.10	How to set a line break inside a cell?	429
5.1.11	How to set or format a Header/Footer?	429
5.1.12	How to set options to print Titles?	429
5.1.13	How to unfreeze the rows and columns in XlsIO?	430
5.1.14	What is the maximum range of Rows and Columns?	431
5.1.15	How to use Named Ranges with XlsIO?	431
5.1.16	How to create and open Excel Template files by using XlsIO?	432
5.1.17	How to open an Excel 2007 Macro Enabled Template?	433
5.1.18	How to Create Template Markers Using XlsIO?	434
5.1.19	How to add chart labels to scatter points?	438
5.2	Advanced.....	439
5.2.1	How to create a Chart with a discontinuous range?.....	439
5.2.2	How to define discontinuous ranges?	442
5.2.3	How to format text within a cell?.....	443
5.2.4	How to suppress the summary rows and columns using XlsIO?	444
5.2.5	How to zip files using the Syncfusion.Compression.Zip namespace?	445
5.2.6	How to zip all the files in subfolders using the Syncfusion.Compression.Zip namespace?	446
5.2.7	Does Essential XlsIO provide support for Client profile?	450

1 Overview

This section covers information on Essential XlsIO, its key features, and prerequisites to use the control, its compatibility with various operating systems and browsers and finally the documentation details complimentary with the product.

1.1 Introduction to Essential XlsIO

Essential XlsIO is a 100% native .NET library that generates fully functional **Microsoft Excel Spreadsheets** in **native Excel** format without depending on Microsoft Excel. Essential XlsIO is a perfect solution for the users who need to read and write Microsoft Excel files. It does not require MS Excel to be installed in the Report generation machine or server.

Essential XlsIO library can be used in any .NET environment including C#, VB.NET and managed C++. It is a Non-UI component that can be used in **ASP.NET**, **Windows Forms**, **WPF** and **Silverlight** applications, without any change in the API. The usage is common for all the environments, except for the part where the created spreadsheet is saved to disk or stream in the case of a Windows Forms/WPF application and streamed to the client browser in the case of Web applications; there are no temporary Excel files created on the server in an ASP.NET application. Essential XlsIO comes with support for working with formulas in the cells and provides data and worksheet manipulation support. It is known for its high performance for generating Excel files with large number of rows and columns. It is possible to create and modify Excel charts inside a workbook and also has options to secure its contents.

XlsIO can read and write Excel files that are compatible with Excel 97 to Excel 2010. The same Excel file created by using XlsIO can be opened by the following versions of MS Excel. There is no need to create separate files for different versions of Excel.

- Excel 97 (.xls)
- Excel 2000 (.xls)
- Excel 2002 (.xls)
- Excel 2003 (.xls)
- SpreadsheetML (.xlsx)
- CSV (.csv)
- Excel 2007 (.xlsx)
- Excel 2010 (.xlsx)

The following image shows a sample worksheet.

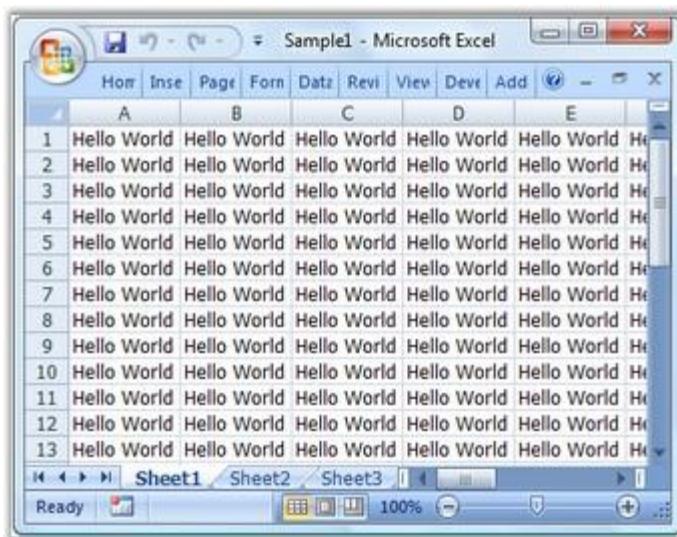


Figure 1: Sample Worksheet

Use Case Scenario

- Desktop Application-A sales tracking application generating a report on sales figures for the past year.
- Web Application-A banking website allowing users to download monthly statements in MS Excel format.

Key Features

- Essential XlsIO provides various formatting options like setting fonts, alignment of content, number formatting, border settings and color-fill settings. It also supports various styles for cells and conditional formatting options.
- It supports range manipulations like copying a range, moving a range, and so on, and Find and Replace option as part of editing the document.
- Various components like chart, pictures and tables can be inserted into the document. It also provides support for insertion of controls like text boxes, check boxes and combo boxes, headers and footers, and links.
- It provides support for page setup options like margin setup, orientation, paper size and page breaks. It also supports several print settings.
- It provides extensive support for using formulae in cells and calculations based on formula function library and calculation engine.
- It provides support for data validation and import/export of data. It also provides data filter and template markers for efficient data-handling.
- Comments can be inserted to any cell in the Excel document by using Essential XlsIO review support.
- Three levels of protection are provided by Essential XlsIO-workbook-level protection, worksheet-level protection and cell-level protection. It also provides encryption and decryption methodology to ensure security of the document.

- Several customizing options like freezing pane, split pane, zooming and macros are supported.
- It provides support for viewing and modifying the properties of the document like Title, Company, Author, Manager, Keywords, and so on.
- Several add-ins for Microsoft Excel are provided.

User Guide Organization

The product comes with numerous samples as well as an extensive documentation to guide you. This User Guide provides detailed information on the features and functionalities of the Essential Chart for MVC. It is organized into the following sections:

- **Overview**-This section gives a brief introduction to our product and its key features.
- **Installation and Deployment**-This section elaborates on the install location of the samples, license, and so on.
- **Getting Started**-This section guides you on getting started with ASP.NET MVC application, controls, and so on.
- **Concepts and Features**-The features of individual controls are illustrated with use case scenarios, code examples and screen shots under this section.
- **Frequently Asked Questions**-This section covers the task-based questions and expert solutions.

Document Conventions

The conventions below will help you to quickly identify the important sections of information, while using the content:

Convention	Icon	Description of the Icon
Note	Note:	Represents important information, to be noted.
Example	Example:	Represents an example.
Tip	Tip:	Represents useful hints, that will help you in using the controls and features.
Additional information	i	Represents additional information on the corresponding topic.

1.2 Prerequisites and Compatibility

This section covers the requirements mandatory for installing Essential XlsIO. It also lists operating systems and browsers compatible with the product.

Prerequisites

The prerequisites details are listed below:

Development Environments	<ul style="list-style-type: none">• Visual Studio 2010 (Ultimate, Premium, Professional and Express)• Visual Studio 2008 (Team System, Professional, Standard & Express)• Visual Studio 2005 (Professional, Standard & Express)• Silverlight 4.0
.NET Framework versions	<ul style="list-style-type: none">• .NET 4.0• .NET 3.5 SP1• .NET 2.0

Compatibility

The compatibility details are listed below:

Operating Systems	<ul style="list-style-type: none">• Windows Server 2008 (32 bit and 64 bit)• Windows 7 (32 bit and 64 bit)• Windows Vista (32 bit and 64 bit)• Windows XP• Windows 2003
-------------------	---

1.3 Documentation

Syncfusion provides the following documentation segments to provide all the necessary information pertaining to Essential XlsIO.

Type of Documentation	Location
-----------------------	----------

Readme	<p>Windows Forms-[drive:]\\Program Files\\Syncfusion\\Essential Studio\\<Version Number>\\Infrastructure\\Data\\Release Notes\\readme.htm</p> <p>ASP.NET-[drive:]\\Program Files\\Syncfusion\\Essential Studio\\<Version Number>\\Infrastructure\\Data\\asp release notes\\readme.htm</p> <p>WPF-[drive:]\\Program Files\\Syncfusion\\Essential Studio\\<Version Number>\\Infrastructure\\Data\\WPF release notes\\readme.htm</p> <p>Silverlight-[drive:]\\Program Files\\Syncfusion\\Essential Studio\\<Version Number>\\Infrastructure\\Data\\Silverlight Release Notes\\readme.htm</p> <p>ASP.NET MVC-[drive:]\\Program Files\\Syncfusion\\Essential Studio\\<Version Number>\\Infrastructure\\Data\\MVC Release Notes\\readme.htm</p>
Release Notes	<p>Windows Forms-[drive:]\\Program Files\\Syncfusion\\Essential Studio\\<Version Number>\\Infrastructure\\Data\\Release Notes\\Release Notes.htm</p> <p>ASP.NET-[drive:]\\Program Files\\Syncfusion\\Essential Studio\\<Version Number>\\Infrastructure\\Data\\asp release notes\\Release Notes.htm</p> <p>WPF-[drive:]\\Program Files\\Syncfusion\\Essential Studio\\<Version Number>\\Infrastructure\\Data\\WPF release notes\\Release Notes.htm</p> <p>Silverlight-[drive:]\\Program Files\\Syncfusion\\Essential Studio\\<Version Number>\\Infrastructure\\Data\\Silverlight Release Notes\\Release Notes.htm</p> <p>ASP.NET MVC-[drive:]\\Program Files\\Syncfusion\\Essential Studio\\<Version Number>\\Infrastructure\\Data\\MVC Release Notes\\Release Notes.htm</p>

	Notes\Release Notes.htm
User Guide (this document)	<p>Online</p> <p>http://help.syncfusion.com/resources (Navigate to the XlsIO User Guide.)</p>  <p>Note: Click Download as PDF to access a PDF version.</p> <p>Installed Documentation</p> <p>Dashboard -> Documentation -> Installed Documentation.</p>
Class Reference	<p>Online</p> <p>http://help.syncfusion.com/resources (Navigate to the Reporting User Guide. Select XlsIO, and then click the Class Reference link found in the upper right section of the page.)</p> <p>Installed Documentation</p> <p>Dashboard -> Documentation -> Installed Documentation.</p>

1.4 Advantages of Using Excel Reports

Microsoft Excel is the most widely used Spreadsheet application in the world, which makes XLS an ideal reporting format for .NET applications. Some of the advantages of using Excel reports over other alternatives [like HTML, PDF etc.] are:

- The report generated can contain rich formats like Charts, Pictures, Multiple Worksheets, Formulae, and so on.
- The end-user can use the Data Visualization power of MS Excel to manipulate the generated reports. For example, the end-user can generate several charts to analyze and understand the sales numbers in a product sales report.
- The popularity of MS Excel guarantees that the end-user is already familiar with using MS Excel.

1.5 Problems of Using MS Excel to Generate Reports

MS Excel was not designed to be a report generation library, so it has several disadvantages compared to XlsIO. Here is a list of some of the problems in using Excel as a reporting component:

- Microsoft, themselves do not recommend using Excel as a report generation server-side component. The reasons are clearly explained in the following Knowledge Base article from Microsoft: <http://support.microsoft.com>
- Here is a quote from the article "Microsoft does not currently recommend, and does not support, Automation of Microsoft Office applications from any unattended, non-interactive client application or component (including ASP, DCOM, and NT Services), because Office may exhibit unstable behavior and/or deadlock when run in this environment."
- **Speed**-Excel automation is about 100 times slower than Essential XlsIO while generating reports.
- **Cost**-Licensing XlsIO is much cheaper than MS Excel licensing options. Here is a link to a Knowledge Base article from Microsoft-<http://support.microsoft.com>.
- **Usability**-Essential XlsIO has a very intuitive object model that is modeled after the Excel object model, making it easy to work. XlsIO also includes some additional helper functionalities like the **ImportDataTable** method which makes programming with XlsIO much easier than using COM automation. Intellisense comments also make the job of programming with XlsIO much easier than Excel automation.

2 Installation and Deployment

2.1 Installation

For step-by-step installation procedure of Essential Studio, refer to the **Installation** topic under **Installation and Deployment** in the **Common UG**:

Common UG -> Installation and Deployment -> Installation topic

See Also

For licensing, patches and information on adding or removing selective components refer the following topics in **Common UG** under **Installation and Deployment**.

- Licensing
- Patches
- Add/Remove Components

2.2 Samples Location

This section covers the location of the installed samples and describes the procedure to run the samples through the sample browser and online.

Sample Installation Locations

Various locations of samples for Essential XlsIO corresponding to each platform are given below:

- **Windows Forms Samples**-The Windows Forms samples are installed under the following location.

*...\\My Documents\\Syncfusion\\EssentialStudio\\Version
Number\\Windows\\XlsIO.Windows\\Samples\\2.0*

- **ASP.NET Samples**-The ASP.NET samples are installed under the following location.

...\\My Documents\\Syncfusion\\EssentialStudio\\Version Number\\Web\\xlsio.web\\Samples\\2.0

- **WPF Samples**-The WPF samples are installed under the following location.

...\\My Documents\\Syncfusion\\EssentialStudio\\Version Number\\WPF\\xlsIO.WPF\\Samples\\3.5

- **Silverlight Samples**-The Silverlight samples are installed under the following location.

...\\My Documents\\Syncfusion\\EssentialStudio\\Version Number\\Silverlight\\xlsIO.Silverlight\\Samples\\3.5

- **ASP.NET MVC Samples**-The ASP.NET MVC samples are installed under the following location.

...\\My Documents\\Syncfusion\\EssentialStudio\\<Version Number>\\MVC



Note: Essential XlsIO does not provide direct support for Silverlight and cannot generate documents in client side. It is only possible to generate the documents in server side and view it in client interface. Samples in the sample browser use web services and generate the documents in the server side.

Viewing Samples

To view the samples:

1. Click Start-->All Programs-->Syncfusion -->Essential Studio <version number> -->Dashboard.
2. Click **Reporting** Edition.

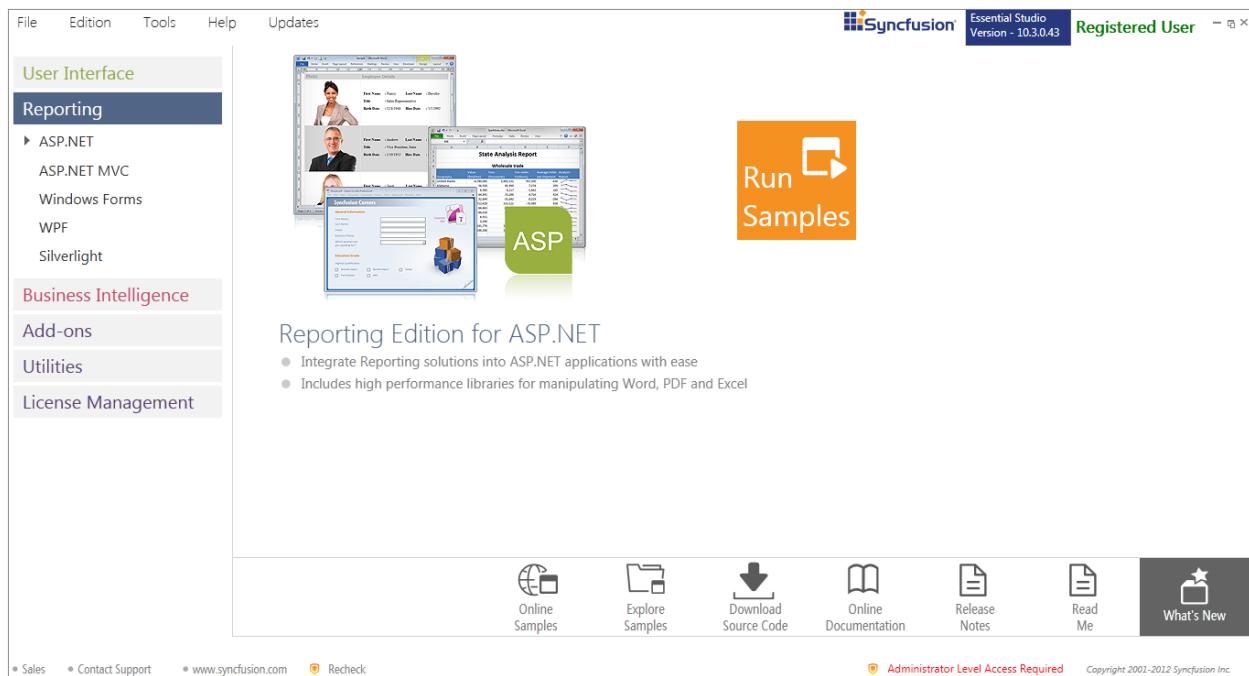


Figure 2: Syncfusion Essential Studio Reporting Dashboard

The steps to view the XlsIO samples in various platforms are discussed below:

Windows Forms

1. In the dashboard window, click **Run Samples** for **Windows Forms** under **Reporting Edition** panel. The **Windows Forms** Sample Browser window is displayed.



Note: You can view the samples in any of the following three ways:

- **Run Samples** - Click to view the locally installed samples.
- **Online Samples** - Click to view online samples.
- **Explore Samples** - Explore Windows Forms samples on disk.

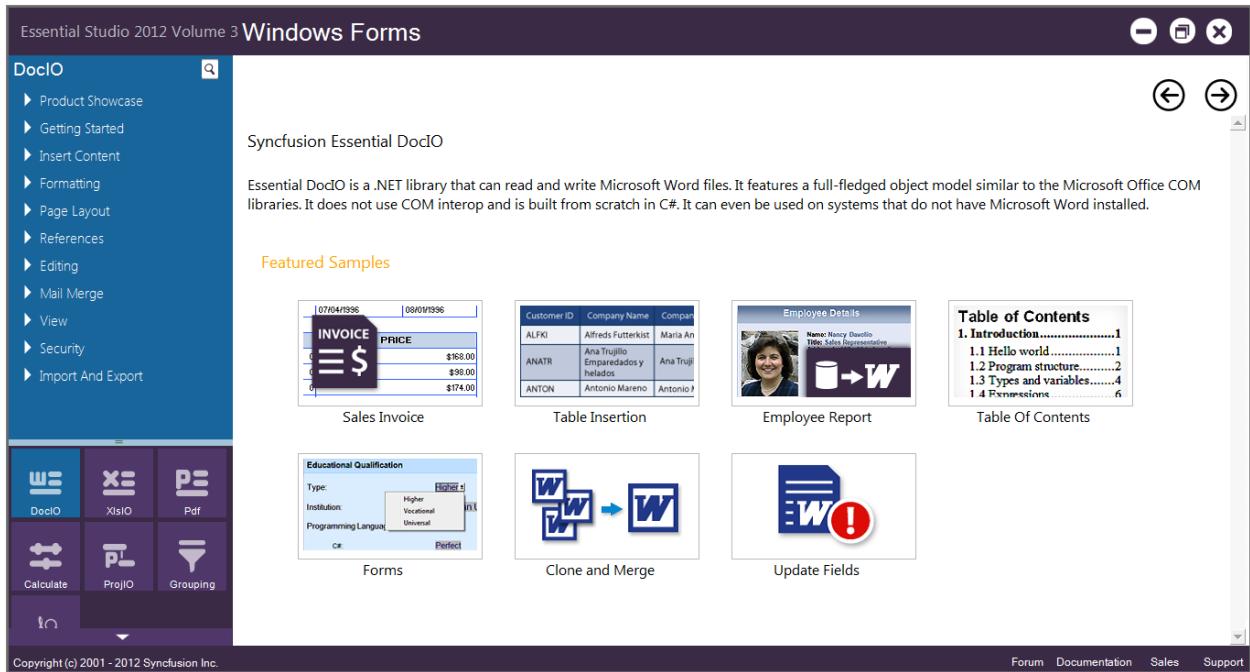


Figure 3: Windows Forms Sample Browser

- Click **XlsIO** from the bottom-left pane. The XlsIO samples are displayed.

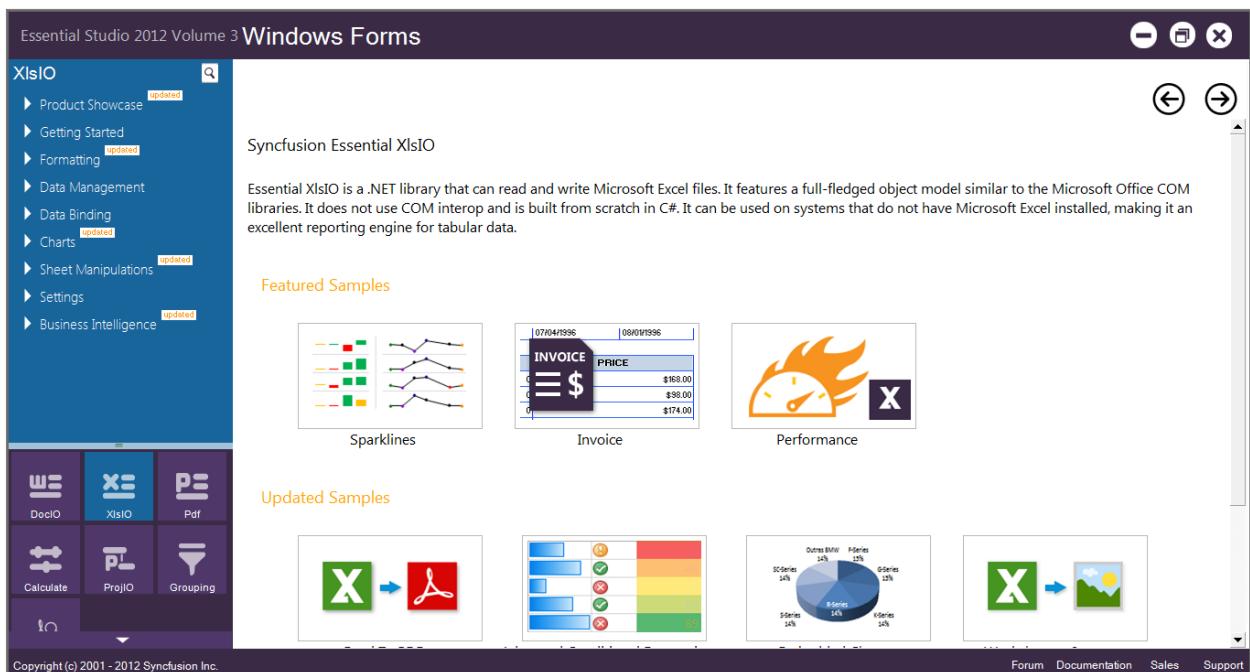


Figure 4: XlsIO Samples Displayed in the Windows Forms Sample Browser

- Select any sample and browse through the features.

ASP.NET

1. In the dashboard window, click **Run Samples** for **ASP.NET** under **Reporting Edition** panel. The **ASP.NET Sample Browser** window is displayed.

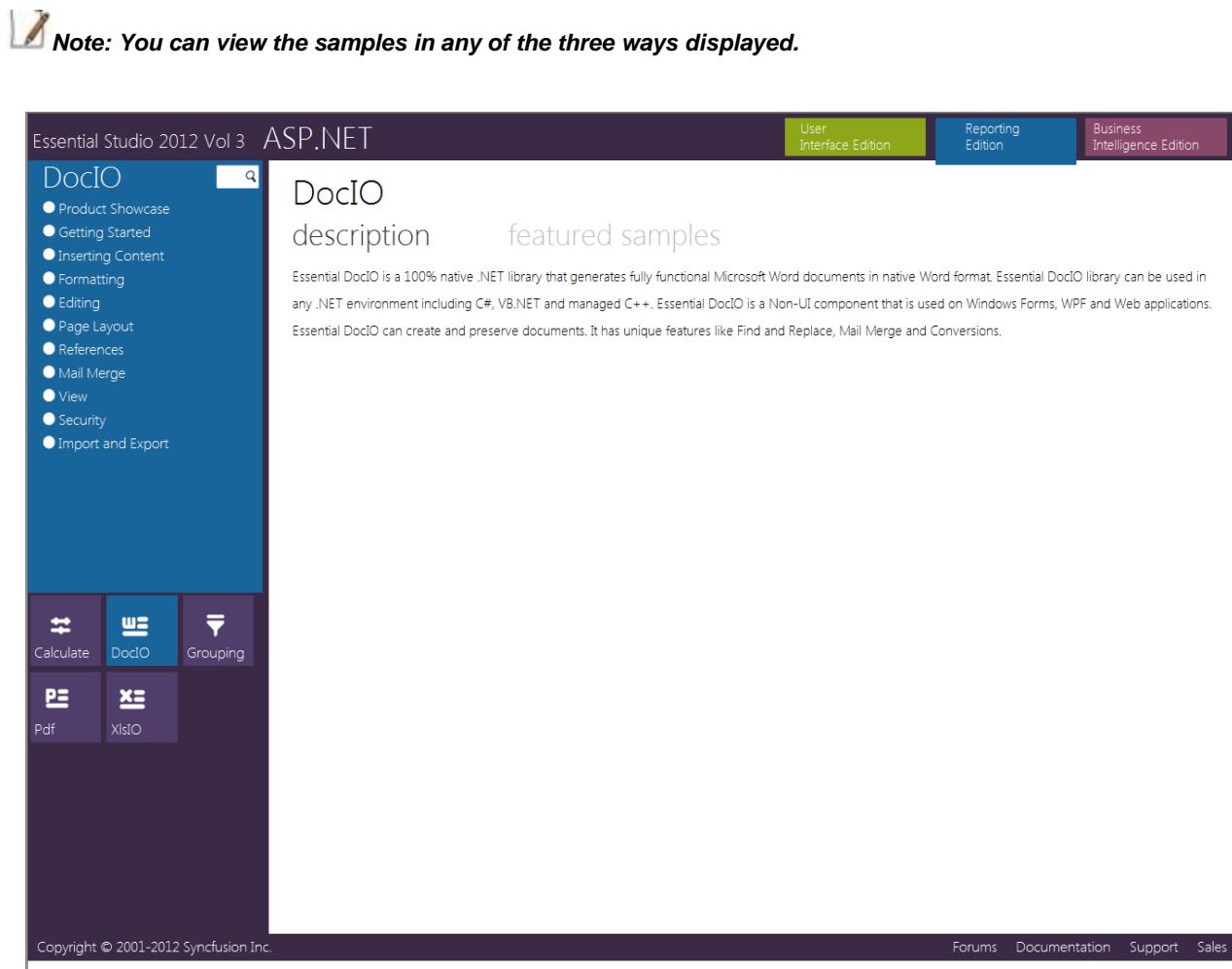


Figure 5: ASP.NET Sample Browser

2. Click **XlsIO** from the bottom-left pane. The XlsIO samples are displayed.

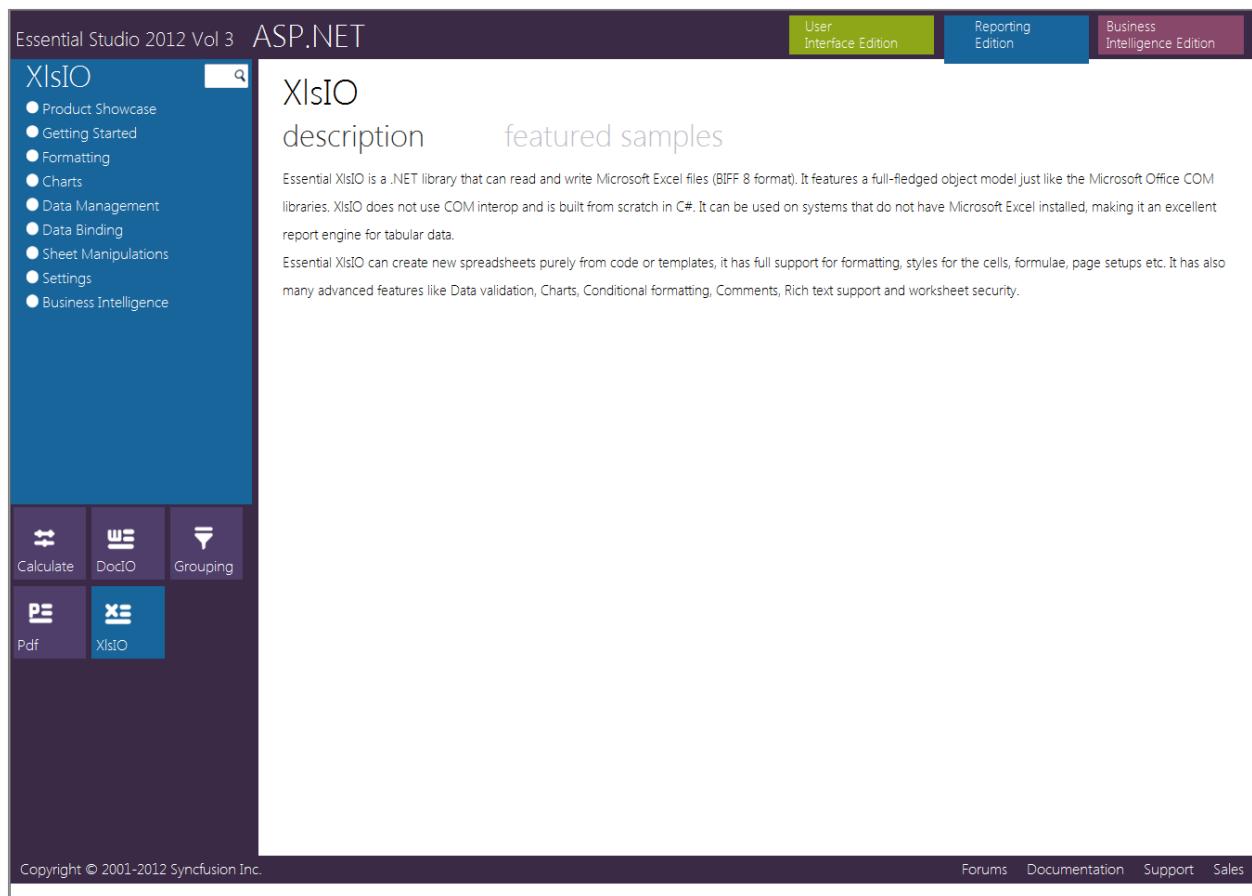


Figure 6: XlsIO Samples Displayed in the ASP.NET Sample Browser

3. Select any sample and browse through the features.

WPF

1. In the dashboard window, click **Run Samples** for **WPF** under **Reporting** Edition panel. The **WPF** Sample Browser window is displayed.



Note: You can view the samples in any of the three ways displayed.

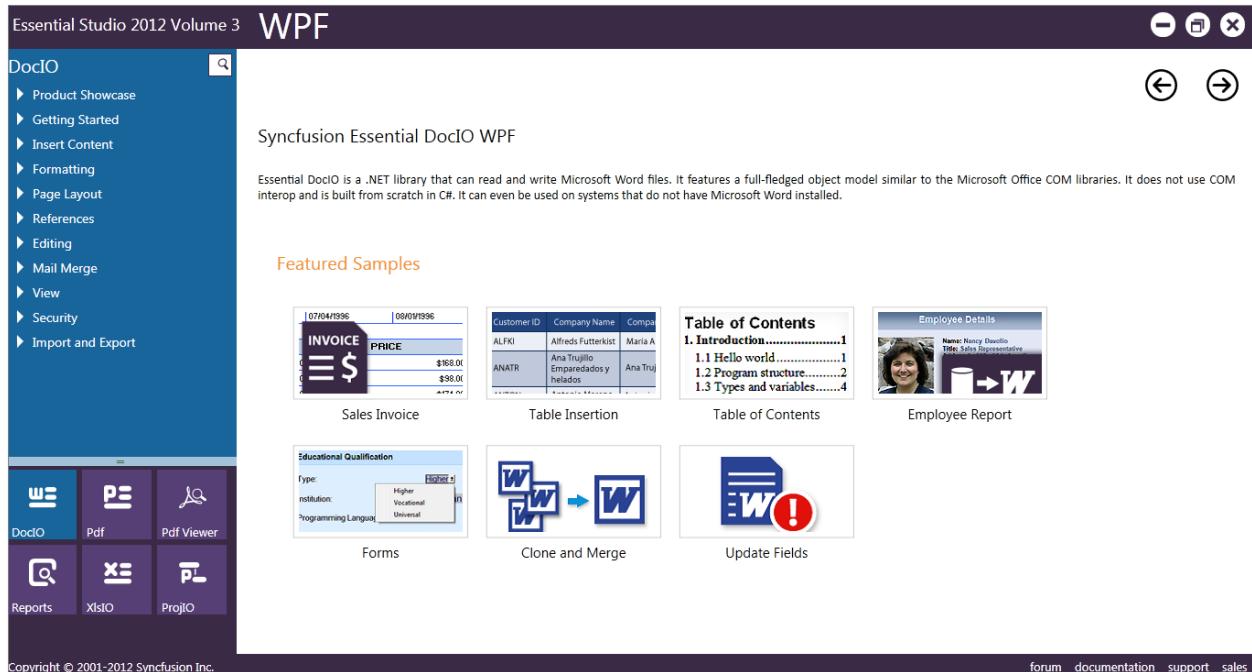


Figure 7: WPF Sample Browser

- Click **XlsIO** from the bottom-left pane. The XlsIO samples are displayed.

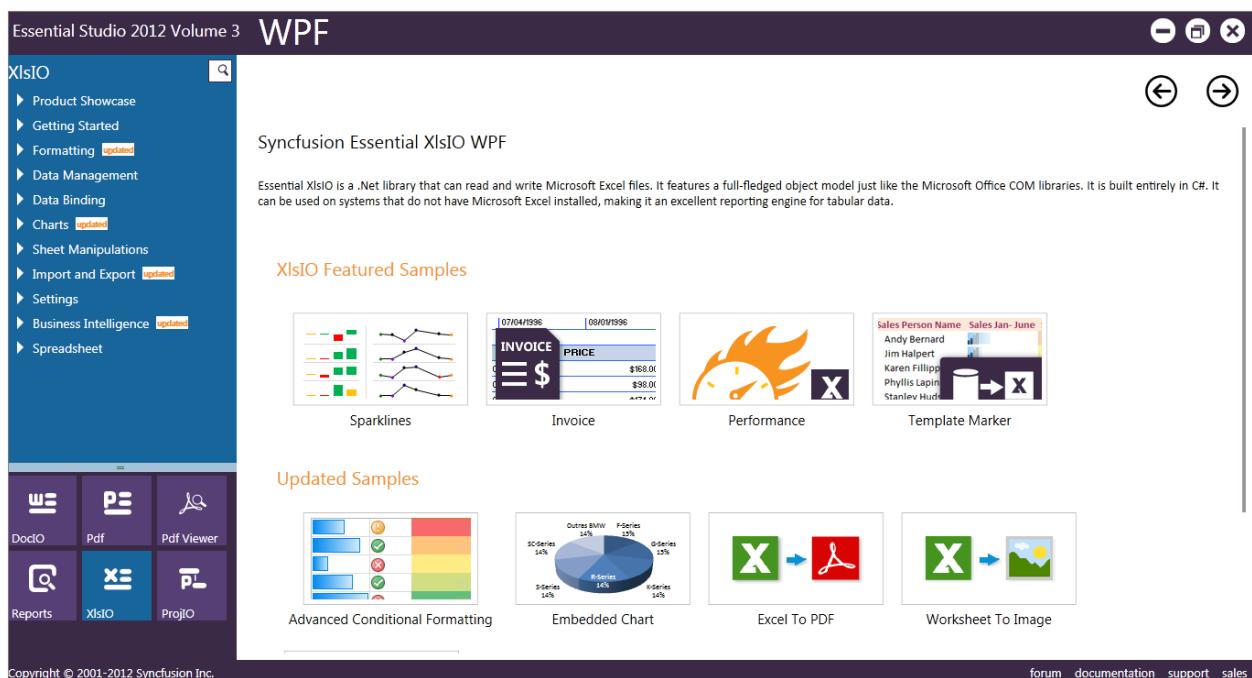


Figure 8: XlsIO Samples Displayed in the WPF Sample Browser

- Select any sample and browse through the features.

Silverlight

1. In the dashboard window, click **Run Samples** for Silverlight under **Reporting Edition** panel. The **Silverlight** Sample Browser window is displayed.

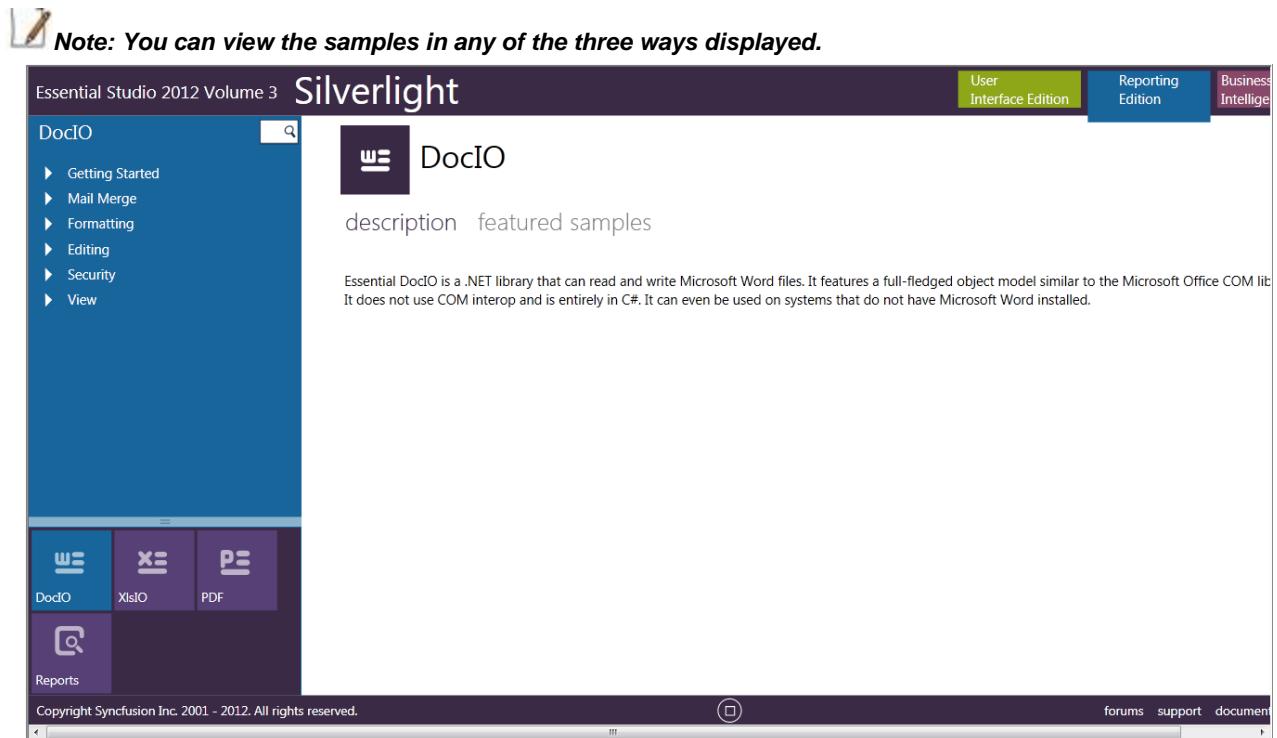


Figure 9: Silverlight Sample Browser

2. Click the **XlsIO** drop-down on the right pane. The XlsIO samples are displayed.



Figure 10: XlsIO Samples Displayed in the Silverlight Sample Browser

3. Select any sample and browse through the features.

ASP.NET MVC

1. In the dashboard window, click **Run Samples** for ASP.NET MVC under **Reporting** Edition panel. The **ASP.NET MVC** Sample Browser window is displayed.



Note: You can view the samples in any of the three ways displayed.

The screenshot shows the ASP.NET MVC sample browser interface. At the top, there's a navigation bar with links for 'Essential Studio 2012 Volume 3' and 'Download Trial'. Below the navigation bar, the main content area has a header 'ASP.NET MVC'.

The left sidebar is titled 'DocIO' and contains a navigation menu with the following items:

- ProductShowCase
- Getting Started
- Editing
- Formatting
- Insert Content
- Mail Merge
- Page Layout
- View
- Security
- References
- Import and Export

Below the menu is a search bar with a magnifying glass icon.

The main content area features several sections:

- Project Status Report**: A screenshot of Microsoft Word showing a document with a header 'NORTHWIND TRADERS' and a table of contents.
- Employee Details**: A screenshot of Microsoft Word showing a document with a photo of a woman and some text.
- Featured Samples** section:
 - Employee Report**: Shows a screenshot of a 'Employee Details' page with a photo of Nancy Davolio and the text 'Name: Nancy Davolio Title: Sales Representative'.
 - Table of Contents**: Shows a screenshot of a 'Table of Contents' page with the following table of contents:

1. Introduction.....	1
1.1 Hello world.....	1
1.2 Program structure.....	2
1.3 Types and variables.....	4
1.4 Expressions.....	6
 - Clone and Merge**: Shows a screenshot of Microsoft Word with three 'W' icons and an arrow pointing from one to another.
 - Forms**: Shows a screenshot of a 'Educational Qualification' form with fields for 'Type' (Higher), 'Institution' (McDonald University), 'Programming Language' (C#), 'Or' (Vocational), and 'Perfect' (Universal).
 - Update Fields**: Shows a screenshot of Microsoft Word with a large 'W!' icon.

Figure 11: ASP.NET MVC Sample Browser

- Click **XlsIO** from the bottom-left pane. The XlsIO samples are displayed.

Essential XlsIO

High performance server side .NET library that can read and write Microsoft Excel files. Full-fledged object model just like the Microsoft Office COM libraries. Built entirely in C# and can be used on systems that do not have Microsoft Excel installed, making it an excellent reporting engine for tabular data.

Featured Samples

- Template Marker
- Sparklines
- Invoice
- Pivot Table

Figure 12: XlsIO Samples Displayed in the ASP.NET MVC Sample Browser

3. Select any sample and browse through the features.

Source Code Location

The default locations of the source code for Essential XlsIO corresponding to each platform are given below:

- **Windows**-The source code for Windows platform is located at the following location.

[System Drive]:\Program Files\Syncfusion\Essential Studio\[Version Number]\Base\XlsIO.Windows\Src

- **WPF**-The source code for WPF platform is located at the following location.

[System Drive]:\Program Files\Syncfusion\Essential Studio\[Version Number]\Base\XlsIO.WPF\Src

- **MVC**-The source code for MVC platform is located at the following location.

[System Drive]:\Program Files\Syncfusion\Essential Studio\[Version Number]\MVC\XlsIO.MVC\Src

- **Web**-The source code for ASP.NET platform is located at the following location.

[System Drive]:\Program Files\Syncfusion\Essential Studio\[Version Number]\Web\XlsIO.Web\Src

2.3 Deployment Requirements

While deploying an application that references Syncfusion Essential XlsIO assembly, the following dependencies must be included in the distribution.

Full Trust Mode

In full trust mode, add references to the following assemblies corresponding to the platform:

XlsIO – Windows Forms, ASP.NET, WPF, ASP.NET MVC

- Syncfusion.Core.dll
- Syncfusion.Compression.Base.dll
- Syncfusion.XlsIO.Base.dll

XlsIO (Full-Trust) – Silverlight

- Syncfusion.Compression.Silverlight.dll
- Syncfusion.XlsIO.Silverlight.dll

Medium/Partial Trust Mode

In medium/partial trust mode, add references to the following assemblies corresponding to the platform:

XlsIO (Partial-Trust) - ASP.NET

- Syncfusion.Core.dll
- Syncfusion.Compression.Base.dll
- Syncfusion.XlsIO.Web.dll

XlsIO (Partial-Trust) - ASP.NET MVC

- Syncfusion.Core.dll
- Syncfusion.Compression.Base.dll
- Syncfusion.XlsIO.MVC.dll

XlsIO (Client Profile) – Windows Forms and WPF

- Syncfusion.Core.dll
- Syncfusion.Compression.Base.dll
- Syncfusion.XlsIO.ClientProfile.dll

3 Getting Started

This section covers the following topics that give information about creating and using Essential XlsIO in an application:

- [Creating a Platform Application](#)
 - [Deploying Essential XlsIO](#)
 - [Web Application Deployment](#)
 - [Feature Summary](#)

In addition, this section renders information about [Spreadsheet](#) parts such as, [Excel Engine](#), [Workbook](#) and [Worksheet](#). It also describes how to [save a workbook](#) and contains the following additional topics:

- [Class Diagram](#)
 - [Using Templates](#)
 - [Improving Performance](#)
 - [Supported Elements](#)

3.1 Class Diagram

The following screen shot illustrates the Class Diagram for Essential XlsIO.

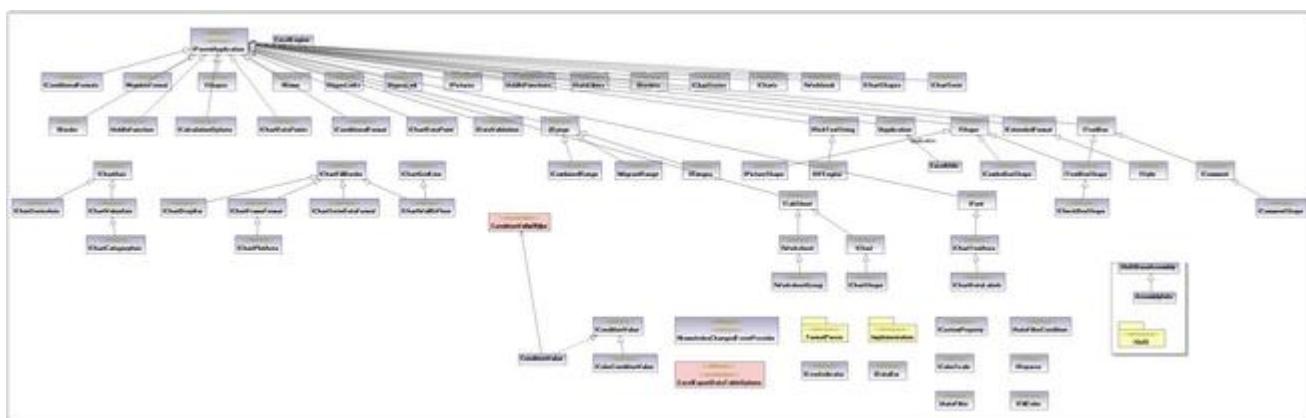


Figure 13: Class Diagram - XlsIO

3.2 Creating a Platform Application

This section illustrates the step-by-step procedure to create the following platform applications.

- Windows
- Web
- WPF
- Silverlight
- ASP.NET MVC

Windows Application

1. Open Microsoft Visual Studio. Go to **File** menu and click **New Project**. In the **New Project** dialog box, select **Windows Forms Application** template, name the project and click **OK**.

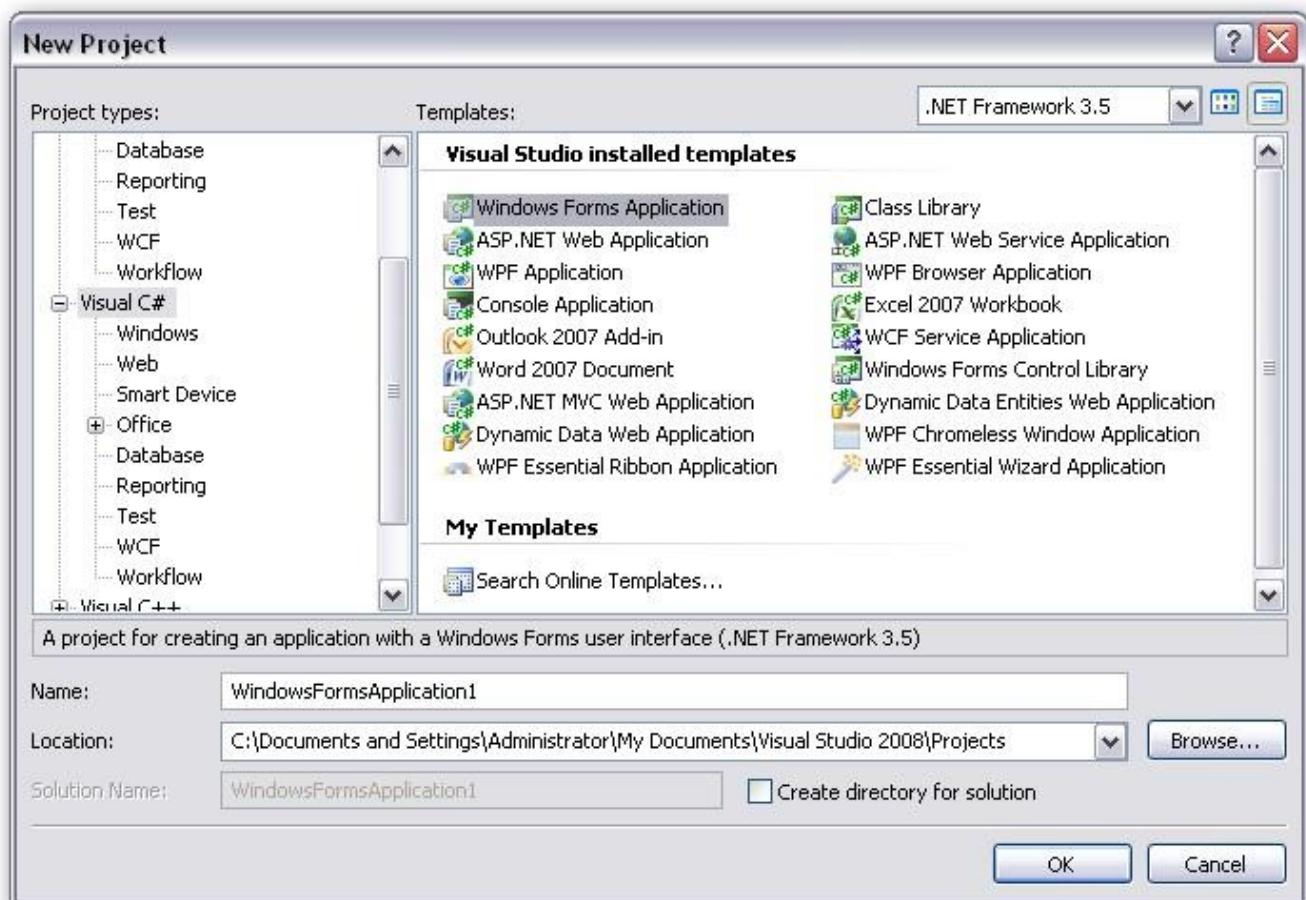


Figure 14: New Project dialog box - Windows Forms Application

A windows application is created.

1. Open the main form of the application in the designer.
2. Add the Syncfusion controls to your VS.NET toolbox if you haven't done so already [This is done automatically when you install Essential Studio].
3. Refer [Deploying Essential XlsIO](#) topic to know how to add XlsIO to the created application.

Web Application

1. Open Microsoft Visual Studio. Go to **File** menu and click **New Project**. In the **New Project** dialog box, select **ASP.NET Web Application** template, name the web application and click **OK**.



Figure 15: New Project dialog box - ASP.NET Application

A Web application is created.

- Now you need to deploy Essential XlsIO into this ASP.NET application. Refer [Deploying Essential XlsIO](#) topic for detailed info.

WPF Application

- Open Microsoft Visual Studio. Go to **File** menu and click **New Project**. In the **New Project** dialog box, select **WPF Application** template, name the project and click **OK**.



Figure 16: New Project dialog box-WPF Application

A new WPF application is created.

- Open the main form of the application in the designer.

3. Add the Syncfusion controls to your VS.NET toolbox if you haven't done so already [This is done automatically when you install Essential Studio].
4. Refer [Deploying Essential XlsIO](#) topic to know how to add XlsIO to the created application.

Silverlight Application

1. Open Microsoft Visual Studio. Go to **File** menu and click **New Project**. In the **New Project** dialog box, select **Silverlight Application** template, name the project and click **OK**.

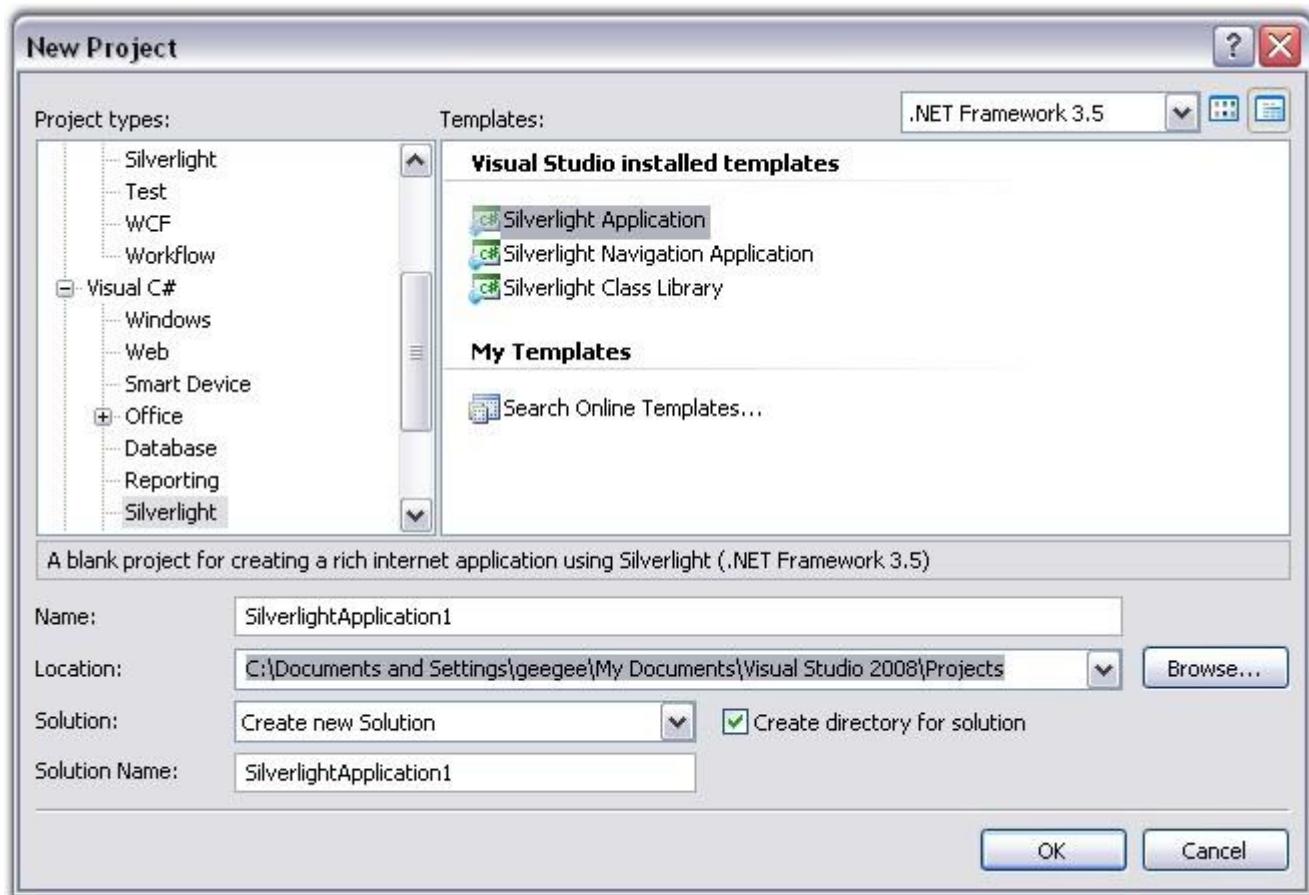


Figure 17: New Project dialog box-Silverlight Application

A **New Silverlight Application** dialog box appears as follows.

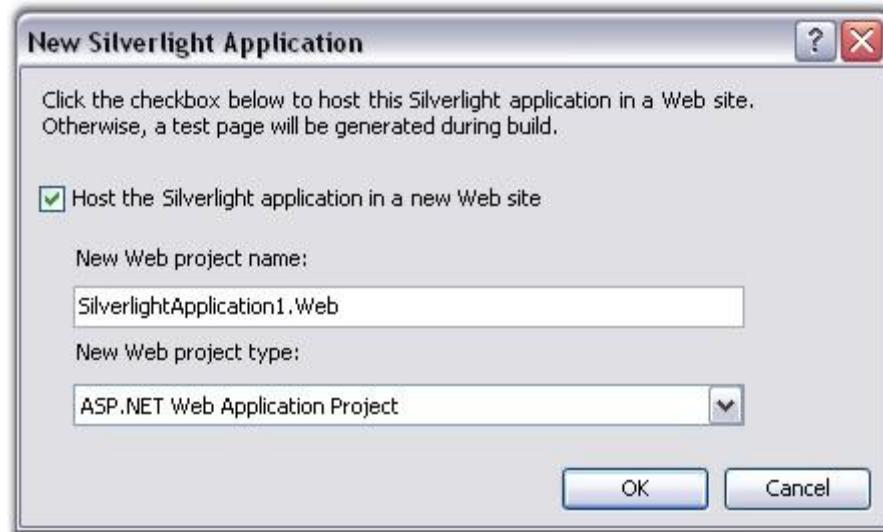


Figure 18: New Silverlight Application dialog box

2. Click OK to host the Silverlight application in a new website.

A new Silverlight application is created.

3. Open the main form of the application in the designer.
4. Add reference to the following assemblies:
 - Syncfusion.XlsIO.Silverlight.dll
 - Syncfusion.Compression.Silverlight.dll

Refer [Deploying Essential XlsIO](#) topic to know how to add XlsIO to the created application.

ASP.NET MVC Application

Refer to **Grid MVC -> Getting Started -> Creating an MVC application** topic to know how to create an MVC application.

To know how to add XlsIO to this application, refer Adding Essential XlsIO to an MVC Application topic under Getting Started section.

3.3 Deploying Essential XlsIO

We have now created a platform application in the previous topic (Creating Platform Application). This section will guide you to deploy Essential XlsIO in those applications under the following topics:

- Windows-This topic illustrates how to deploy XlsIO in a Windows application.
- ASP.NET-This topic illustrates how to deploy XlsIO in a Web application.
- WPF-This topic illustrates how to deploy XlsIO in a WPF application.
- Silverlight-This topic illustrates how to deploy XlsIO in a Silverlight application.
- ASP.NET MVC-This topic illustrates how to deploy XlsIO in an ASP.NET MVC application.

3.3.1 Windows

Now, you have created a Windows application (refer [Creating a Platform Application](#)). This section covers the following:

- Deploying Essential XlsIO in a Windows Application
- Creating and adding an Excel document (with worksheets) to the Application

Deploying Essential XlsIO in a Windows Application

The following steps will guide you to deploy Essential XlsIO:

1. Go to **Solution Explorer** of the application you have created. Right-click the **Reference** folder and then click **Add References**.
2. Add the following assemblies as references in the application.
 - Syncfusion.Core.dll
 - Syncfusion.Compression.dll
 - Syncfusion.XlsIO.base.dll

 **Note:** For detailed documentation on Windows Application deployment, see:
http://www.syncfusion.com/support/user/uploads/DeployingWindowsApplication_bdaf76f7.pdf

Essential XlsIO is deployed in the Windows application.

Creating and Adding an Excel Document (With Worksheets) to the Application

The following steps will guide you to create and add XlsIO document to this application:

1. Add the following C# code to import the Syncfusion.XlsIO namespace.

[C#]

```
using Syncfusion.XlsIO;
```

The Syncfusion.XlsIO namespace is imported.

2. Create an instance of XlsIO by using the following code.

[C#]

```
// New instance of XlsIO is created.[Equivalent to launching MS Excel with no workbooks open].
// Instantiate the spreadsheet creation engine.
ExcelEngine excelEngine = new ExcelEngine();
```

[VB.NET]

```
' New instance of XlsIO is created.[Equivalent to launching MS Excel with no workbooks open].
' Instantiate the spreadsheet creation engine.
Dim excelEngine As ExcelEngine = New ExcelEngine()
```

An instance of XlsIO is created.

 See [Excel Engine](#), for more details.

3. Create an instance of the Excel application through the **IApplication** interface.

[C#]

```
// Instantiate the Excel application object.
IApplication application = excelEngine.Excel;
```

[VB.NET]

```
' Instantiate the Excel application object.
```

```
Dim application As IApplication = excelEngine.Excel
```

An Excel document is created.

4. Create a workbook. A newly created workbook has three worksheets by default. You can change the number of worksheets, using the **Create** method of **IWorkBook** as shown in the following code.

[C#]

```
// A new workbook is created.[Equivalent to creating a new workbook in MS  
Excel).  
// The new workbook will have 5 worksheets.  
IWorkbook workbook = application.Workbooks.Create(5);
```

[VB.NET]

```
' A new workbook is created.[Equivalent to creating a new workbook in MS  
Excel].  
' The new workbook will have 5 worksheets.  
Dim workbook As IWorkbook = application.Workbooks.Create(5)
```

A workbook with the mentioned number of worksheets is created in the Excel document.

 See [Workbook](#) and [Worksheet](#), for more details.

5. Access the worksheet in the workbook and set the data for the given range, say "A1".

[C#]

```
// The first worksheet object in the worksheets collection is accessed.  
IWorksheet sheet = workbook.Worksheets[0];  
  
// Inserting sample text into the first cell of the first worksheet.  
sheet.Range["A1"].Text = "Hello World";
```

[VB.NET]

```
' The first worksheet object in the worksheets collection is accessed.  
Dim sheet As IWorksheet = workbook.Worksheets(0)
```

```
' Inserting sample text into the first cell of the first worksheet.  
sheet.Range("A1").Text = "Hello World"
```

The string "Hello World" is written to the cell **A1** of the document.

6. Save and close the workbook.

[C#]

```
// Saving the workbook to disk.  
workbook.SaveAs("Sample.xls");  
  
// Closing the workbook.  
workbook.Close();
```

[VB.NET]

```
' Saving the workbook to disk.  
workbook.SaveAs("Sample.xls")  
  
' Closing the workbook.  
workbook.Close()
```

The Workbook is saved and closed.

 To know more about saving the workbook, see [Save](#).

7. Dispose the Excel engine. Note that the engine should be disposed after completing workbook operations.

[C#]

```
// Dispose the Excel engine.  
excelEngine.Dispose();
```

[VB.NET]

```
' Dispose the Excel engine.  
excelEngine.Dispose()
```

The Excel engine is disposed.

This completes the creation of an Excel document in the Windows Application, using Essential XlsIO.

The following screen shot shows the Excel document generated by the above given procedure.

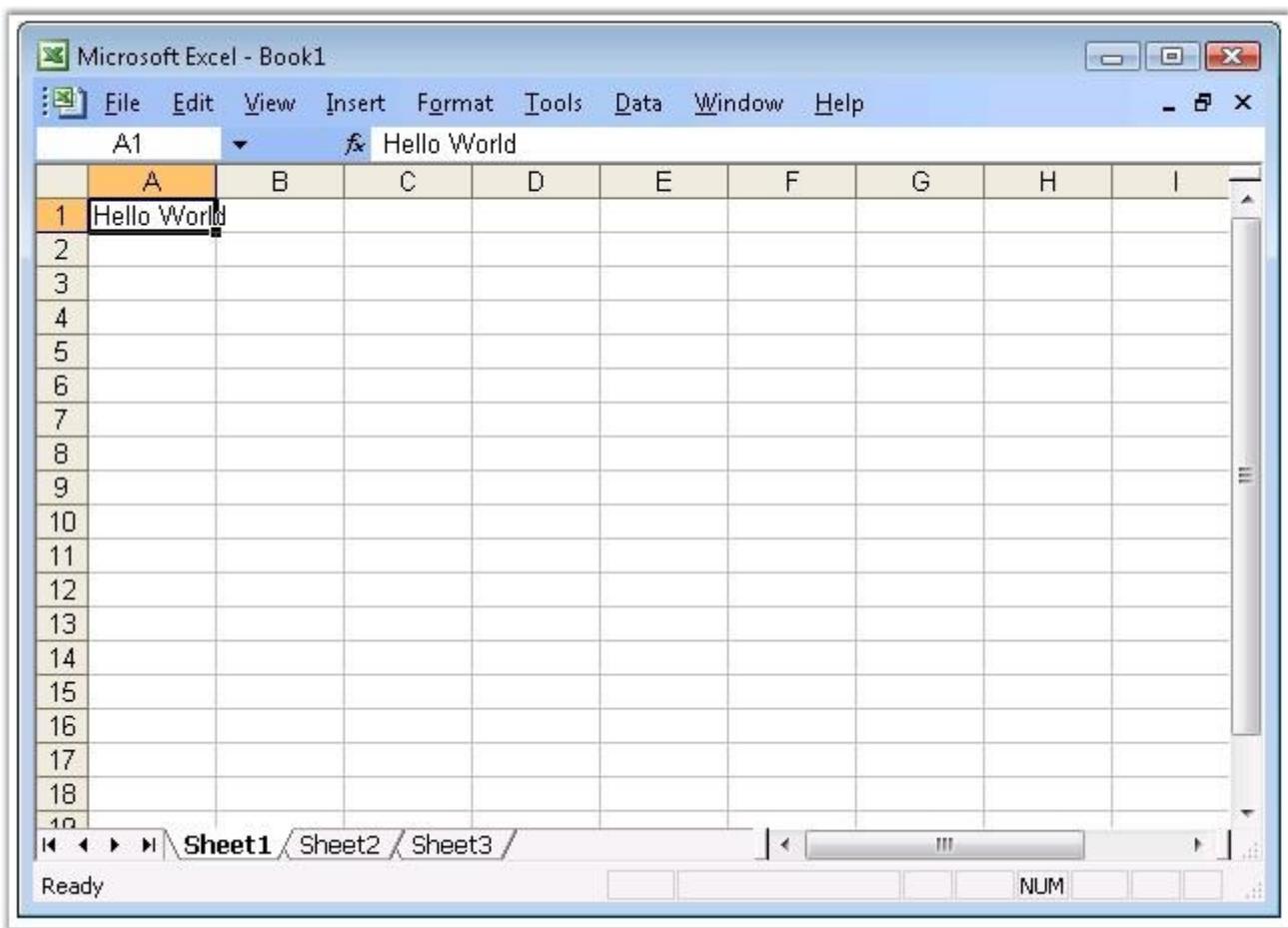


Figure 19: Spreadsheet created in the Windows Application

It is possible to serialize the XlsIO object model to any of the following file formats:

- BIFF8
- BIFF12
- SpreadsheetML
- CSV

3.3.2 ASP.NET

Now, you have created a ASP.NET application (refer to [Creating a Platform Application](#)). This section covers the following:

- Deploying Essential XlsIO in an ASP.NET Application
- Creating and adding an Excel document (with worksheets) to the Application

Deploying Essential XlsIO in an ASP.NET Application

This section provides information and instructions for deploying ASP.NET applications that use Essential XlsIO for ASP.NET. This is in addition to the section on [Deploying Essential Studio for ASP.NET \(Common-->Deploying Essential Studio for ASP.NET\)](#) in the Getting Started guide. Essential XlsIO ships with .NET Framework 2.0 (Visual Studio 2005) version of the C# and VB.NET samples which are installed beneath 2.0 directories. During installation, application directories are created in IIS for each of the C# and VB.NET samples.

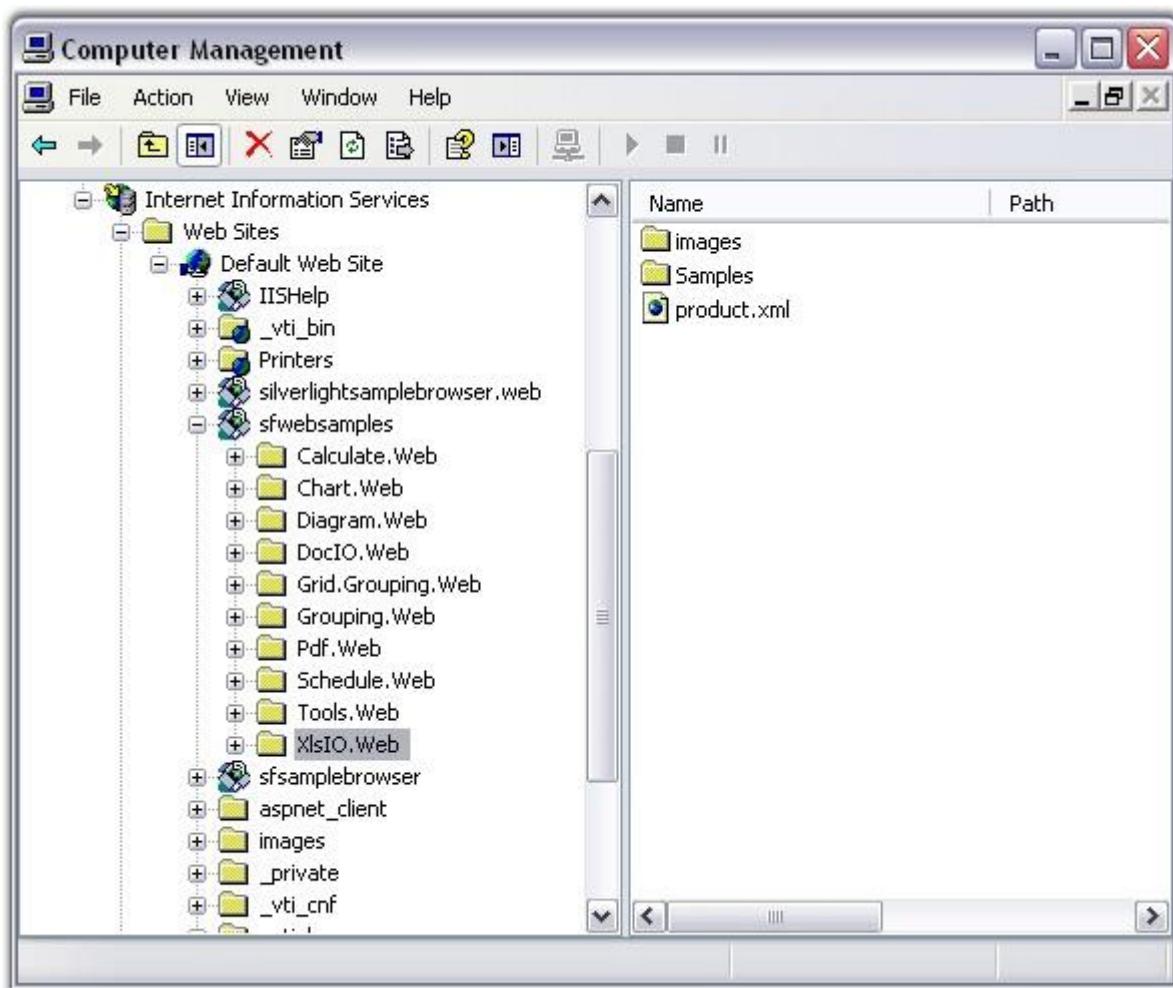


Figure 20: Sample Application Directories in IIS

The following steps will guide you to deploy Essential XlsIO in an ASP.NET application:

1. Marking the Application Directory

The appropriate directory, usually where the aspx files are stored, must be marked as **Application in IIS**.

2. Syncfusion Assemblies

The following Syncfusion assemblies need to be in the **bin** folder that is beside the aspx files:

- Syncfusion.Core.dll
- Syncfusion.Compression.Base.dll
- Syncfusion.XlsIO.Base.dll

They can also be in the GAC, in which case, they should be referenced in the Web.config file, using the code as follows:

[ASPX]

```
<configuration>
  <system.web>
    <compilation>
      <assemblies>
        <add assembly="Syncfusion.XlsIO.Base, Version=x.x.x.x, Culture=neutral,
          PublicKeyToken=3D67ED1F87D44C89"/>
      </assemblies>
    </compilation>
    ...
  </system.web>
</configuration>
```



Note: X.X.X.X in the above code corresponds to the correct version number of the Essential Studio version that you are currently using.

Please refer to the document in the following location, for step-by-step process of Syncfusion assemblies deployment in ASP.NET:

http://www.syncfusion.com/support/user/uploads/webdeployment_c883f681.pdf

 **Note:** 1) Application with Essential XlsIO needs the following dependent assemblies for deployment:

- Syncfusion.Core.dll
- Syncfusion.Compression.Base.dll
- Syncfusion.XlsIO.Base.dll

- 2) All the assemblies should be built from the same version.
- 3) Missing out Compression.Base will raise **Application Exception - TypeInitialization**.

3. Data Files

If you have an xml, mdb, or other data files, ensure that they have sufficient security permission. Authenticated users should have full control over the files and directories, in order to give ASP.NET code, enough permission to open files at run time.

The following JavaScript files should be added in the ASPX page:

- jquery-1.3.2.min.js
- MicrosoftAjax.js

The above-mentioned script files are added by using the following code:

[ASPx]

```
<script src="<% Url.Content("~/Scripts/jquery-1.3.2.min.js") %>"  
type="text/javascript"></script>  
<script src="<% Url.Content("~/Scripts/MicrosoftAjax.js") %>"  
type="text/javascript"></script>
```

With this step, the process of deploying XlsIO in the ASP.NET application gets completed.

Creating and Adding an Excel Document (With Worksheets) to the Application

The following code sample will guide you to create and add an Excel document to this application.

[C#]

```
// New instance of XlsIO is created.[Equivalent to launching MS Excel with  
no workbooks open].
```

```
// The instantiation process consists of two steps.

// Step 1: Instantiate the spreadsheet creation engine.
ExcelEngine excelEngine = new ExcelEngine();

// Step 2: Instantiate the excel application object.
IApplication application = excelEngine.Excel;

// A new workbook is created.[Equivalent to creating a new workbook in MS
// Excel].
// The new workbook will have 3 worksheets.
IWorkbook workbook = application.Workbooks.Create(3);

// The first worksheet object in the worksheets collection is accessed.
IWorksheet sheet = workbook.Worksheets[0];

// Inserting sample text into the first cell of the first worksheet.
sheet.Range["A1"].Text = "Hello World";

// Saving the workbook to disk.
workbook.SaveAs("Sample.xls", ExcelSaveType.SaveAsXLS, Response,
ExcelDownloadType.Open);

// Closing the workbook.
workbook.Close();

// Dispose the Excel engine
excelEngine.Dispose();
```

The sample Excel document created through the above procedure is shown below.

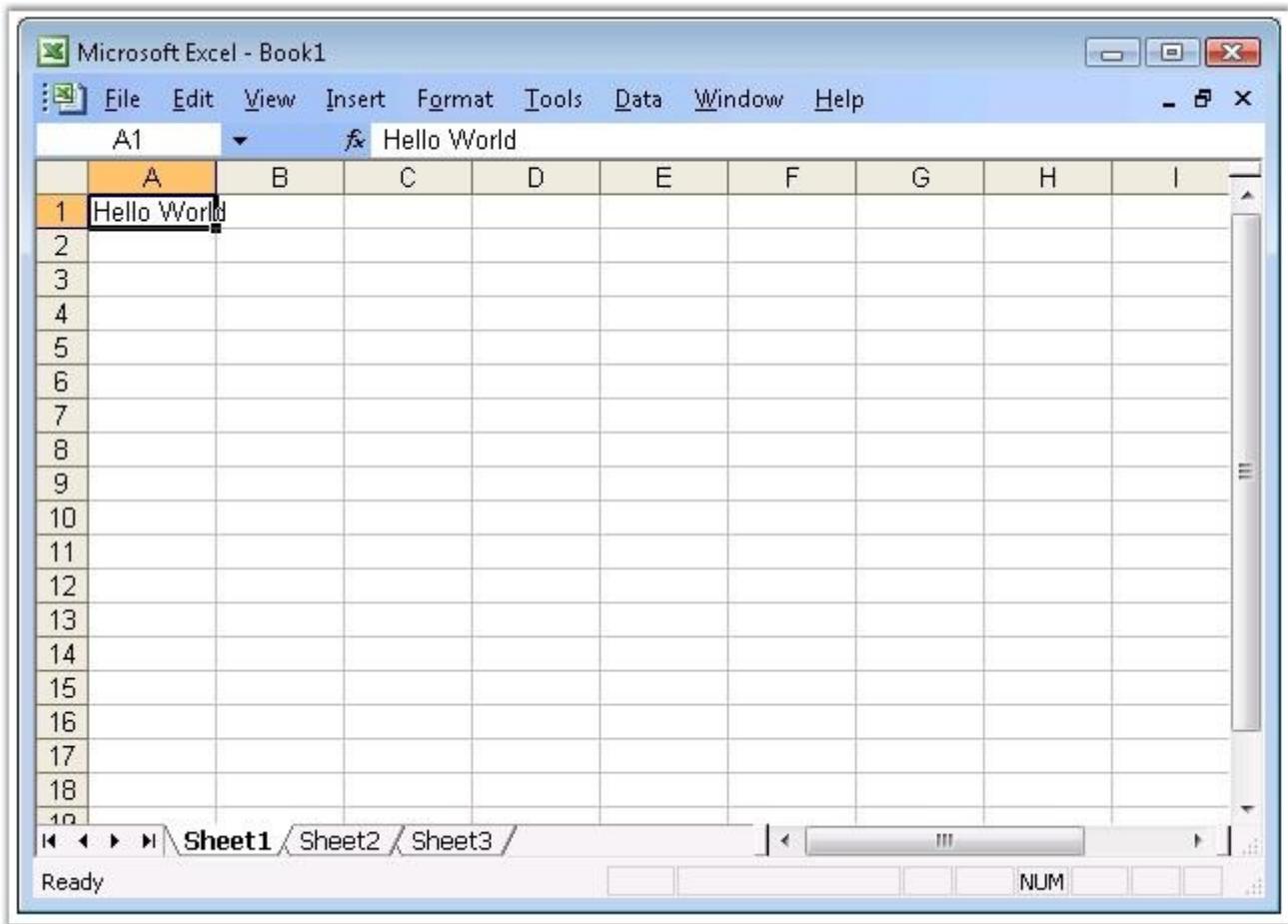


Figure 21: Spreadsheet created in the ASP.NET Application

3.3.3 WPF

Now, you have created a WPF application (refer [Creating a Platform Application](#)). This section covers the following:

- Deploying Essential XlsIO in a WPF Application
- Creating and adding an Excel document (with worksheets) to the application

Deploying Essential XlsIO in a WPF Application

The following steps will guide you to deploy Essential XlsIO:

1. Go to Solution Explorer of the application you have created. Right-click Reference folder and then click Add References.

2. Add the below mentioned assemblies as references in the application.

- Syncfusion.core.dll
- Syncfusion.compression.dll
- Syncfusion.XlsIO.base.dll



Note: There is no toolbox support for XlsIO in WPF application.

Essential XlsIO is now deployed in the WPF application.

Creating and Adding an Excel Document (With Worksheets) to the Application

Following steps will guide you to create and add XlsIO document to this application:

1. Add the following C# code to import the Syncfusion.XlsIO namespace:

[C#]

```
using Syncfusion.XlsIO;
```

The Syncfusion.XlsIO namespace is imported.

2. Create an instance of XlsIO using the following code:

[C#]

```
// New instance of XlsIO is created.[Equivalent to launching MS Excel with  
no workbooks open].  
// The instantiation process consists of two steps.  
  
// Step 1: Instantiate the spreadsheet creation engine.  
ExcelEngine excelEngine = new ExcelEngine();
```

[VB.NET]

```
' New instance of XlsIO is created.[Equivalent to launching MS Excel with no  
workbooks open].  
' The instantiation process consists of two steps.  
  
' Step 1: Instantiate the spreadsheet creation engine.  
Dim excelEngine As ExcelEngine = New ExcelEngine()
```

An instance of XlsIO is created.

3. Create an instance of the Excel application through the IApplication interface.

[C#]

```
// Step 2: Instantiate the excel application object.  
IApplication application = excelEngine.Excel;
```

[VB.NET]

```
' Step 2: Instantiate the excel application object.  
Dim application As IApplication = excelEngine.Excel
```

An Excel document is created.

4. Create a workbook. A newly created workbook has three worksheets by default. You can change the number of worksheets, using the Create method of IWorkBook as shown in the following code.

[C#]

```
// A new workbook is created.[Equivalent to creating a new workbook in MS  
Excel).  
// The new workbook will have 5 worksheets.  
IWorkbook workbook = application.Workbooks.Create(5);
```

[VB.NET]

```
' A new workbook is created.[Equivalent to creating a new workbook in MS  
Excel].  
' The new workbook will have 5 worksheets.  
Dim workbook As IWorkbook = application.Workbooks.Create(5)
```

A workbook with the mentioned number of worksheets is created in the Excel document.

5. Access the worksheet in the workbook and set the data for the given range, say "A1".

[C#]

```
// The first worksheet object in the worksheets collection is accessed.  
IWorksheet sheet = workbook.Worksheets[0];
```

```
// Inserting sample text into the first cell of the first worksheet.  
sheet.Range["A1"].Text = "Hello World";
```

[VB.NET]

```
' The first worksheet object in the worksheets collection is accessed.  
Dim sheet As IWorksheet = workbook.Worksheets(0)  
  
' Inserting sample text into the first cell of the first worksheet.  
sheet.Range("A1").Text = "Hello World"
```

The string "Hello World" is written to the cell **A1** of the document.

6. Save and close the workbook.

[C#]

```
// Saving the workbook to disk.  
workbook.SaveAs("Sample.xls");  
  
// Closing the workbook.  
workbook.Close();
```

[VB.NET]

```
' Saving the workbook to disk.  
workbook.SaveAs("Sample.xls")  
  
' Closing the workbook.  
workbook.Close()
```

The Workbook is saved and closed.

7. Dispose the Excel engine. Note that the engine should be disposed after completing workbook operations.

[C#]

```
// Dispose the Excel engine  
excelEngine.Dispose();
```

[VB .NET]

```
' Dispose the Excel engine  
excelEngine.Dispose()
```

The Excel engine is disposed.

This completes the creation of an Excel document in the WPF Application, using Essential XlsIO.

The figure below shows the Excel document generated by the above given procedure:

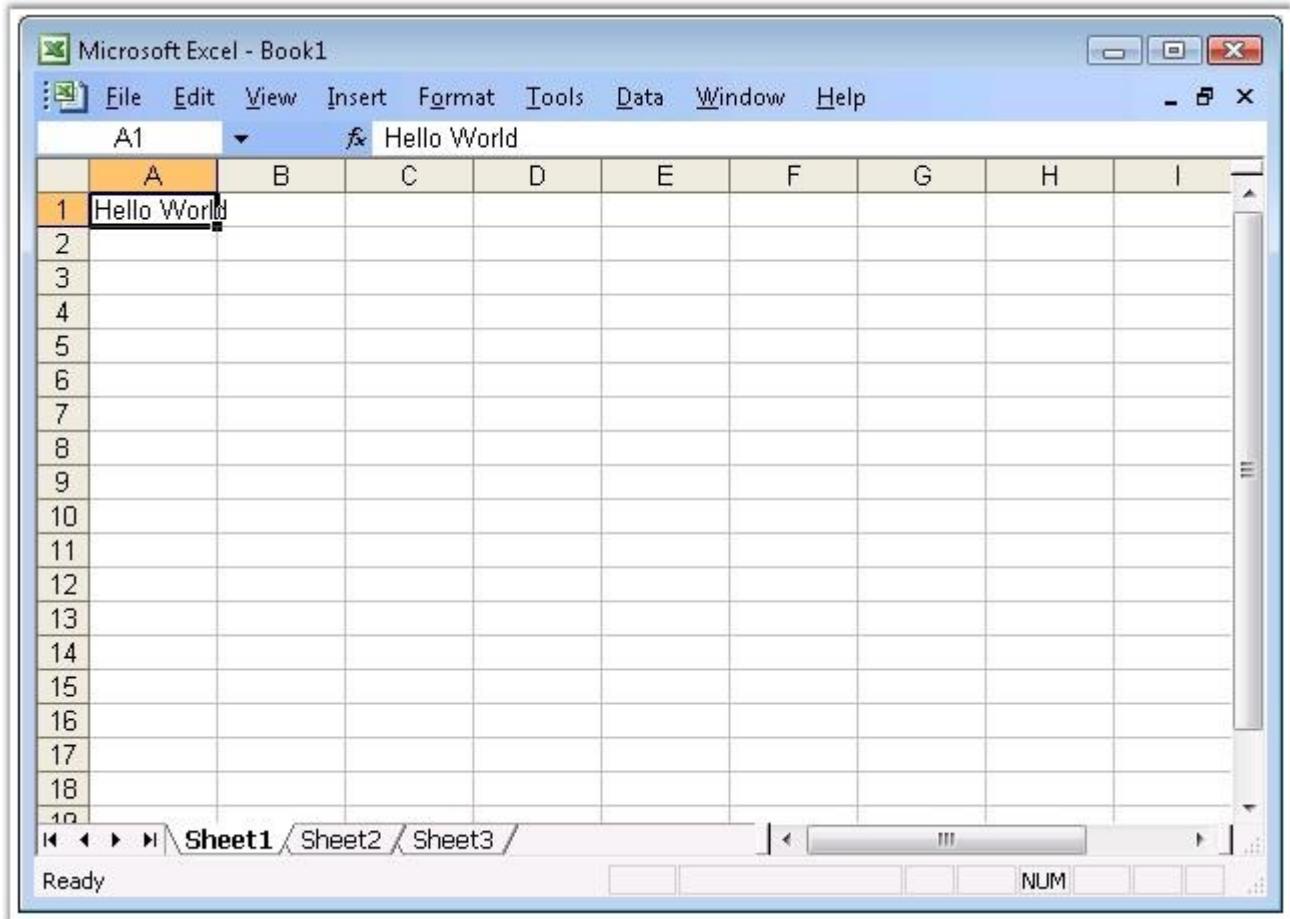


Figure 22: Spreadsheet created in the WPF Application

3.3.4 Silverlight

Now, you have created a Silverlight application (refer [Creating a Platform Application](#)). This section covers the following:

- Deploying Essential XlsIO in a Silverlight Application
- Creating and adding an Excel document (with worksheets) to the application

Deploying Essential XlsIO in a Silverlight Application

The following steps will guide you to deploy Essential XlsIO:

1. Open the **MainPage.xaml** of the application in the designer.
2. Add the **Syncfusion.Compression.Silverlight.dll** and **Syncfusion.XlsIO.Silverlight.dll** assemblies as references to the application.

Essential XlsIO is now deployed in your Silverlight application.

Creating and Adding an Excel Document (With Worksheets) to the Application

Essential XlsIO for Silverlight has support for creation and manipulation of richly formatted Excel [97-2003] format spreadsheets from scratch on the client side. Advanced features like Data Validation, Conditional Formatting and Charts can also be used in this approach.

Following steps will guide you to create a simple spreadsheet:

1. Add references to the following assemblies.
 - Syncfusion.Compression.Silverlight.dll
 - Syncfusion.XlsIO.Silverlight.dll
2. The next step is to add reference to the following namespace.
 - **Syncfusion.XlsIO** (using Syncfusion.XlsIO)

[C#]

```
using Syncfusion.XlsIO;
```

3. Instantiate the Excel Engine.

[C#]

```
// New instance of XlsIO is created.[Equivalent to launching MS Excel with no workbooks open].
// The instantiation process consists of two steps.

// Step 1: Instantiate the spreadsheet creation engine.
ExcelEngine excelEngine = new ExcelEngine();
```

[VB.NET]

```
' New instance of XlsIO is created.[Equivalent to launching MS Excel with no workbooks open].
' The instantiation process consists of two steps.

' Step 1: Instantiate the spreadsheet creation engine.
Dim excelEngine As ExcelEngine = New ExcelEngine()
```

4. Instantiate the Excel application through the **IApplication** interface which represents an Excel application.

[C#]

```
// Step 2: Instantiate the excel application object.
IApplication application = excelEngine.Excel;
```

[VB.NET]

```
' Step 2: Instantiate the excel application object.
Dim application As IApplication = excelEngine.Excel
```

An Excel document is created.

5. Create a workbook. A newly created workbook has three worksheets by default. You can change the count of the worksheets by using the **Create** method of [IWorkbook](#).

[C#]

```
// A new workbook is created.[Equivalent to creating a new workbook in MS Excel].
// The new workbook will have 5 worksheets.
IWorkbook workbook = application.Workbooks.Create(5);
```

[VB.NET]

```
' A new workbook is created.[Equivalent to creating a new workbook in MS
Excel].
' The new workbook will have 5 worksheets.
Dim workbook As IWorkbook = application.Workbooks.Create(5)
```

A workbook with the mentioned number of worksheets is created in the Excel document.

6. Access the [worksheet](#) in the workbook and set the data for the given Range, say "A1".

[C#]

```
// The first worksheet object in the worksheets collection is accessed.
IWorksheet sheet = workbook.Worksheets[0];

// Inserting sample text into the first cell of the first worksheet.
sheet.Range["A1"].Text = "Hello World";
```

[VB .NET]

```
' The first worksheet object in the worksheets collection is accessed.
Dim sheet As IWorksheet = workbook.Worksheets(0)

' Inserting sample text into the first cell of the first worksheet.
sheet.Range("A1").Text = "Hello World"
```

The string "Hello World" is written to the cell A1 of the document.

7. Save to stream and close the workbook. Also, dispose the Excel engine.



Note: *The engine should be disposed after completing workbook operations.*

[C#]

```
// Save the file on to disk.
SaveFileDialog sfd = new SaveFileDialog();
sfd.DefaultExt = ".xls";
sfd.Filter = "Files (*.xls)|*.xls";
if (sfd.ShowDialog() == true)
{
    using (Stream stream = sfd.OpenFile())
    {
        workbook.SaveAs(stream);
```

```

    // Closing the workbook.
    workbook.Close();

    // Dispose the Excel Engine.
    excelEngine.Dispose();
}
}

```

[VB .NET]

```

' Save the file on to disk.
Dim sfd As SaveFileDialog = New SaveFileDialog()
sfd.DefaultExt = ".xls"
sfd.Filter = "Files(*.xls) | *.xls"
If sfd.ShowDialog() = True Then
    Using stream As Stream = sfd.OpenFile()
        workbook.SaveAs(stream)

        ' Closing the workbook.
        workbook.Close()

        ' Dispose the Excel Engine.
        excelEngine.Dispose()
    End Using
End If

```

The Workbook is saved and closed and the Excel engine is disposed.



Note: This is a very basic usage scenario where "Hello World" is inserted into the cell "A1" of the spreadsheet. The more advanced usage scenarios of creating complex spreadsheets from scratch are explained in detail in this user guide.

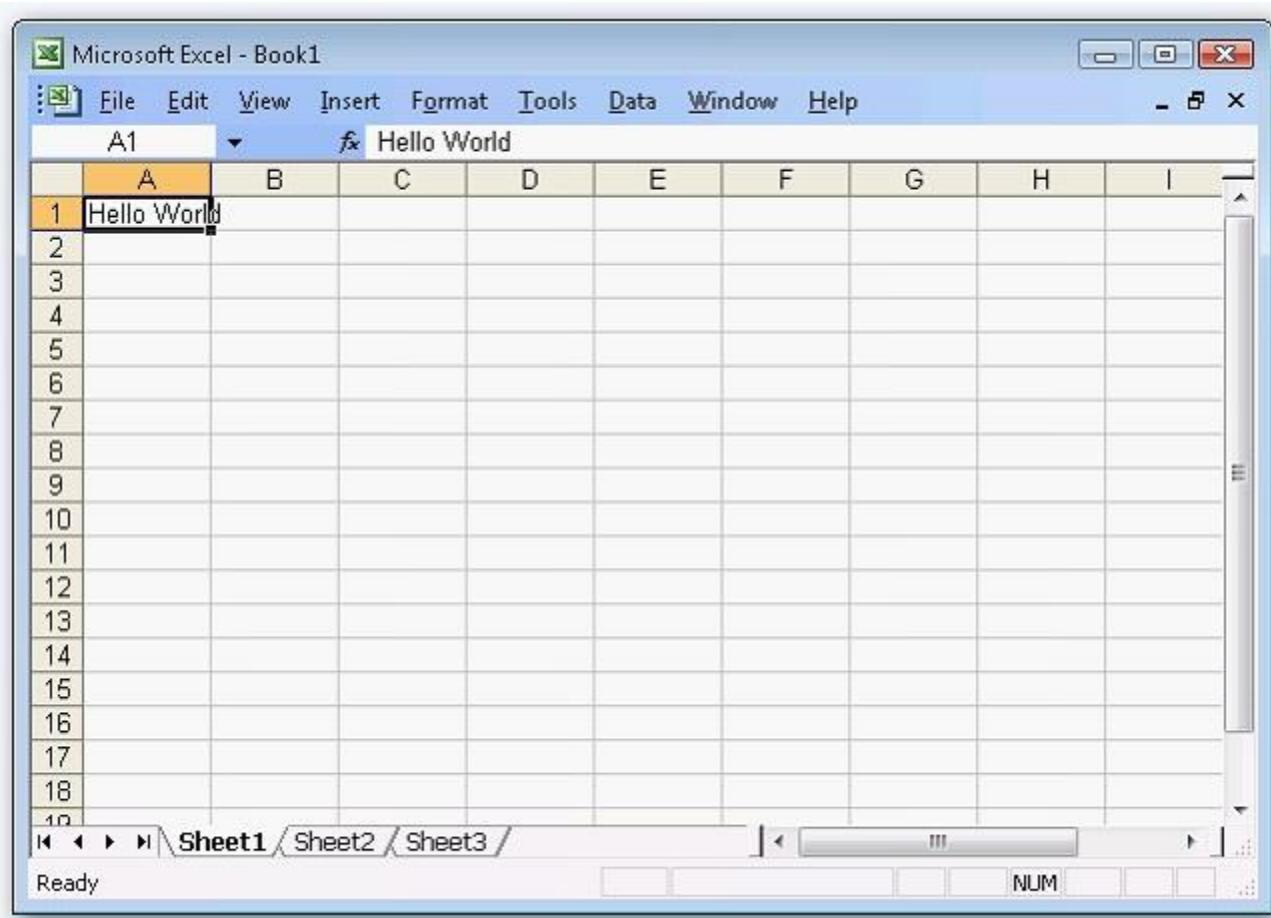


Figure 23: Spreadsheet created in the Silverlight Application

A spreadsheet has been created in the Silverlight application.

Following are the list of supported elements in XlsIO for Silverlight.

Element	xls		xlsx
	Read	Write	
Font settings	Yes	Yes	
Alignments	Yes	Yes	Not Supported
Number formatting	Yes	Yes	

Border settings	Yes	Yes	
Fill settings	Yes	Yes	
Autofit	No	No	
Cell styles	Yes	Yes	
RGB Colors	Yes (Indexed)	Yes (Indexed)	
Conditional formatting	Yes	Yes	
Hide/unhide rows/cols	Yes	Yes	
Hide/Unhide worksheet	Yes	Yes	
Copy/Move worksheet	Yes	Yes	
Sheet protection	Yes	Yes	
Workbook protection	Yes	Yes	
Sheet format[sheet name, tab color]	Yes	Yes	
Bitmap Images	Yes	Yes	
Vector Images	Yes	Yes	
Charts	Yes	Yes	
Hyperlinks	Yes	Yes	
Header/Footer	Yes	Yes	
Pivot tables	No	No	
Auto Shapes	No	No	
Text Box	Yes	Yes	
Check Box	Yes	Yes	
Combo Box	Yes	Yes	

Page setup [Margin,origin,page size]	Yes	Yes	
Page breaks	Yes	Yes	
Background image	Yes	Yes	
Print settings[Print area,Print titles,page order]	Yes	Yes	
Formulas	Yes	Yes	
Calculation options	Yes	Yes	
Names	Yes	Yes	
Formula auditing [Ignore error]	Yes	Yes	
Autofilter	Yes	Yes	
Data validation	Yes	Yes	
Template marker	Yes	Yes	
Outlines[group/ungroup, summary settings]	Yes	Yes	
Comments	Yes	Yes	
Freeze pane, split pane	Yes	Yes	
View[Zoom,show/hide gridline,show/hide headings], horizontal/vertical scroll bars	Yes	Yes	
Macros	No	No	

Encryption	Yes	Yes	
Decryption	No	No	
Ole Objects	No	No	
Track changes	No	No	
Streams	Yes	Yes	
Tables	No	No	

3.3.5 ASP.NET MVC

Now, you have created a ASP.NET MVC application (refer [Creating a Platform Application](#)). This section covers the following:

- Deploying Essential XlsIO in an ASP.NET MVC Application
- Create and add a XlsIO document with pages to the application

Deploying Essential XlsIO in an ASP.NET MVC Application

The following steps will guide you to deploy Essential XlsIO:

1. Add the **Syncfusion.XlsIO.Base** assembly into the Reference folder to deploy Essential XlsIO to the application.
2. Add the following assembly reference under **<compilation>** tag of Web.config file.

[XML]

```

<compilation debug="true">
    <assemblies>
        ...
        <add assembly="Syncfusion.XlsIO.Base, Version=x.x.x.x,
Culture=neutral, PublicKeyToken=3d67ed1f87d44c89"/>
        ...
    </assemblies>

```

```
</compilation>
```



Note: X.X.X.X in the above code corresponds to the correct version number of the Essential Studio version that you are currently using.

3. Add the following under <namespaces> tag of Web.config file.

[XML]

```
<namespaces>
  ...
  <add namespace="Syncfusion.XlsIO"/>
</namespaces>
```

Essential XlsIO is now deployed in your ASP.NET MVC application.

Creating and Adding an Excel Document (With Worksheets) to the Application

The following steps illustrate how to create a worksheet in an MVC application:

Step 1: View

Add a **Button** to the aspx page in the View, as illustrated by the following code:

[ASPX]

```
<%Html.BeginForm("createsheet", "GettingStarted", FormMethod.Post);%>
<input type="submit" value="Create Document" />
<% Html.EndForm();%>
```

Step 2: Create a Custom ActionResult Class

In an MVC application, the controller returns an action result. In particular, it returns an action that derives from the base ActionResult class.

But, if you want to return some other type of content to a browser, such as an image, a XlsIO file, or a Microsoft Excel document, you need to create your own action result. The following code example illustrates how to create a custom action result:

[C#]

```
public class ExcelResult : ActionResult
{
    private IWorkbook m_source;
    private string m_filename;
    private HttpResponse m_response;
    private ExcelDownloadType m_downloadType;
    private ExcelHttpContentType m_contentType;
    private string m_separator;

    public string FileName
    {
        get
        {
            return m_filename;
        }
        set
        {
            m_filename = value;
        }
    }

    public IWorkbook Source
    {
        get
        {
            return m_source as IWorkbook;
        }
    }

    public HttpResponse Response
    {
        get
        {
            return m_response;
        }
    }

    public ExcelDownloadType DownloadType
    {
        set
        {
            m_downloadType = value;
        }
        get
        {
            return m_downloadType;
        }
    }
}
```

```

        }
    }

    public ExcelHttpContentType ContentType
    {
        set
        {
            m_contentType = value;
        }
        get
        {
            return m_contentType;
        }
    }

    public ExcelResult(IWorkbook source, string fileName, HttpResponse
response, ExcelDownloadType downloadType, ExcelHttpContentType
contentType)
    {
        this.FileName = fileName;
        this.m_source = source;
        m_response = response;
        DownloadType = downloadType;
        ContentType = contentType;
    }

    public override void ExecuteResult(ControllerContext context)
    {
        if (context == null)
            throw new ArgumentNullException("Context");
        this.m_source.SaveAs(fileName, Response, DownloadType, ContentType);
    }
}

```

Every action result must inherit the base ActionResult class. The base ActionResult class has one method that you must implement: the **ExecuteResult** method. The ExecuteResult method is called to generate the content, created by the action result. This method streams the output Excel file to the browser.

A controller action cannot return an action result directly. For example, if you want to return a view from a controller action, you don't return an object of type ViewResult; instead, you call the **View** method. The View method instantiates a new object of type ViewResult and returns it to the browser.

An extension method named **SaveAsActionResult** has to be created to add the Controller class. The SaveAsActionResult method returns an object of type, ExcelResult.

The following code illustrates the **SaveAsActionResult** method:

[C#]

```
public static ExcelResult SaveAsActionResult(this ExcelEngine _engine, this
IWorkbook _workbook, string filename, ExcelHttpContentType contentType)
{
    ExcelSaveType saveType, HttpResponse response, ExcelDownloadType
DownloadType, return new ExcelResult(_engine, _workbook, filename,
response, DownloadType, contentType);
}
```

Step 3: Controller

Add the following code in the Controller's action result:

[C#]

```
public ActionResult createsheet()
{
    return View();
}

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult createsheet(string SaveOption, string Browser)
{
    //Step 1 : Instantiate the spreadsheet creation engine.
    ExcelEngine excelEngine = new ExcelEngine();
    //Step 2 : Instantiate the excel application object.
    IApplication application = excelEngine.Excel;

    // Creating new workbook
    IWorkbook workbook = application.Workbooks.Create(3);
    IWorksheet sheet = workbook.Worksheets[0];
    sheet.Range["A1:A5"].Text = "HelloWorld";
    return excelEngine.SaveAsActionResult(workbook, "Sample.xls",
HttpContext.ApplicationInstance.Response,
ExcelDownloadType.Open,ExcelHttpContentType.Excel97);
}
```

Step 4: Run the Application

Now try running the project by selecting **Start Debugging** option from the **Debug** menu. The following dialog box appears:



Figure 24: Opening <File> Dialog Box

The dialog box shown above provides options to download and launch the generated file.

This step completes the process of generating an Excel document in an MVC application.

3.4 Web Application Deployment

Web application by default is deployed in full trust mode. This section discusses the deployment in medium or partial trust scenarios.

Deploying in Medium Trust or Partial Trust Scenarios

There are two such scenarios in which Syncfusion assemblies might be deployed.

Example 1

If the Syncfusion Assemblies are in **GAC** (Global Assembly Cache), and the **Web Application** is running in **medium** trust, then the Syncfusion assemblies actually runs in **full** trust. Hence this scenario is fully supported and there are no additional steps necessary for deployment.

Example 2

Say, the Syncfusion Assemblies are present in the application's **bin** folder and the Web Application is running in **medium** trust, then the Syncfusion assemblies will run in **medium** trust.

You have to use following assemblies instead of **XlsIO.Base** to work with **XlsIO** in medium trust level:

- Syncfusion.Core.dll
- Syncfusion.Compression.Base.dll
- Syncfusion.XlsIO.Web.dll

No additional changes are required, except the assembly references.



Note: There will be reasonable performance lag, while using **XlsIO** in medium trust, for large files.

3.5 Feature Summary

This section provides a list of important features of Essential XlsIO with their definition and usage.

- **Formatting:** Essential XlsIO provides various formatting options like setting fonts, alignment of content, number formatting, border settings and color-fill settings. It also supports various styles for cells and conditional formatting options.

The following screen shot shows various formatting options provided in Essential XlsIO.

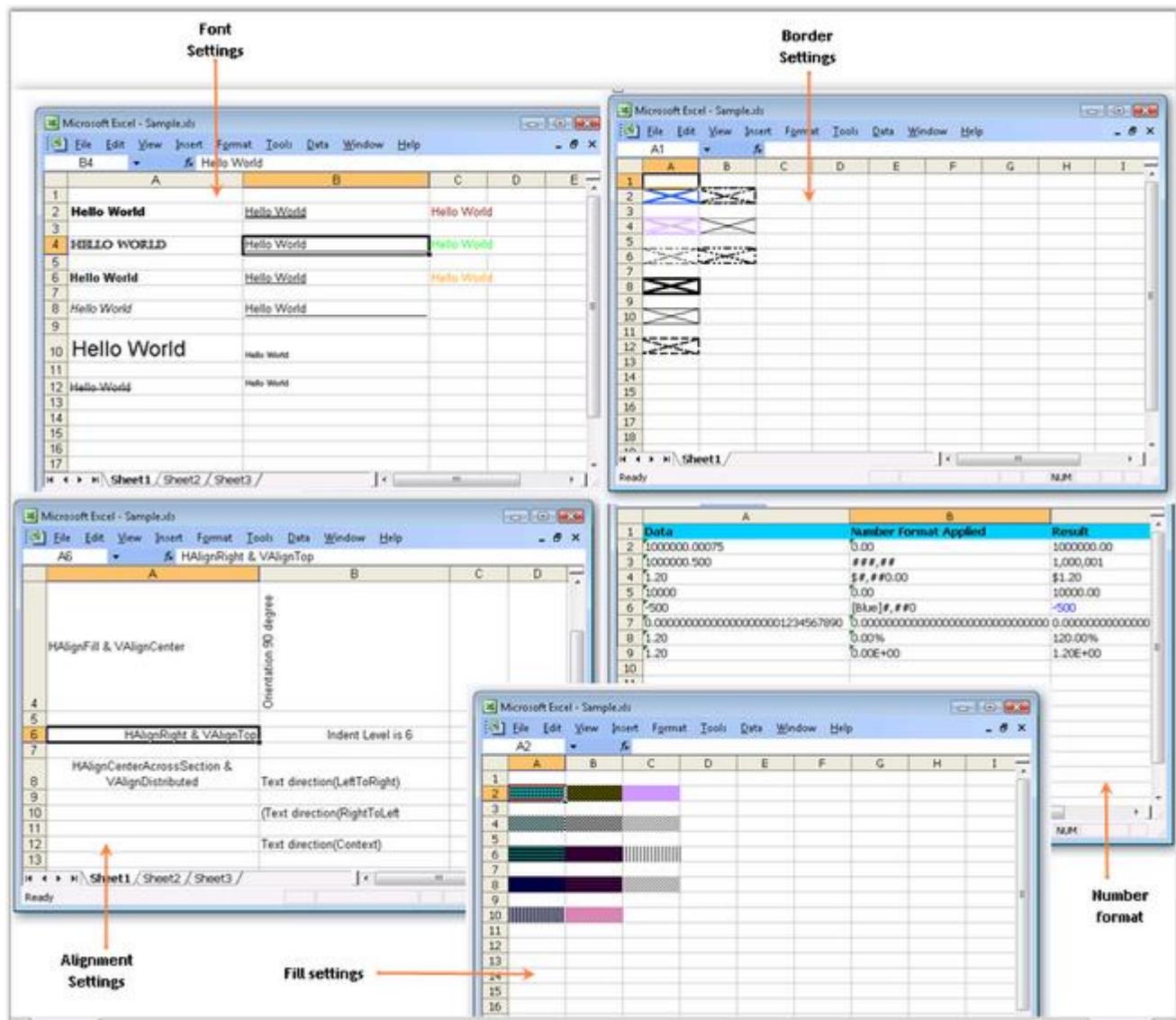


Figure 25: Various Formatting options

- **Editing:** Essential XlsIO supports range manipulations like copying a range, moving a range, and so on, and Find and Replace option as part of editing the document.

The following screen shot shows the Find and Replace dialog box.



Figure 26: Find and Replace

- **Insert Options:** Various components like chart, pictures and tables can be inserted into the document. It also provides support for insertion of controls like text boxes, check boxes and combo boxes, headers and footers, and links.

The following screen shot shows the insert options available in Essential XlsIO.

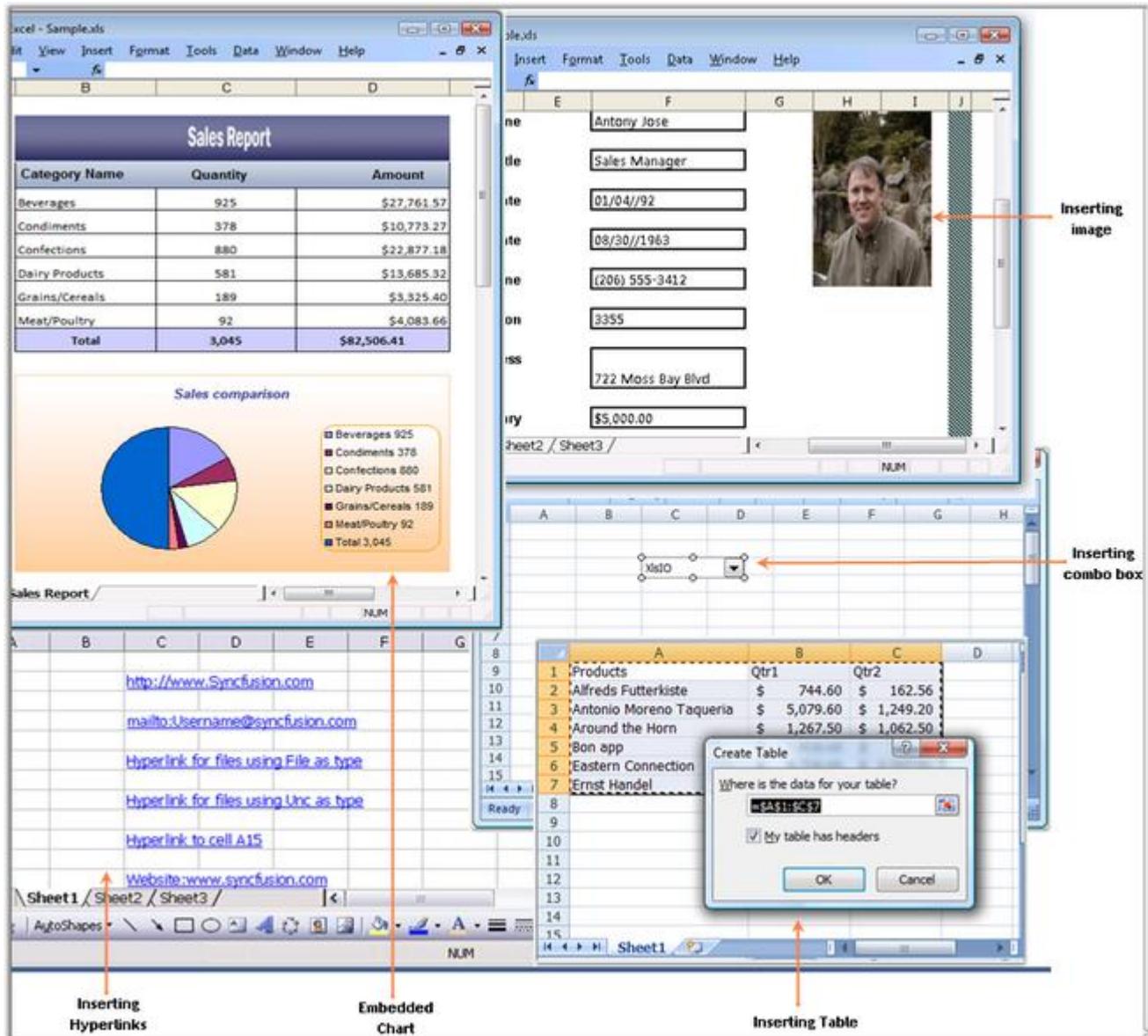


Figure 27: Various Insertion Options

- It provides support for data validation and import/export of data. It also provides data filter and template markers for efficient data handling.

The following screen shot shows data validation, import and export, and template marker features provided by Essential XlsIO.

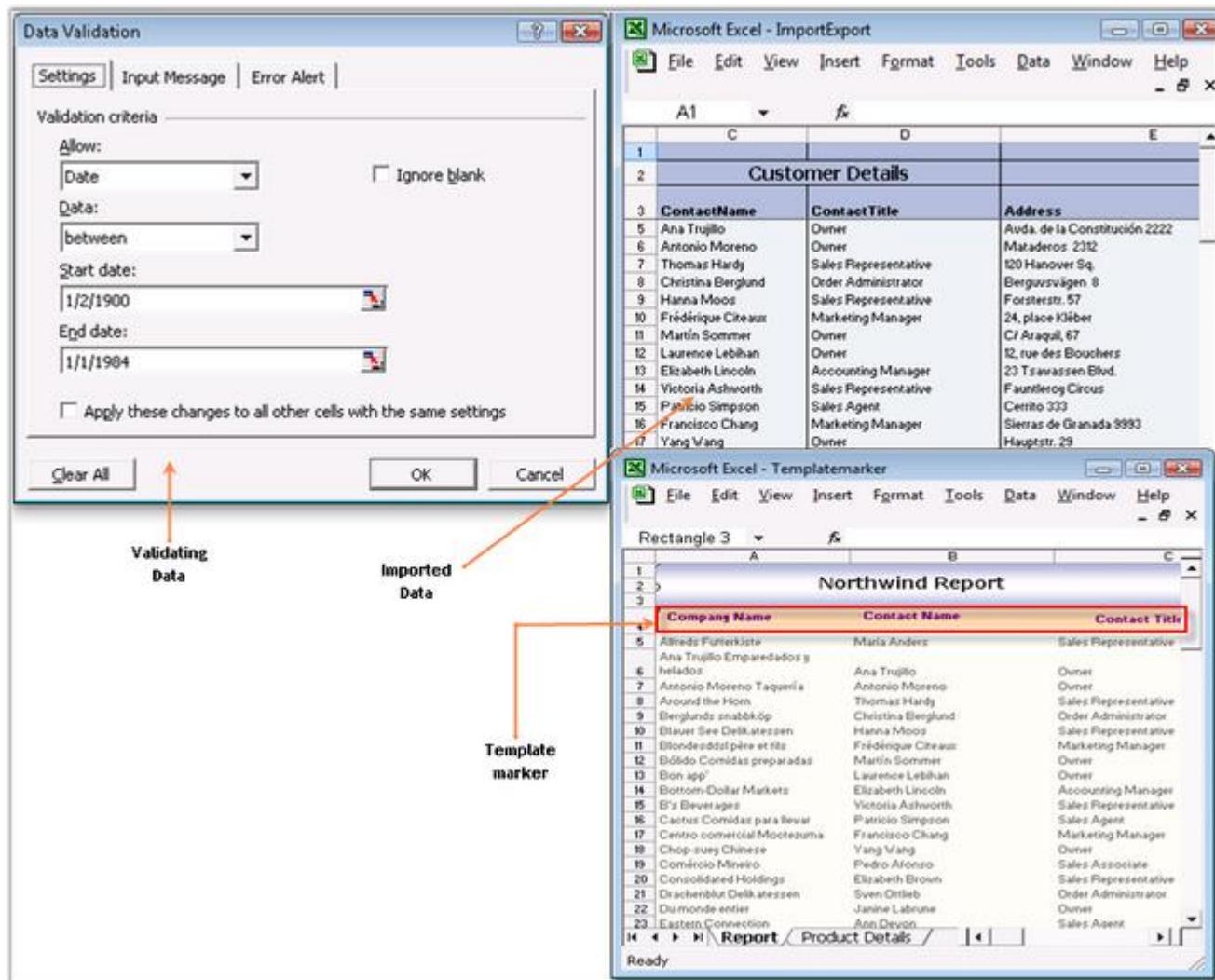


Figure 28: Data-related Features

- Add-Ins:** Essential XlsIO provides several add-ins for Microsoft Excel.

The following image shows the Bloomberg add-in.



Figure 29: Bloomberg Add-in

3.6 Spreadsheet

A spreadsheet is a grid that organizes data into columns and rows. Spreadsheets make it easy to display information, and people can insert formulas to work with the data. This section covers the following topics:

3.6.1 Excel Engine

ExcelEngine class provides access to **IApplication** interface that represents an Excel application. This class has the **Dispose** method, which is responsible for disposing the objects after closing the workbook.

Public Properties

The following table lists the public properties and their corresponding descriptions of **ExcelEngine** class:

Properties	Description
Excel	Interface to the XlsIO Application which gives access to all supported functions.
ThrowNotSavedOnDestroy	Dispose will throw an ExcelWorkbookNotSaved exception when the workbook is not saved and this property is set to True. Default value is False.
ActiveCell	Returns a range object that represents the active cell in the active window (the window on top) or in the specified window. If the window isn't displaying a worksheet, this property fails. It is a Read-Only property.
ActiveSheet	Returns an object that represents the active sheet (the sheet on top) in the active workbook or in the specified window or workbook. Returns nothing if no sheet is active. It is a Read-Only property.

ActiveWorkbook	Returns a Workbook object that represents the workbook in the active window (the window on top). It is a Read-Only property. Returns nothing if there are no windows open or if either the Info window or the Clipboard window is the active window.
ArgumentsSeparator	Formula arguments separator. This can be used to change the separator used in the formulas as per the culture.
Build	Returns the Microsoft Excel build number. It is a Read-Only property.
ChangeStyleOnCellEdit	If this property is set to True, then if some cells have reference to the same style, changes will influence all these cells. Default value is False.
CSVSeparator	Represents CSV Separator. Using for Auto recognize file type.
DecimalSeparator	Sets or returns the character used for the decimal separator as a String.
DefaultFilePath	Returns or sets the default path that Microsoft Excel uses when it opens files.
DefaultVersion	Gets/sets default Excel version. This value is used in create methods.
DeleteDestinationFile	Indicates whether XlsIO should delete the destination file before saving into it. Default value is True.
FixedDecimal	All data entered after this property is set to True will be formatted with the number of fixed decimal places set by the FixedDecimalPlaces property.
FixedDecimalPlaces	Returns or sets the number of fixed decimal places used when the FixedDecimal property is set to True.
Path	Returns the complete path to the application, excluding the final separator and name of the application. It is a Read-Only property.

PathSeparator	Returns the path separator character ("\"). It is a Read-Only property.
Range	Returns a Range object that represents a cell or a range of cells.
RowSeparator	Gets/sets row separator for array parsing.
SheetsInNewWorkbook	Returns or sets the number of sheets that Microsoft Excel automatically inserts into new workbooks.
StandardFont	Returns or sets the name of the standard font.
StandardFontSize	Returns or sets the standard font size, in points.
StandardHeight	Returns or sets the standard (default) height of all the rows in the worksheet, in points. This value is used only for newly created worksheets.
StandardHeightFlag	Returns or sets the standard (default) height option flag, which defines that standard (default) row height and book default font height do not match. This value is used only for newly created worksheets.
StandardWidth	Returns or sets the standard (default) width of all the columns in the worksheet. This value is used only for newly created worksheets.
ThousandsSeparator	Sets or returns the character used for the thousands separator as a String.
UseNativeOptimization	Indicates to use unsafe code.
UseNativeStorage	Indicates whether we should use native storage (standard windows COM object) or .NET implementation to open Excel 97-2003 files. This .NET storage doesn't depends on windows APIs and are developed with fully managed code. By default, it is set to true [uses native storage].
UserName	Returns or sets the name of the current user.

UseSystemSeparators	True (default) if the system separators of Microsoft Excel are enabled.
Workbooks	Returns a Workbooks collection that represents all the open workbooks.
Worksheets	For an Application object, this returns a Sheets collection that represents all the worksheets in the active workbook. For a Workbook object, this returns a Sheets collection that represents all the worksheets in the specified workbook.

Public Methods

The following table provides the list of methods and their corresponding descriptions of ExcelEngine class:

Methods	Description
CentimetersToPoints	Converts a measurement from centimeters to points (one point equals 0.035 centimeters).
ConvertUnits	Converts units.
InchesToPoints	Converts a measurement from inches to points.
Save	Saves changes to the active workbook.

Public Events

The following table lists the events of the ExcelEngine class and their corresponding descriptions:

Events	Description
OnPasswordRequired	This event is fired when user tries to open password protected workbook without specifying password. It is used to obtain password.
OnWrongPassword	This event is fired when user specified wrong password when trying to open password protected workbook. It is used to obtain correct password.

ProgressEvent	When workbook is read from stream and position of the stream changed, this event is raised.
---------------	---

3.6.2 Workbook

IWorkbooks

IWorkbooks is a collection of all the workbook objects that are currently open in the Excel application. This is responsible for adding new workbooks, opening existing workbooks, creating new workbooks and pasting workbooks.

Public methods exposed by this interface are given in the following table:

Public Methods

Methods	Description
Add	Add another workbook to the collection.
Close	Closes the workbook object.
Create	Creates new workbook with three worksheets, by default.
Open	Opens an existing document.
OpenFromXml	Opens SpreadsheetML document.
OpenReadOnly	Opens the workbook as read-only, when the workbook is already opened.
PasteWorkbook	Pastes workbook from the clipboard.

 There are various options for opening an existing workbook that allows us to skip parsing, select the version of the files, provide password and open as a stream. Opening from **disk.ExcelOpenType** has an option **Automatic** that enables the automatic selection of the version, as given below:

[C#]

```
excelEngine.Excel.Workbooks.Open("Excel2003.xls", ExcelOpenType.Automatic);
excelEngine.Excel.Workbooks.Open("Excel2007.xlsx", ExcelOpenType.Automatic);
```

For more details on **Open** method and its overloads, see Class Reference in [documentation](#).

IWorkbook

IWorkbook is used to represent a single workbook in the collection. This interface is responsible for customizing a single workbook and applies settings such as document properties, protection, find and replace that are common to the whole workbook. **Save/SaveAs** method of IWorkbook is responsible for saving any changes made to worksheets. Note that any changes made after Save/SaveAs statement will not be reflected in the output.

Public Methods

Method	Description
Activate	Activates the first window associated with the workbook.
AddFont	Adds font to the inner fonts collection and makes this font read-only.
Clone	Creates copy of the current instance.
Close	Close the workbook.
CopyToClipboard	Copies workbook to the clipboard.
CreateHFEEngine	Creates header/footer engine.
CreateTemplateMarkersProcessor	Creates object that can be used for template markers processing.
FindAll	Finds all the data that matches given string.
FindFirst	Finds the matching data that appears first in the workbook.
GetNearestColor	Gets the nearest color.
GetPaletteColor	Returns Color object from workbook palette by its index.

Protect	Protects the workbook.
Replace	Replaces the given data.
ResetPalette	Recovers palette to default values.
Save	Saves changes to the same workbook.
SaveAs	Saves changes to another workbook.
SaveAsXml	Save as Xml [SpreadsheetML].
SetColorOrGetNearest	Gets/sets the nearest color.
SetPaletteColor	Sets user color for specified element in Color table.
SetSeparators	Sets separators for formula parsing.
SetWriteProtectionPassword	This method sets write protection password.
Unprotect	Removes protection.

Public Properties

Property	Description
ActiveSheet	Returns an object that represents the active sheet (the sheet on top) in the active workbook or in the specified window or workbook. Returns nothing if no sheet is active. This is a Read-Only property.
ActiveSheetIndex	Gets/sets index of the active sheet.
AddInFunctions	Returns collection of all workbook's add-in functions. This is a Read-Only property.
Allow3DRangesInDataValidation	Indicates whether to allow usage of 3D ranges in DataValidation list property (MS Excel does not allow usage of 3D ranges in DataValidation list property).
Application	Application object for this object (Inherited from IParentApplication).
ArgumentsSeparator	Formula arguments separator.

Author	Returns or sets the author of the comment.
BuiltInDocumentProperties	Returns collection that represents all the built-in document properties for the specified workbook.
CalculationOptions	Returns calculation options. This is a Read-Only property.
Charts	Collection of the chart objects.
CodeName	Name which is used by macros to access the workbook items.
CustomDocumentProperties	Returns collection that represents all the custom document properties for the specified workbook. This is a Read-Only property.
Date1904	True if the workbook uses the 1904 date system.
DetectDateTimeInValue	Indicates whether library should try to detect string value passed to Value (and Value2) property as DateTime. Setting this property to false can increase performance greatly for such operations especially on Framework 1.0 and 1.1. Default value is True.
DisableMacrosStart	This property allows users to disable load of macros from document. Excel on file open will simply skip macros and will work as if document does not contain them. This option works only when file contains macros (HasMacros property is True).
DisplayedTab	Index of tab which will be displayed on document open.
DisplayWorkbookTabs	Indicates whether tabs are visible.
HasMacros	Indicates whether the opened workbook contains VBA macros.
IsCellProtection	Indicates if cell is protected.
IsRightToLeft	Indicates whether worksheet is displayed right to left.

IsWindowProtection	Indicates if window is protected.
MaxColumnCount	Returns maximum column count for each worksheet in this workbook. This is a Read-Only property.
MaxRowCount	Returns maximum row count for each worksheet in this workbook. This is a Read-Only property.
Names	For an Application object, returns a Names collection that represents all the names in the active workbook. For a Workbook object, returns a Names collection that represents all the names in the specified workbook (including all worksheet-specific names).
Palette	<p>Get Palette of colors which an Excel document can have. Here is a table of color indexes to places in the color tool box provided by Excel application:</p> <pre>----- 1 2 3 4 5 6 7 8 ----- 1 00 51 50 49 47 10 53 54 2 08 45 11 09 13 04 46 15 3 02 44 42 48 41 40 12 55 4 06 43 05 03 07 32 52 14 5 37 39 35 34 33 36 38 01 ----- ----- 6 16 17 18 19 20 21 22 23 7 24 25 26 27 28 29 30 31 ----- -----</pre>
PasswordToOpen	Gets/sets password to encrypt document.
ReadOnly	True if the workbook has been opened as Read-only.
ReadOnlyRecommended	True to display a message when the file is opened, recommending that the file be opened as read-only.
RowSeparator	Gets/sets row separator for array parsing.
Saved	This property is set to true if no changes have been made to the specified workbook since it was last saved.
StandardFont	Returns or sets the name of the standard font.

StandardFontSize	Returns or sets the standard font size, in points.
Styles	Returns a Styles collection that represents all the styles in the specified workbook.
TabSheets	Returns collection of tab sheets.
ThrowOnUnknownNames	Indicates whether exception should be thrown when unknown name was found in a formula.
Version	Gets/sets Excel version.
Worksheets	Returns a Sheets collection that represents all the worksheets in the specified workbook. Read-Only Sheets object.

Public Events

Event	Description
OnFileSaved	This event is handled after workbook is successfully saved.
OnReadOnlyFile	This event is handled when trying to save to a read-only file.

3.6.3 Worksheet

The worksheet object is a member of the **Worksheets** collection. The Worksheets collection contains all the worksheet objects in a workbook. **IWorksheet** interface is responsible for applying settings that are sheet-oriented. This controls the worksheet visibility, importing data from various data sources, and row and column manipulation. Following are the members of the **IWorksheets**.

Public Methods

Method	Description
Activate	Makes the current sheet the active sheet. Equivalent to clicking the sheet's tab.

Method	Description
AutofitColumn	Autofits specified column.
AutofitRow	Autofits specified row.
Clear	Clears worksheet data. Removes all formatting and merges.
ClearData	Clears worksheet. Only the data is removed from each cell.
ColumnWidthToPixels	Converts column width into pixels.
CopyToClipboard	Copies worksheet into the clipboard.
CreateRangesCollection	Creates new instance of IRanges and group different ranges.
CreateTemplateMarkersProcessor	Creates object that can be used for template markers processing.
DeleteColumn	Removes specified column (with formulas update).
DeleteRow	Removes specified row (with formulas update).
ExportDataTable	Exports sheet data as data table.
FindAll	Finds all matching data.
FindFirst	Finds the first data that matches the constraint.
GetBoolean	Gets bool value from the cell.
GetColumnWidth	Returns width from ColumnInfoRecord if there is corresponding ColumnInfoRecord or StandardWidth if not.
GetColumnWidthInPixels	Returns width in pixels from ColumnInfoRecord if there is corresponding ColumnInfoRecord or StandardWidth if not.
GetDefaultColumnStyle	Returns default column style.
GetDefaultRowStyle	Returns default row style.
GetError	Gets error value from cell.

Method	Description
GetFormula	Returns formula value corresponding to the cell.
GetFormulaBoolValue	Gets formula bool value from cell.
GetFormulaErrorValue	Gets formula error value from cell.
GetFormulaNumberValue	Returns formula number value corresponding to the cell.
GetFormulaStringValue	Returns formula string value corresponding to the cell.
GetNumber	Returns number value corresponding to the cell.
GetRowHeight	Returns height from RowRecord if there is a corresponding RowRecord. Otherwise returns StandardHeight.
GetRowHeightInPixels	Returns height from RowRecord if there is a corresponding RowRecord. Otherwise returns StandardHeight.
GetText	Returns string value corresponding to the cell.
ImportArray	Overloaded.
Import DataColumn	Imports data from a DataColumn into worksheet.
Import DataTable	Imports data from a DataTable into worksheet.
Import DataView	Imports data from a DataView into worksheet.
Insert Column	Inserts Column.
Insert Row	Inserts row.
Intersect Ranges	Intersects two ranges.
Is Column Visible	Checks if Column with specified index is visible to end-user or not.
Is Row Visible	Checks if Row with specified index is visible to user or not.
Merge Ranges	Merges two ranges.

Method	Description
Move	Moves worksheet.
PixelsToColumnWidth	Converts pixels into column width (in characters).
Protect	Protects worksheet with or without password.
Remove	Removes worksheet from parent worksheets collection.
RemovePanes	Removes panes from a worksheet.
Select	Selects current tab sheet.
SetBlank	Sets blank in specified cell.
SetBoolean	Sets value in the specified cell.
SetColumnWidth	Sets column width.
SetColumnWidthInPixels	Sets column width in pixels.
SetDefaultColumnStyle	Sets style for the whole column.
SetDefaultRowStyle	Sets style for the whole rows.
SetError	Sets error in the specified cell.
SetFormula	Sets formula in the specified cell.
SetFormulaBoolValue	Sets formula bool value.
SetFormulaErrorValue	Sets formula error value.
SetFormulaNumberValue	Sets formula number value.
SetFormulaStringValue	Sets formula string value.
SetNumber	Sets value in the specified cell.
SetRowHeight	Sets row height.
SetRowHeightInPixels	Sets row height in pixels.
SetText	Sets text in the specified cell.

Method	Description
SetValue	Sets value in the specified cell.
ShowColumn	Shows/Hides the specified column.
ShowRow	Shows/Hides the specified row.
Unprotect	Unprotects worksheet's content with password.
Unselect	Unselects current tab sheet.
MarkAsFinal	Marks the workbook as final and read-only.

Public Properties

Property	Description
ActivePane	Identifier of pane with active cell cursor.
AutoFilters	Returns collection of worksheet's autofilters. This is a Read-Only property.
Cells	Returns all used cells in the worksheet. This is a Read-Only property.
Charts	Returns charts collection. This is a Read-Only property.
CodeName	Name that is used by macros to access the workbook items. This is a Read-Only property.
Columns	For a Worksheet object, returns an array of Range objects that represents all the columns on the specified worksheet.
Comments	Comments collection.
CustomProperties	Returns collection of custom properties. This is a Read-Only property.
DisplayPageBreaks	This property is set to true if page breaks (both automatic and manual) on the specified worksheet are displayed.

Property	Description
FirstVisibleColumn	Index to first visible column in right pane(s).
FirstVisibleRow	Index to first visible row in bottom pane(s).
HorizontalSplit	Position of the horizontal split (by, 0 = No horizontal split): Unfrozen pane: Height of the top pane(s) (in twips = 1/20 of a point) Frozen pane: Number of visible rows in top pane(s)
HPageBreaks	Returns an HPageBreaks collection that represents the horizontal page breaks on the sheet. This is a Read-Only property.
HyperLinks	Collection of all worksheet's hyperlinks.
Index	Returns the index number of the object within the collection of similar objects. This is a Read-Only property.
IsFreezePanes	Defines whether freezed panes are applied.
IsGridLinesVisible	This property is set to true if gridlines are visible; false otherwise.
IsPasswordProtected	Indicates if the worksheet is password protected.
IsRightToLeft	Indicates whether worksheet is displayed right to left.
IsRowColumnHeadersVisible	This property is set to true if row and column headers are visible; false otherwise.
IsSelected	Indicates whether tab of this sheet is selected. This is a Read-Only property.
IsStringsPreserved	Indicates if all values in the workbook are preserved as strings.
MergedCells	Returns all merged ranges. This is a Read-Only property.
MigrantRange	Returns instance of migrant range - row and column of this range object can be changed by user. This is a Read-Only property.

Property	Description
Name	Gets/sets name of the tab sheet.
Names	For a Worksheet object, returns a Names collection that represents all the worksheet-specific names (names defined with the "WorksheetName!" prefix). Read-Only Names object.
PageSetup	Returns a PageSetup object that contains all the page setup settings for the specified object. This is a Read-Only property.
Pictures	Returns pictures collection. This is a Read-Only property.
ProtectContents	Indicates if current sheet is protected.
ProtectDrawingObjects	This property is set to true if objects are protected. This is a Read-Only property.
Protection	Gets protected options. For setting the protection options, use "Protect" method.
ProtectScenarios	This property is set to true if the scenarios of the current sheet are protected. This is a Read-Only property.
Range	Returns a Range object that represents a cell or a range of cells.
Rows	For a Worksheet object, returns an array of Range objects that represents all the rows on the specified worksheet.
SplitCell	Return split cell range.
StandardHeight	Returns or sets the standard (default) height of all the rows in the worksheet, in points.
StandardHeightFlag	Returns or sets the standard (default) height option flag, which defines that standard (default) row height and book default font height do not match.
StandardWidth	Returns or sets the standard (default) width of all

Property	Description
	the columns in the worksheet.
TabColor	Gets/sets tab color.
TabColorRGB	Gets/sets tab color.
TabIndex	Returns index in the parent ITabSheets collection. This is a Read-Only property.
Type	Returns or sets the worksheet type.
UsedCells	Returns all accessed cells. This is a Read-Only property. WARNING: This property creates Range object for each cell in the worksheet, and creates new array each time user calls to it. It can cause huge memory usage especially if called frequently.
UsedRange	Returns a Range object that represents the used range on the specified worksheet. This is a Read-Only property.
UseRangesCache	Indicates whether all created range objects should be cached. Default value is false.
VerticalSplit	Position of the vertical split (px, 0 = No vertical split): Unfrozen pane: Width of the left pane(s) (in twips = 1/20 of a point) Frozen pane: Number of visible columns in left pane(s).
Visibility	Control visibility of worksheet to end-user.
VPageBreaks	Returns a VPageBreaks collection that represents the vertical page breaks on the sheet. This is a Read-Only property.
Workbook	Returns parent workbook. This is a Read-Only property.
Zoom	Zoom factor of document. Value must be in range from 10 till 400.

3.7 Saving a Workbook

Essential XlsIO is a Non-UI component that can be used on Windows Forms, Web Forms and WPF applications.

Any changes made in a new or existing worksheet will be affected, only if it is saved to a disk or a stream. XlsIO supports saving files to different formats in stream and disk by using the SaveAs method of IWorkbook. The workbook can be saved to stream/disk/response. The only code that is specific for the usage of XlsIO in a Windows Forms application and WPF application, is the saving of the spreadsheet to disk, and for Web Forms applications, it is the streaming of the spreadsheet to the client browser.

The following is the code sample to save the document to disk.

Saving Worksheet in Windows and WPF Applications

```
[C#]
// Saving the workbook to disk.
workbook.SaveAs("Sample.xls");
```

```
[VB .NET]
'Saving the workbook to disk.
workbook.SaveAs("Sample.xls");
```

Saving Worksheet in Web Applications

```
[C#]
// Stream the workbook to the client browser.
workbook.SaveAs("Sample.xls", ExcelSaveType.SaveAsXLS, Response,
ExcelDownloadType.Open);
```

```
[VB .NET]
'Stream the workbook to the client browser.
workbook.SaveAs("Sample.xls", ExcelSaveType.SaveAsXLS, Response,
ExcelDownloadType.Open)
```

Similarly, you can open an xlsx file inside the browser by using the following code sample.

[C#]

```
workbook.Version = ExcelVersion.Excel2007;
// Stream the workbook to the client browser.
workbook.SaveAs("Sample.xlsx", Response, ExcelDownloadType.Open);
```

[VB.NET]

```
workbook.Version = ExcelVersion.Excel2007
'Stream the workbook to the client browser.
workbook.SaveAs("Sample.xlsx", Response, ExcelDownloadType.Open);
```

Following code sample allows to prompt for the Save dialog box, to save the created file in some location in disk.

[C#]

```
// Stream the workbook to the client browser.
workbook.SaveAs("Sample.xls", ExcelSaveType.SaveAsXLS, Response,
ExcelDownloadType.PromptDialog);
```

[VB.NET]

```
'Stream the workbook to the client browser.
workbook.SaveAs("Sample.xls", ExcelSaveType.SaveAsXLS, Response,
ExcelDownloadType.PromptDialog)
```

For more information on overloads of the workbook's Save method, refer [Class Reference](#) in the online documentation.

This section explains saving the files to the below formats.

3.7.1 Excel97to2003

XlsIO provides various overloads to save an Excel workbook. By default, when you save a workbook with **.xls** extension, it is saved to the Biff8 format [Excel97to2003 format].

[C#]

```
[IWorksheet].SaveAs("[Desired File Name.xls]");

// Example
myWorkBook.SaveAs("Sample.xls");
```

[VB .NET]

```
[Workbook].SaveAs("[Desired File Name.xls]")

' Example
myWorkBook.SaveAs("Sample.xls")
```

You can also save to Excel97to2003 format, while opening an Excel 2007 file, by setting the version as given below.

[C#]

```
workbook.Version = ExcelVersion.Excel97to2003;
```

[VB .NET]

```
workbook.Version = ExcelVersion.Excel97to2003
```

For more Information refer:

[Excel 2007 \[.Xlsx-Biff12 format\]](#), [SpreadsheetML](#), [CSV format](#)

3.7.2 XLSX

Excel 2007 version of MS Excel has various advanced features, and overcomes the drawbacks of previous versions. Essential XlsIO introduces basic support for Excel 2007 and Excel 2010 **xlsx** format that includes support to read and write basic elements (listed below) into the document.

Here is a sample code sample that opens an **xlsx** file, makes some changes, and saves it as an **xlsx** file.

[C#]

```
//Open an existing Excel 2007 file. Note that you should select the
ExcelOpenType when opening
//.xlsx files
IWorkbook workbook = excelEngine.Excel.Workbooks.Open("Excel2007.xlsx",
ExcelOpenType.Automatic);

//The first worksheet object in the worksheets collection is accessed.
IWorksheet sheet = workbook.Worksheets[0];

//Write data
sheet.Range["C3:O28"].Text = "Hello world";

//Select the version to be saved
workbook.Version = ExcelVersion.Excel2007; //or
//workbook.Version = ExcelVersion.Excel2010;

//Save it as "Excel2007" format.
workbook.SaveAs("Sample.xlsx");
```

[VB.NET]

```
'Open an existing Excel 2007 file. Note that you should select the
ExcelOpenType when opening
'.xlsx files
Dim workbook As IWorkbook =
excelEngine.Excel.Workbooks.Open("Excel2007.xlsx", ExcelOpenType.Automatic)

'Select the version to be saved.
workbook.Version = ExcelVersion.Excel2007 'or
'workbook.Version = ExcelVersion.Excel2010

'The first worksheet object in the worksheets collection is accessed.
Dim sheet As IWorksheet = workbook.Worksheets(0)

'Write data
sheet.Range("C3:O28").Text = "Hello world"

'Save it as "Excel 2007" format.
workbook.SaveAs("Sample.xlsx")
```



Note: You can use the very same API to work with the xlsx file or any other older format.

You can also set the **default version** of the workbook when you want to work with the same format.

[C#]

```
ExcelEngine excelEngine = new ExcelEngine();
IApplication application = excelEngine.Excel;

//Select the default version as Excel 2007 or Excel 2010;
application.DefaultVersion = ExcelVersion.Excel2007; //or
//application.DefaultVersion = ExcelVersion.Excel2010;

//Open an existing Excel 2007 file.
IWorkbook workbook = excelEngine.Excel.Workbooks.Open("Excel2007.xlsx");

//Save it as "Excel2007" format.
workbook.SaveAs("Sample.xlsx");
```

[VB.NET]

```
Dim excelEngine As ExcelEngine = New ExcelEngine()
Dim application As IApplication = excelEngine.Excel

'Set the default version as Excel 2007;
application.DefaultVersion = ExcelVersion.Excel2007 'Or
'application.DefaultVersion = ExcelVersion.Excel2010

'Open an existing Excel 2007 file.
Dim workbook As IWorkbook =
excelEngine.Excel.Workbooks.Open("Excel2007.xlsx")

'Save it as "Excel 2007" format.
workbook.SaveAs("Sample.xlsx")
```

Essential XlsIO also allows to open an existing .xls file and save it to the .xlsx format [with supported elements], or open an .xlsx file and save it to the .xls format.

[C#]

```
//Open an existing Excel 2007 file. Note that you should select the
ExcelOpenType when opening
//.xlsx files
IWorkbook workbook = excelEngine.Excel.Workbooks.Open("Excel2007.xlsx",
ExcelOpenType.Automatic);

//Select the version to be saved.
```

```

workbook.Version = ExcelVersion.Excel97to2003;

//The first worksheet object in the worksheets collection is accessed.
IWorksheet sheet = workbook.Worksheets[0];

//Write data
sheet.Range["C3:O28"].Text = "Hello world";

//Save it as "Excel 97 to 2003" format.
workbook.SaveAs("Sample.xls");

```

[VB.NET]

```

'Open an existing Excel 2007 file. Note that you should select the
ExcelOpenType when opening
'.xlsx files
Dim workbook As IWorkbook =
excelEngine.Excel.Workbooks.Open("Excel2007.xlsx", ExcelOpenType.Automatic)

'Select the version to be saved.
workbook.Version = ExcelVersion.Excel97to2003

'The first worksheet object in the worksheets collection is accessed.
Dim sheet As IWorksheet = workbook.Worksheets(0)

'Write data
sheet.Range("C3:O28").Text = "Hello world"

'Save it as "Excel 97 to 2003" format.
workbook.SaveAs("Sample.xls")

```

Xlsx File Format Support List

Support for Excel 2007 file formats in XlsIO	
XML-based File Format Support	
	Gets/sets cells (text, date time, time span, error, number, Boolean, formula).
	New dimensions: 2^20 x 2^14.
	Range operations such as copy/move range, insert/remove row/column, formula updates after these operations.

Support for Excel 2007 file formats in XlsIO	
	Merged cells support.
	Row and column settings (default style, height/width, visibility).
	Named ranges support.
	Ability to open 2007 files and save into 2003 format (without unsupported items).
	AutoFilters (existing functionality).
	Read/write data validation (existing functionality).
	<ul style="list-style-type: none"> - Read/Write conditional formatting (existing functionality). - Increase possible rules number (in Excel 97-2003 there can be only three rules). - New visualizations [Data bar, Icon sets and Color scales].
	Adjust row/column height & width, insert rows/cols, group/ungroup [with summary settings], freeze pane and split pane.
	Images operations (insert/remove/move/resize/open/save, without fill).
	Read/Write hyperlinks (existing functionality).
	Comments (open/save/move/resize/add/remove/get and set rtf text, author, without fill).
	Document properties (built-in and custom) and sheet level properties.
	Page Setup (all existing properties including header/footer images and page breaks), page layout, zoom, sheet alignment [right to left] and page break preview.
	Worksheet properties (tab color, background

Support for Excel 2007 file formats in XlsIO	
	image, hide and rename).
	Ignore error indicator, show/hide gridlines and gridline color.
	Gets/sets RTF string.
	Worksheet protection with or without password, workbook window or structure protection without password.
	Encryption and Decryption.
	TextBox, CheckBox and Combo Box – Read/Write support.
	Pivot Table creation and formatting.
	Tables – Read/Write and Styles support (Table Formulas are not supported).
	Read/Write Excel 2007 Formulas.
	Chart - existing functionality except the options given below which are not supported. 1. Secondary axes [partial]. 2. Marker filling options. 3. Drop lines. 4. Chart referring to values in other worksheet/workbook.
Objects Preservation while Opening and Saving XLSX Format	
	Pivot Tables
	Shapes (auto shapes, Image styles).
Cell Styles Support	
	Read/Write/Set Cell styles (All styles in Biff8

Support for Excel 2007 file formats in XlsIO	
	format).
	Themes
	32-bit colors
	Excel 2007 Built-In Styles.
	Gradient Fill
	New count restriction - ~65000.

The following screen shot shows the output file generated by Essential XlsIO, with all the basic features supported [Multiple conditional formatting].

sample.xlsx [Last saved by user] - Microsoft Excel

F24 50000

	B	C	D	E	F	G	H	I	J	K
6										
7	Antony Jose									
8										
9										
10										
11										
12										
13										
14										
15										
16										
17										
18										
19										
20										
21										
22										
23										
24										
25										
26										
27										

Name Antony Jose

Title Sales Manager

Birth Date 01/04/92

Hire date 08/30/1963

Home phone (206) 555-3412

Extension 3355

Home address 722 Moss Bay Blvd

Salary \$50,000.00

Antony Jose graduated from St. Andrews University, Scotland, with a BSC degree in 1976. Upon joining the company as a sales representative in 1992, he spent 6 months in an orientation program at the Seattle office and then returned to his permanent post in London. He was promoted to sales manager in March 1993.

Sheet1 Sheet2 Sheet3

Ready 100%

Figure 30: Sample .xlsx file created with XlsIO

For More Information Refer:

[Excel 97 to 2003, SpreadsheetML, CSV format](#)

3.7.2.1 Reducing size of Excel 2007 & Excel 2010 files

The default compression technique, using which Essential XlsIO compresses files uses .NET compression. A new compression technique has been implemented that will considerably reduce the size of the compressed XLSX files.

Use Case Scenarios

The reduced size of the compressed file will result in reduced data transfer between applications.

How Compression level can be set

The Compression level can be set at IApplication interface. This will set the level for all the workbooks created using the same instance of Excel Engine.

[C#]

```
ExcelEngine excelEngine = new ExcelEngine();
IApplication application = excelEngine.Excel;
application.DefaultVersion = ExcelVersion.Excel2007; // or ExcelVersion
= ExcelVersion.Excel2010;
application.CompressionLevel
Syncfusion.Compression.CompressionLevel.Best;

workbook.SaveAs(fileName);
workbook.Close();
excelEngine.Dispose();
```

[VB]

```
Dim excelEngine As New ExcelEngine()
Dim application As IApplication = excelEngine.Excel
application.DefaultVersion = ExcelVersion.Excel2007 ' or ExcelVersion =
ExcelVersion.Excel2010
application.CompressionLevel =
Syncfusion.Compression.CompressionLevel.Best

workbook.SaveAs(fileName)
```

```
workbook.Close()
excelEngine.Dispose()
```

Following are the list of enumerations available:

Enum	Description
NoCompression	File will not be compressed.
BestSpeed	Fast compression with more size than normal compression.
BelowNormal	Compression speed and size will be between Normal and BestSpeed.
Normal	Both size and speed will be Normal.
AboveNormal	Takes more time to compress, with reduced file size.
Best	Slow compression, file size reduced to the best level.

3.7.3 SpreadsheetML

SpreadsheetML is an XML dialect developed by Microsoft to represent the information in an Excel workbook. SpreadsheetML allows you to save Excel workbooks as XML documents, and to open them in Excel. Microsoft created a format that allows you to save, in an XML-based file, almost every Excel customization (including formulas, data, and formatting).

SpreadsheetML file can be created with Office 2003 by selecting the **Save as type:** as **XML Spreadsheet (*.xml)**.

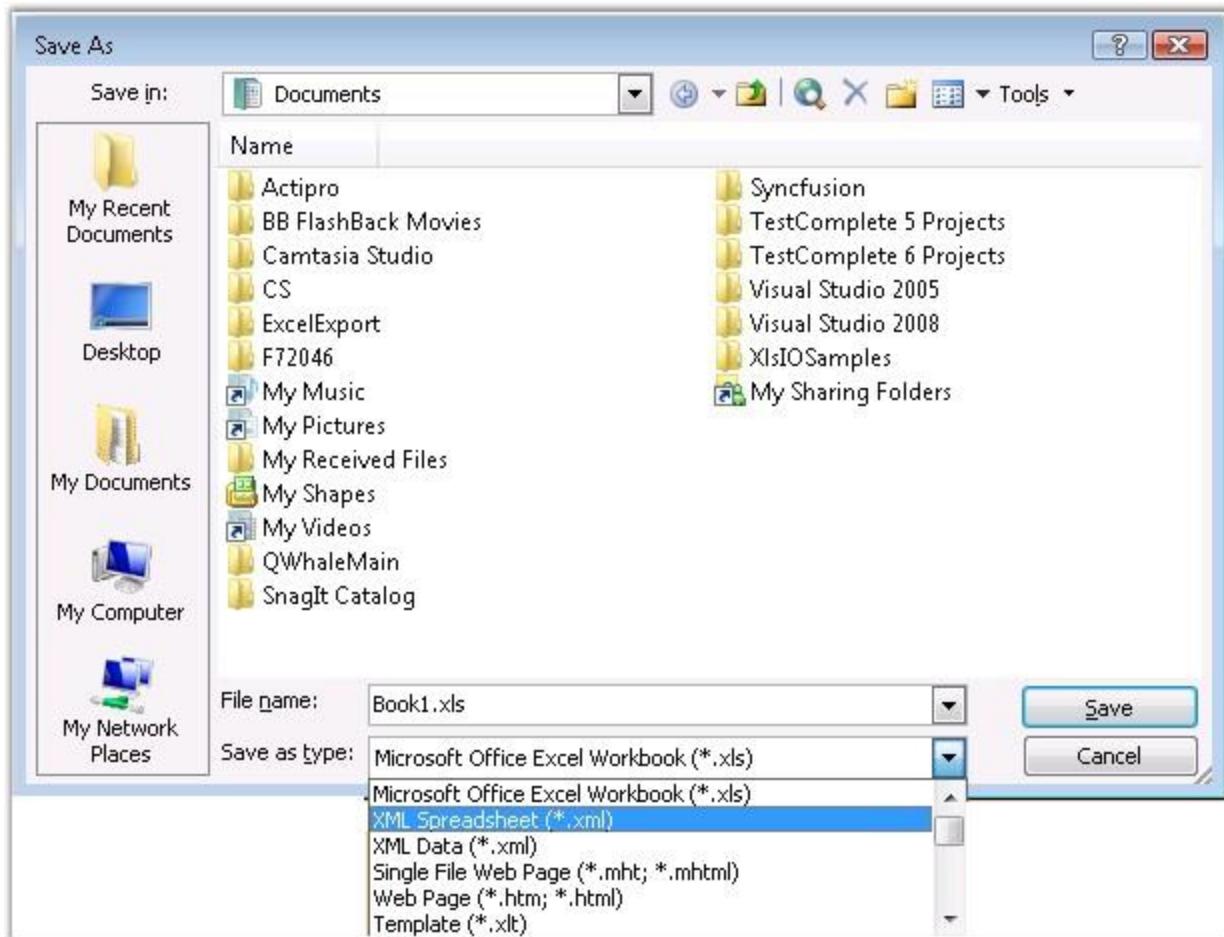


Figure 31: Saving as Xml Spreadsheet

Saving Excel Workbooks as XML By Using Essential XlsIO

Essential XlsIO provides support for reading and writing objects to SpreadsheetML format. Creating a SpreadsheetML file from scratch, has no difference when compared to the API used for Excel97-2003 format, except for the way it is saved.

Here is the code sample for creating a SpreadsheetML file.

[C#]

```
//Create a new workbook
IWorkbook workbook = excelEngine.Excel.Workbooks.Create(3);
```

```
//The first worksheet object in the worksheets collection is accessed.  
IWorksheet sheet = workbook.Worksheets[0];  
  
//Write data  
sheet.Range["C3:O28"].Text = "Hello world";  
  
//Save as SpreadsheetML.  
workbook.SaveAsXml("Sample.xml", ExcelXmlSaveType.MSExcel);
```

[VB .NET]

```
'Create a new workbook  
Dim workbook As IWorkbook = excelEngine.Excel.Workbooks.Create(3)  
  
'The first worksheet object in the worksheets collection is accessed.  
Dim sheet As IWorksheet = workbook.Worksheets(0)  
  
'Write data  
sheet.Range("C3:O28").Text = "Hello world"  
  
'Save as SpreadsheetML.  
workbook.SaveAsXml("Sample.xml",ExcelXmlSaveType.MSExcel)
```

Following code example illustrates how to open an existing SpreadsheetML file.

[C#]

```
//Open an existing SpreadsheetMl file.  
IWorkbook workbook = excelEngine.Excel.Workbooks.Open("spreadsheetml.xml",  
ExcelXmlOpenType.MSExcel);  
  
//The first worksheet object in the worksheets collection is accessed.  
IWorksheet sheet = workbook.Worksheets[0];  
  
//Write data  
sheet.Range["C3:O28"].Text = "Hello world";  
  
//Save as SpreadsheetML.  
workbook.SaveAsXml("Sample.xml", ExcelXmlSaveType.MSExcel);
```

[VB .NET]

```
'Open an existing SpreadsheetMl file.
```

```

Dim workbook As IWorkbook =
excelEngine.Excel.Workbooks.Open("spreadsheetml", ExcelXmlOpenType.MSExcel)

'The first worksheet object in the worksheets collection is accessed.
Dim sheet As IWorksheet = workbook.Worksheets(0)

'Write data
sheet.Range("C3:O28").Text = "Hello world"

'Save as SpreadsheetML.
workbook.SaveAsXml("Sample.xml", ExcelXmlSaveType.MSExcel)

```



Note: Currently XlsIO cannot parse the Document Properties apart from elements that are not supported by MS Excel. Colors created by XlsIO will choose the closest color to it when exported to the SpreadsheetML format. For more information refer [Excel 97 to 2003](#), [Excel 2007 \[Xlsx-Biff12 format\]](#), and [CSV format](#).

3.7.4 CSV Format

The Comma Separated Value (CSV) file format is a file type that stores tabular data. The CSV file format is often used to exchange data between disparate applications. The file format, as it is used in Microsoft Excel, has become a pseudo standard throughout the industry, even among non-Microsoft platforms.

XlsIO provides support for reading and writing CSV files. The following code example illustrates how to open a .csv file.

[C#]

```

// Opening a File.
IWorkbook workbook = application.Workbooks.Open("CSVfile.csv", "", "");

```

[VB.NET]

```

' Opening a File.
Dim workbook As IWorkbook = application.Workbooks.Open("CSVfile.csv", "", "")

```

While saving files, you have options to save as Unicode, ASCII, and other Non-Unicode encoding. The following code example illustrates how to save a file to the CSV format.

[C#]

```
// Saving the workbook to disk.
sheet.SaveAs("Sample.csv", ",", Encoding.ASCII);
```

[VB.NET]

```
' Saving the workbook to disk.
sheet.SaveAs("Sample.csv", ",", Encoding.ASCII)
```

For More Information Refer:

[Excel 97 to 2003](#), [Excel 2007 \[.Xlsx-Biff12 format\]](#), [SpreadsheetML](#)

3.8 Using Templates

This is the most widely used functionality of XlsIO as it is the easiest and the most convenient to use. An existing spreadsheet is used as a template for generating new spreadsheets. There are several advantages of using the Template-based approach.

- The spreadsheet can be formatted by using the MS Excel GUI, and the data alone can be inserted dynamically by using XlsIO. This saves the problem of writing code for formatting by using the XlsIO API.
- There are some features like VBA Macros which cannot be created by using the XlsIO's API. Even these features can be preserved by XlsIO, when a spreadsheet is opened and saved programmatically.
- Editing an existing data in the spreadsheet is also possible by opening the template with XlsIO.

[C#]

```
// New instance of XlsIO is created.[Equivalent to launching MS Excel with
no workbooks open].
// The instantiation process consists of two steps.

// Step 1: Instantiate the spreadsheet creation engine.
ExcelEngine excelEngine = new ExcelEngine();

// Step 2: Instantiate the excel application object.
IApplication application = excelEngine.Excel;
```

```

// Open an existing spreadsheet which will be used as a template for
generating the new spreadsheet.
// After opening the spreadsheet, the workbook object represents the
complete in-memory object model of the template spreadsheet.
IWorkbook workbook =
application.Workbooks.Open("SpreadsheetFromTemplate.xls");

// The first worksheet object in the worksheets collection is accessed.
IWorksheet sheet = workbook.Worksheets[0];

// Inserting some additional data.
sheet.Range["A1"].Text = "New text that was not present in the template";

// Saving the workbook to disk. This spreadsheet is the result of opening
and modifying
// an existing spreadsheet and then saving the result to a new workbook.
workbook.SaveAs("Sample.xls");

// Close the workbook.
workbook.Close();

// Dispose the Excel engine
excelEngine.Dispose();

```

[VB.NET]

```

' New instance of XlsIO is created.[Equivalent to launching MS Excel with no
workbooks open].
' The instantiation process consists of two steps.

' Step 1: Instantiate the spreadsheet creation engine.
Dim excelEngine As ExcelEngine = New ExcelEngine()

' Step 2: Instantiate the excel application object.
Dim application As IApplication = excelEngine.Excel

' Open an existing spreadsheet which, will be used as a template for
generating the new spreadsheet.
' After opening the spreadsheet, the workbook object represents the complete
in-memory object model of the template spreadsheet.
Dim workbook As IWorkbook =
application.Workbooks.Open("../..\..\..\..\Data\SpreadsheetFromTemplate.xls")

' The first worksheet object in the worksheets collection is accessed.
Dim sheet As IWorksheet = workbook.Worksheets(0)

```

```
' Inserting some additional data.  
sheet.Range("A1").Text = "New text that was not present in the template"  
  
' Saving the workbook to disk. This spreadsheet is the result of opening and  
modifying  
' an existing spreadsheet and then saving the result to a new workbook.  
workbook.SaveAs("Sample.xls")  
  
' Closing the workbook.  
workbook.Close()  
  
' Dispose the Excel engine  
excelEngine.Dispose()
```

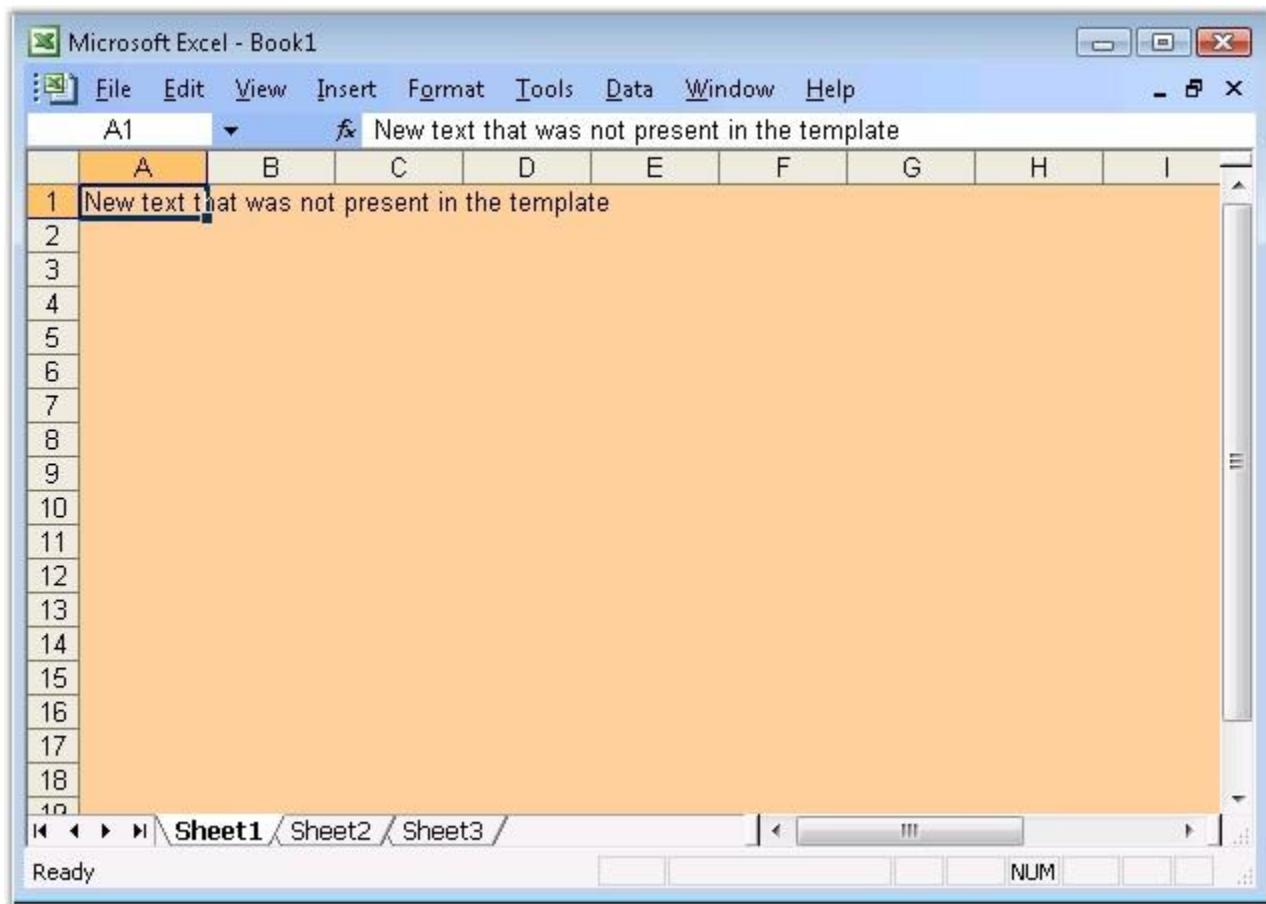


Figure 32: XlsIO with Spreadsheet from Template

XlsIO enables to open a spreadsheet in Read-Only mode, even if the spreadsheet is already open in your machine. The following code example illustrates how to do

this.

[C#]

```
// New instance of XlsIO is created.[Equivalent to launching MS Excel with no workbooks open].
// The instantiation process consists of two steps.

// Step 1: Instantiate the spreadsheet creation engine.
ExcelEngine excelEngine = new ExcelEngine();

// Step 2: Instantiate the Excel application object.
IApplication application = excelEngine.Excel;

// Open an existing spreadsheet which will be used as a template for generating the new spreadsheet.
// After opening the spreadsheet, the workbook object represents the complete in-memory object model of the template spreadsheet.
IWorkbook workbook =
application.Workbooks.OpenReadOnly("SpreadsheetFromTemplate.xls");
```

[VB.NET]

```
' New instance of XlsIO is created.[Equivalent to launching MS Excel with no workbooks open].
' The instantiation process consists of two steps.

' Step 1: Instantiate the spreadsheet creation engine.
Dim excelEngine As ExcelEngine = New ExcelEngine()

' Step 2: Instantiate the Excel application object.
Dim application As IApplication = excelEngine.Excel

' Open an existing spreadsheet which will be used as a template for generating the new spreadsheet.
' After opening the spreadsheet, the workbook object represents the complete in-memory object model of the template spreadsheet.
Dim workbook As IWorkbook =
application.Workbooks.OpenReadOnly("SpreadsheetFromTemplate.xls")
```

3.9 Improving Performance

Essential XlsIO can create large reports in few seconds.

 **Tips to improve the Performance**

- Use [default styles](#), which can be used to apply styles for the whole column instead of having to apply for each cell.
- Minimize AutoFit manipulations.
- Get **UsedRange** globally. It is recommended to get the UsedRange in loops as follows.

[C#]

```
for(int i = 0;i<sheet.UsedRange.LastRow;i++)

// Do not use the following method.
int lastRow = sheet.UsedRange.LastRow
for(int i=0;i<lastRow;i++)
```

- Use [IMigrantRange](#) to optimize memory performance while dealing with large strings.
- Use [global styles](#), rather than using different cell styles for each cell/range.
- Use **Begin** and **End** call while using more than one global style for a worksheet.
- Use **Value** and **Value2** property only when the data type is unknown. Value/Value2 property checks the data type of the given string and hence this consumes time.
- Files parsing can be optimized by setting **IApplication.UseFastRecordParsing = false** or **true** (true – fast mode, but less error checks and false – slower but more reliable).
- To extract values little faster, use **Unsafe** code option of **IApplication** interface as follows.

[C#]

```
application.DataProviderType = ExcelDataProviderType.Unsafe;
```

- Make use of **GetText**, **SetText**, **GetNumber** and **SetNumber** methods that enable users to get/set values without range object.
- Set **IWorkbook.DetectDateTimeInValue** property to **false** with **Value2** property, if you are sure that the given value is not of **DateTime** data type which improves time performance.
- Use of **BeginUpdate** and **EndUpdate** methods for large blocks of Data Validation greatly improves the performance.
- Use **DataProvider.Unsafe** option to increase performance while deleting large number of rows or columns.
- Use [CompressionLevel](#) to reduce the size of the file.

3.10 Supported Elements

This section includes the list of various supported and non-supported Excel elements of Essential XlsIO in various platforms. It has the following sections:

3.10.1 Windows, ASP.NET, WPF, ASP.NET MVC

The list of various supported and non-supported Excel elements of Essential XlsIO in Windows, ASP.NET, WPF and ASP.NET MVC platforms is given in the following table. XLS represents Excel 97 to 2003 format and XLSX represents Excel 2007 and Excel 2010 formats.



Note: Some Excel 2003 files generated using Essential XlsIO may fail when validated using Office File Validation of Office 2010.

Element	xls			xlsx			xls to xlsx
	Read	Write	Preserve	Read	Write	Preserve	
Document Properties	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Font settings	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Alignment s	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Number formatting	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Border settings	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Fill settings	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Cell styles	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Conditiona l formatting	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Element	xls			xlsx			xls to xlsx
	Read	Write	Preserve	Read	Write	Preserve	
Cell size[Row/column height/width, autofit]	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Hide/unhide rows/cols	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Hide/Unhide worksheet	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Copy/Move worksheet	Yes	Yes	Yes	Yes	Yes	Yes	-
Sheet protection	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Workbook protection	Yes	Yes	Yes	Yes [Without password]	Yes [Without password]	Yes [Without password]	Yes (Without password)
Sheet format[sheet name, tab color]	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Image	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Charts	Yes	Yes	Yes	Partial	Partial	Partial	Partial
Hyperlinks	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Header/Footer	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Pivot tables	No	No	Yes	No	Yes	Yes	No

Element	xls			xlsx			xls to xlsx
	Read	Write	Preserve	Read	Write	Preserve	
Pivot Chart	No	No	Yes	No	No	Yes	No
Auto Shapes	No	No	Yes	No	No	Yes	No
Text Box	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Check Box	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Combo Box	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Page setup [Margin,origin,page size]	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Page breaks	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Background image	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Print settings[Print area,Print titles,page order]	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Formulas	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Calculation options	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Names	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Formula auditing [Ignore]	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Element	xls			xlsx			xls to xlsx
	Read	Write	Preserve	Read	Write	Preserve	
error]							
Autofilter	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Data validation	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Template marker	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Outlines[group/ungroup, summary settings]	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Comments	Yes	Yes	Yes	Yes partial	Yes partial	Yes partial	Yes
Freeze pane, split pane	Yes	Yes	Yes	Yes	Yes	Yes	Yes
View[Zoom,show/hide gridline,show/hide headings], horizontal/ vertical scroll bars	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Macros	No	No	Yes	No	No	Yes	No
Encryption and Decryption (partial – just default)	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Element	xls			xlsx			xls to xlsx
	Read	Write	Preserve	Read	Write	Preserve	
algorithm)							
Ole Objects	No	No	No	No	No	No	No
Track changes	No	No	No	No	No	No	No
Themes	-	-	-	Yes partial	No	Yes partial	-
Cell gradient	-	-	-	Yes	Yes	Yes	-
Advanced CF [icon,databases,Colors,scales]	-	-	-	Yes	Yes	Yes	-
Tables	No	No	No	Yes	Yes	Yes	No
RGB colors	Yes	Yes	Indexed color	Yes	Yes	Yes	Yes
OLE Objects	No	No	Yes (Full Trust Only)	Yes (Full Trust Only)	No	No	No

3.10.2 Silverlight

The list of various supported and non-supported Excel elements of Essential XlsIO in Silverlight platform is given in the following table. Xls represents Excel 97 to 2003 format and XLSX represents both Excel 2007 and Excel 2010 file formats.

 **Note:** Some Excel 2003 files generated using Essential XlsIO may fail when validated using Office File Validation of Office 2010.

Element	xls		xlsx	
	Read	Write	Read	Write
Document Properties	Yes	Yes	Yes	Yes
Font settings	Yes	Yes	Yes	Yes
Alignments	Yes	Yes	Yes	Yes
Number formatting	Yes	Yes	Yes	Yes
Border settings	Yes	Yes	Yes	Yes
Fill settings	Yes	Yes	Yes	Yes
Cell styles	Yes	Yes	Yes	Yes
RGB Colors	Yes	Yes	Yes	Yes
Conditional formatting	Yes	Yes	Yes	Yes
Cell size[Row/Col height/width, autofit]	Yes	Yes	Yes	Yes
Hide/unhide rows/cols	Yes	Yes	Yes	Yes
Hide/Unhide worksheet	Yes	Yes	Yes	Yes
Copy/Move worksheet	Yes	Yes	Yes	Yes
Sheet protection	Yes	Yes	Yes	Yes
Workbook protection	Yes	Yes	Yes	Yes
Sheet format[sheet name,tab color]	Yes	Yes	Yes	Yes
Bitmap Images	Yes	Yes	Yes	Yes
Vector images	No	No	No	No

Element	xls		xlsx	
	Read	Write	Read	Write
Charts	Yes	Yes	Yes	Yes
Hyperlinks	Yes	Yes	Yes	Yes
Header/Footer	Yes	Yes	Yes	Yes
Pivot tables	No	No	No	Yes
Auto Shapes	No	No	No	No
Text box	Yes	Yes	Yes	Yes
Combo box	Yes	Yes	Yes	Yes
Check box	Yes	Yes	Yes	Yes
Page setup [Margin,origin,page size]	Yes	Yes	Yes	Yes
Page breaks	Yes	Yes	Yes	Yes
Background image	Yes	Yes	No	No
Print settings[Print area, Print titles, page order]	Yes	Yes	Yes	Yes
Formulas	Yes	Yes	Yes	Yes
Calculation options	Yes	Yes	Yes	Yes
Names	Yes	Yes	Yes	Yes
Formula auditing[Ignore error]	Yes	Yes	Yes	Yes
Autofilter	Yes	Yes	Yes	Yes
Data validation	Yes	Yes	Yes	Yes

Element	xls		xlsx	
	Read	Write	Read	Write
Template marker	Yes	Yes	Yes	Yes
Outlines[group/ungroup, summary settings]	Yes	Yes	Yes	Yes
Comments	Yes	Yes	Yes	Yes
Freeze pane, split pane	Yes	Yes	Yes	Yes
View[Zoom,show/hide gridline,show/hide headings], horizontal/vertical scroll bars	Yes	Yes	Yes	Yes
Macros	No	No	No	No
Encryption	Yes	Yes	Yes	Yes
Decryption	No	No	No	No
OLE Objects	No	No	No	No
Track changes	No	No	No	No
Streams	Yes	Yes	Yes	Yes
Themes	-	-	Yes	Yes
Cell gradient	-	-	Yes	Yes
Advanced CF [icon, databars, Color scales]	-	-	Yes	Yes
Tables	-	-	Yes	Yes

4 Concepts and Features

This section illustrates all the Microsoft Excel equivalent features of Essential XlsIO. The structure of this section is designed in such a way that the topics are arranged as per the Excel 2007 menu structure as shown in the following screen shot.

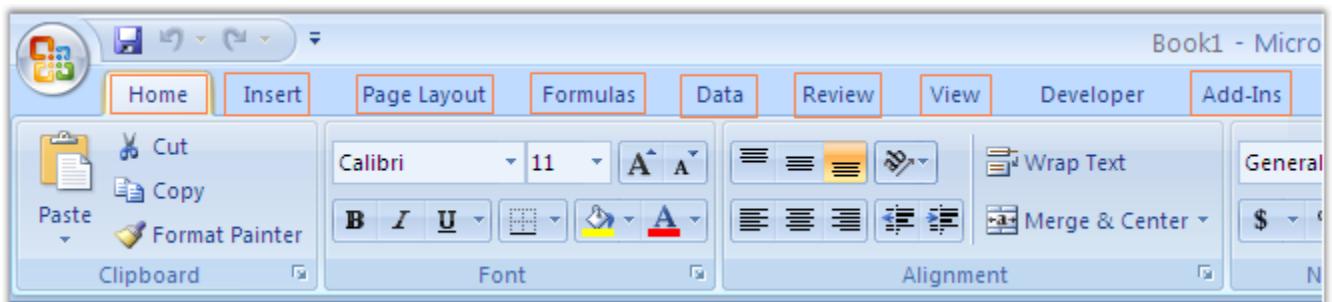


Figure 33: Excel 2007 Menu Structure

4.1 Home

This particular section explains all the basic features of Excel supported by XlsIO that are listed under Home menu.

4.1.1 Formatting

Data formatting gives a better readability and clear understanding of the data in the worksheet. Numerous formatting options are available like underlining, coloring, and so on, which organizes and clearly identifies the data within a worksheet. Minute refining options such as aligning the titles to center across the columns gives a professional appearance to the worksheet.

Microsoft Excel provides various options to format these worksheets. XlsIO is also enriched with formatting options similar to MS Excel. This section explains the formatting capabilities of XlsIO in the following subsections.

4.1.1.1 Font Settings

MS Excel provides support to customize the font settings through the **Format Cells** dialog box. Font tab in the format dialog box provides options to set the font name, size, color, and so on.

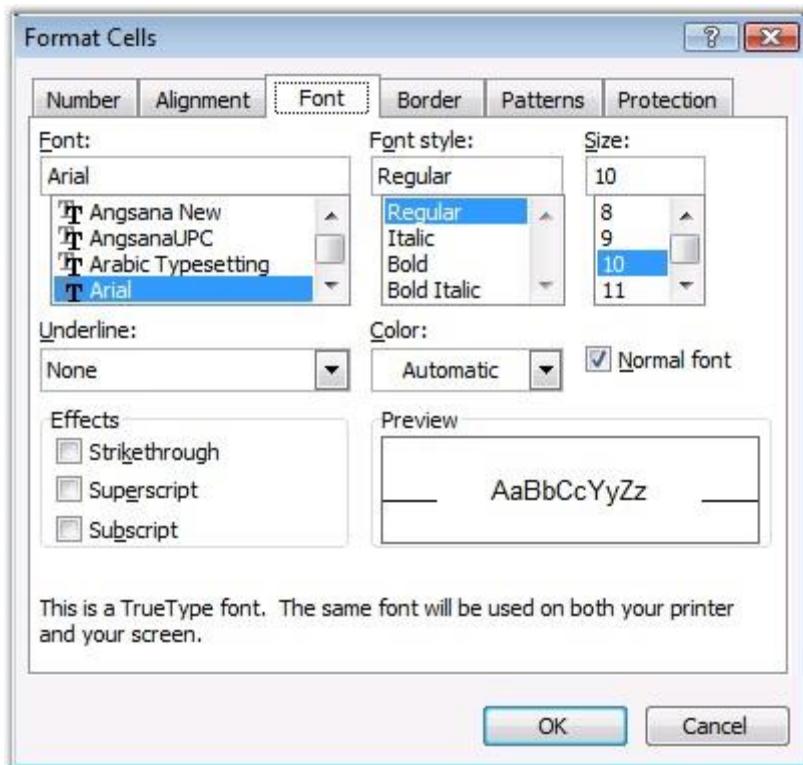


Figure 34: Font settings in MS Excel

Font Settings in XlsIO

XlsIO also has API support for specifying the font style for the text in the cells. **CellStyle** property exposes various font settings which is illustrated in the following code.

```
[C#]

// Setting Font Type.
sheet.Range["A2"].CellStyle.Font.FontName = "Arial Black";
sheet.Range["A4"].CellStyle.Font.FontName = "Castellar";

// Setting Font Styles.
sheet.Range["A6"].CellStyle.Font.Bold = true;
sheet.Range["A8"].CellStyle.Font.Italic = true;

// Setting Font Size.
```

```

sheet.Range[ "A10" ].CellStyle.Font.Size = 18;

// Setting Font Effects.
sheet.Range["A12"].CellStyle.Font.Strikethrough = true;
sheet.Range["B10"].CellStyle.Font.Subscript = true;
sheet.Range["B12"].CellStyle.Font.Superscript = true;

// Setting UnderLine Types.
sheet.Range["B2"].CellStyle.Font.Underline = ExcelUnderline.Double;
sheet.Range["B4"].CellStyle.Font.Underline =
ExcelUnderline.DoubleAccounting;
sheet.Range["B6"].CellStyle.Font.Underline = ExcelUnderline.Single;
sheet.Range["B8"].CellStyle.Font.Underline =
ExcelUnderline.SingleAccounting;

// Setting Font Color.
sheet.Range["C2"].CellStyle.Font.Color = ExcelKnownColors.Lavender;
sheet.Range["C4"].CellStyle.Font.Color = ExcelKnownColors.Light_blue;
sheet.Range["C6"].CellStyle.Font.Color = ExcelKnownColors.Indigo;

```

[VB.NET]

```

' Setting Font Type.
sheet.Range("A2").CellStyle.Font.FontName = "Arial Black"
sheet.Range("A4").CellStyle.Font.FontName = "Castellar"

' Setting Font Styles.
sheet.Range("A6").CellStyle.Font.Bold = True
sheet.Range("A8").CellStyle.Font.Italic = True

' Setting Font Size.
sheet.Range("A10").CellStyle.Font.Size = 18

' Setting Font Effects.
sheet.Range("A12").CellStyle.Font.Strikethrough = True
sheet.Range("B10").CellStyle.Font.Subscript = True
sheet.Range("B12").CellStyle.Font.Superscript = True

' Setting UnderLine Types.
sheet.Range("B2").CellStyle.Font.Underline = ExcelUnderline.Double
sheet.Range("B4").CellStyle.Font.Underline = ExcelUnderline.DoubleAccounting
sheet.Range("B6").CellStyle.Font.Underline = ExcelUnderline.Single
sheet.Range("B8").CellStyle.Font.Underline = ExcelUnderline.SingleAccounting

' Setting Font Color.
sheet.Range("C2").CellStyle.Font.Color = ExcelKnownColors.Lavender

```

```
sheet.Range("C4").CellStyle.Font.Color = ExcelKnownColors.Light_blue
sheet.Range("C6").CellStyle.Font.Color = ExcelKnownColors.Indigo
```

Editing Rich Text

XlsIO provides support for reading and writing rich text by using the **IRichTextString** interface. It enables to format each character in the cell with different font styles.



Note: Currently XlsIO cannot write formatted rich text.

[C#]

```
// Insert Rich Text.
IRange range = sheet.Range["A1"];
range.Text = "RichText";
IRichTextString rtf = range.RichText;

// Formatting first 4 characters.
IFont redFont = workbook.CreateFont();
redFont.Bold = true;
redFont.Italic = true;
redFont.RGBColor = Color.Red;
rtfSetFont(0, 3, redFont);

// Formatting last 4 characters.
IFont blueFont = workbook.CreateFont();
blueFont.Bold = true;
blueFont.Italic = true;
blueFont.RGBColor = Color.Blue;
rtfSetFont(4, 7, blueFont);
```

[VB .NET]

```
' Insert Rich Text.
Dim range As IRange = sheet.Range("A1")
range.Text = "RichText"
Dim rtf As IRichTextString = range.RichText

' Formatting first 4 characters.
Dim redFont As IFont = workbook.CreateFont()
redFont.Bold = True
redFont.Italic = True
redFont.RGBColor = Color.Red
rtf.SetFont(0, 3, redFont)
```

```
' Formatting last 4 characters.  
Dim blueFont As IFont = workbook.CreateFont()  
blueFont.Bold = True  
blueFont.Italic = True  
blueFont.RGBColor = Color.Blue  
rtfSetFont(4, 7, blueFont)
```

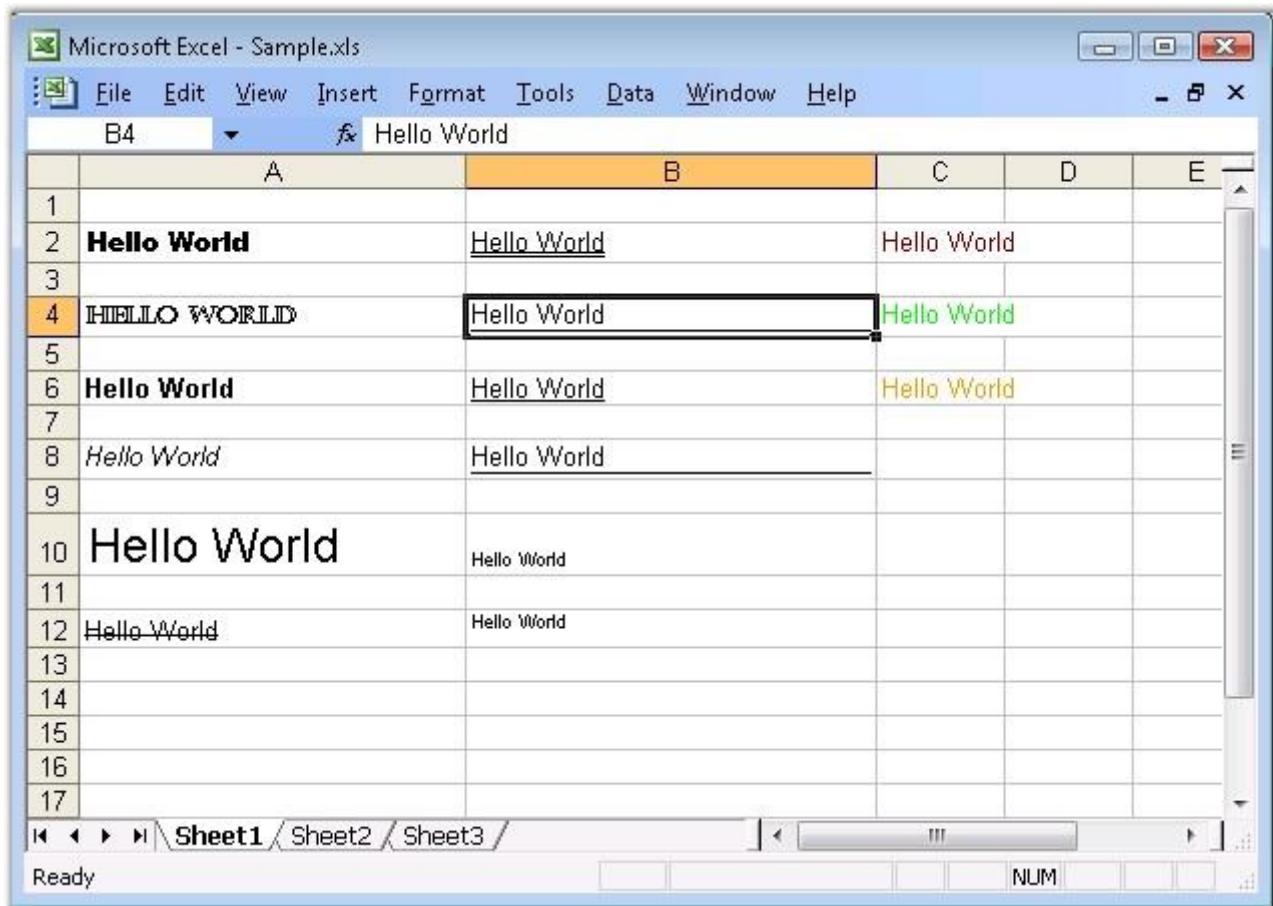


Figure 35: XlsIO with Font Settings

4.1.1.2 Alignment Settings

The following are some of the alignment settings available in Excel.

Text Alignment

Text has to be aligned inside the cells to properly fit in any data. This is done in Excel by using the Horizontal and Vertical alignment settings either through the **Formatting** toolbar or through the options provided by the **Alignment** tab in the **Format Cells** dialog box. In addition to the Left, Center, and Right alignments, Horizontal alignment option goes further and allows text to be justified. This can be especially useful, if the text is significantly long. Vertical alignment option allows you to align the contents of a cell towards the Top, Middle or Bottom area of a cell.

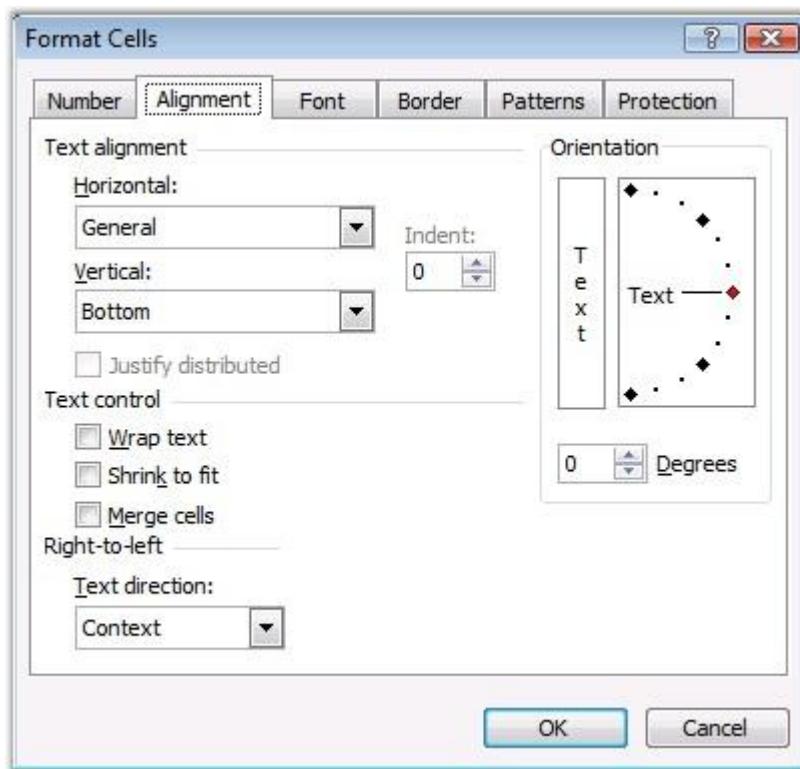


Figure 36: Alignment Settings in MS Excel

Indentation

In some circumstances, you may neither want to center the text nor keep it left or right aligned. In such cases, indentation can be done. Indentation consists of "pushing" the text to the left or right, without aligning it to center. To indent a text, you need to specify the number of units in the **Indent** box.

Alignment Settings in XlsIO

XlsIO supports alignment properties similar to Excel. The following code example illustrates the alignment settings that can be applied to the cells by using XlsIO.

[C#]

```
// Text Alignment Setting (Horizontal Alignment).
sheet.Range["A2"].CellStyle.HorizontalAlignment = ExcelHAlign.HAlignCenter;
sheet.Range["A4"].CellStyle.HorizontalAlignment = ExcelHAlign.HAlignFill;
sheet.Range["A6"].CellStyle.HorizontalAlignment = ExcelHAlign.HAlignRight;
sheet.Range["A8"].CellStyle.HorizontalAlignment =
ExcelHAlign.HAlignCenterAcrossSelection;

// Text Alignment Setting (Vertical Alignment).
sheet.Range["A2"].CellStyle.VerticalAlignment = ExcelVAlign.VAlignBottom;
sheet.Range["A4"].CellStyle.VerticalAlignment = ExcelVAlign.VAlignCenter;
sheet.Range["A6"].CellStyle.VerticalAlignment = ExcelVAlign.VAlignTop;
sheet.Range["A8"].CellStyle.VerticalAlignment =
ExcelVAlign.VAlignDistributed;

// Text Indent Setting.
sheet.Range["B6"].CellStyle.IndentLevel = 6;
```

[VB.NET]

```
' Text Alignment Setting (Horizontal Alignment).
sheet.Range("A2").CellStyle.HorizontalAlignment = ExcelHAlign.HAlignCenter
sheet.Range("A4").CellStyle.HorizontalAlignment = ExcelHAlign.HAlignFill
sheet.Range("A6").CellStyle.HorizontalAlignment = ExcelHAlign.HAlignRight
sheet.Range("A8").CellStyle.HorizontalAlignment =
ExcelHAlign.HAlignCenterAcrossSelection

' Text Alignment Setting (Vertical Alignment).
sheet.Range("A2").CellStyle.VerticalAlignment = ExcelVAlign.VAlignBottom
sheet.Range("A4").CellStyle.VerticalAlignment = ExcelVAlign.VAlignCenter
sheet.Range("A6").CellStyle.VerticalAlignment = ExcelVAlign.VAlignTop
sheet.Range("A8").CellStyle.VerticalAlignment =
ExcelVAlign.VAlignDistributed

' Text Indent Setting.
sheet.Range("B6").CellStyle.IndentLevel = 6
```

Text Control

The Text Control section provides three options: Wrap Text, Shrink To Fit, and Merge Cells.

At times, the text you enter in a cell will be wider than the cell. In such situations, the text may be hidden beyond the edge of the cell. Although one solution to this problem is to resize the cell, there are two additional solutions. They are, to Shrink the text to fit the cell, or Wrap the text so that it is displayed in multiple lines within the cell.

Yet another solution could be to merge multiple cells, so that the text can be fully displayed.

XlsIO allows to set these text control options by using the following APIs.

[C#]

```
// Merging of Cells.  
sheet.Range["A16:C16"].Merge();  
  
// Wrapping Text.  
sheet.Range["A14"].WrapText = true;
```

[VB .NET]

```
' Merging of Cells.  
sheet.Range("A16:C16").Merge()  
  
' Wrapping Text.  
sheet.Range("A14").WrapText = True
```

Orientation

The Orientation section enables you to bend the text to a fixed angle. There are two ways to set an angle. By dragging the small red diamond, one can specify the desired angle. You can also click one of the arrows of the **Degrees** spin button.

[C#]

```
// Text Orientation Settings.  
sheet.Range["B2"].CellStyle.Rotation = 60;  
sheet.Range["B4"].CellStyle.Rotation = 90;
```

[VB .NET]

```
' Text Orientation Settings.  
sheet.Range("B2").CellStyle.Rotation = 60;
```

```
sheet.Range("B4").CellStyle.Rotation = 90
```

Text Direction

You can specify the text orientation by using the **ReadingOrder** property. The following code example illustrates this.

[C#]

```
// Text Direction Setting.  
sheet.Range("B8").CellStyle.ReadingOrder =  
Syncfusion.XlsIO.ExcelReadingOrderType.LeftToRight;  
sheet.Range("B10").CellStyle.ReadingOrder =  
Syncfusion.XlsIO.ExcelReadingOrderType.RightToLeft;  
sheet.Range("B12").CellStyle.ReadingOrder =  
Syncfusion.XlsIO.ExcelReadingOrderType.Context;
```

[VB .NET]

```
' Text Direction Setting.  
sheet.Range("B8").CellStyle.ReadingOrder =  
Syncfusion.XlsIO.ExcelReadingOrderType.LeftToRight  
sheet.Range("B10").CellStyle.ReadingOrder =  
Syncfusion.XlsIO.ExcelReadingOrderType.RightToLeft  
sheet.Range("B12").CellStyle.ReadingOrder =  
Syncfusion.XlsIO.ExcelReadingOrderType.Context
```

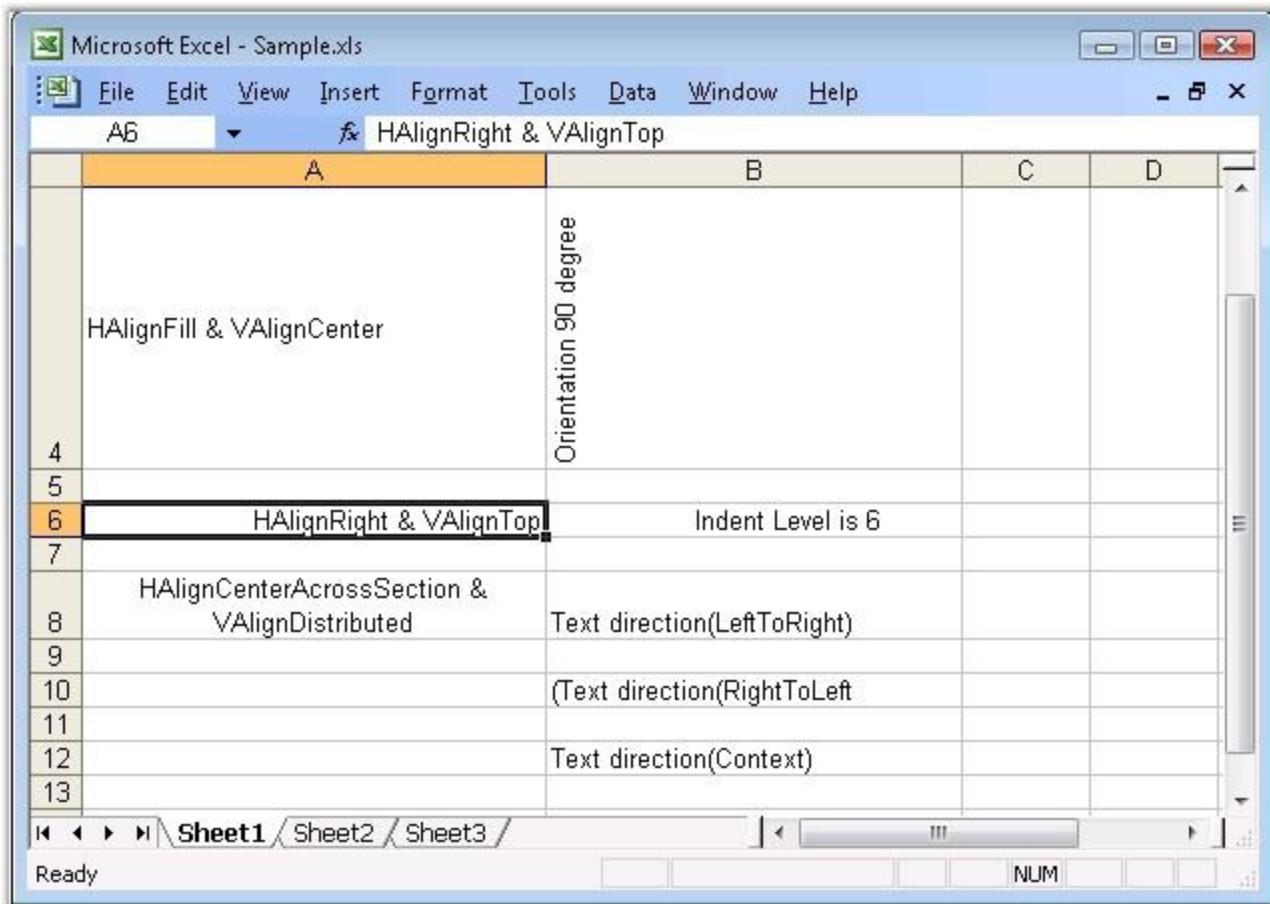


Figure 37: XlsIO with Alignment Settings

4.1.1.3 Number Formatting

Number Formats are little code that help you control the appearance of numbers in the Excel. A number in a cell is displayed depending on the number format applied to it. The same number can be displayed in different formats also. For example, 1.5 might represent a half teaspoon in one spreadsheet, while the same 1.5 would represent somebody's age, another spreadsheet's percentage, or so on.

Microsoft Excel recognizes the numbers in various formats: Accounting, Scientific, Fractions, and Currency. MS Excel allows to set these number formats by using the Number tab in the **Format Cells** dialog box.

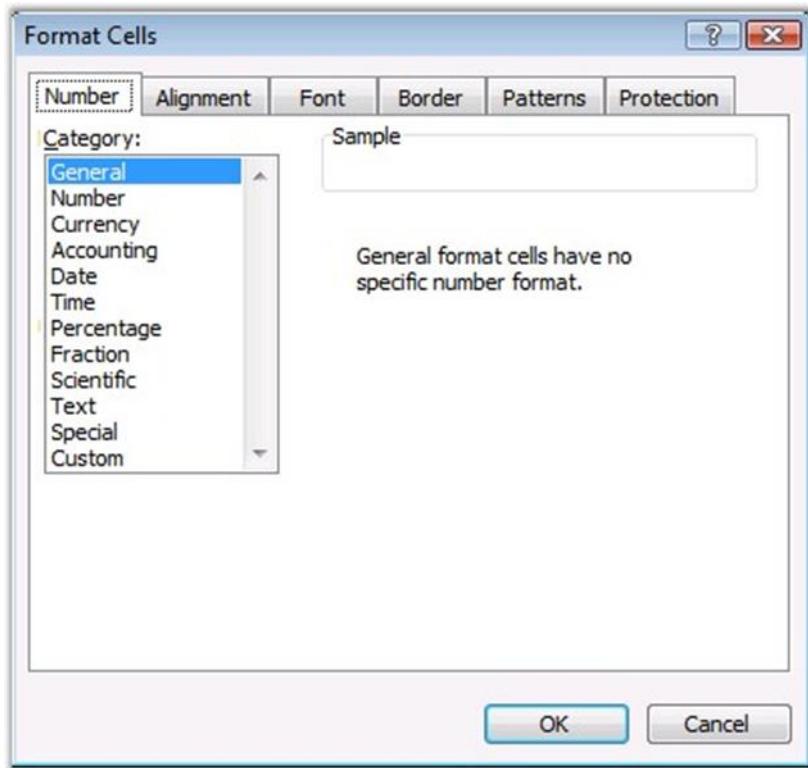


Figure 38: Number formatting in Excel

A number format consists of up to 4 items, separated by semicolons. Each of the items is an individual number format. The first, by default, applies to positive numbers, the second to negative numbers, the third to zeros, and the forth to text. If you don't apply any special formatting to text, Excel uses the 'General' number format, which basically means anything that will fit.

Long Numbers

You can make long numbers easier to read by inserting a thousands separator (a comma for US regional settings). The #,##0 number format is used. This format does not allow Excel to use the shorter scientific format, so the longest numbers merely show #####, signifying that they don't fit in their cells. You can either **autofit** the column or specify the column width in order to avoid showing #####.

Hide Zeros

Excel allows to hide zero values from displaying, when the user is not interested to show the values if it is "0". This can be done by setting custom format as **General** or **0.00**.

Leading Zeros

Excel removes the zero(0) in the zip code. The reason is that zero is not required to display the value of the number. However, since number formats are not about the value, but they are about the appearance, you can use a special format to retain the leading zeros, **00000**.

Special Number Formats

In addition to the 00000 Zip Code number format, Excel also has 00000-0000 for Zip-Plus-Four, 000-00-0000 for Social Security numbers, and (000)000-0000 for telephone numbers. If you live in a location that needs different zip codes, you can either design your own custom number formats.

The following table shows various custom formatting codes.

Number Code	Description
General	General number format.
0 (zero)	Digit placeholder. This code pads the value with zeros to fill the format.
#	Digit placeholder. This code does not display extra zeros.
?	Digit placeholder. This code leaves a space for insignificant zeros but does not display them.
. (period)	Decimal number.
%	Percentage. Microsoft Excel multiplies by 100 and adds the % character.
, (comma)	Thousands separator. A comma followed by a placeholder scales the number by a thousand.
E+ E- e+ e-	Scientific notation.
Text Code	Description
\$ - + / () : space	These characters are displayed in the number. To display any other character, enclose the character in quotation marks or precede it with a backslash.
\character	This code displays the character you specify. Note Typing !, ^, &, ', ~, {, }, =, <, or > automatically places a backslash in front of the character.

Number Code	Description
"text"	This code displays the text.
*	<p>This code repeats the next character in the format to fill the column width.</p> <p>Note: Only one asterisk per section of a format is allowed.</p>
_ (underscore)	<p>This code skips the width of the next character. This code is commonly used as "_" (without the quotation marks) to leave space for a closing parenthesis in a positive number format when the negative number format includes parentheses.</p> <p>This allows the values to line up at the decimal point.</p>
@	Text placeholder.
Date Code	Description
m	Month as a number without leading zeros (1-12).
mm	Month as a number with leading zeros (01-12).
mmm	Month as an abbreviation (Jan - Dec).
mmmm	Unabbreviated Month (January - December).
d	Day without leading zeros (1-31).
dd	Day with leading zeros (01-31).
ddd	Week day as an abbreviation (Sun - Sat).
dddd	Unabbreviated week day (Sunday - Saturday).
yy	Year as a two-digit number (for example, 96).
yyyy	Year as a four-digit number (for example, 1996).
Time Code	Description
h	Hours as a number without leading zeros (0-23).
hh	Hours as a number with leading zeros (00-23).

Number Code	Description
m	Minutes as a number without leading zeros (0-59).
mm	Minutes as a number with leading zeros (00-59).
s	Seconds as a number without leading zeros (0-59).
ss	Seconds as a number with leading zeros (00-59).
AM/PM am/pm	Time based on the twelve-hour clock.
Miscellaneous Code	Description
[BLACK], [BLUE], [CYAN], [GREEN], [MAGENTA], [RED], [WHITE], [YELLOW], [COLOR n]	These codes display the characters in the specified colors. Note: n is a value from 1 to 56 and refers to the nth color in the color palette.
[Condition value]	Condition may be <, >, =, >=, <=, <> and value may be any number. Note: A number format may contain up to two conditions.

Number Formatting in XlsIO

XlsIO provides API support for reading and writing the various built-in and custom number formats in a cell by using the **NumberFormat** property of **IRange**. To set the various number formats, use the appropriate typed properties [Number, DateTime, TimeSpan].

The following code example illustrates how to set the format for **Number**.

```
[C#]
// Applying Number Format.
sheet.Range["C2"].Number = 1000000.00075;
sheet.Range["B2"].Text = "0.00";
sheet.Range["C2"].NumberFormat = "0.00";

sheet.Range["C3"].Number = 1000000.500;
sheet.Range["B3"].Text = "###,##";
sheet.Range["C3"].NumberFormat = "###,##";

sheet.Range["C5"].Number = 10000;
```

[VB .NET]

The following code example illustrates how to set the format for **Percentage**.

[C#]

```
// Applying Number Format.
sheet.Range["C8"].Number = 1.20;
sheet.Range["B8"].Text = "0.00%";
sheet.Range["C8"].Number Format = "0.00%";
```

[VB.NET]

```
' Applying Number Format.
sheet.Range("C8").Number = 1.2
sheet.Range("B8").Text = "0.00%"
sheet.Range("C8").Number Format = "0.00%"
```

The following code example illustrates how to set the format for **DateTime**.

[C#]

```
sheet.Range["B2"].Number Format = "m/d/yyyy";
sheet.Range["B2"].DateTime = new DateTime(2005, 12, 25);
```

[VB.NET]

```
sheet.Range("B2").Number Format = "m/d/yyyy"
sheet.Range("B2").DateTime = New DateTime(2005,12,25)
```

The following code example illustrates how to set the format for **Currency**.

[C#]

```
// Applying Number Format.
sheet.Range["C4"].Number = 1.20;
sheet.Range["B4"].Text = "$#,##0.00";
sheet.Range["C4"].NumberFormat = "$#,##0.00";
```

[VB.NET]

```
' Applying Number Format.
sheet.Range("C4").Number = 1.2
sheet.Range("B4").Text = "$#,##0.00"
```

```
sheet.Range("C4").NumberFormat = "$#,##0.00"
```

Note that you can also set the custom format and other formats such as Accounting, by using the **NumberFormat** property as illustrated in the preceding code example.

The following code example illustrates entering data in the **Text** format.

[C#]

```
sheet.Range["C4"].Text = "1.20";
```

[VB .NET]

```
sheet.Range("C4").Text = "1.20"
```

XlsIO provides the following properties to get/set the data in the cells.

- **DisplayText**-Gets the text that is displayed in the cell. This is a read-only property, which returns a cell value that is displayed after the number format application.
- **Value**-Gets/sets the string from it. Since user can assign different data types, Value property parses that input string to determine which type was used, i.e., it sequentially checks whether it is an empty cell, formula, bool, error, number, or date time.
- **Value2**-This object returns/sets the cell value. This property works in the following way. It first checks whether the specified object has the type known for it (DateTime, TimeSpan, Double, Int). If yes, then it uses the corresponding typed properties (DateTime, TimeSpan, Number). Otherwise, it calls Value property with String data type.

The only difference between the Value2 property and the Value property is that the Value2 property does not use the Currency and Date data types. Also, it does not support FormulaArray value.

IsStringsPreserved property of IWorksheet is used modify its behavior and return bool, double or datetime. Otherwise, it returns Value property.

[C#]

```
sheet.IsStringsPreserved = true;
sheet.Range[1, 1].Value = "1";
```

```
sheet.Range[1, 2].Value = "Test";
sheet.Range[1, 3].Value = "=SUM(F1:F10)";
```

You can use these properties in the following way.

- If you have an object, and independent of its type, use Value2.
- If you have a string that can have different data types and do not want to parse it, use Value property.
- When you know the exact data (cell) type, use typed properties.

The following code example illustrates how to get the display text and value in the cell.

[C#]

```
sheet.Range["C4"].Number = 1.20;
sheet.Range["B4"].Text = "$#,##0.00";

Console.WriteLine(sheet.Range["C4"].DisplayText);
Console.WriteLine(sheet.Range["C4"].Value2.ToString());
```

[VB .NET]

```
sheet.Range("C4").Number = 1.20
sheet.Range("B4").Text = "$#,##0.00"

Console.WriteLine(sheet.Range("C4").DisplayText)
Console.WriteLine(sheet.Range("C4").Value2.ToString())
```

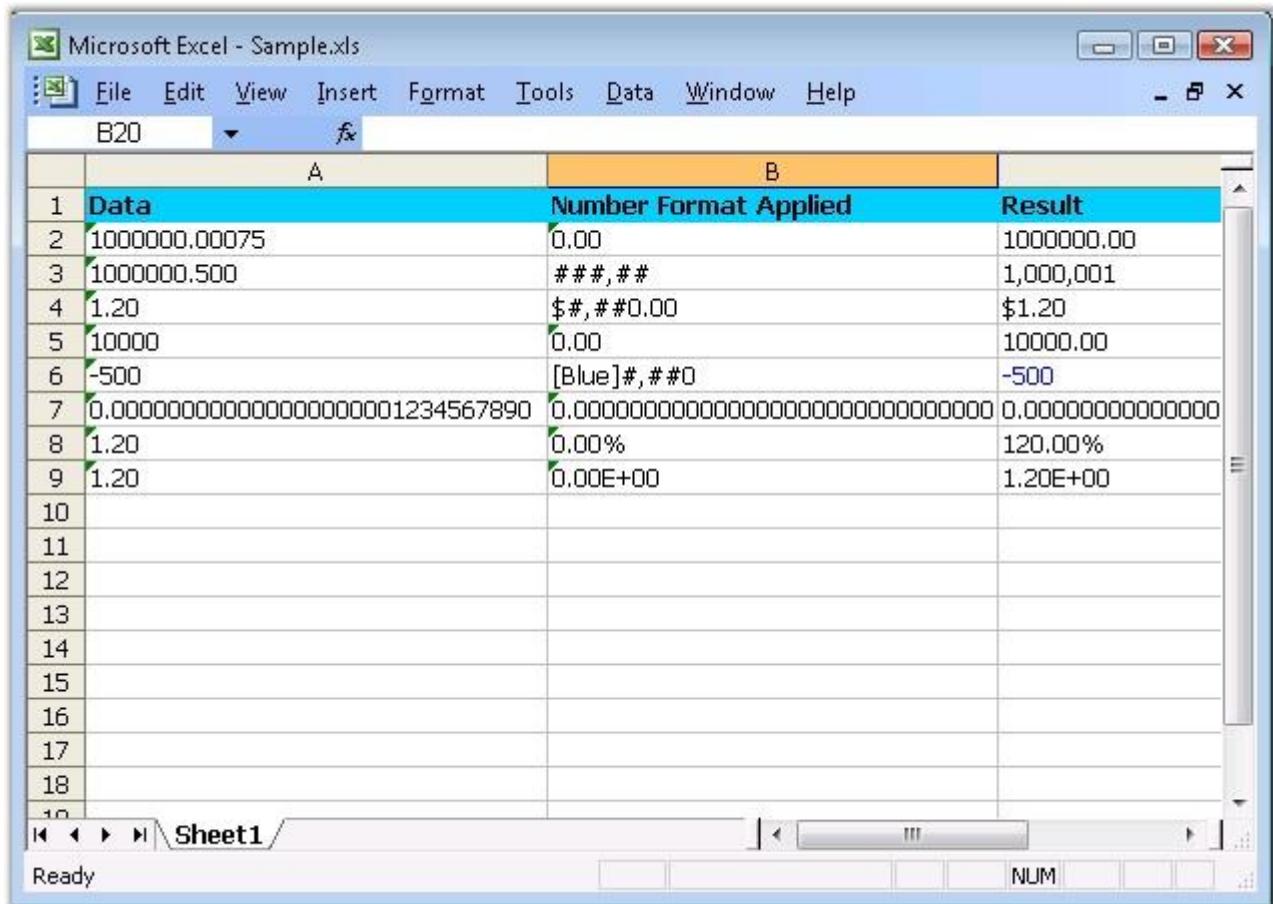


Figure 39: XlsIO with Number Formatting

4.1.1.4 Border Settings

Microsoft Excel provides a default appearance for a cell background. For example, it surrounds the cell with a gray border and a white background. You can control this default appearance through the **Formatting** toolbar or the **Border** tab in the **Format Cells** dialog box.

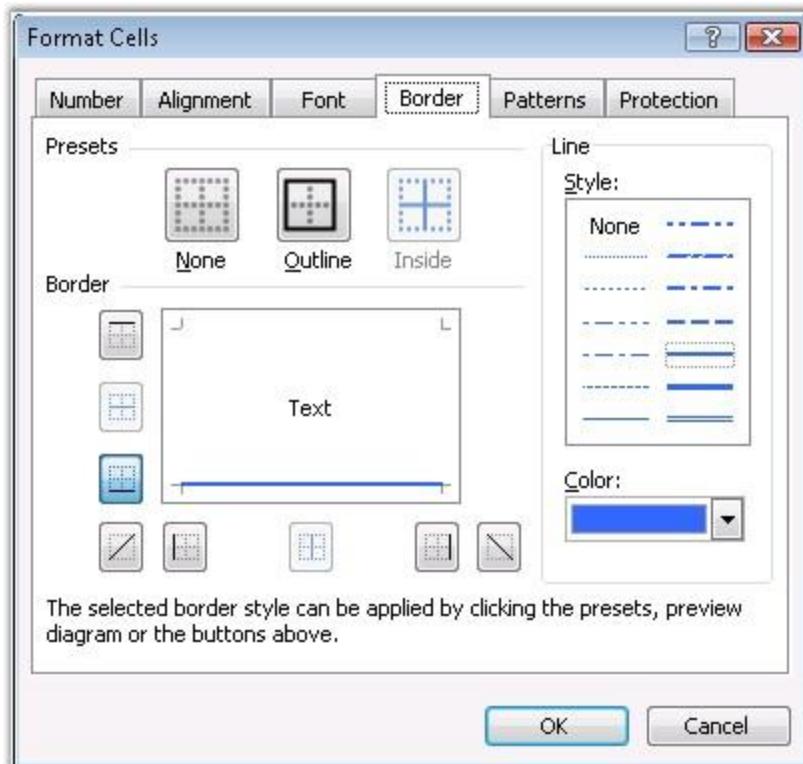


Figure 40: Format cells Dialog of MS Excel - Border tab

Border Settings in XlsIO

XlsIO provides support to insert and format borders through the **IBorder** interface. The following code example illustrates how this can be done.

[C#]

```
// The first worksheet object in the Worksheets collection is accessed.
IWorksheet sheet = workbook.Worksheets[0];

// Setting Border Line Styles.
sheet.Range["A2"].CellStyle.Borders.LineStyle = ExcelLineStyle.Medium;
sheet.Range["A4"].CellStyle.Borders.LineStyle = ExcelLineStyle.Double;
sheet.Range["A6"].CellStyle.Borders.LineStyle = ExcelLineStyle.Dash_dot;
sheet.Range["A8"].CellStyle.Borders.LineStyle = ExcelLineStyle.Thick;
sheet.Range["A10"].CellStyle.Borders.LineStyle = ExcelLineStyle.Thin;
sheet.Range["A12"].CellStyle.Borders.LineStyle =
ExcelLineStyle.Medium_dashed;
sheet.Range["B2"].CellStyle.Borders.LineStyle =
ExcelLineStyle.Slanted_dash_dot;
```

```

sheet.Range["B4"].CellStyle.Borders.LineStyle = ExcelLineStyle.Hair;
sheet.Range["B6"].CellStyle.Borders.LineStyle =
ExcelLineStyle.Medium_dot_dot;

// Setting the Border Color for Cell "A2".
sheet.Range["A2"].CellStyle.Borders[ExcelBordersIndex.DiagonalDown].Color =
ExcelKnownColors.Blue;
sheet.Range["A2"].CellStyle.Borders[ExcelBordersIndex.DiagonalUp].Color =
ExcelKnownColors.Blue;
sheet.Range["A2"].CellStyle.Borders[ExcelBordersIndex.EdgeBottom].Color =
ExcelKnownColors.Blue;
sheet.Range["A2"].CellStyle.Borders[ExcelBordersIndex.EdgeLeft].Color =
ExcelKnownColors.Blue;
sheet.Range["A2"].CellStyle.Borders[ExcelBordersIndex.EdgeRight].Color =
ExcelKnownColors.Blue;
sheet.Range["A2"].CellStyle.Borders[ExcelBordersIndex.EdgeTop].Color =
ExcelKnownColors.Blue;

```

[VB.NET]

```

' The first worksheet object in the worksheets collection is accessed.
Dim sheet As IWorksheet = workbook.Worksheets(0)

' Setting Border Line Styles.
sheet.Range("A2").CellStyle.Borders.LineStyle = ExcelLineStyle.Medium
sheet.Range("A4").CellStyle.Borders.LineStyle = ExcelLineStyle.Double
sheet.Range("A6").CellStyle.Borders.LineStyle = ExcelLineStyle.Dash_dot
sheet.Range("A8").CellStyle.Borders.LineStyle = ExcelLineStyle.Thick
sheet.Range("A10").CellStyle.Borders.LineStyle = ExcelLineStyle.Thin
sheet.Range("A12").CellStyle.Borders.LineStyle =
ExcelLineStyle.Medium_dashed
sheet.Range("B2").CellStyle.Borders.LineStyle =
ExcelLineStyle.Slanted_dot
sheet.Range("B4").CellStyle.Borders.LineStyle = ExcelLineStyle.Hair
sheet.Range("B6").CellStyle.Borders.LineStyle =
ExcelLineStyle.Medium_dot_dot

' Setting the Border Color for Cell "A2".
sheet.Range("A2").CellStyle.Borders(ExcelBordersIndex.DiagonalDown).Color =
ExcelKnownColors.Blue
sheet.Range("A2").CellStyle.Borders(ExcelBordersIndex.DiagonalUp).Color =
ExcelKnownColors.Blue
sheet.Range("A2").CellStyle.Borders(ExcelBordersIndex.EdgeBottom).Color =
ExcelKnownColors.Blue
sheet.Range("A2").CellStyle.Borders(ExcelBordersIndex.EdgeLeft).Color =
ExcelKnownColors.Blue
sheet.Range("A2").CellStyle.Borders(ExcelBordersIndex.EdgeRight).Color =

```

```
ExcelKnownColors.Blue
sheet.Range("A2").CellStyle.Borders(ExcelBordersIndex.EdgeTop).Color =
ExcelKnownColors.Blue
```

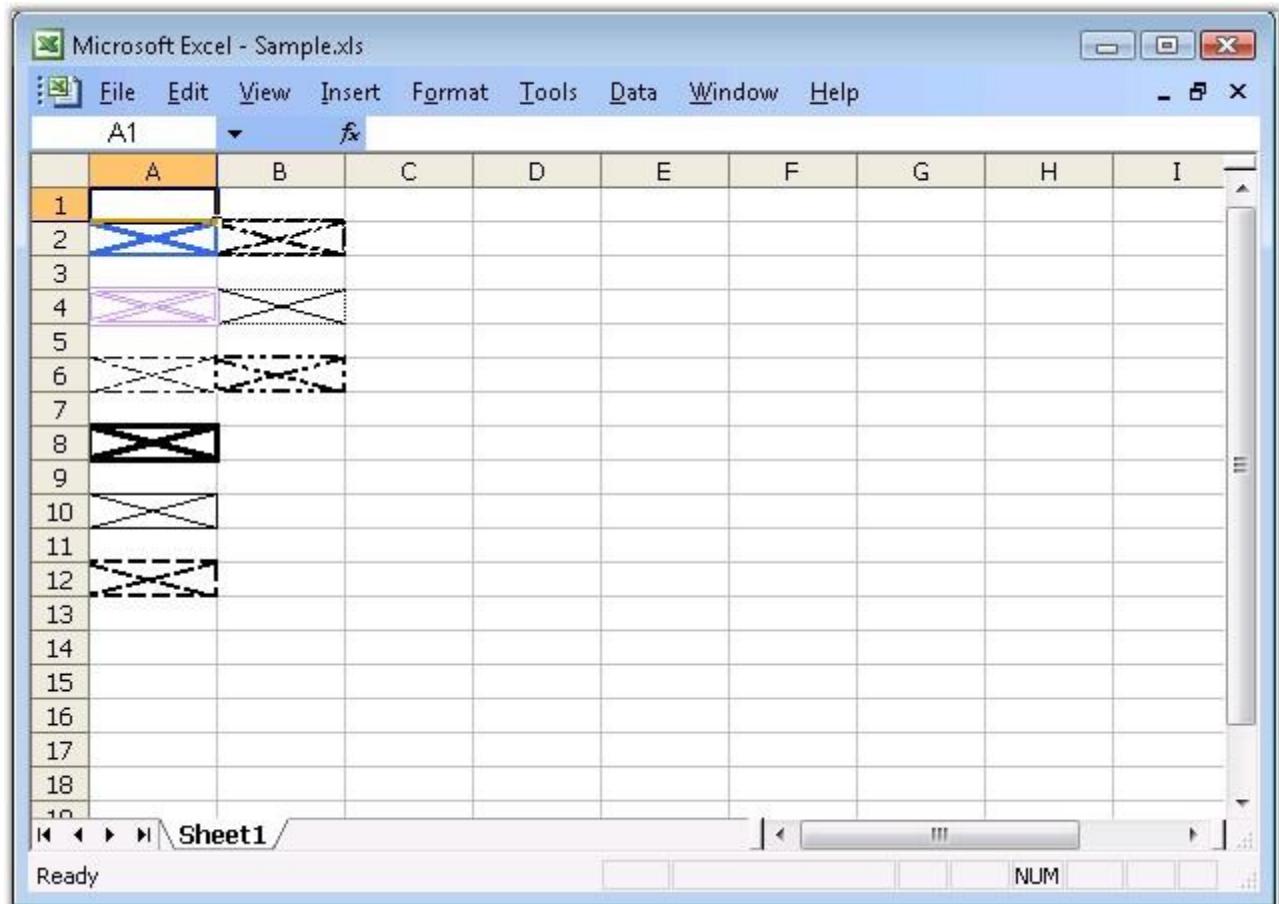


Figure 41: XlsIO with Border Settings

You can also set the borders for a range as follows.

[C#]

```
sheet.Range["C2"].BorderAround();
sheet.Range["C4"].BorderInside(ExcelLineStyle.Dash_dot,Color.Red);
```

[VB .NET]

```
sheet.Range("C2").BorderAround()
```

```
sheet.Range("C4").BorderInside(ExcelLineStyle.Dash_dot, Color.Red)
```

4.1.1.5 Fill Settings

This section illustrates the fill settings available in Excel.

Color

MS Excel provides support to format its cells, rows, and columns with various colors and patterns. This can be done by using the **Fill Color** button and an associated palette.

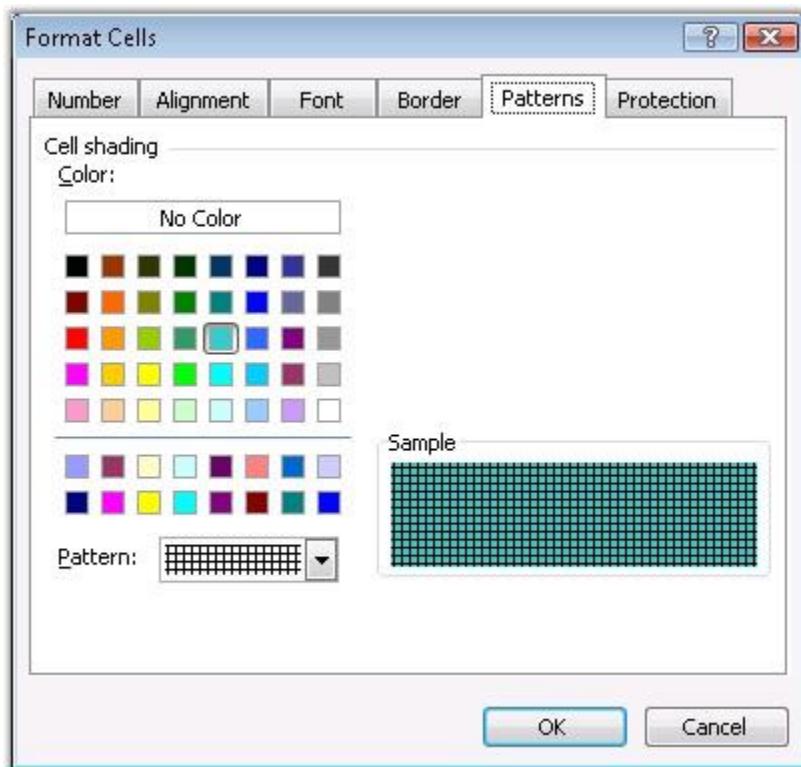


Figure 42: Format Cells Dialog Box

A color is a 4-byte number of the format 00BBGGRR, where RR, GG, and BB values are the Red, Green, and Blue values, each of which is between 0 and 255 (&HFF). If all component values are 0, the RGB color is 0, which is black. If all component values are 255 (&HFF), the RGB color is 16,777, 215 (&H00FFFFFF), or white. All other color combinations of values for the red, green, and blue components.

Color Pallet

Excel supports colors for fonts and background fills through what is called the **Color Pallet**. The Pallet is an array or series of 56 RGB colors. The value of each of those 56 colors may be any of the 16 million available colors, but the Pallet, and thus the number of distinct colors in a workbook, is limited to 56 colors. The RGB values in the Pallet are accessed by the **ColorIndex** property. The ColorIndex is an offset or index in the Pallet, and thus has a value between 1 and 56. In the default, unmodified Pallet, the 3rd element in the Pallet is the RGB value 255 (&HFF), which is red.

When you format a cell's background to red, for example, you are actually assigning to the **ColorIndex** property of the Interior a value of 3. Excel reads the 3 in the ColorIndex property, and goes to the 3rd element of the Pallet to get the actual RGB color. If you modify the Pallet, say by changing the 3rd element from red (255 = &HFF) to blue (16,711,680 = &HFF0000), all items that were once red are changed to blue. This is because the value of the 3rd element in the Pallet has been changed from red to blue, while the ColorIndex property remains equal to 3.

You can change the values in the default pallet by modifying the Colors array of the workbook. You can also get the colors in the palette by using the **Palette** property.

Colors in XlsIO

XlsIO provides support for adding new colors to the color palette that are not available in the standard MS Excel color palette, by using the **SetPaletteColor** method. If you have modified a workbook's Pallet, you can reset the pallet back to the default values, by using the **ResetPalette** method.

The following code example illustrates how to set the color palette.

```
[C#]

// Creating color palette.
string[] known = Enum.GetNames( typeof( KnownColor ) );
Color[] palette = workbook.Palette;

for( int i = ( int )ExcelKnownColors.Custom0; i < palette.Length; i++ )
{
    KnownColor value = ( KnownColor )Enum.Parse( typeof( KnownColor ), known[i] );
    workbook.SetPaletteColor( i, Color.FromKnownColor( value ) );
}

palette = workbook.Palette;
```

```

int pos = 0;
for( int j=1; j<100; j++ )
{
    for( int i=1; i<=3; i++ )
    {
        ExcelKnownColors knownEnm = (ExcelKnownColors)pos;
        sheet.Range[ j, i ].CellStyle.ColorIndex = knownEnm;
        sheet.Range[ j, i ].Text = palette[ pos ].Name + ", " +
            string.Format( "R{0}:G{1}:B{2}:A{3}", palette[ pos
] .R,
                palette[ pos ].G, palette[ pos ].B, palette[ pos ].A
);
        pos++;
    }

    if( pos >= palette.Length ) break;
}
if( pos >= palette.Length ) break;
}

```

[VB.NET]

```

' Creating color palette.
Dim known As String() = System.Enum.GetNames(GetType(KnownColor))
Dim palette As Color() = workbook.Palette

Dim i As Integer = CInt(ExcelKnownColors.Custom0)

Do While i < palette.Length
    Dim value As KnownColor =
CType(System.Enum.Parse(GetType(KnownColor), known(i)), KnownColor)
    workbook.SetPaletteColor(i, Color.FromKnownColor(value))
    i += 1
Loop

palette = workbook.Palette
Dim pos As Integer = 0
For j As Integer = 1 To 99
    For i = 1 To 3
        Dim knownEnm As ExcelKnownColors = CType(pos, ExcelKnownColors)
        sheet.Range(j, i).CellStyle.ColorIndex = knownEnm
        sheet.Range(j, i).Text = palette(pos).Name & ", " &
String.Format("R{0}:G{1}:B{2}:A{3}", palette(pos).R, palette(pos).G,
palette(pos).B, palette(pos).A)
        pos += 1

    If pos >= palette.Length Then

```

```

        Exit For
    End If
Next i

If pos >= palette.Length Then
    Exit For
End If
Next j

```

XlsIO also enables to get or set the closest RGB color in the pallet by using the **SetColorOrGetNearest** method. It returns the ColorIndex value of the color in the Pallet, that is closest to a given RGBLong color value. The method used here considers every RGB color to be a spatial location in a 3-dimensional space, where the axes are Red, Green, and Blue components of an RGB Long value. "Closest" is taken in the geometrical sense, the distance between two colors in a 3-dimensional space with axes of Red, Green, and Blue values, that is, a color is identified spatially by the values of the Red, Green, and Blue components. The distances between the spatial location of RGBLong and each Color of the pallet is computed and the ColorIndex that minimizes this distance is returned. The distance between RGBLong and each Color(ColorIndex) value is computed by the simple Pythagorean distance (without Square root):

$$\text{Dist} = (\text{R1-R2})^2 + (\text{G1-G2})^2 + (\text{B1-B2})^2$$

where R1, G1, and B1 are the components of RGBLong and R2, G2, and B2 are the components of each Color(ColorIndex) value.

Pattern

Excel provides various pattern styles for highlighting cells. These can be applied through the **Pattern** tab in the **Format Cells** dialog box.

XlsIO includes APIs to specify the above **background** pattern for a cell. The following code example illustrates this.

```

[C#]

// Setting the Pattern Types.
sheet.Range["A2"].CellStyle.FillPattern = ExcelPattern.Angle;
sheet.Range["A4"].CellStyle.FillPattern = ExcelPattern.DarkDownwardDiagonal;

// Setting the Pattern Color.
sheet.Range["A2"].CellStyle.FillBackground = ExcelKnownColors.Aqua;
sheet.Range["A4"].CellStyle.FillBackground = ExcelKnownColors.Pale_blue;

```

[VB.NET]

```
' Setting the Pattern Types.  
sheet.Range("A2").CellStyle.FillPattern = ExcelPattern.Angle  
sheet.Range("A4").CellStyle.FillPattern = ExcelPattern.DarkDownwardDiagonal  
  
' Setting the Pattern Color.  
sheet.Range("A2").CellStyle.FillBackground = ExcelKnownColors.Aqua  
sheet.Range("A4").CellStyle.FillBackground = ExcelKnownColors.Pale_blue
```

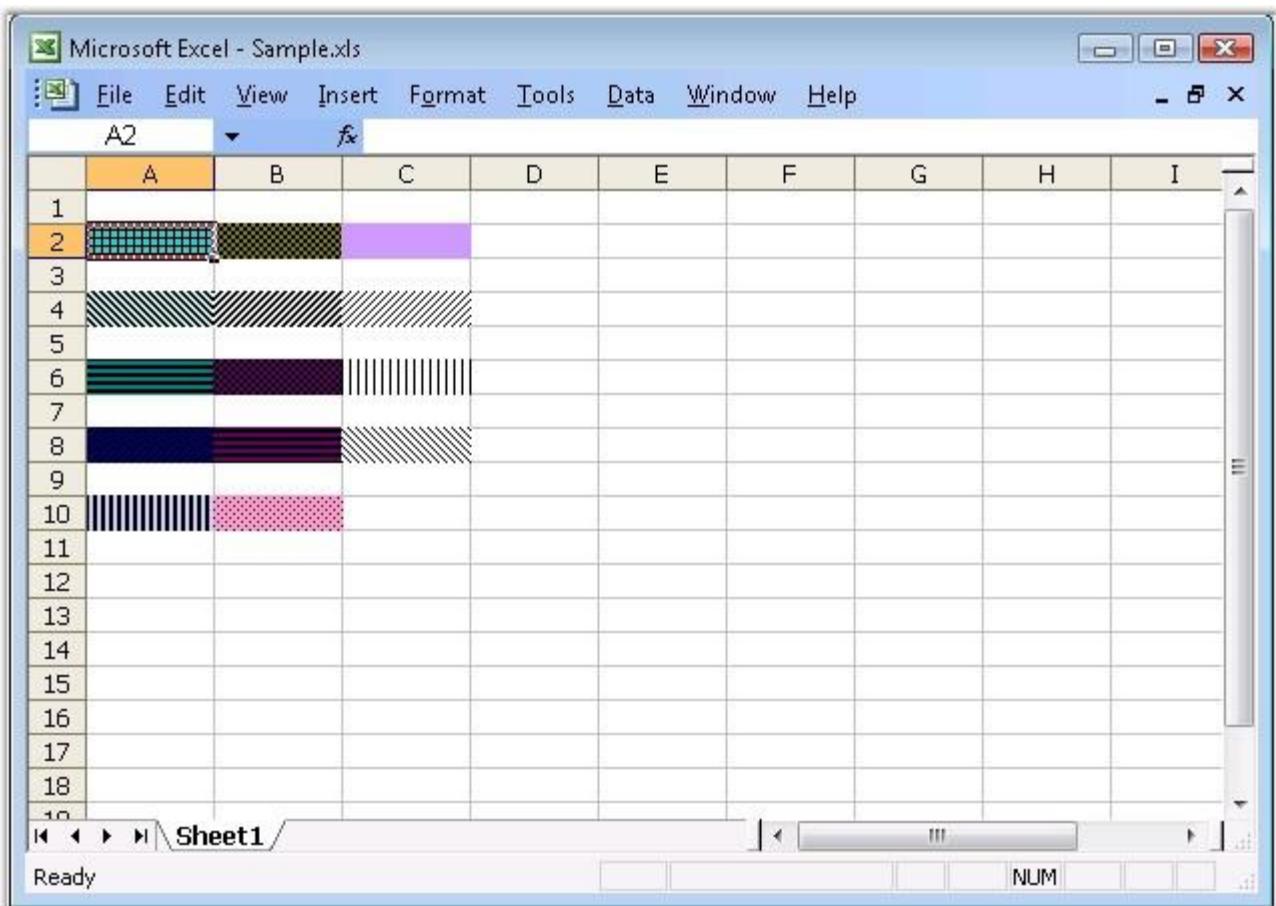


Figure 43: Excel with Different Fill Patterns

4.1.1.6 Styles

Microsoft Excel provides a faster and customized technique to apply a particular user-created format to a series of cells.

If a sample set of formatting is applied a number of times for cells in a worksheet, a style can be created and saved with the workbook, and used whenever we format information with the same attributes. Excel provides options to create, edit and modify the styles.

This section explains how XlsIO enables users to create and use such styles with the following categories as in MS Excel.

- Cell Styles-This section explains the creation of various styles by using XlsIO.
- Conditional Formatting-This section explains about various conditional formatting techniques implemented by using XlsIO.

4.1.1.6.1 Cell Styles

Microsoft Excel provides users, the ability to create and apply styles to cells, by accruing a few benefits. First, it gives the users a way to create a consistent-looking document, without the need to do plenty of direct formatting. Second, it gives the users the ability to quickly change the formatting of all cells that use a particular style.

This section explains various styles created by using XlsIO. Following are the styles discussed in this section.

4.1.1.6.1.1 Default Styles

Microsoft Excel provides support to create styles by using the **Style** dialog box (Go to the **Format** menu and click **Styles** command). It also permits to modify and add new styles, which can be applied to a range of cells.

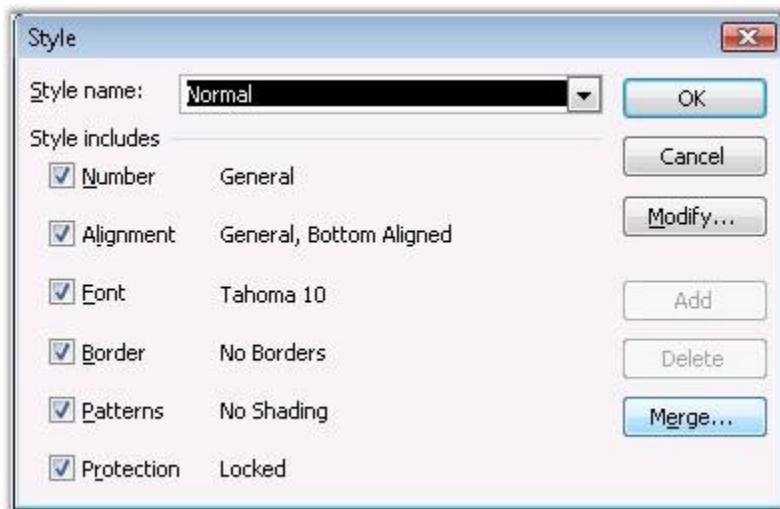


Figure 44: Style Dialog Box

Applying Default style in XlsIO

XlsIO provides various ways to apply styles. **IStyle** interface is used for creating styles. You can set the default styles created with groups of styles to a range of rows and columns. This is the most optimized approach to format rows and columns with large number of cells with same styles.

Following code example illustrates how to create and apply default styles for a range of rows and columns.

[C#]

```
// Define the default styles that need to be applied to rows and columns.
IStyle rowStyle = workbook.Styles.Add("RowStyle");
rowStyle.Color = Color.LightCoral;
IStyle columnStyle = workbook.Styles.Add("ColumnStyle");
columnStyle.Color = Color.Orange;

//Set Column Default Style
sheet.SetDefaultRowStyle(1, 2, rowStyle);

//Set Column Default Style
sheet.SetDefaultColumnStyle(1, 2, columnStyle);
```

[VB .NET]

```
'Define the default styles that need to be applied to rows and columns
Dim rowStyle As IStyle = workbook.Styles.Add("RowStyle")
rowStyle.Color = Color.LightCoral
Dim columnStyle As IStyle = workbook.Styles.Add("ColumnStyle")
columnStyle.Color = Color.Orange

'Set Column Default Style
sheet.SetDefaultRowStyle(1, 2, rowStyle)

'Set Column Default Style
sheet.SetDefaultColumnStyle(1, 2, columnStyle)
```



Note: Applying custom styles will override the original styles.

See Also

[Global Styles](#)

4.1.1.6.1.2 Global Styles

XlsIO provides support for adding and modifying common (or global) styles that can be applied to one or more cells in a workbook. These styles can be created and applied to several ranges of cells in the workbook. Note that the usage of common styles to format spreadsheets is the recommended approach, since setting a separate style for each cell can reduce the performance considerably.

 **Note:** If you want to apply more than one style for cells, enclose the style within the Begin and End calls. This will improve the performance.

```
[C#]

// Formatting

// Global styles should be used when the same style needs to be applied to
more than
// one cell. This usage of a global style reduces memory usage.

// Header Style
IStyle headerStyle = workbook.Styles.Add("HeaderStyle");

// Add custom colors to the palette.
headerStyle.BeginUpdate();
workbook.SetPaletteColor(8, Color.FromArgb(255, 174, 33));
headerStyle.Color = Color.FromArgb(255, 174, 33);
headerStyle.Font.Bold = true;
headerStyle.Borders[ExcelBordersIndex.EdgeLeft].LineStyle =
ExcelLineStyle.Thin;
headerStyle.Borders[ExcelBordersIndex.EdgeRight].LineStyle =
ExcelLineStyle.Thin;
headerStyle.Borders[ExcelBordersIndex.EdgeTop].LineStyle =
ExcelLineStyle.Thin;
headerStyle.Borders[ExcelBordersIndex.EdgeBottom].LineStyle =
ExcelLineStyle.Thin;
headerStyle.EndUpdate();

// Body Style
IStyle bodyStyle = workbook.Styles.Add("BodyStyle");

// Add custom colors to the palette.
bodyStyle.BeginUpdate();
```

```

workbook.SetPaletteColor(9, Color.FromArgb(239, 243, 247));
bodyStyle.Color = Color.FromArgb(239, 243, 247);
bodyStyle.Borders[ExcelBordersIndex.EdgeLeft].LineStyle =
ExcelLineStyle.Thin;
bodyStyle.Borders[ExcelBordersIndex.EdgeRight].LineStyle =
ExcelLineStyle.Thin;
bodyStyle.EndUpdate();

// Apply the defined styles.
// Apply Body Style.
sheet.UsedRange.CellStyleName = "BodyStyle";

// Apply Header style.
sheet.Rows[0].CellStyleName = "HeaderStyle";

```

[VB.NET]

```

' Formatting

' Global styles should be used when the same style needs to be applied to
more than
' one cell. This usage of a global style reduces memory usage.

' Header Style
Dim headerStyle As IStyle = workbook.Styles.Add("Header Style")

' Add custom colors to the palette.
headerStyle.BeginUpdate()
workbook.SetPaletteColor(8,Color.FromArgb(255,174,33))
headerStyle.Color = Color.FromArgb(255,174,33)
headerStyle.Font.Bold = True
headerStyle.Borders(ExcelBordersIndex.EdgeLeft).LineStyle =
ExcelLineStyle.Thin
headerStyle.Borders(ExcelBordersIndex.EdgeRight).LineStyle =
ExcelLineStyle.Thin
headerStyle.Borders(ExcelBordersIndex.EdgeTop).LineStyle =
ExcelLineStyle.Thin
headerStyle.Borders(ExcelBordersIndex.EdgeBottom).LineStyle =
ExcelLineStyle.Thin
headerStyle.EndUpdate()

' Body Style
Dim bodyStyle As IStyle = workbook.Styles.Add("BodyStyle")

' Add custom colors to the palette.
bodyStyle.BeginUpdate()

```

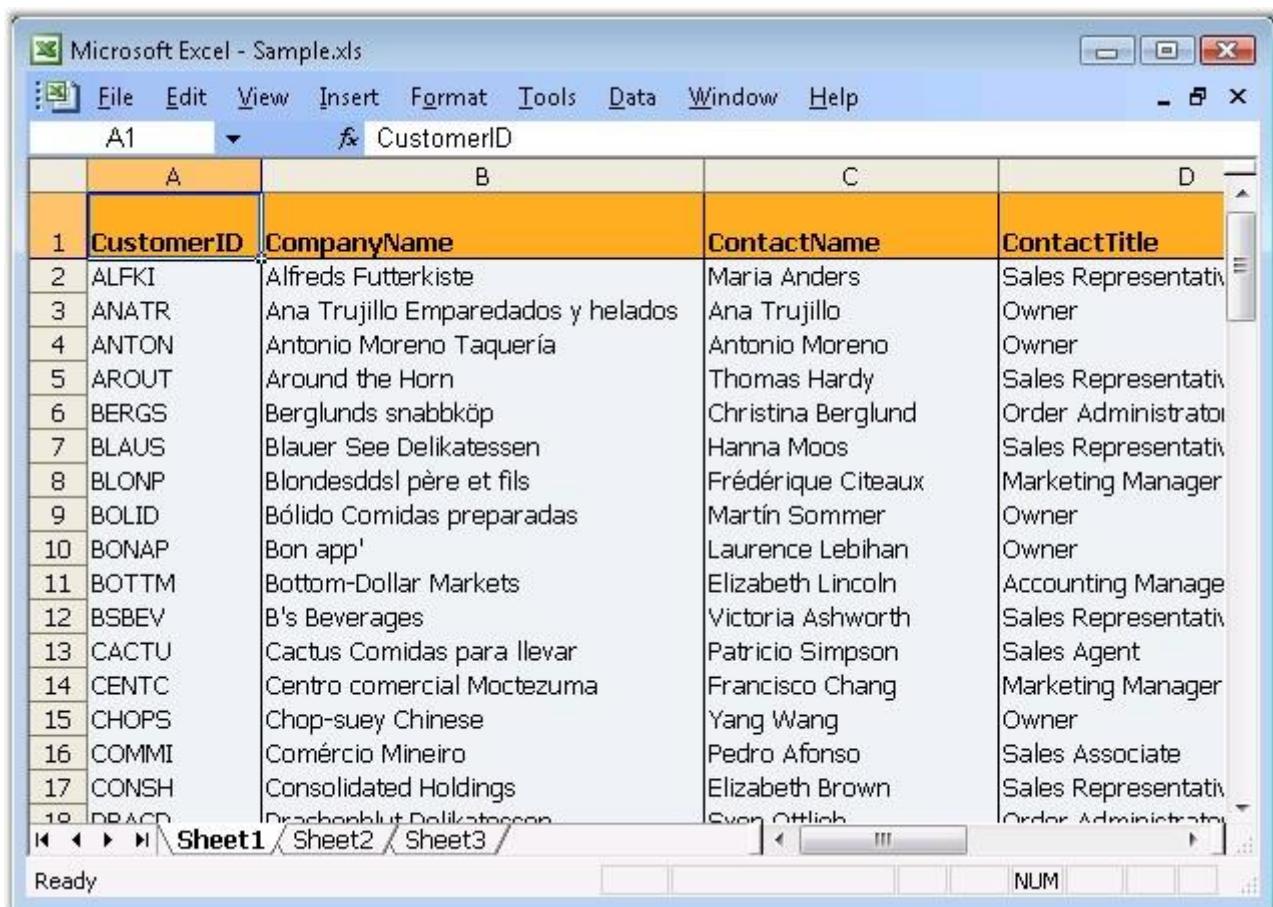
```

workbook.SetPaletteColor(9,Color.FromArgb(239,243,247))
bodyStyle.Color = Color.FromArgb(239,243,247)
bodyStyle.Borders(ExcelBordersIndex.EdgeLeft).LineStyle =
ExcelLineStyle.Thin
bodyStyle.Borders(ExcelBordersIndex.EdgeRight).LineStyle =
ExcelLineStyle.Thin
bodyStyle.EndUpdate()

' Apply the defined styles.
' Apply Body Style.
sheet.UsedRange.CellStyleName = "BodyStyle"

' Apply Header style.
sheet.Rows[0].CellStyleName = "HeaderStyle"

```



The screenshot shows a Microsoft Excel window titled "Microsoft Excel - Sample.xls". The window displays a table with 19 rows of data. The first row contains column headers: "CustomerID", "CompanyName", "ContactName", and "ContactTitle". The data rows show various company names like "Alfreds Futterkiste", "Ana Trujillo Emparedados y helados", etc., along with their contact details. The table has a light gray background, and the header row has a darker gray background. The columns are labeled A, B, C, and D at the top. The status bar at the bottom shows "Ready".

	A	B	C	D
1	CustomerID	CompanyName	ContactName	ContactTitle
2	ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative
3	ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner
4	ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner
5	AROUT	Around the Horn	Thomas Hardy	Sales Representative
6	BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator
7	BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative
8	BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager
9	BOLID	Bólido Comidas preparadas	Martin Sommer	Owner
10	BONAP	Bon app'	Laurence Lebihan	Owner
11	BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager
12	BSBEV	B's Beverages	Victoria Ashworth	Sales Representative
13	CACTU	Cactus Comidas para llevar	Patricia Simpson	Sales Agent
14	CENTC	Centro comercial Moctezuma	Francisco Chang	Marketing Manager
15	CHOPS	Chop-suey Chinese	Yang Wang	Owner
16	COMMI	Comércio Mineiro	Pedro Afonso	Sales Associate
17	CONSH	Consolidated Holdings	Elizabeth Brown	Sales Representative
18	DRACD	Dirección de Desarrollo	Susan Ottlisch	Order Administrator

Figure 45: XlsIO with Global Styles

For More Information Refer:

[Default Styles](#)**4.1.1.6.2 Conditional Formatting**

Conditional formatting is a feature by which the contents of a cell are dynamically formatted, based on a value. It allows you to define and apply formatting to some cells, text, and numbers, based on the criteria that is set. For example, you can format a time sheet, to point out the overtime obtained by the employee. You can also use it to track the best sales employees in a company, by setting a quota that makes a cell range particular.

In MS Excel, click the **Format** menu and then click **Conditional Formatting**. You can use any criteria of your choice. The formatting can be applied to cells' values or a particular formula.

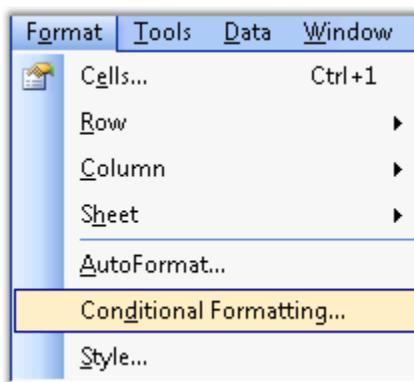


Figure 46: Conditional Formatting option displayed in Format Menu

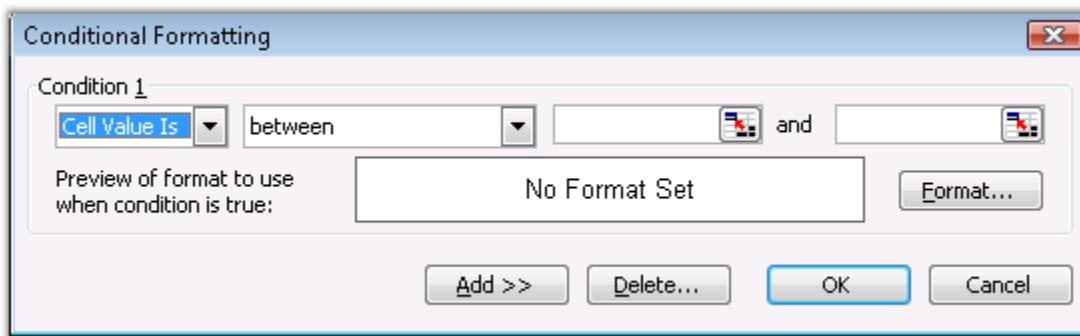


Figure 47: Using Conditional Formatting Feature in MS Excel

Note: Excel allows the addition of a maximum of three conditions for the same cell in the Biff8 format. However, this restriction is removed in Excel 2007 formats.

Conditional Formatting in XlsIO

XlsIO allows you to create conditional formats by using **IConditionFormats**; various conditions can be set by using its properties. XlsIO also provides support for applying more than three conditional formats in the same cell in the **.xlsx** format.

The following code creates and applies various conditional formats in XlsIO.

```
[C#]

// Applying conditional formatting to "A1" for format type as
CellValue(Between).
IConditionalFormats condition = sheet.Range["A1"].ConditionalFormats;

// Adding formats to IConditionalFormats collection.
IConditionalFormat condition1 = condition.AddCondition();
sheet.Range["A1"].Text = "Enter a Number between 10 to 20";
condition1.FirstFormula = "10";
condition1.SecondFormula = "20";

// Setting format properties.
condition1.Operator = ExcelComparisonOperator.Between;
condition1.FormatType = ExcelCFType.CellValue;
condition1.BackColor = ExcelKnownColors.Light_Orange;
condition1.IsBold = true;
condition1.IsItalic = true;

// Applying conditional formatting to "A3" for format type as
CellValue(Equal).
IConditionalFormats condition2 = sheet.Range["A3"].ConditionalFormats;

// Adding formats to the IConditionalFormats collection.
IConditionalFormat condition3 = condition2.AddCondition();
sheet.Range["A3"].Text = "Enter the Number as 1000";

// Setting format properties.
condition3.FormatType = ExcelCFType.CellValue;
condition3.Operator = ExcelComparisonOperator.Equal;
condition3.FirstFormula = "1000";
condition3.FontColor = ExcelKnownColors.Magenta;

// Applying conditional formatting to "A5" for format type as CellValue (Not
between).
IConditionalFormats condition4 = sheet.Range["A5"].ConditionalFormats;

// Adding formats to the IConditionalFormats collection.
IConditionalFormat condition5 = condition4.AddCondition();
sheet.Range["A5"].Text = "Enter a Number not between 100 to 200";
```

```

// Setting format properties.
condition5.FormatType = ExcelCFType.CellValue;
condition5.Operator = ExcelComparisonOperator.NotBetween;
condition5.FirstFormula = "100";
condition5.SecondFormula = "200";
condition5.FillPattern = ExcelPattern.DarkVertical;

// Applying conditional formatting to "A7" for format type as
// CellValue(LessOrEqual).
IConditionalFormats condition6 = sheet.Range["A7"].ConditionalFormats;

//Adding formats to IConditionalFormats collection
IConditionalFormat condition7 = condition6.AddCondition();
sheet.Range["A7"].Text = "Enter a Number which is less than or equal to
1000";

// Setting format properties.
condition7.FormatType = ExcelCFType.CellValue;
condition7.Operator = ExcelComparisonOperator.LessOrEqual;
condition7.FirstFormula = "1000";
condition7.BackColor = ExcelKnownColors.Light_green;

// Applying conditional formatting to "A9" for format type as
// CellValue(NotEqual).
IConditionalFormats condition8 = sheet.Range["A9"].ConditionalFormats;

// Adding formats to the IConditionalFormats collection.
IConditionalFormat condition9 = condition8.AddCondition();
sheet.Range["A9"].Text = "Enter a Number which is not equal to 1000";

// Setting format properties.
condition9.FormatType = ExcelCFType.CellValue;
condition9.Operator = ExcelComparisonOperator.NotEqual;
condition9.FirstFormula = "1000";
condition9.BackColor = ExcelKnownColors.Lime;

```

[VB.NET]

```

' Applying conditional formatting to "A1" for format type as
// CellValue(Between).
Dim condition As IConditionalFormats = sheet.Range("A1").ConditionalFormats

' Adding formats to the IConditionalFormats collection.
Dim condition1 As IConditionalFormat = condition.AddCondition()
sheet.Range("A1").Text = "Enter a Number between 10 to 20"

```

```

condition1.FirstFormula = "10"
condition1.SecondFormula = "20"

' Setting format properties.
condition1.Operator = ExcelComparisonOperator.Between
condition1.FormatType = ExcelCFType.CellValue
condition1.BackColor = ExcelKnownColors.Light_Orange
condition1.IsBold = True
condition1.IsItalic = True

' Applying conditional formatting to "A3" for format type as
CellValue(Equal).
Dim condition2 As IConditionalFormats = sheet.Range("A3").ConditionalFormats

' Adding formats to the IConditionalFormats collection.
Dim condition3 As IConditionalFormat = condition2.AddCondition()
sheet.Range("A3").Text = "Enter the Number as 1000"

' Setting format properties.
condition3.FormatType = ExcelCFType.CellValue
condition3.Operator = ExcelComparisonOperator.Equal
condition3.FirstFormula = "1000"
condition3.FontColor = ExcelKnownColors.Magenta

' Applying conditional formatting to "A5" for format type as CellValue(Not
between).
Dim condition4 As IConditionalFormats = sheet.Range("A5").ConditionalFormats

' Adding formats to the IConditionalFormats collection.
Dim condition5 As IConditionalFormat = condition4.AddCondition()
sheet.Range("A5").Text = "Enter a Number not between 100 to 200"

' Setting format properties.
condition5.FormatType = ExcelCFType.CellValue
condition5.Operator = ExcelComparisonOperator.NotBetween
condition5.FirstFormula = "100"
condition5.SecondFormula = "200"
condition5.FillPattern = ExcelPattern.DarkVertical

' Applying conditional formatting to "A7" for format type as
CellValue(LessOrEqual).
Dim condition6 As IConditionalFormats = sheet.Range("A7").ConditionalFormats

' Adding formats to the IConditionalFormats collection.
Dim condition7 As IConditionalFormat = condition6.AddCondition()
sheet.Range("A7").Text = "Enter a Number which is less than or equal to"

```

```
1000"

' Setting format properties.
condition7.FormatType = ExcelCFType.CellValue
condition7.Operator = ExcelComparisonOperator.LessOrEqual
condition7.FirstFormula = "1000"
condition7.BackColor = ExcelKnownColors.Light_green

' Applying conditional formatting to "A9" for format type as
CellValue(NotEqual).
Dim condition8 As IConditionalFormats = sheet.Range("A9").ConditionalFormats

' Adding formats to the IConditionalFormats collection.
Dim condition9 As IConditionalFormat = condition8.AddCondition()
sheet.Range("A9").Text = "Enter a Number which is not equal to 1000"

' Setting format properties.
condition9.FormatType = ExcelCFType.CellValue
condition9.Operator = ExcelComparisonOperator.NotEqual
condition9.FirstFormula = "1000"
condition9.BackColor = ExcelKnownColors.Lime
```

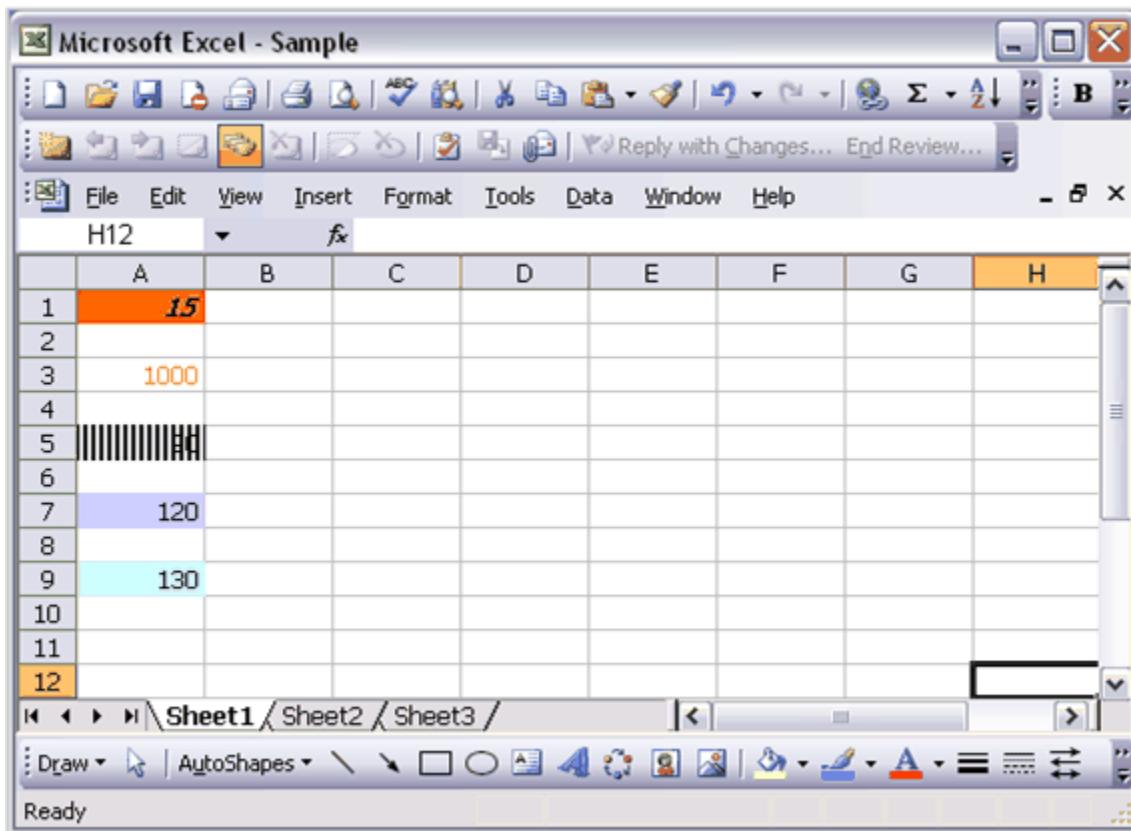


Figure 48: XlsIO with Conditional Formatting

Reading Conditional Formats in XlsIO

XlsIO also provides support for reading conditional formats. Following code example illustrates this.

[C#]

```
// Read conditional formatting settings.
this.textBox1.Text =
sheet.Range["A1"].ConditionalFormats[0].FormatType.ToString();
this.textBox2.Text =
sheet.Range["A1"].ConditionalFormats[0].Operator.ToString();
this.textBox3.Text =
sheet.Range["A1"].ConditionalFormats[0].BackColor.ToString();
```

[VB .NET]

```
' Read conditional formatting settings.
Me.textBox1.Text =
sheet.Range("A1").ConditionalFormats(0).FormatType.ToString()
Me.textBox2.Text =
sheet.Range("A1").ConditionalFormats(0).Operator.ToString()
Me.textBox3.Text =
sheet.Range("A1").ConditionalFormats(0).BackColor.ToString()
```

Removing Conditional Formats in MS-Excel

With Microsoft Excel, you can delete conditional formats from the selected cells or from the entire sheet.

To remove the conditional formats in MS-Excel:

1. Select the cell that has the conditional format that you want to delete.
2. Click **Conditional Formatting** on **Home** tab, and then select **Clear Rules**. The Clear Rules property can be applied to the selected cells or to the entire sheet.

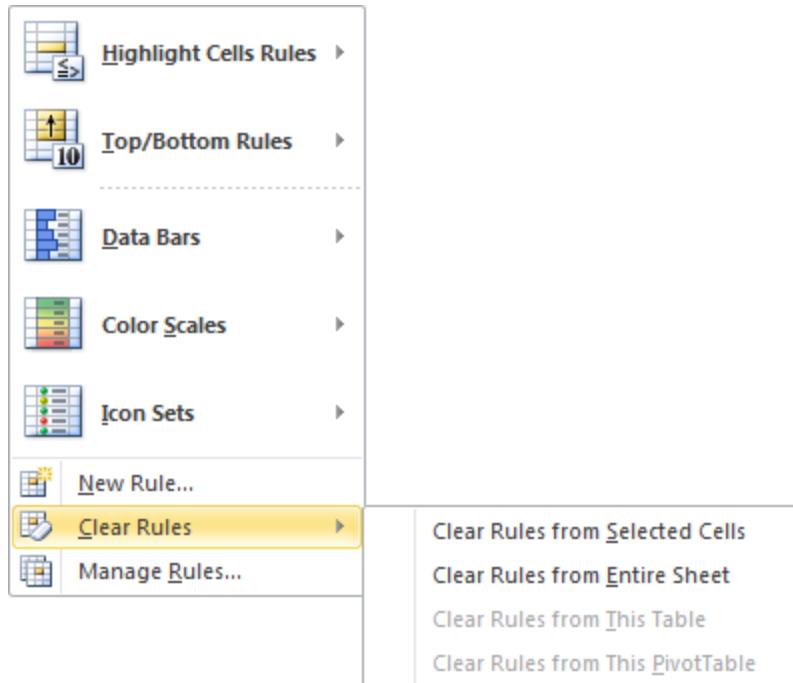


Figure 49: Removing Conditional Format by using Excel

Removing Conditional Formats in XlsIO

XlsIO also provides support for removing the conditional formats. Following are the methods for removing the conditional formats associated with the **IConditionalFormat** interface.

- **Remove**
- **RemoveAt**

Removing Conditional Formats at specified range

XlsIO removes the conditional formats at specified range by using **Remove Method**.

The following code example illustrates this.

[C#]

```
// Removing conditional format at the specified range.  
sheet.Range["E5"].ConditionalFormats.Remove();
```

[VB .NET]

```
' Removing conditional format at the specified range.  
sheet.Range["E5"].ConditionalFormats.Remove()
```

Removing Conditional Formats at specified index value

XlsIO removes the conditional formats at specified index value by using **RemoveAt** Method

Following code example illustrates this.

[C#]

```
// Removing Conditional Format at the specified Range  
sheet.Range["E5"].ConditionalFormats.RemoveAt(0);
```

[VB .NET]

```
' Removing Conditional Format at the specified Range  
sheet.Range["E5"].ConditionalFormats.RemoveAt(0)
```

Removing Conditional Formats from entire sheet

XlsIO also provides support for removing conditional formats from the entire sheet. Following code example illustrates this.

[C#]

```
// Removing Conditional Formatting Settings From Entire Sheet.  
sheet.UsedRange.Clear(ExcelClearOptions.ClearConditionalFormats);
```

[VB .NET]

```
'Removing Conditional Formatting Settings From Entire Sheet.

sheet.UsedRange.Clear(ExcelClearOptions.ClearConditionalFormats)
```

Using FormulaR1C1 property in Conditional Formats

XlsIO returns or sets the formula for the conditional format by using R1C1-style notation. Following code example illustrates this.

[C#]

```
// Using FormulaR1C1 property in Conditional Formatting

IConditionalFormats condition =
worksheet.Range["E5:E18"].ConditionalFormats;

IConditionalFormat condition1 = condition.AddCondition();

condition1.FirstFormulaR1C1 = "=R[1]C[0]";
condition1.SecondFormulaR1C1 = "=R[1]C[1]" ;
```

[VB.NET]

```
' Using FormulaR1C1 property in Conditional Formatting

Dim condition As IConditionalFormats =
sheet.Range("E5:E18").ConditionalFormats

Dim condition1 As IConditionalFormat = condition.AddCondition()

condition1.FirstFormulaR1C1 = "=R[1]C[0]"
condition1.SecondFormulaR1C1 = "=R[1]C[1]"
```

[Advanced Conditional Formatting](#)

See Also

[Cell Styles](#)

4.1.1.6.2.1 Advanced Conditional Formatting

Excel 2007 introduces a new formatting that highlights cells once it meets respective constraints. Three new visualizations such as **Data Bars**, **Color Scales**, and **Icon Sets** help you to explore large datasets, identify trends and exceptions, and quickly compare data.

Data Bars

Data Bars give you an opportunity to create visual effects in your data that help you see how the value of a cell compares with other cells.

Excel compares the values in each of the selected cells, and draws a data bar in each cell representing the value of that cell relative to the other cells in the selected range. This bar provides a clear visual cue for users, making it easier to pick out larger and smaller values in a range.

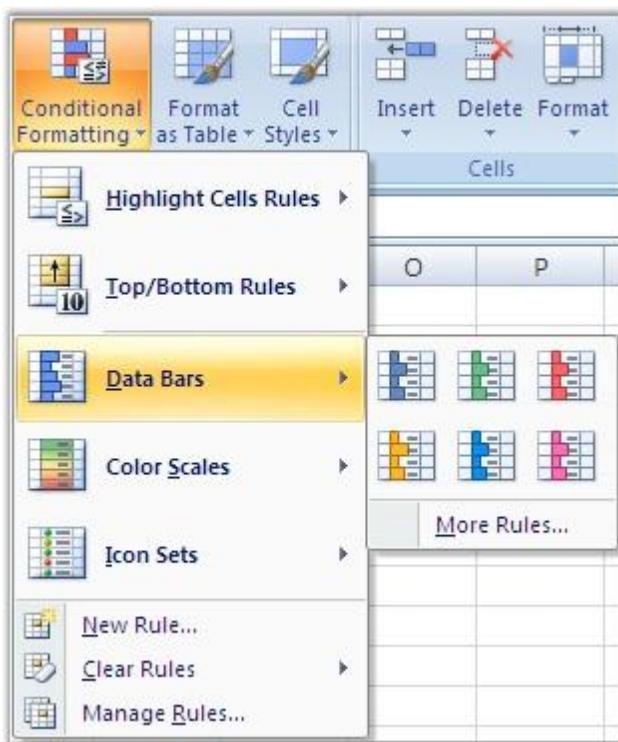


Figure 50: Data Bars

MS Excel enables to set these formats through the **Conditional Formatting** menu. It also allows to set the criteria through the **New Formatting Rule** dialog box shown below.

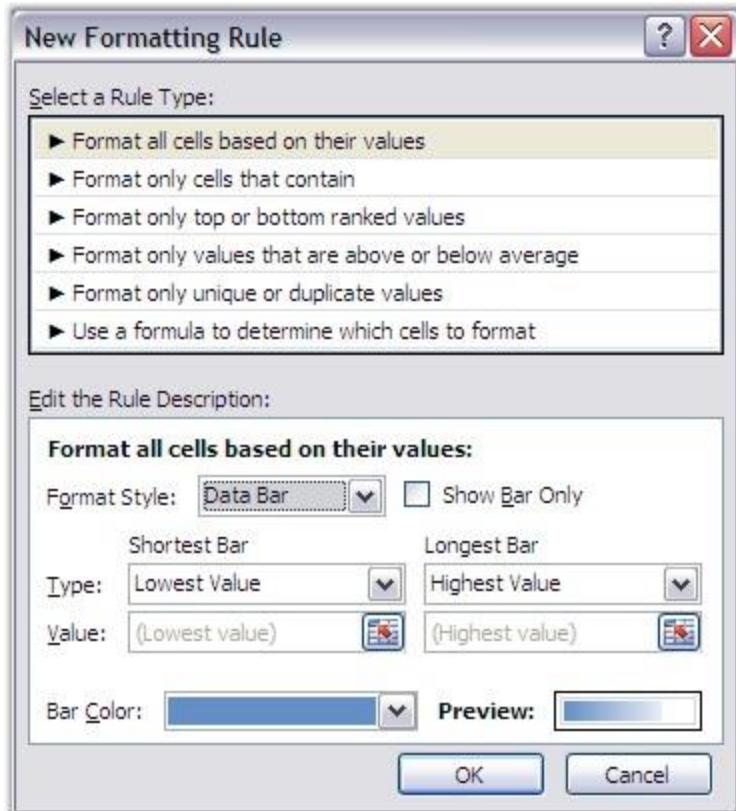


Figure 51: New Formatting Rule Dialog for setting Data Bar

Color Scales

Color Scales let you create visual effects in your data, to see how the value of a cell compares with the values in a range of cells. A color scale uses cell shading, as opposed to bars, to communicate relative values. This is especially useful when you want to communicate more about your data, beyond the relative size of the value of a cell.



Figure 52: Color Scales

You can customize the criteria through the New Formatting Rule dialog box in MS Excel.

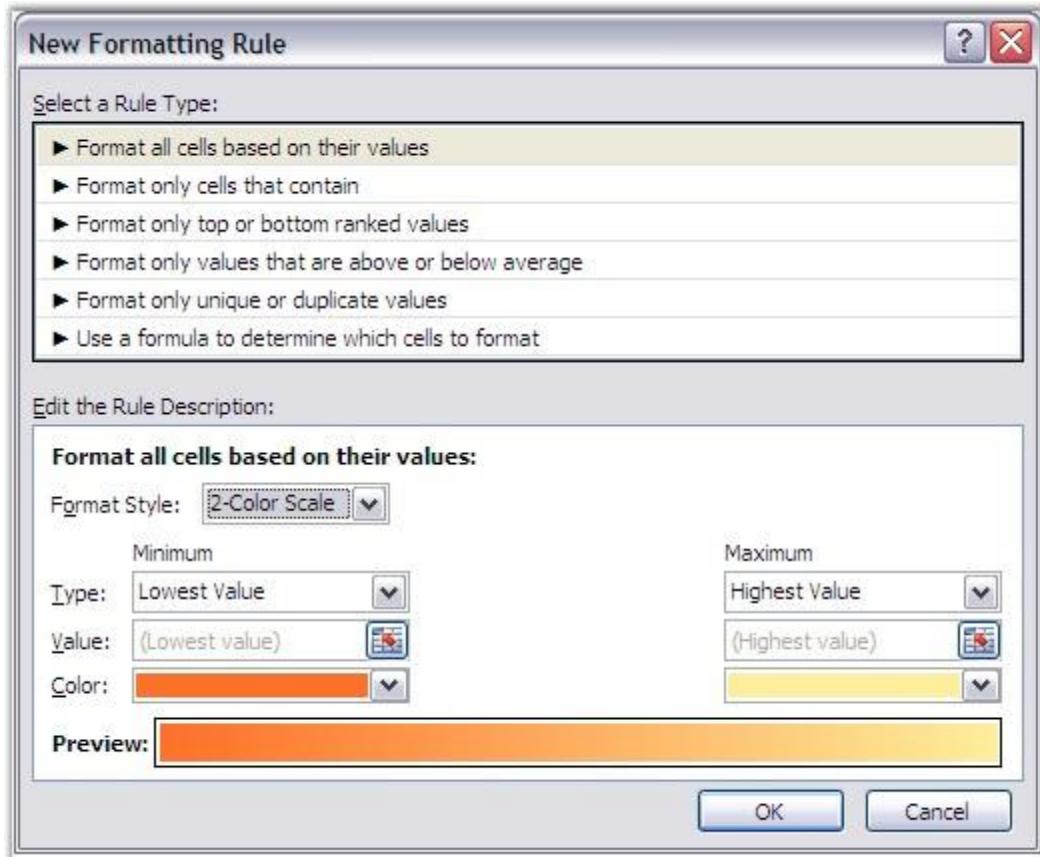


Figure 53: New Formatting Dialog Box for setting Color Scales

Icon Sets

Icon Sets give you an opportunity to create visual effects in your data, to see how the value of a cell compares with other cells. Excel 2007 offers several choices of icon sets. You can choose the icons that are most appropriate for the data you are using. Icon sets come in three sizes, so as you increase or decrease the font size, the icon becomes larger or smaller, appropriately.

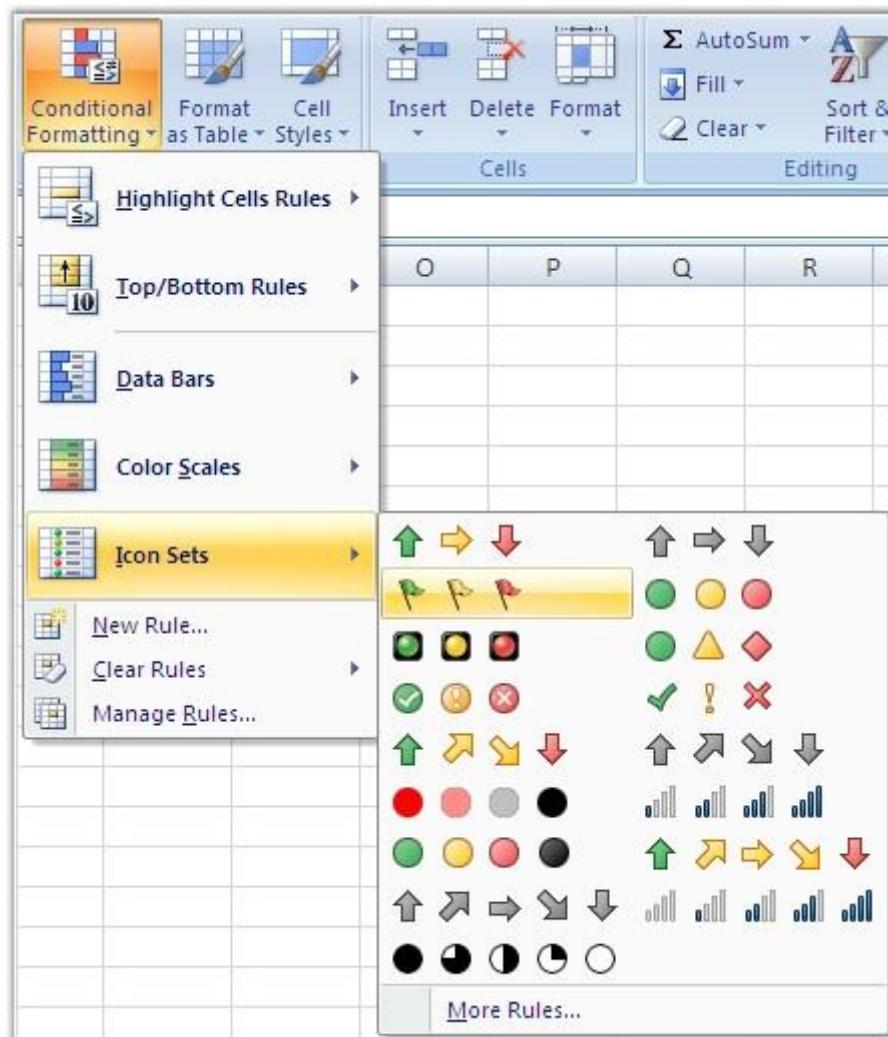


Figure 54: Icon Sets

It is possible to hide the value of the cell and just draw the icon, while applying a conditional formatting rule for icon sets, by using the **New Formatting Rule** dialog box.

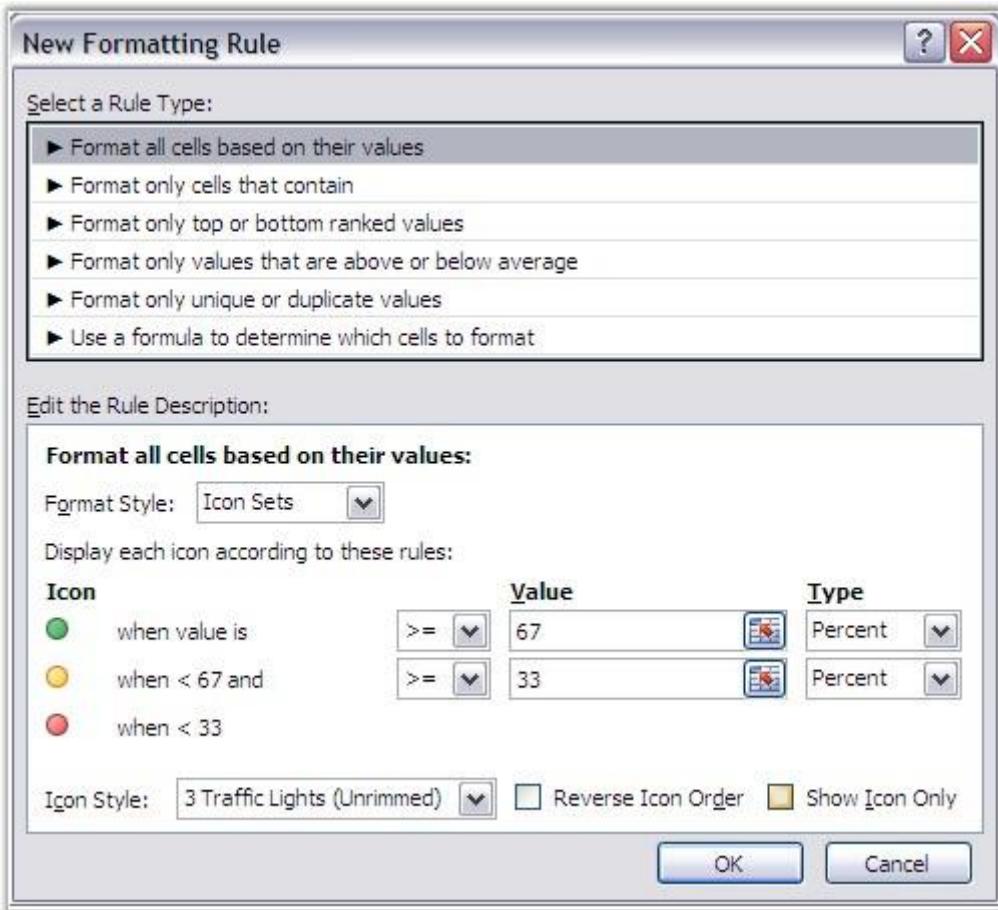


Figure 55: New Formatting Rule for setting Icon Sets

Visualizations in XlsIO

XlsIO also provides support for these visualizations through **IDataBar**, **IconSet** and **IColorScale** interfaces. You can also customize these visualizations by specifying the criteria, by using XlsIO. Following code example illustrates how to apply and customize various visualizations.

```
[C#]

//Add condition for the range
IConditionalFormats conditionalFormats =
worksheet.Range["C7:C46"].ConditionalFormats;
IConditionalFormat conditionalFormat = conditionalFormats.AddCondition();
```

```
//Set Data bar and icon set for the same cell
//Set the conditionalFormat type
conditionalFormat.FormatType = ExcelCFType.DataBar;
IDataBar dataBar = conditionalFormat.DataBar;

//Set the constraint
dataBar.MinPoint.Type = ConditionValueType.LowestValue;
dataBar.MinPoint.Value = "0";
dataBar.MaxPoint.Type = ConditionValueType.HighestValue;
dataBar.MaxPoint.Value = "0";

//Set color for Bar
dataBar.BarColor = Color.FromArgb(156, 208, 243);

//Hide the value in data bar
dataBar.ShowValue = false;
dataBar.MaxPoint = new ConditionValue(ConditionValueType.HighestValue,
"0");
dataBar.BarColor = Color.Red;
dataBar.ShowValue = false;

//Add another condition in the same range
conditionalFormat = conditionalFormats.AddCondition();

//Set Icon conditionalFormat type
conditionalFormat.FormatType = ExcelCFType.IconSet;
IIconSet iconSet = conditionalFormat.IconSet;
iconSet.IconSet = ExcelIconsetType.FourRating;
iconSet.IconCriteria[0].Type = ConditionValueType.LowestValue;
iconSet.IconCriteria[0].Value = "0";
iconSet.IconCriteria[1].Type = ConditionValueType.HighestValue;
iconSet.IconCriteria[1].Value = "0";
iconSet.ShowIconOnly = true;
```

```
//Sets Icon sets for another range
conditionalFormats = worksheet.Range["E7:E46"].ConditionalFormats;
conditionalFormat = conditionalFormats.AddCondition();
conditionalFormat.FormatType = ExcelCFType.IconSet;
iconSet = conditionalFormat.IconSet;
iconSet.IconSet = ExcelIconsetType.ThreeSymbols;
iconSet.IconCriteria[0].Type = ConditionValueType.LowestValue;
iconSet.IconCriteria[0].Value = "0";
iconSet.IconCriteria[1].Type = ConditionValueType.HighestValue;
iconSet.IconCriteria[1].Value = "0";
iconSet.ShowIconOnly = true;

//Sets Color scale conditional format type
conditionalFormats = worksheet.Range["D7:D46"].ConditionalFormats;
conditionalFormat = conditionalFormats.AddCondition();
conditionalFormat.FormatType = ExcelCFType.ColorScale;
IColorScale colorScale = conditionalFormat.ColorScale;

//Sets 3 - color scale.
colorScale.SetConditionCount(3);

colorScale.Criteria[0].FormatColorRGB = Color.FromArgb(230, 197, 218);
colorScale.Criteria[0].Type = ConditionValueType.LowestValue;
colorScale.Criteria[0].Value = "0";

colorScale.Criteria[1].FormatColorRGB = Color.FromArgb(244, 210, 178);
colorScale.Criteria[1].Type = ConditionValueType.Percentile;
colorScale.Criteria[1].Value = "50";

colorScale.Criteria[2].FormatColorRGB = Color.FromArgb(245, 247, 171);
colorScale.Criteria[2].Type = ConditionValueType.HighestValue;
colorScale.Criteria[2].Value = "0";
```

[VB .NET]

```
'Add condition for the range
Dim formats As IConditionalFormats = sheet.Range("C7:C46").ConditionalFormats
Dim format As IConditionalFormat = formats.AddCondition()

'Set Data bar and icon set for the same cell
'Set the format type
format.FormatType = ExcelCFType.DataBar
Dim dataBar As IDataBar = format.DataBar

'Set the constraint
dataBar.MinPoint.Type = ConditionValueType.LowestValue
dataBar.MinPoint.Value = "0"
dataBar.MaxPoint.Type = ConditionValueType.HighestValue
dataBar.MaxPoint.Value = "0"

'Set color for Bar
dataBar.BarColor = System.Drawing.Color.FromArgb(156, 208, 243)

'Hide the value in data bar
dataBar.ShowValue = False

'Add another condition in the same range
format = formats.AddCondition()

'Set Icon format type
format.FormatType = ExcelCFType.IconSet
Dim iconSet As IIIconSet = format.IconSet
iconSet.IconSet = ExcelIconsetType.FourRating
iconSet.IconCriteria(0).Type = ConditionValueType.LowestValue
iconSet.IconCriteria(0).Value = "0"
iconSet.IconCriteria(1).Type = ConditionValueType.HighestValue
iconSet.IconCriteria(1).Value = "0"
```

```
iconSet.ShowIconOnly = True

'Sets Icon sets for another range
formats = sheet.Range("E7:E46").ConditionalFormats
format = formats.AddCondition()
format.FormatType = ExcelCFType.IconSet
iconSet = format.IconSet
iconSet.IconSetType = ExcelIconSetType.ThreeSymbols
iconSet.IconCriteria(0).Type = ConditionValueType.LowestValue
iconSet.IconCriteria(0).Value = "0"
iconSet.IconCriteria(1).Type = ConditionValueType.HighestValue
iconSet.IconCriteria(1).Value = "0"
iconSet.ShowIconOnly = True

'Sets Color Scale conditional format type
formats = sheet.Range("D7:D46").ConditionalFormats
format = formats.AddCondition()
format.FormatType = ExcelCFType.ColorScale
Dim colorScale As IColorScale = format.ColorScale

'Sets 3 - color scale.
colorScale.SetConditionCount(3)

colorScale.Criteria(0).FormatColorRGB = System.Drawing.Color.FromArgb(230, 197, 218)
colorScale.Criteria(0).Type = ConditionValueType.LowestValue
colorScale.Criteria(0).Value = "0"

colorScale.Criteria(1).FormatColorRGB = System.Drawing.Color.FromArgb(244, 210, 178)
colorScale.Criteria(1).Type = ConditionValueType.Percentile
colorScale.Criteria(1).Value = "50"

colorScale.Criteria(2).FormatColorRGB = System.Drawing.Color.FromArgb(245, 247, 171)
```

```
colorScale.Criteria(2).Type = ConditionValueType.HighestValue
colorScale.Criteria(2).Value = "0"
```

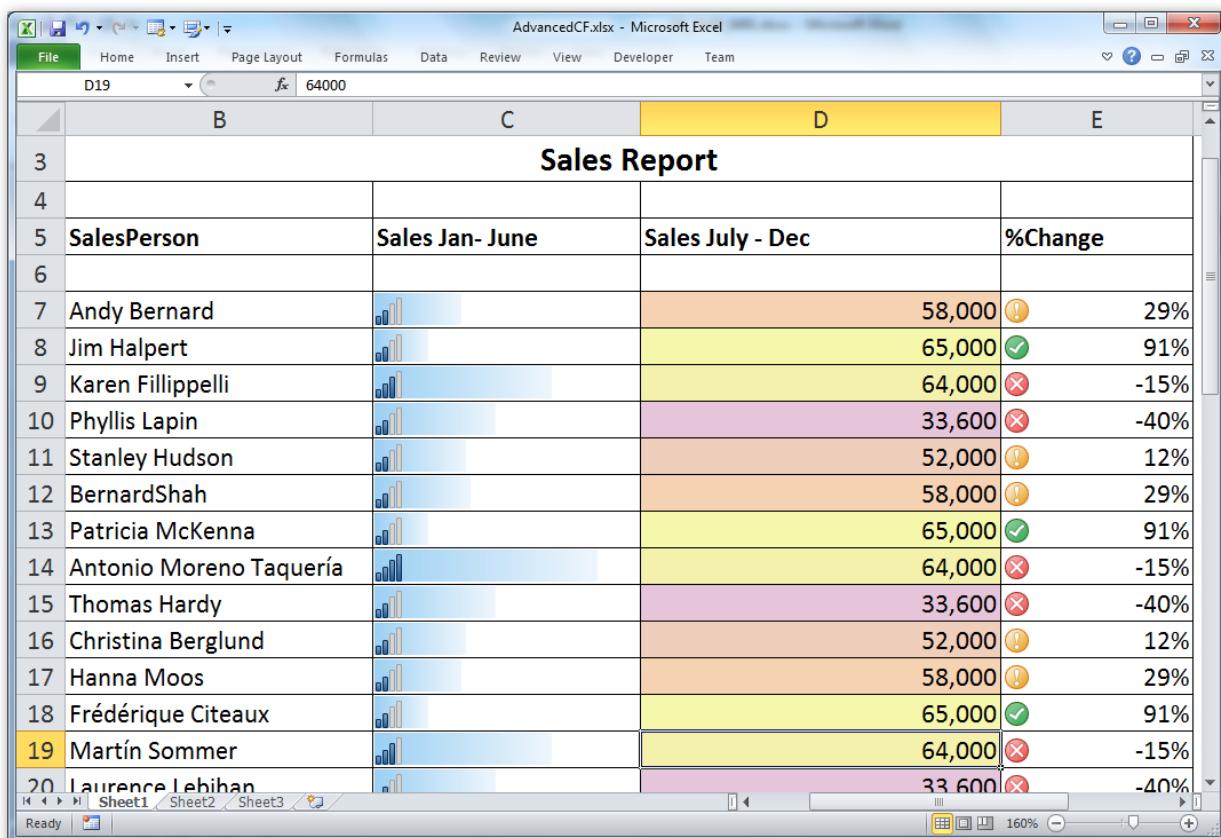


Figure 56: XlsIO Visualization

Note: XlsIO visualization has been enhanced with backward compatibility for Advanced Conditional Formatting.

4.1.2 Editing

MS Excel provides support to clear/move/copy/fill data in the cells, just by focusing the cell.

XlsIO allows to edit the contents of the cell(s) [say range] with simple and user friendly APIs. This section explains various editing capabilities of XlsIO in the following subsections.

4.1.2.1 Range Manipulation

The **IRange** interface represents a single cell or a group of cells in a worksheet. XlsIO has several useful methods for manipulating the data and formatting it in the ranges. This section discusses the topics listed below.

- Accessing a Range
- IMigrant Range
- Copying/Moving a Range
- Clearing a Range
- Getting Used Range

Accessing a Range

Range of cells can be accessed through the **IRange** interface. Following code example illustrates this.

```
// Get cell range.
IRange this[string name] { get; }

// Gets/sets cell by row and index.
IRange this[int row, int column] { get; }

// Get cell range.
IRange this[string name, bool IsR1C1Notation] { get; }

// Get cells range.
IRange this[int row, int column, int lastRow, int lastColumn] { get; }
```



Note: Here row and column indexes in the range are "one based". Following code example explains various ways of accessing cells.

[C#]

```
// Method 1 to Access a Range.
sheet.Range["A7"].Text = "Accessing a Range (Method 1)";

// Method 2 to Access a Range.
sheet.Range[9, 1].Text = "Accessing a Range (Method 2);
```

```
// Method 3 to Access a Range(using defined names).
sheet.Range["Name"].Text = "Accessing a Range (Method 3)";

// Accessing a Range of cells (Method 1).
sheet.Range["A13:C13"].Text = "Accessing a Range of Cells (Method 1)";

// Accessing a Range of cells (Method 2).
sheet.Range[15, 1, 15, 3].Text = "Accessing a Range of Cells (Method 2)";

// Accessing a Range of cells (Method 3 using defined names).
sheet.Range["Name1"].Text = "Accessing a Range of Cells (Method 3);
```

[VB.NET]

```
' Method 1 to Access a Range.
sheet.Range("A7").Text = "Accessing a Range (Method 1)"

' Method 2 to Access a Range.
sheet.Range(9, 1).Text = "Accessing a Range (Method 2)"

' Method 3 to Access a Range(using defined names).
sheet.Range("Name").Text = "Accessing a Range (Method 3)"

' Accessing a Range of cells (Method 1).
sheet.Range("A13:C13").Text = "Accessing a Range of Cells (Method 1)"

' Accessing a Range of cells (Method 2).
sheet.Range(15, 1, 15, 3).Text = "Accessing a Range of Cells (Method 2)"

' Accessing a Range of cells (Method 3 using defined names).
sheet.Range("Name1").Text = "Accessing a Range of Cells (Method 3)"
```

Accessing Discontinuous Ranges

You can also access different discontinuous ranges and add them to the **RangesCollection**, so that the same format is applied to different ranges. Following code example explains the same.

[C#]

```
IRange range1 = sheet.Range[ "A1:A2" ];
IRange range2 = sheet.Range["C1:C2" ];

RangesCollection ranges = new RangesCollection( engine.Excel, sheet );
```

```
ranges.Add( range1 );
ranges.Add( range2 );
ranges.Text = "Test";
```

[VB.NET]

```
Dim range1 As IRange = sheet.Range("A1:A2")
Dim range2 As IRange = sheet.Range("C1:C2")

Dim ranges As RangesCollection = New RangesCollection(engine.Excel, sheet)

ranges.Add(range1)
ranges.Add(range2)
ranges.Text = "Test"
```

IMigrantRange

The **IMigrantRange** interface can be used to access the worksheet range and manipulate it. This is an optimal method of writing strings with better memory performance. Following code example illustrates this.

[C#]

```
// Writing Data.
for (int row = 1; row <= rowCount; row++)
{
    for (int column = 1; column <= colCount; column++)
    {
        // Writing values.
        migrantRange.ResetRowColumn(row, column);
        migrantRange.Text = "Test";
    }
}
```

[VB.NET]

```
' Writing Data.
Dim row As Integer

For row = 1 To rowCount Step row + 1
Dim column As Integer
    For column = 1 To colCount Step column + 1
```

```
' Writing values.
    migrantRange.ResetRowColumn(row, column)
    migrantRange.Text = "Test"
    Next
Next
```

Copying/Moving Range

Moving and copying cells is a very common procedure when you are creating or editing your worksheets. XlsIO provides support to copy a range of cells from one end to another. **CopyTo** method enables copying range of cells from the source to destination. It has an option to copy all the formats or only specific formats to the destination range by using the **ExcelCopyRangeOptions** enumerator. Following values can be set for the ExcelCopyRangeOptions.

Member Name	Description
None	No flags.
UpdateFormulas	Indicates whether update formula during copy. WARNING: you should always specify this flag if your operations could change the position of Array formula.
UpdateMerges	Indicates whether update merges during copy.
CopyStyles	Indicates that we have to copy styles during range copy.
CopyShapes	Indicates that we have to copy shapes during range copy.
CopyErrorIndicators	Indicates that we have to copy error indicators during range copy.
CopyConditionalFormats	Indicates that we have to copy conditional formats during range copy.
All	All flags.

Following code example illustrates how to copy a range of cells from the source to destination preserving only cell styles.

```
[C#]

// Copying a Range.
Dim source As IRange = sheet.Range("A1")
Dim des As IRange = sheet.Range("A5")
source.CopyTo(des)
```

[VB.NET]

```
' Copying a Range.
Dim source As IRange = sheet.Range("A1")
Dim des As IRange = sheet.Range("A5")
source.CopyTo(des)
```

MoveTo method is used for moving a range of cells to the destination. The only difference between copy and move operation is that Move will not create a copy in the source. This is similar to the **Cut** and **Paste** option in Excel.



Note: *Move does not update formulas.*

[C#]

```
// Moving a Range.
Dim source As IRange = sheet.Range("A1")
Dim des As IRange = sheet.Range("A5")
source.MoveTo(des)
```

[VB.NET]

```
' Moving a Range.
Dim source As IRange = sheet.Range("A1")
Dim des As IRange = sheet.Range("A5")
source.MoveTo(des)
```

Clearing a Range

While editing Excel workbooks, one of the most common action that users perform is clearing or deleting cells. Clearing cells mean erasing everything within them, whereas deleting actually deletes the entire cell. You can clear the cell content by using the **Clear** method. XlsIO also provides options to clear styles or data alone.

Following code example illustrates how to clear a range along with its formatting.

[C#]

```
// Clearing a Range and its formatting.
sheet.Range["A4"].Clear(true);
```

[VB .NET]

```
' Clearing a Range and its formatting.
sheet.Range("A4").Clear(True)
```

Getting Used Range

XlsIO enables to get the range of cells used in a given sheet. This will help the user to apply the same format to all the cells used in the worksheet. You can also get the first row/column, last row/column, and number of rows/columns used in the sheet, by using the various methods of IRange.



Note: By default, XlsIO considers a cell as used, even if there exists some formatting. You can disable this behavior, and make XlsIO consider a cell as used, only when there exists data, by using the **UsedRangeIncludesFormatting** property.

Following code example is used to format the Used Range.

[C#]

```
// Modifying only the Used Ranges.
sheet.UsedRange.ColumnWidth = 20;
sheet.UsedRange.RowHeight = 20;
```

[VB .NET]

```
' Modifying only the Used Ranges.
sheet.UsedRange.ColumnWidth = 20
sheet.UsedRange.RowHeight = 20
```

4.1.2.2 Find and Replace

Find and Replace feature in Excel enables to navigate between large spreadsheets. It carries out a simultaneous search in Microsoft Excel values, formulas and also comments.

XlsIO also has support for finding and replacing contents in a worksheet. It has various options to find the first matching entry, find all the matching entries, and replace the found content with various data and data sources.

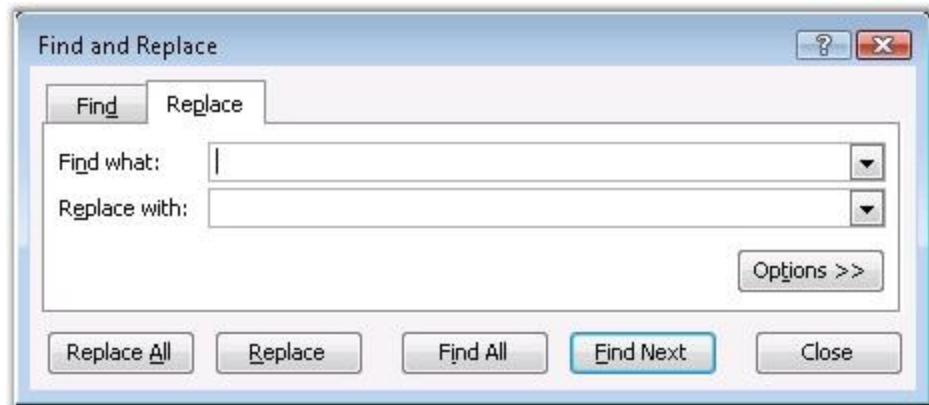


Figure 57: Find and Replace Dialog Box

XlsIO has the following common find and replace methods and properties, and their usage. This section describes all the methods listed below.

- FindFirst
- FindAll
- FindStringStartswith
- FindStringEndswith
- Replace

Following are the possible types of params of the **ExcelFindType** enumerator in the FindFirst and FindAll methods.

Member Name	Description
Text	Represents the Text Finding type.
Formula	Represents the Formula Finding type.
FormulaStringValue	Represents the FormulaStringValue Finding type.
Error	Represents the Error Finding type.
Number	Represents the Number Finding type.
FormulaValue	Represents the FormulaValue Finding type.

Following are the possible types of params of the **ExcelFindOptions** enumerator in the FindFirst and FindAll methods.

Member Name	Description
Match Case	Matches case while finding the value.
MatchEntireCell	Matches the whole word being searched while finding the value.

Find First

This method has overloads for searching the first cell with the specified typed value. The **ExcelFindType** enumerator provides options to set the data type of the string (i.e. value and formula value) to be searched, and the **ExcelFindOptions** enumerator provides the options to match the strings associated with the find value.

Methods	Description
FindFirst(Boolean)	This method searches for the cell with specified bool value.
FindFirst (DateTime)	This method searches for the cell with specified DateTime value.
FindFirst (TimeSpan)	This method searches for the cell with specified TimeSpan value.
FindFirst (Double, ExcelFindType)	This method searches for the cell with specified double value.
FindFirst (String, ExcelFindType, ExcelFindOptions)	This method searches for the cell with specified string value, again based on the Find options

[C#]

```
//FindFirst with Number
IRange result = sheet.FindFirst(1000000.00075, ExcelFindType.Number);

// Gets the cell display text
```

```

txtDisplay.Text = result.DisplayText.ToString();

//Find First with Match case
IRange result = sheet.FindFirst("Simple text", ExcelFindType.Text,
ExcelFindOptions.MatchCase);

// Gets the cell display text.
txtDisplay.Text = result.DisplayText.ToString();

//Find First with MatchEntireCellContent
IRange result = sheet.FindFirst("Simple text", ExcelFindType.Text,
ExcelFindOptions.MatchEntireCellContent);

// Gets the cell display text
txtDisplay.Text = result.DisplayText.ToString();

```

[VB]

```

'FindFirst with Number
Dim result As Range = sheet.FindFirst(1000000.00075,
ExcelFindType.Number)

'Gets the cell display text
txtDisplay.Text = result.DisplayText.ToString()

'Find First with Match case
Dim result As IRange = sheet.FindFirst("Simple text",
ExcelFindType.Text, ExcelFindOptions.MatchCase)

'Gets the cell display text.
txtDisplay.Text = result.DisplayText.ToString()

'Find First with MatchEntireCellContent
Dim result As IRange = sheet.FindFirst("Simple text",

```

```
ExcelFindType.Text, ExcelFindOptions.MatchEntireCellContent)

'Gets the cell display text.
txtDisplay.Text = result.DisplayText.ToString()
```

FindAll

This method searches all the cells, and returns all the entries in the sheet that matches the specified data.

Methods	Description
FindAll(Boolean)	This method searches for all the cells with specified bool value.
FindAll(DateTime)	This method searches for all the cells with specified DateTime value.
FindAll(TimeSpan)	This method searches for all the cells with specified TimeSpan value.
FindAll(Double, ExcelFindType)	This method searches for all the cells with specified double value.
FindAll (String, ExcelFindType, ExcelFindOptions)	This method searches for all the cells with specified string value, again based on the Find options

[C#]

```
//Find All with Text
IRange[] result =sheet .FindAll ("Simple Text",ExcelFindType.Text );

//Find All with Simple text and Match Case
IRange[] result = sheet.FindAll("Simple text", ExcelFindType.Text,
ExcelFindOptions.MatchCase);
```

```
//Find All with Simple Text and MatchEntireCellContent
IRange[] result = sheet.FindAll("Simple text", ExcelFindType.Text,
ExcelFindOptions.MatchEntireCellContent);
```

[VB]

```
'Find All with Text
Dim result() As IRange =sheet.FindAll ("Simple Text",ExcelFindType.Text)

'Find All with Simple text and Match Case
Dim result() As IRange = sheet.FindAll("Simple text", ExcelFindType.Text,
ExcelFindOptions.MatchCase)

'Find All with Simple Text and MatchEntireCellContent
Dim result() As IRange = sheet.FindAll("Simple text", ExcelFindType.Text,
ExcelFindOptions.MatchEntireCellContent)
```

FindStringStartswith

This method has overloads to search for the first cell that starts with the specified value. The **ExcelFindType** enumerator provides options to set the data type of the string (i.e. value and formula value) to be searched.

Methods	Description
FindStringStartsWith(String , ExcelFindType)	These methods search for cells that start with the specified string value for the given find type.
FindStringStartsWith(String, ExcelFindType,bool)	These methods searchfor cells which start with the specified string value, for the given find type and boolean value.

[C#]

```
//Starts with Simple Text
IRange result = result = sheet.FindStringStartsWith("Sim",
```

```

ExcelFindType.Text);

//StartsWith the Number with Text format
IRange result = sheet.FindStringStartsWith("$8", ExcelFindType.Text);

//Startswith the Text with MatchCase
IRange result = sheet.FindStringStartsWith("Si", ExcelFindType.Text, false);

```

[VB]

```

' Starts with Simple Text
Dim result As IRange = result = sheet.FindStringStartsWith("Sim",
ExcelFindType.Text)

'StartsWith the Number with Text format
Dim result As IRange = sheet.FindStringStartsWith("$8", ExcelFindType.Text)

'Startswith the Text with MatchCase
Dim result As IRange = sheet.FindStringStartsWith("Si", ExcelFindType.Text,
False)

```

FindStringEndswith

This method has overloads to search for cells that have the first cell ending with the specified typed value. **ExcelFindType** enumerator provides options to set the data type of the value and formula value/string to be searched.

Methods	Description
FindStringEndsWith (String, ExcelFindType)	These methods search for the cell which ends with the specified string value, for the given find type.
FindStringStartsWith (String, ExcelFindType,bool)	These methods search for the cell which ends with the specified string value, for the given find type and bool value.

[C#]

```
//Ends with Text
IRange result = result = sheet.FindStringEndsWith("Text",
ExcelFindType.Text);

//EndsWith the Number with Text format
IRange result = sheet.FindStringEndsWith("00", ExcelFindType.Text);

//Endswith the Text with MatchCase
IRange result = sheet.FindStringEndsWith("Case",ExcelFindType.Text, false);
```

[VB]

```
'Starts with Simple Text
Dim result As IRange = result = sheet.FindStringEndsWith("Text ",
ExcelFindType.Text)

'StartsWith the Number with Text format
Dim result As IRange = sheet.FindStringEndsWith("00", ExcelFindType.Text)

'Startswith the Text with MatchCase
Dim result As IRange = sheet.FindStringEndsWith("Case", ExcelFindType.Text,
False)
```

Replace

This method enables to replace a string, with the data of various data types and data sources, such as data table, data column and array. Following are the overloads for the **Replace** method.

Methods	Description
Replace(String, DateTime)	Replaces specified string by specified value.
Replace(String, Double)	Replaces specified string by specified value.

Replace(String, String)	Replaces specified string by specified value.
Replace(String, DataColumn, Boolean)	Replaces specified string by data column values.
Replace(String, DataTable, Boolean)	Replaces specified string by data table values.
Replace(String, Double[], Boolean)	Replaces specified string by data from array.
Replace(String, Int32[], Boolean)	Replaces specified string by data from array.
Replace(String, String[], Boolean)	Replaces specified string by data from array.

Following code example illustrates how to replace strings with various data.

[C#]

```
// Replacing the Text.
sheet.Replace("Find and Replace", "New Find and Replace");

// Replacing a date value by using datetime.
sheet.Replace("Datevalue", DateTime.Now);

// Replace using array value.
sheet.Replace("Arrayvalue", new string[] { "ArrayValue1", "ArrayValue2",
"ArrayValue3" }, true);

// Replacing a data table by calling a function SampleDataTable().
DataTable table = SampleDataTable();
sheet.Replace("DataTable", table, true);
```

[VB .NET]

```
' Replacing the Text.
sheet.Replace("Find and Replace", "New Find and Replace")

' Replacing a date value by using datetime.
sheet.Replace("Datevalue", DateTime.Now)

' Replace using array value.
sheet.Replace("Arrayvalue", New String() {"ArrayValue1", "ArrayValue2",
"ArrayValue3"}, True)

' Replacing a data table by calling a function SampleDataTable().
Dim table As DataTable = SampleDataTable()
```

```
sheet.Replace("DataTable", table, True)
```

4.1.3 Cells

Spreadsheets are made up of cells. Excel allows to manipulate these cells to customize the sheet for user needs.

This section explains the following cell manipulations.

- Cell Size-This section explains various options to change the cell height and width.
- Insert-This section explains how XlsIO APIs are used to insert rows and columns in the spreadsheet.
- Delete-This section explains how XlsIO APIs are used to delete rows and columns in the spreadsheet.
- Visibility-This section explains various options in XlsIO are used to control the visibility of the spreadsheet.

4.1.3.1 Insert

XlsIO has support for dynamically inserting rows and columns into a new/existing worksheet. Inserting rows/columns will allow the other rows/columns to move down/right by one step, and accommodate the new rows/columns.

MS Excel allows to insert rows/columns through the **Insert** menu option.

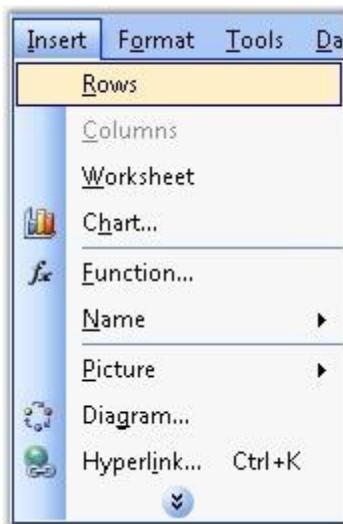


Figure 58: Insert Menu in Excel

Following code example illustrates inserting rows/columns.

[C#]

```
// Inserting Rows.  
sheet.InsertRow(3);  
  
// Inserting Columns.  
sheet.InsertColumn(2);
```

[VB.NET]

```
' Inserting Rows.  
sheet.InsertRow(3)  
  
' Inserting Columns.  
sheet.InsertColumn(2)
```

XlsIO also allows you to insert multiple rows and columns. The following code example illustrates this.

[C#]

```
// Inserting multiple columns.  
sheet.InsertColumn(colIndex,colCount);
```

```
' Inserting multiple rows.  
sheet.InsertRow(rowIndex, rowCount);
```

[VB.NET]

```
' Inserting multiple columns.  
sheet.InsertColumn(colIndex, colCount)  
  
' Inserting multiple rows.  
sheet.InsertRow(rowIndex, rowCount)
```

You can also preserve the previous or next row/column formats by using XlsIO. Following code example illustrates this.

[C#]

```
sheet.InsertRow(rowIndex, count, ExcelInsertOptions.FormatAsBefore);
```

[VB.NET]

```
sheet.InsertRow(rowIndex, count, ExcelInsertOptions.FormatAsBefore)
```



Note: Here row and column index of Insert methods are "one based".

Following table lists the options provided by the **ExcelInsertOptions** enumerator.

Member name	Description
FormatAsBefore	Indicates that after insert operation, inserted rows/columns must be formatted as row above or column left.
FormatAsAfter	Indicates that after insert operation, inserted rows/columns must be formatted as row below or column right.
FormatDefault	Indicates that after insert operation, inserted rows/columns must have default format.

4.1.3.2 Delete

It is often necessary to delete unwanted cells, rows and columns in a spreadsheet, when you want to manipulate cells. MS Excel provides various options to delete cells, rows and columns. You can delete a cell by right-clicking on it, and selecting the **Delete** option from the context menu. On selecting the Delete option, the **Delete** dialog box prompts for an option to be selected as shown in the following screen shot.



Figure 59: Delete Dialog Box in Excel

To delete a cell in XlsIO, you can make use of the **Clear** method. Following code example demonstrates this.

[C#]

```
// Shifts cell left after deletion.
mySheet.Range["A1:E1"].Clear(ExcelMoveDirection.MoveLeft);

// Shifts cell up after deletion.
mySheet.Range["A1:A6"].Clear(ExcelMoveDirection.MoveUp);
```

[VB.NET]

```
' Shifts cell left after deletion.
mySheet.Range("A1:E1").Clear(ExcelMoveDirection.MoveLeft)

' Shifts cell up after deletion.
mySheet.Range("A1:A6").Clear(ExcelMoveDirection.MoveUp)
```

Delete Rows and Columns

MS Excel allows to delete rows and columns in a spreadsheet, by selecting and deleting the rows, through the context menu that appears on right-clicking.

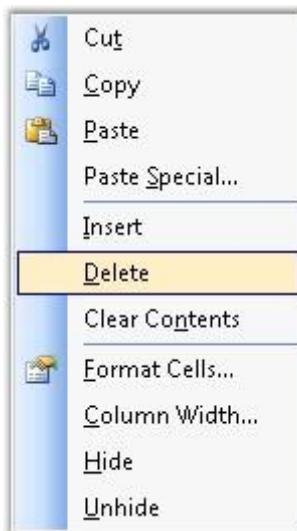


Figure 60: Context menu on right-clicking the cell

Deleting a row, will move the below rows one step up and deleting a column, will move the columns to the right, one step to the left respectively.

XlsIO allows deleting rows and columns by using the **IWorksheet.DeleteRow** and **IWorksheet.DeleteColumn** methods. Following code example illustrates how to delete rows and columns.

[C#]

```
// Deleting Row.
sheet.DeleteRow(3);

// Deleting Column.
sheet.DeleteColumn(2);
```

[VB .NET]

```
' Deleting Rows.
sheet.DeleteRow(3)

' Deleting Columns.
sheet.DeleteColumn(2)
```

You can also delete multiple rows as follows.

[C#]

```
// Deleting Rows.  
sheet.DeleteRow(startRow, NoOfRows);  
  
// Deleting Columns.  
sheet.DeleteColumn(startColumn, NoOfColumn);
```

[VB.NET]

```
' Deleting Rows.  
sheet.DeleteRow(startRow, NoOfRows)  
  
' Deleting Columns.  
sheet.DeleteColumn(startColumn, NoOfColumn)
```



Note: Deletion by using above method is more efficient than looping.



Note: Row/Column index of these methods are "one based".

4.1.3.3 Formats

Excel cells have various formats that can be applied to customize the cells as per the users needs. These formats include sizing up of the cell, showing or hiding cells, protecting the cells, and organizing the sheet that contains the cell.

XlsIO provides support for such formatting with simple APIs. This section explains the support for below formats.

- Cell Size-This section explains various methods that are used to resize cells.
- Visibility-This section explains various options that control the cell's visibility.
- Sheet Organization-This section explains various manipulation sheets that contain cells.
- Protection-This section explains how a cell and sheet can be protected from editing.

4.1.3.3.1 Changing Cell Size

This section deals with customizing the cell width and height as per the users needs. There are two ways to change the cell size. They are as follows.

- Changing the row height and column width manually, with the values specified.
- Changing the row height and column width automatically, to fit the text into the cell.

This is discussed in the following topics.

4.1.3.3.1.1 Row Height and Column Width

MS Excel allows to set the row height and column width by using the **Format** menu option. Go to the **Format** menu, click **Row/Column** and then click **Height/Width** option.

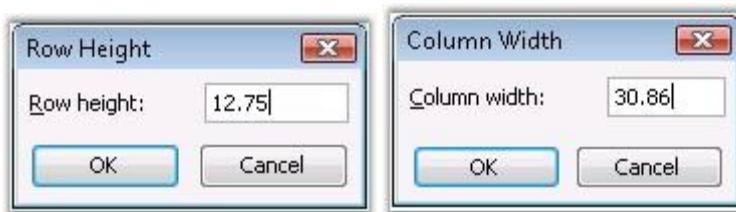


Figure 61: Row Height and Column Width dialog boxes in Excel

Specifying Row Height and Column Width in XlsIO

XlsIO allows to specify a column width of 0 to 255 in a spreadsheet. This value represents the number of characters that can be displayed in a cell that is formatted with the standard font. The default column width is 8.43 characters, which is the default value of MS Excel. If a column has a width of 0, the column is hidden.

Similarly, you can specify a row height of 0 to 409. Note that this is the restriction of MS Excel. This value represents the height measurement in points (1 point equals approximately 1/72 inch or 0.035 cm). The default row height is 12.75 points. If a row has a height of 0, the row is hidden.

XlsIO provides support for setting the **RowHeight** and **ColumnWidth** properties in a worksheet. You can also set the column width and row height in pixels, by using the **IWorksheet.SetColumnWidthInPixel** and **IWorksheet.SetRowHeightInPixel** methods respectively.

```
[C#]
// Changing the Column Width.
sheet.Range["A1"].ColumnWidth = 20;
```

```
sheet.Range["B1"].ColumnWidth = 30;
sheet.Range["C1"].ColumnWidth = 40;
sheet.Range["D1"].ColumnWidth = 50;

// Changing the Row Height.
sheet.Range["A2"].RowHeight = 20;
sheet.Range["A4"].RowHeight = 35;
sheet.Range["A5"].RowHeight = 50;
sheet.Range["A7"].RowHeight = 60;
```

[VB.NET]

```
' Changing the Column Width.
sheet.Range("A1").ColumnWidth = 20
sheet.Range("B1").ColumnWidth = 30
sheet.Range("C1").ColumnWidth = 40
sheet.Range("D1").ColumnWidth = 50

' Changing the Row Height.
sheet.Range("A2").RowHeight = 20
sheet.Range("A4").RowHeight = 35
sheet.Range("A5").RowHeight = 50
sheet.Range("A7").RowHeight = 60
```

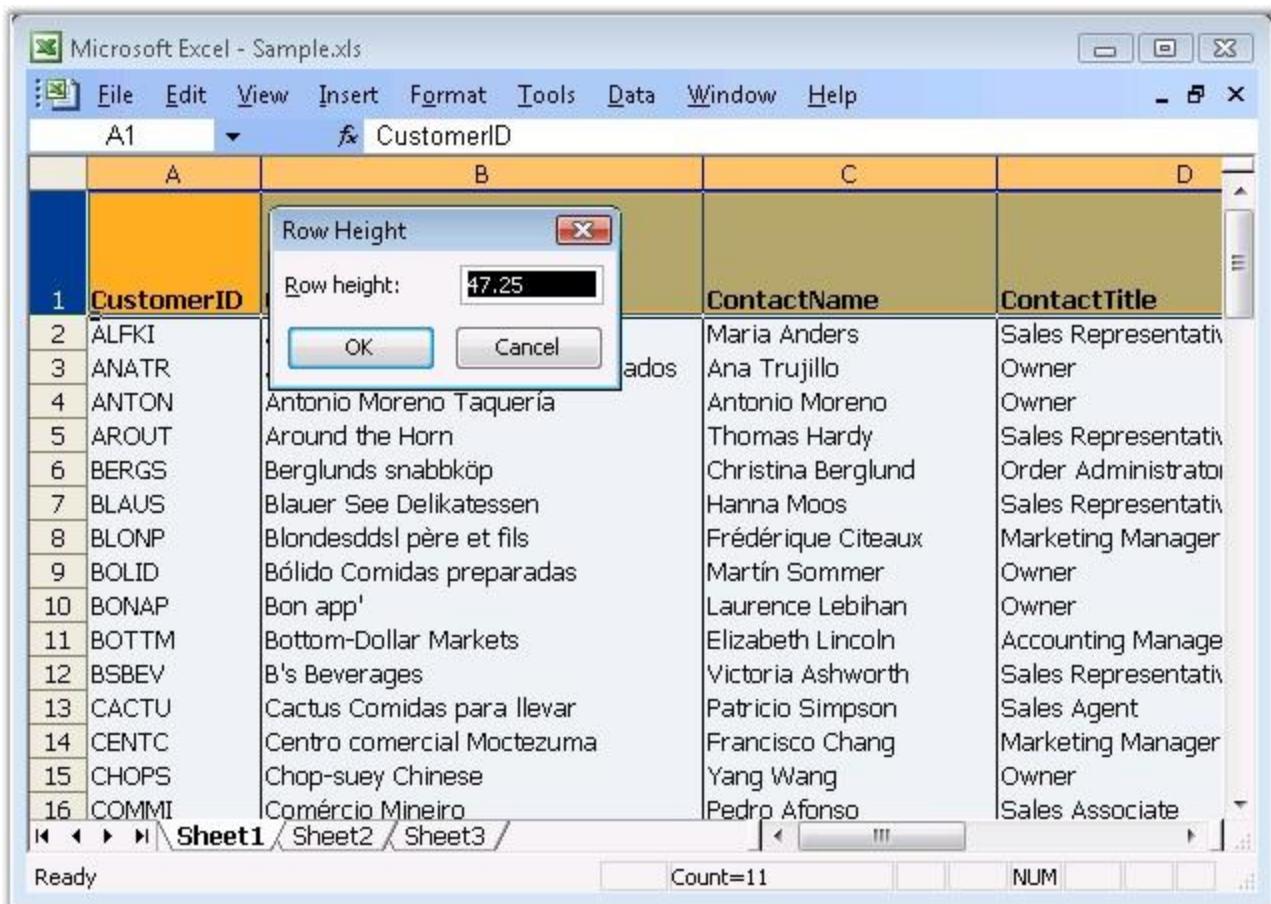


Figure 62: Setting row height in XlsIO

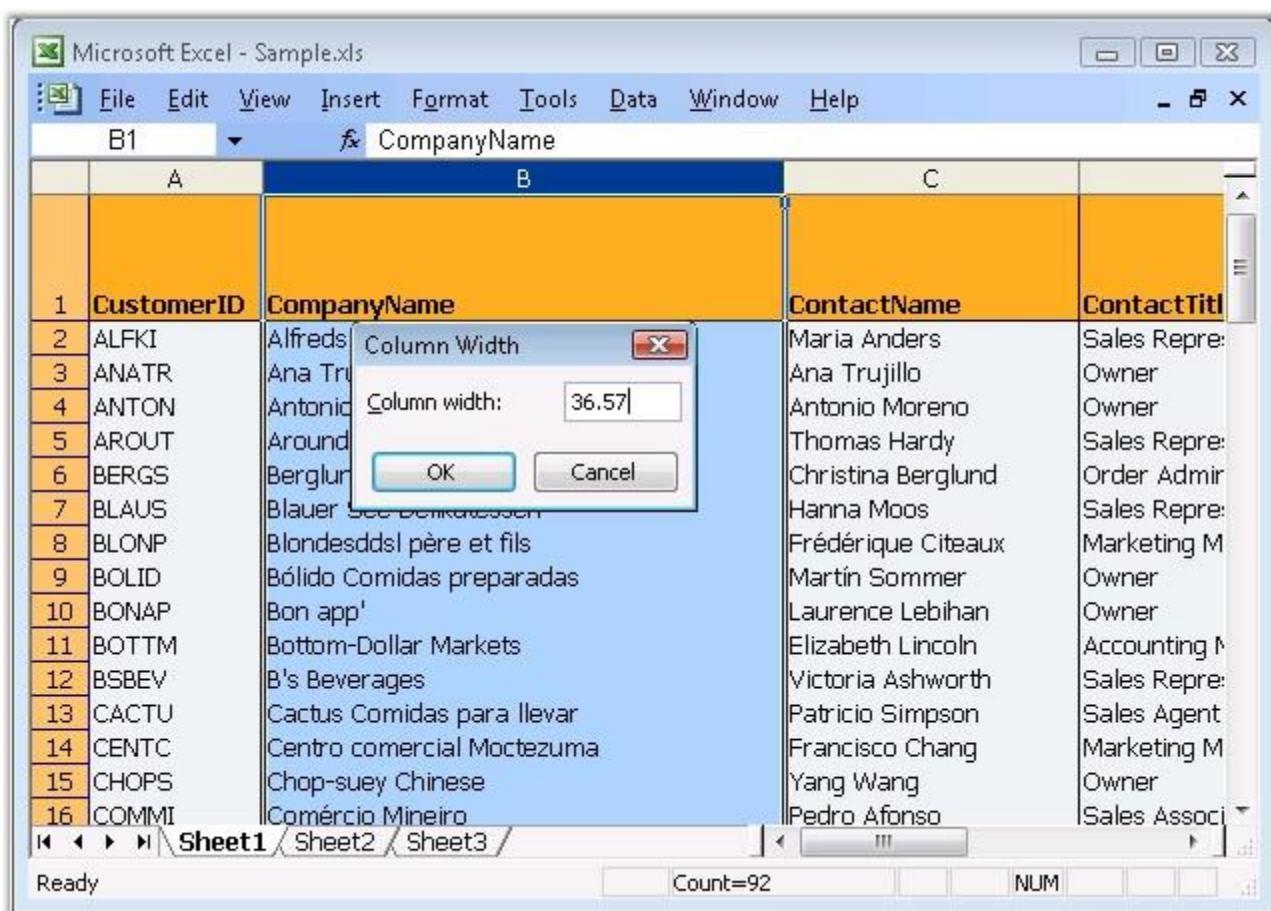


Figure 63: Setting column width in XlsIO

4.1.3.3.1.2 AutoFit Rows and Columns

AutoFit is the option in MS Excel that can be enabled or disabled through the **Format** menu. AutoFit is the name given to the automatic width (or height) adjustment, to fit the contents of a cell, row or column.

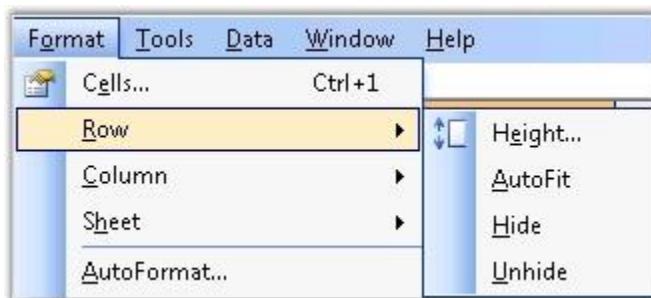


Figure 64: AutoFit in Excel

XlsIO also provides support to autofit the contents in a worksheet. You can autofit a range of cells, or a single column/row, or autofit within the given range.

This section demonstrates various autofit techniques supported by XlsIO.

AutoFit Single Row/Column

XlsIO allows to resize the cells in a column or row, based on the row/column index given. Following code example illustrates autofitting a single row/column.

[C#]

```
sheet.Range["E1"].Text = "This is the Long Text";

// AutoFit applied to a Single Column.
sheet.AutofitColumn(5);

// AutoFit applied to a Single Row.
sheet.AutofitRow(1);
```

[VB.NET]

```
sheet.Range("E1").Text = "This is the Long Text"

' AutoFit applied to a Single Column.
sheet.AutofitColumn(5)

' AutoFit applied to a Single Row.
sheet.AutofitRow(1)
```



Note: These indexes are "one based".

You can also autofit single row/column as follows.

[C#]

```
sheet.Rows[0].AutofitRows();
sheet.Columns[0].AutofitColumns();
```

[VB .NET]

```
sheet.Rows[0].AutofitRows()
sheet.Columns[0].AutofitColumns()
```



Note: Here column and row indexes are "zero based".

AutoFit Multiple Rows/Columns

It is also possible to autofit multiple rows/column based on the range specified as follows.

[C#]

```
// Entering text inside the Cells.
sheet.Range["A1:D1"].Text = "This is the Long Text";
sheet.Range["E1"].Text = "This is the Long Text";
sheet.Range["A2:A5"].Text = "This is the Long Text using Autofit Columns and
Rows";

// AutoFit Applied to a Range.
sheet.Range["A1:D1"].AutofitColumns();
sheet.Range["A2:A5"].AutofitRows();

//Autofits all the columns used in the worksheet.
sheet.UsedRange.AutofitColumns();
```

[VB .NET]

```
' Entering text inside the Cells.
sheet.Range("A1:D1").Text = "This is the Long Text"
sheet.Range("E1").Text = "This is the Long Text"
sheet.Range("A2:A5").Text = "This is the Long Text using Autofit Columns and
Rows"

' AutoFit Applied to a Range .
sheet.Range("A1:D1").AutofitColumns()
sheet.Range("A2:A5").AutofitRows()

'Autofits all the columns used in the worksheet.
sheet.UsedRange.AutofitColumns()
```

AutoFit within a Range of Cells

XlsIO also allows to autofit a row/column based on the content in a range of cells within the cells.

```
// AutoFit columns within a Range.
IWorksheet.Range[int startRow, int startColumn, int lastRow, int
lastColumn].AutofitColumns()

// AutoFit rows within a Range.
IWorksheet.Range[int startRow, int startColumn, int lastRow, int
lastColumn].AutofitRows();
```

Following code example illustrates autofitting within a range of cells.

[C#]

```
// Entering text inside the Cells.
sheet.Range[B2:I20].Text = "Autofit";
sheet.Range[1, 2].Value = "A very long text, This should be ignored by
mySheet.Range[5, 2, 19, 2].AutofitColumns()";
sheet.Rows[4].RowHeight = 90;
sheet.Rows[15].RowHeight = 90;

// AutoFit columns applied within a Range.
mySheet.Range[5, 2, 19, 2].AutofitColumns();

// AutoFit rows applied within a Range.
mySheet.Range[5, 2, 13, 4].AutofitRows();
```

[VB.NET]

```
' Entering text inside the Cells.
sheet.Range[B2:I20].Text = "Autofit"
sheet.Range[1, 2].Value = "A very long text, This should be ignored by
mySheet.Range[5, 2, 19, 2].AutofitColumns()"
sheet.Rows[4].RowHeight = 90
sheet.Rows[15].RowHeight = 90

' AutoFit columns applied within a Range.
mySheet.Range[5, 2, 19, 2].AutofitColumns()

' AutoFit rows applied within a Range.
mySheet.Range[5, 2, 13, 4].AutofitRows()
```

Here, though the cell "B2" has long text, autofit will not be applied to this column as the cell inside the range[5, 2, 19, 2] has text smaller than that. Similarly, row height for Row 15 will not be affected with AutoFit rows as the range[5, 2, 13, 4] has row height smaller than Row 15.



Note:

- 1) If a Range text is text wrapped, **AutoFitColumn** method will not be applied on it.
- 2) If a Range is merged, **AutoFit** methods will not be applied on it. Note that this is the behavior of MS Word.
- 3) Implementation of **AutoFit** methods are more time consuming. Use these methods in minimal for better performance.

4.1.3.3.2 Cell Visibility

Controlling the visibility of a cell is one of the most useful features in MS Excel. It allows to show/hide the rows and columns according to the needs of the user, and produce fairly readable content, when there is large data in the worksheet.

Following section explains the support provided by XlsIO to hide/unhide sheets and rows/columns.

4.1.3.3.2.1 Hide/Unhide Rows and Columns

Excel allows to hide a row/column by using the Hide command, but a row or column also becomes hidden when you change its row height or column width to zero. Also, you can show a hidden row/column by using the Unhide command.

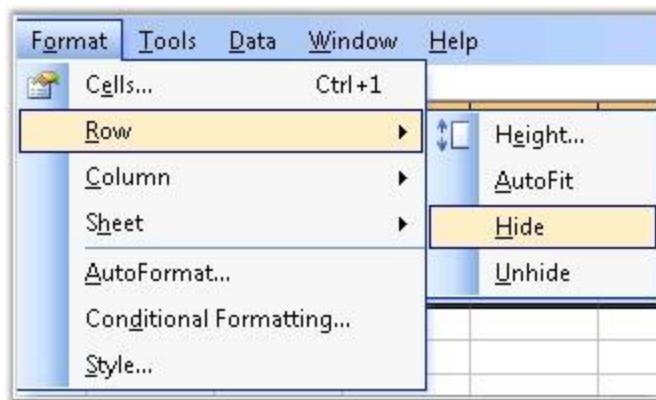


Figure 65: Hiding a row

Hiding and Unhiding Rows in XlsIO

XlsIO provides support for hiding/unhiding rows and columns. This can be done by using **ShowRow** and **ShowColumn** methods.

[C#]

```
// Hiding the First Column and Second Row.
sheet.ShowColumn( 1, false );
sheet.ShowRow( 2, false );

// Hiding the Fifth Column and Fifth Row.
sheet.ShowColumn( 5, false );
sheet.ShowRow( 5, false );

// Unhiding the Fifth Column and Second Row.
sheet.ShowColumn( 5, true );
sheet.ShowRow( 2, true );
```

[VB.NET]

```
' Hiding the First Column and Second Row.
sheet.ShowColumn(1, False)
sheet.ShowRow(2, False)

' Hiding the Fifth Column and Fifth Row.
sheet.ShowColumn(5, False)
sheet.ShowRow(5, False)

' Unhiding the Fifth Column and Second Row.
```

```
sheet.ShowColumn(5, True)
sheet.ShowRow(2, True)
```

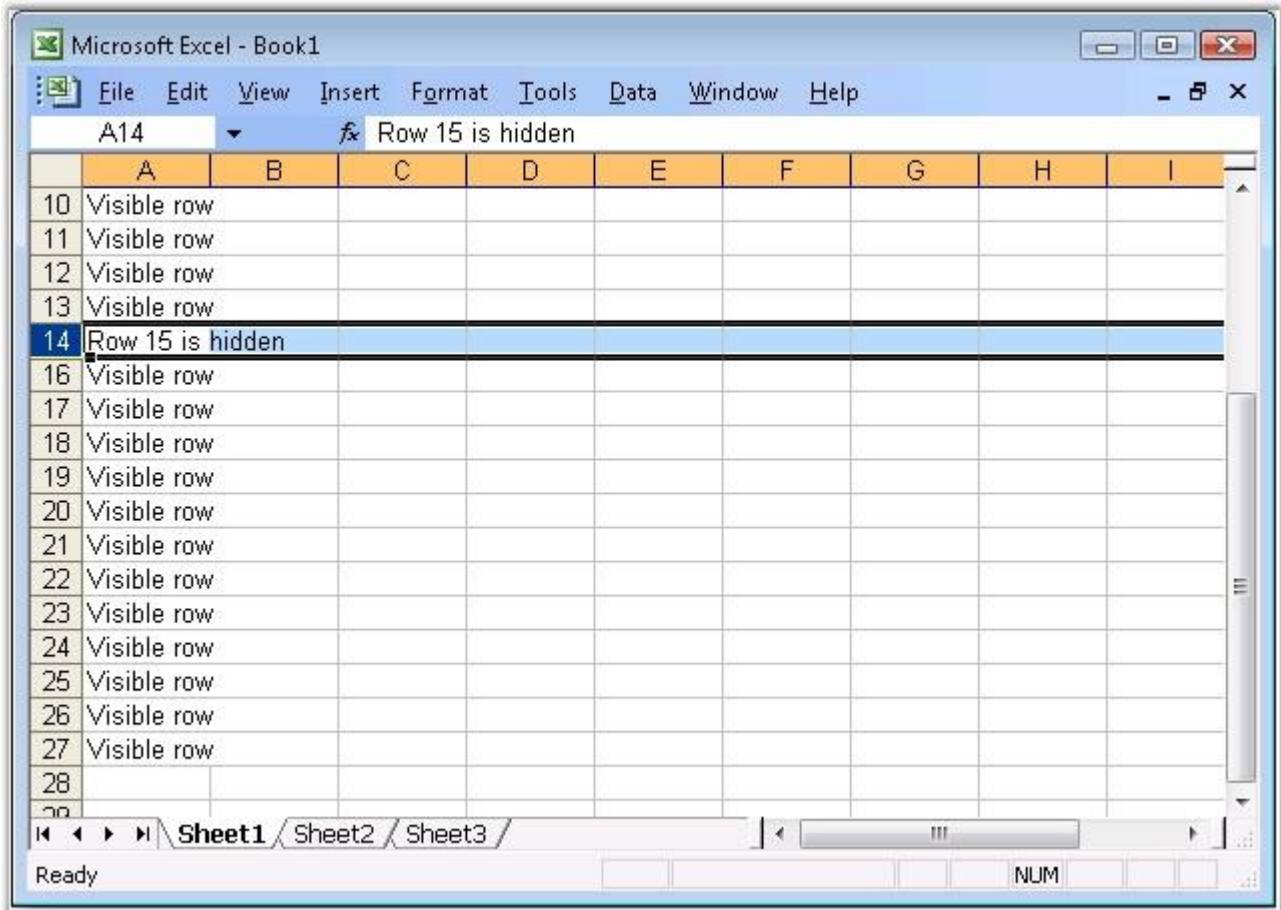


Figure 66: Worksheet with a hidden row

XlsIO also provides options to provide focus to a particular row/column, when it is opened by using the **TopVisibleRow** and **LeftVisibleColumn** properties respectively.

[C#]

```
//Scrolls to 40th row
sheet.TopVisibleRow = 40;

//Scrolls to 7 column when opening
sheet.LeftVisibleColumn = 7;
```

[VB .NET]

```
'Scrolls to 40th row
sheet.TopVisibleRow = 40

'Scrolls to 7 column when opening
sheet.LeftVisibleColumn = 7
```

This is especially useful, when the spreadsheet has large number of records, and the user wants to view a particular row/column that has some details, without scrolling to that row/column after opening it. Note that these row and column indexes are "one based".

4.1.3.3.2.2 Hide/Unhide Worksheet

Excel has the sheet tab bar that appears at the bottom of the screen with tab scrolling buttons displayed on the left side. Excel provides an option to show/hide a sheet from the user view. This is done by selecting the "Hide" item in the context menu of the sheet.

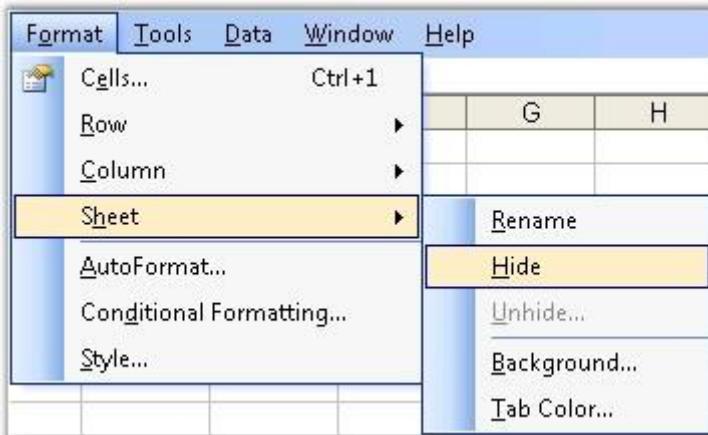


Figure 67: Hiding a Worksheet

Hiding and Unhiding a Worksheet in XlsIO

XlsIO also allows to hide/unhide worksheets by using the **Visibility** property. Following APIs are used to hide/unhide worksheets.

[C#]

```
sheet.Visibility = WorksheetVisibility.Hidden;
```

[VB .NET]

```
sheet.Visibility = WorksheetVisibility.Hidden
```

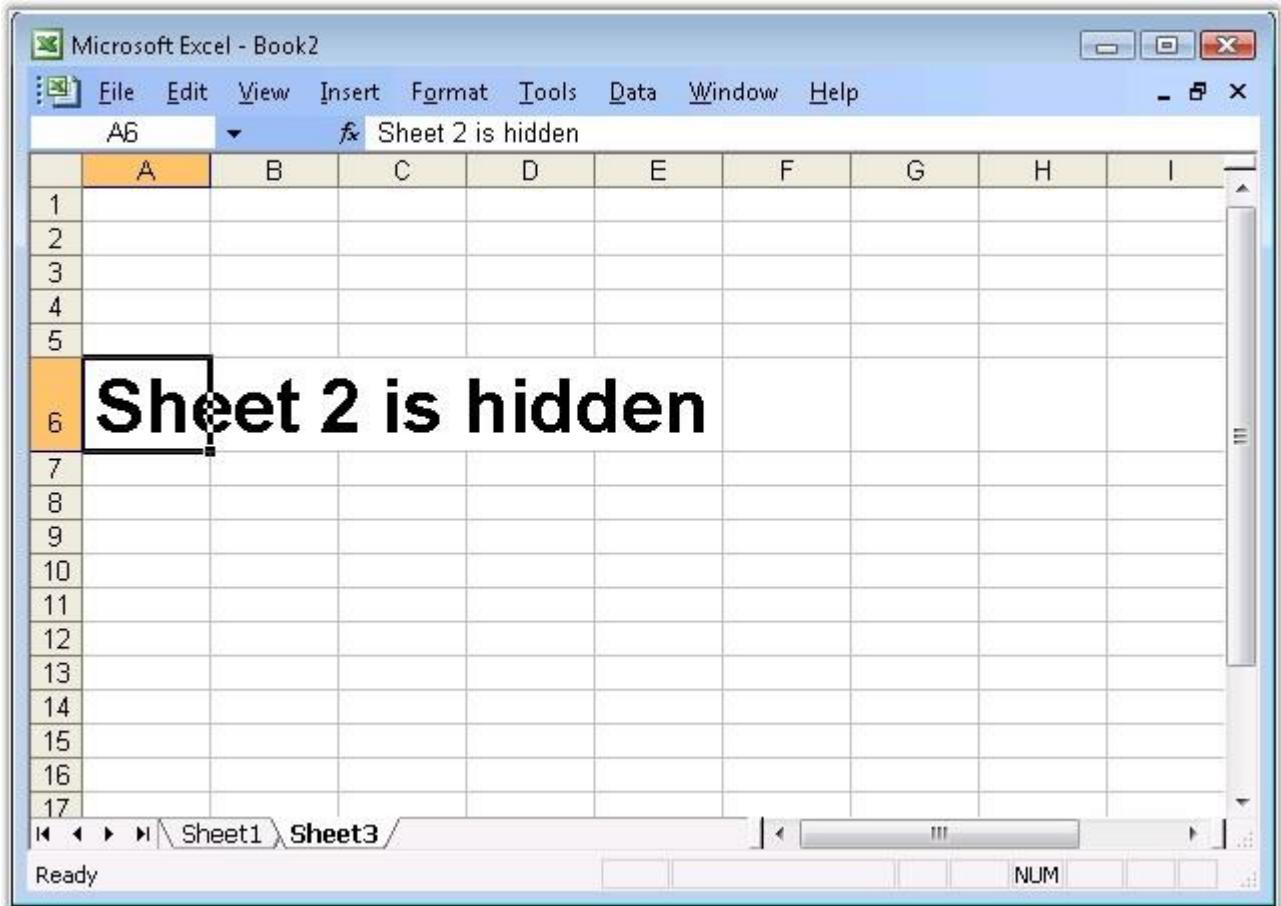


Figure 68: Hidden Worksheet

You can also hide all the tabs in the worksheet by using the **DisplayWorkbookTabs** option in the **IWorkbook**.

XlsIO also provides an option to activate a worksheet, while opening it in the workbook, which is equivalent to clicking a worksheet in MS Excel. This is done by using the **Activate** method.

[C#]

```
sheet.Activate();
```

[VB .NET]

```
sheet.Activate()
```

4.1.3.3.2.3 Adding Feature to show or hide a particular range of rows/columns

Hiding and Unhiding Rows in XlsIO

XlsIO provides support for hiding/unhiding rows and columns. This can be done by using **ShowRow**, **ShowColumn** and **ShowRange** methods as shown in the following code sample:

[C#]

```
// Hiding the First Column and Second Row.  
sheet.ShowColumn( 1, false );  
sheet.ShowRow( 2, false );  
  
// Hiding the Fifth Column and Fifth Row.  
sheet.ShowColumn( 5, false );  
sheet.ShowRow( 5, false );  
  
// Unhiding the Fifth Column and Second Row.  
sheet.ShowColumn( 5, true );  
sheet.ShowRow( 2, true );  
  
IRange range = sheet[1, 4];  
//Hiding the first to thrid row and first to thrid column  
sheet.ShowRange(range, false);  
  
IRange firstRange = ws[1, 1,3,3];  
IRange secondRange = ws[5, 5, 7, 7];  
RangesCollection rangeCollection = new RangesCollection(app, ws);  
rangeCollection.Add(firstRange);  
rangeCollection.Add(secondRange);  
//Hiding the collection of ranges  
ws.ShowRange(rangeCollection, false);
```

[VB .NET]

```

'Hiding the First Column and Second Row.
sheet.ShowColumn(1, False)
sheet.ShowRow(2, False)

'Hiding the Fifth Column and Fifth Row.
sheet.ShowColumn(5, False)
sheet.ShowRow(5, False)

'Unhiding the Fifth Column and Second Row.
sheet.ShowColumn(5, True)
sheet.ShowRow(2, True)

IRange range = sheet[1, 4];
//Hiding the first to thrid row and first to thrid column
sheet.ShowRange(range, false);

IRange firstRange = ws[1, 1, 3, 3];
IRange secondRange = ws[5, 5, 7, 7];
RangesCollection rangeCollection = new RangesCollection(app, ws);
rangeCollection.Add(firstRange);
rangeCollection.Add(secondRange);
//Hiding the collection of ranges
ws.ShowRange(rangeCollection, false);

```

Methods

Prototype	Description
ShowRange (IRange, bool)	Shows or hides a particular range
ShowRange(IRange[], bool)	Shows or hides a particular array of ranges
ShowRange(RasngesCollection, bool)	Shows or hides a particular collection of ranges

4.1.3.3.3 Sheet Organization

Excel has options to consolidate data from different worksheets, or move a sheet to another workbook, or insert a sheet in between the worksheets. Also, the sheet tab color can be formatted, and the sheets can be named as per the users needs. This section explains how sheets can be organized. Below sections explains the XlsIO's ability to organize sheets.

- Copy/Move Worksheet-This section explains how a worksheet can be copied from another worksheet with or without certain formatting.
- Sheet Format-This section explains various formats that can be applied to a sheet.

4.1.3.3.3.1 Copy/Move Worksheet

When you copy/move rows and columns, Microsoft Excel copies or moves all the data that it contains, including formulas and their resulting values, comments, cell formats, and hidden cells.

Copying Worksheets

Copying worksheets can be internal or external. XlsIO provides support for copying a worksheet within a workbook, and also from one workbook to another. This feature can be used to merge together several workbooks. Following code example illustrates how to copy a sheet with its entire contents to another sheet.

[C#]

```
// Open the Source WorkBook.
IWorkbook sourceWorkbook =
application.Workbooks.Open(@"..\..\..\..\..\Data\SourceWorkbookTemplate.xls");
;

// Open the Destination WorkBook.
IWorkbook destinationWorkbook =
application.Workbooks.Open(@"..\..\..\..\..\Data\DestinationWorkbookTemplate.xls");

// Copy the first worksheet from the Source workbook to the destination
// workbook.
destinationWorkbook.Worksheets.AddCopy(sourceWorkbook.Worksheets[0]);

// Activate the newly added worksheet in the destination workbook.
destinationWorkbook.ActiveSheetIndex = 1;
```

```
// Saving the workbook to disk.
destinationWorkbook.SaveAs("Sample.xls");

// Close the workbook.
destinationWorkbook.Close();
```

[VB.NET]

```
' Open the Source WorkBook.
Dim sourceWorkbook As IWorkbook =
application.Workbooks.Open("../..\..\..\..\..\Data\SourceWorkbookTemplate.xls")

' Open the Destination WorkBook.
Dim destinationWorkbook As IWorkbook =
application.Workbooks.Open("../..\..\..\..\..\Data\DestinationWorkbookTemplate.xls")

' Copy the first worksheet from the Source workbook to the destination
workbook.
destinationWorkbook.Worksheets.AddCopy(sourceWorkbook.Worksheets(0))

' Activate the newly added worksheet in the destination workbook.
destinationWorkbook.ActiveSheetIndex = 1

' Saving the workbook to disk.
destinationWorkbook.SaveAs("Sample.xls")

' Close the workbook.
destinationWorkbook.Close()
```

You can also specify copy options while copying a worksheet, if you are interested in improving the performance, and if you are interested in ignoring certain formatting while copying through the **ExcelWorksheetCopyFlags** enumerator. Following are the values for this enumerator.

Member name	Description
None	No flags.
ClearBefore	Represents the ClearBefore copy flags.
CopyNames	Copies Names.
CopyCells	Copies whole Cells.
CopyRowHeight	Copies Row Height.

CopyColumnHeight	Copies Column Height.
CopyOptions	CopyOptions copy flags.
CopyMerges	Copies Merges.
CopyShapes	Copies Shapes.
CopyConditionFormats	Represents the CopyConditionFormats copy flags.
CopyAutoFilters	Copies AutoFilters.
CopyDataValidations	Copies Data Validations.
CopyPageSetup	Copy page setup (page breaks, paper orientation, header, footer and other properties).
CopyAll	Represents the CopyAll copy flags.
CopyWithoutNames	Represents the CopyWithoutNames copy flags.

The following code example illustrates copying worksheets.

[C#]

```
// Open the Source WorkBook.
IWorkbook sourceWorkbook =
application.Workbooks.Open(@"..\..\..\..\..\Data\SourceWorkbookTemplate.xls");
;

// Open the Destination WorkBook.
IWorkbook destinationWorkbook =
application.Workbooks.Open(@"..\..\..\..\..\Data\DestinationWorkbookTemplate.xls");

// Copy the first worksheet from the Source workbook to the destination
// workbook with data validations.
destinationWorkbook.Worksheets.AddCopy(sourceWorkbook.Worksheets[0],
ExcelWorksheetCopyFlags.CopyDataValidations);

// Activate the newly added worksheet in the destination workbook.
destinationWorkbook.ActiveSheetIndex = 1;

// Saving the workbook to disk.
destinationWorkbook.SaveAs("Sample.xls");
```

```
// Close the workbook.
destinationWorkbook.Close();
```

[VB.NET]

```
' Open the Source WorkBook.
Dim sourceWorkbook As IWorkbook =
application.Workbooks.Open("../..\..\..\..\..\Data\SourceWorkbookTemplate.xls")

' Open the Destination WorkBook.
Dim destinationWorkbook As IWorkbook =
application.Workbooks.Open("../..\..\..\..\..\Data\DestinationWorkbookTemplate.xls")

' Copy the first worksheet from the Source workbook to the destination
workbook with data validations.
destinationWorkbook.Worksheets.AddCopy(sourceWorkbook.Worksheets(0),
ExcelWorksheetCopyFlags.CopyDataValidations)

' Activate the newly added worksheet in the destination workbook.
destinationWorkbook.ActiveSheetIndex = 1

' Saving the workbook to disk.
destinationWorkbook.SaveAs("Sample.xls")

' Close the workbook.
destinationWorkbook.Close()
```

You can also copy a worksheet before and after a particular worksheet by using the **AddCopyBefore** and **AddCopyAfter** methods respectively.

Moving a Worksheet

XlsIO also allows moving worksheets from one position to another. This is similar to dragging a worksheet in MS Excel. This can be performed by using the **Move** method. Following code example illustrates how a worksheet is moved to the 2nd position.

[C#]

```
sheet.Move(2);
```

[VB.NET]

```
sheet.Move(2)
```

4.1.3.3.2 Convert To Image

Essential XlsIO can convert a worksheet to an image of type bitmap or metafile based on the input range of rows and columns with all basic formats preserved. The sheet can be converted and saved to disk or stream. The converted image can be inserted in a pdf by using Essential PDF.

For more information on insertion of images in PDF, refer the following link:

http://help.syncfusion.com/ug_82sp/Reporting_XlsIO/defaultWPF.html

[C#]

```
// Convert as bitmap.
Image img = sheet.ConvertToImage(1, 1, 10, 20);
img.Save("Sample.png", ImageFormat.Png);

// Converts to MetaFile.
Image img = sheet.ConvertToImage(1, 1, 10, 20, ImageType.Metafile, null);
img.Save("Sample.emf", ImageFormat.Emf);

// Converts and save as stream.
MemoryStream stream = new MemoryStream();
sheet.ConvertToImage(1, 1, 10, 20, ImageType.Metafile, stream);
```

[VB]

```
' Convert as bitmap.
Dim img As Image = sheet.ConvertToImage(1, 1, 10, 20)
img.Save("Sample.png", ImageFormat.Png)

' Converts to Metafile.
Dim img As Image = sheet.ConvertToImage(1, 1, 10, 20, ImageType.Metafile,
Nothing)
img.Save("Sample.emf", ImageFormat.Emf)

' Converts and save as stream.
Dim stream As MemoryStream = New MemoryStream()
sheet.ConvertToImage(1, 1, 10, 20, ImageType.Metafile, stream)
```

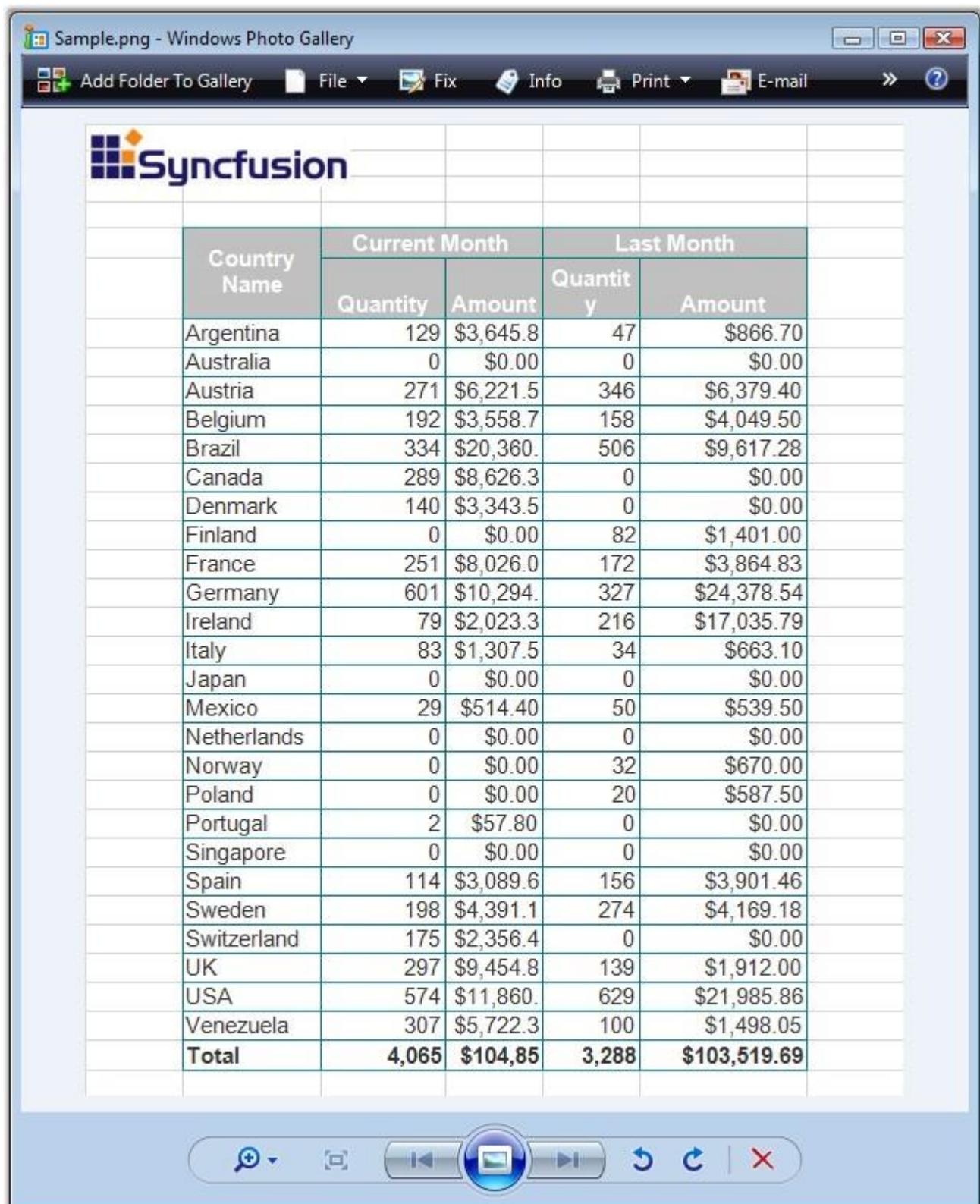


Figure 69: Worksheet converted to image by using XlsIO

Essential XlsIO can convert a worksheet based on the input range of the rows and columns which does not support the following elements:

- Subscript/Superscript
- RTF
- Shrink to fit
- Shapes (except TextBox shape and Image)
- Charts and Chart Worksheet
- Complex conditional formatting
- Gradient fill is partially supported

4.1.3.3.3 Sheet Format

Excel provides various options to format a sheet. This includes setting the tab color, naming sheets, and clearing the data in the sheet. This section demonstrates how these formats can be applied to the worksheets in XlsIO.

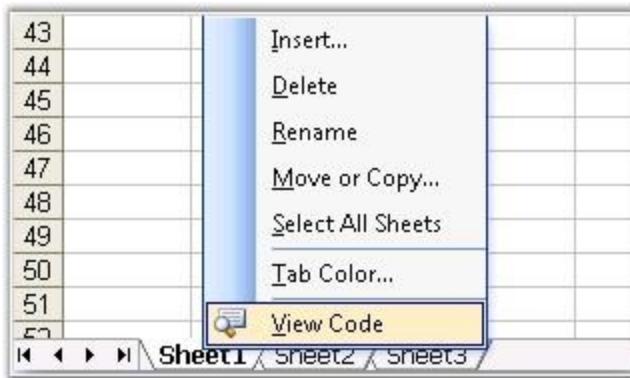


Figure 70: Menu for formatting a Sheet

Worksheet Formatting in XlsIO

Sheet Naming

Sheets can be named/renamed to find out the sheet or provide a hint on the data in the sheet, in a large workbook that has a number of worksheets. You can name a worksheet by using the **IWorksheet.Name** property.

```
[C#]
sheet.Name = "Result";
```

[VB .NET]

```
sheet.Name = "Result"
```

Tab Color

Tab Colors are set to highlight a particular sheet that has some important data. This is done in Excel, by selecting the "Tab Color" option in the sheet context menu. You can set the tab color through the **TabColor** property, as given below.

[C#]

```
sheet.TabColor = ExcelKnownColors.Blue;
```

[VB .NET]

```
sheet.TabColor = ExcelKnownColors.Blue
```

Clear

You can clear all data/data with formatting in a sheet, by using the **Clear** and **ClearData** methods of the **IWorksheet**.

[C#]

```
sheet.Clear();
```

[VB .NET]

```
sheet.Clear()
```

4.1.3.3.3.4 Sheet to HTML

XlsIO provides support to convert a worksheet or workbook to HTML with basic formatting preserved. The following code example illustrates how to do this.

[C#]

```
// Save an Excel sheet as HTML file.  
sheet.SaveAsHtml("Sample.html");  
  
// Save a workbook as HTML file.
```

```
workbook.SaveAsHtml("Sample.html", HtmlSaveOptions.Default);
```

[VB.NET]

```
' Save an Excel sheet as HTML file.  
sheet.SaveAsHtml("Sample.html")  
  
' Save a workbook as HTML file.  
workbook.SaveAsHtml("Sample.html", HtmlSaveOptions.Default)
```

Save Options

XlsIO also provides various save options to control images and text in an Excel file. It enables you to save a worksheet with the displayed text or value in the cell to HTML file. The following code example illustrates this.

[C#]

```
HtmlSaveOptions options = new HtmlSaveOptions();  
options.TextMode = HtmlSaveOptions.GetText.DisplayText;  
optionsImagePath = @"..\..\Output\";  
sheet.SaveAsHtml("Sample.html", options);
```

[VB.NET]

```
Dim options As New HtmlSaveOptions()  
options.TextMode = HtmlSaveOptions.GetText.DisplayText  
optionsImagePath = "..\..\Output\"  
sheet.SaveAsHtml("Sample.html", options)
```

4.1.3.3.4 Protection

Excel provides various options to protect worksheet and workbook elements. Protection prevents a user from accidentally or deliberately changing, moving, or deleting important data. There are various options to protect worksheets and workbooks.

Refer to the [Changes](#) section for more details.

This section explains how cell protection can be applied in MS Excel by using XlsIO.

4.1.3.3.4.1 Lock Cells

Cell modification can be prevented by locking the cell, by using the **Protection** tab in the **Format Cells** dialog box.

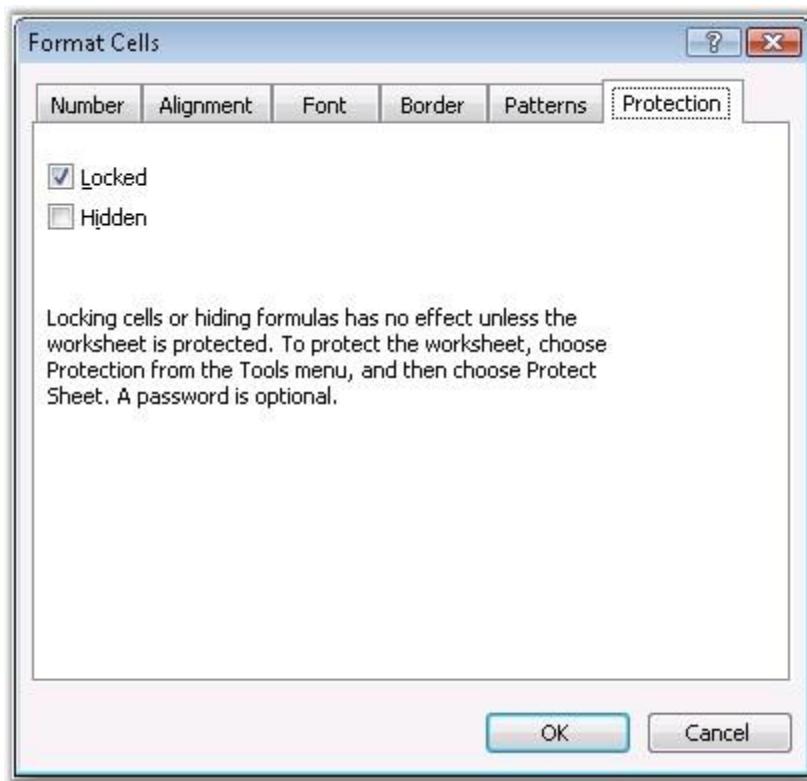


Figure 71: Format Cells dialog - Protection

This will prompt the following error message at run time, when a user tries to modify the cell.



Figure 72: Error on modifying the protected cell

Locking and Unlocking in XlsIO

XlsIO supports locking and unlocking cells by using the cell's **Locked** property, which can be manipulated to make certain cells editable in a protected worksheet. Please note that locking/unlocking a cell in an unprotected worksheet has no effect. For protecting the worksheet, see [Worksheet Protection](#).

[C#]

```
// Opening the Existing (Protected) Worksheet from a Workbook
IWorkbook workbook =
application.Workbooks.Open("CellProtectionTemplate.xls");

// Unlocking the cell which, need to be edited.
sheet.Range["A1"].CellStyle.Locked = false;
```

[VB.NET]

```
' Opening the Existing (Protected) Worksheet from a Workbook
Dim workbook As IWorkbook =
application.Workbooks.Open("CellProtectionTemplate.xls")

' Unlocking the cells which, need to be edited.
sheet.Range("A1").CellStyle.Locked = False
```

4.1.4 Clipboard

The office clipboard in Excel is an extension of the Windows clipboard, and allows for easy transfer of data, by using the **Copy** and **Paste** commands. XlsIO provides support to copy a worksheet or workbook to the clipboard. It can be done by using the **CopyToClipboard** method as follows.

[C#]

```
sheet.CopyToClipboard();
workbook.CopyToClipboard();
```

[VB.NET]

```
sheet.CopyToClipboard();
workbook.CopyToClipboard();
```

4.2 Insert

Excel inserts various elements in spreadsheets that demonstrate the intention of the user. This section explains the below features supported by XlsIO.

- Illustrations-This topic explains how a picture can be inserted in a worksheet.
- Charts-This topic explains various charts, and how they can be created by using XlsIO.
- Links-This topic explains how different types of hyperlinks can be inserted through the XlsIO's APIs.
- Header/Footer-This topic explains how a header/footer can be inserted in a spreadsheet.
- Tables-This topic explains about XlsIO's support to read and write tables in a spreadsheet.
- Pivot Tables-This topic explains how pivot tables in templates are supported by XlsIO.
- Shapes-This topic illustrates the shapes supported in XlsIO.
- OLE Objects-This topic illustrates OLE Object support in XlsIO

4.2.1 Illustrations

Excel allows to insert a picture through the **Insert** menu -> **Picture**, and then clicking **From file**. It allows to customize the image by sizing, formatting and positioning the image.

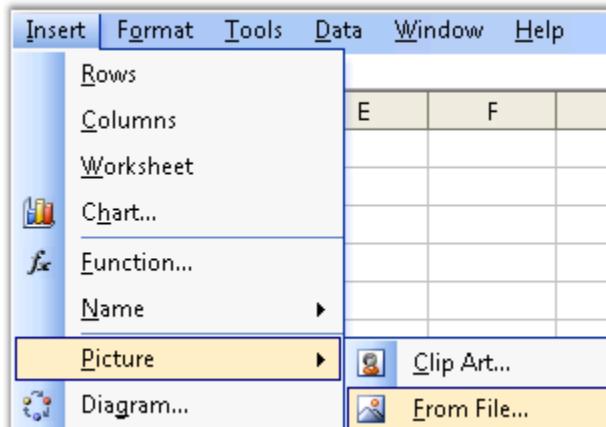


Figure 73: Inserting picture from file

Essential XlsIO has advanced API support for working with images.

It supports the insertion of **Scalar** and **Vector** images in a worksheet. It is also possible to position and set the properties for the image at the desired location. **IPictureShape** is used for inserting and formatting pictures.

[C#]

```
// Inserting image
IPictureShape shape = sheet.Pictures.AddPicture(1, 1, "sample.jpg");
shape.Top = 1157;
shape.Height = 808;
shape.Left = 1417;
shape.Width = 1121;
```

[VB.NET]

```
' Inserting image
Dim shape As IPictureShape = sheet.Pictures.AddPicture(1, 1, "sample.jpg")
shape.Top = 1157
shape.Height = 808
shape.Left = 1417
shape.Width = 1121
```

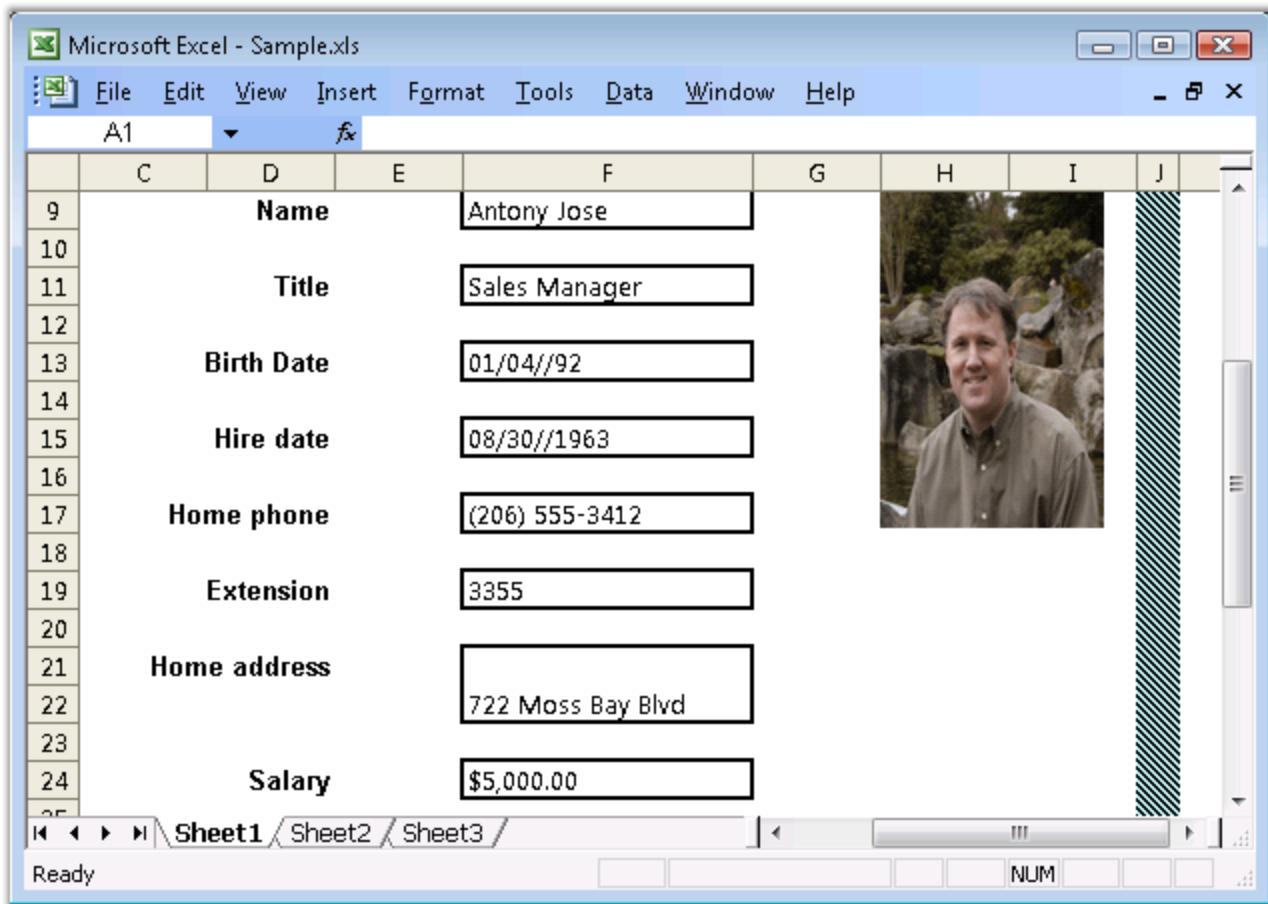


Figure 74: Document with Image created through XlsIO and viewed in Excel

Barcodes and **Charts** can also be inserted in a spreadsheet by using XlsIO's Image Insertion API's. The barcode/chart is rendered to an image by using the **Essential Barcode**/ **Essential Chart**, and then inserted into the spreadsheet as an image.

XlsIO can also extract images from an existing spreadsheet.

```
[C#]
// Read Image.
this.pictureBox1.Image = sheet.Pictures[0].Picture;
```

```
[VB.NET]
' Read Image.
Me.pictureBox1.Image = sheet.Pictures(0).Picture
```



Figure 75: Image from an xls file read by XlsIO and rendered in a Panel

4.2.2 Charts

This section explains the usage of the Essential XlsIO's Chart APIs.

Charts can convey much more than numbers and makes it easier to see the meaning behind the numbers. Essential XlsIO has advanced support for creating and modifying Excel charts inside a workbook. There is an option to choose between creating an Embedded Chart (chart is embedded inside a worksheet) or a Chart Worksheet (chart is a separate worksheet).

4.2.2.1 Embedded Chart

Essential XlsIO has APIs for creating an embedded chart. The **IChartShape** interface represents the embedded chart's in-memory, and this object can be used to format and modify the chart settings for chart area, plot area, and chart title area, with gradient, texture, patterns and pictures.

IChartFrameFormat can be used to change the format of the chart. **IChartSeries** is used to format the series. XlsIO provides options to enable/disable Legends and Data Tables by using the **HasLegend** and **HasDataTable** properties. You can also resize and position the embedded chart in a worksheet.

A chart in XlsIO can be created either through the Data Range of the chart, or by adding series one by one.

Following code example illustrates how to create a chart through the Data Range.

[C#]

```
// Clustered Column Chart.
IChartShape chart = sheet.Charts.Add();

// Set Chart Type.
chart.ChartType = ExcelChartType.Column_Clustered

// Set Data Range.
chart.DataRange = sheet.Range["A1:E5"];

// Specify Series
chart.IsSeriesInRows = false;

// Chart Title
chart.ChartTitle = "Sales comparison";

// X-axis title
chart.PrimaryCategoryAxis.Title = "Fruit Types";

// Y-axis title
chart.PrimaryValueAxis.Title = "Months";

// Show Data Table.
chart.HasDataTable = true;

// Format Chart Area.
IChartFrameFormat chartArea = chart.ChartArea;

// Border
```

```
// Style
chartArea.Border.LinePattern = ExcelChartLinePattern.Solid;

// Color
chartArea.Border.LineColor = Color.Blue;

// Weight
chartArea.Border.LineWeight = ExcelChartLineWeight.Hairline;

// Area

// Fill Effects
chartArea.Fill.FillType = ExcelFillType.Gradient;

// Two Color
chartArea.Fill.GradientColorType = ExcelGradientColor.TwoColor;

// Set two colors.
chartArea.Fill.BackColor = Color.FromArgb(205,217,234);
chartArea.Fill.ForeColor = Color.White;

// Plot Area
IChartFrameFormat chartPlotArea = chart.PlotArea;

// Border

// Style
chartPlotArea.Border.LinePattern = ExcelChartLinePattern.Solid;

// Color
chartPlotArea.Border.LineColor = Color.Blue;

// Weight
chartPlotArea.Border.LineWeight = ExcelChartLineWeight.Hairline;

// Fill Effects
chartPlotArea.Fill.FillType = ExcelFillType.Gradient;

// Two Color
chartPlotArea.Fill.GradientColorType = ExcelGradientColor.TwoColor;

// Set two colors.
chartPlotArea.Fill.BackColor = Color.FromArgb(205,217,234);
chartPlotArea.Fill.ForeColor = Color.White;
```

```

// Format Data Series.
IChartSerie chartAppleSerie = chart.Series["Apples"];

// Color of first serie.
chartAppleSerie.SerieFormat.AreaProperties.ForegroundColor = Color.Red;
chartAppleSerie = chart.Series["Oranges"];

// Color of second serie.
chartAppleSerie.SerieFormat.AreaProperties.ForegroundColor = Color.Orange;
chartAppleSerie = chart.Series["Grapes"];

// Color of third serie.
chartAppleSerie.SerieFormat.AreaProperties.ForegroundColor = Color.Purple;
chartAppleSerie = chart.Series["Banana"];

// Color of fourth serie.
chartAppleSerie.SerieFormat.AreaProperties.ForegroundColor = Color.Yellow;

// Embedded chart position.
chart.TopRow = 10;
chart.BottomRow = 40;
chart.LeftColumn = 5;
chart.RightColumn = 15;

```

[VB.NET]

```

' Clustered Column Chart.
Dim chart As IChartShape = sheet.Charts.Add()

' Set Chart Type.
chart.ChartType = ExcelChartType.Column_Clustered

' Set Data Range.
chart.DataRange = sheet.Range("A1:E5")

' Specify Series.
chart.IsSeriesInRows = False

' Chart Title
chart.ChartTitle = "Sales comparison"

' X-axis title
chart.PrimaryCategoryAxis.Title = "Fruit Types"

' Y-axis title

```

```
chart.PrimaryValueAxis.Title = "Months"

' Show Data Table.
chart.HasDataTable = True

' Format Chart Area.
Dim chartArea As IChartFrameFormat = chart.ChartArea

' Border

' Style
chartArea.Border.LinePattern = ExcelChartLinePattern.Solid

' Color
chartArea.Border.LineColor = Color.Blue

' Weight
chartArea.Border.LineWeight = ExcelChartLineWeight.Hairline

' Area

' Fill Effects
chartArea.Fill.FillType = ExcelFillType.Gradient

' Two Color
chartArea.Fill.GradientColorType = ExcelGradientColor.TwoColor

' Set two colors.
chartArea.Fill.BackColor = Color.FromArgb(205,217,234)
chartArea.Fill.ForeColor = Color.White

' Plot Area
Dim chartPlotArea As IChartFrameFormat = chart.PlotArea

' Border

' Style
chartPlotArea.Border.LinePattern = ExcelChartLinePattern.Solid

' Color
chartPlotArea.Border.LineColor = Color.Blue

' Weight
chartPlotArea.Border.LineWeight = ExcelChartLineWeight.Hairline

' Fill Effects
```

```

chartPlotArea.Fill.FillType = ExcelFillType.Gradient

' Two Color
chartPlotArea.Fill.GradientColorType = ExcelGradientColor.TwoColor

' Set two colors.
chartPlotArea.Fill.BackColor = Color.FromArgb(205,217,234)
chartPlotArea.Fill.ForeColor = Color.White

' Format Data Series.
Dim chartAppleSerie As IChartSerie = chart.Series("Apples")

' Color of first serie.
chartAppleSerie.SerieFormat.AreaProperties.ForegroundColor = Color.Red
chartAppleSerie = chart.Series("Oranges")

' Color of second serie.
chartAppleSerie.SerieFormat.AreaProperties.ForegroundColor = Color.Orange
chartAppleSerie = chart.Series("Grapes")

' Color of third serie.
chartAppleSerie.SerieFormat.AreaProperties.ForegroundColor = Color.Purple
chartAppleSerie = chart.Series("Banana")

' Color of fourth serie.
chartAppleSerie.SerieFormat.AreaProperties.ForegroundColor = Color.Yellow

' Embedded chart position.
chart.TopRow = 10
chart.BottomRow = 40
chart.LeftColumn = 5
chart.RightColumn = 15

```

Following code example illustrates how to create charts by adding Series.

[C#]

```

// Inserting sample data for the chart.
sheet.Range["A1"].Text = "Month";
sheet.Range["B1"].Text = "Product A";
sheet.Range["C1"].Text = "Product B";

// Months
sheet.Range["A2"].Text = "Jan";

```

```

sheet.Range["A3"].Text = "Feb";
sheet.Range["A4"].Text = "Mar";
sheet.Range["A5"].Text = "Apr";
sheet.Range["A6"].Text = "May";

// Random Data.
Random r = new Random();
for(int i=2;i<=6;i++)
{
    for(int j=2;j<=3;j++)
    {
        sheet.Range[i,j].Number = r.Next(0,500);
    }
}

// Embedded Chart.
IChartShape chart = sheet.Charts.Add();

// Setting chart type.
chart.ChartType = ExcelChartType.Line;

// Setting the Chart Title.
chart.ChartTitle = "Product Sales comparison";

// Product A.
IChartSerie productA = chart.Series.Add("ProductA");
productA.Values = sheet.Range["B2:B6"];
productA.CategoryLabels = sheet.Range["A2:A6"];

// Product B.
IChartSerie productB = chart.Series.Add("ProductB");
productB.Values = sheet.Range["C2:C6"];
productB.CategoryLabels = sheet.Range["A2:A6"];

```

[VB.NET]

```

' Inserting sample data for the chart.
sheet.Range("A1").Text = "Month"
sheet.Range("B1").Text = "Product A"
sheet.Range("C1").Text = "Product B"

' Months
sheet.Range("A2").Text = "Jan"
sheet.Range("A3").Text = "Feb"
sheet.Range("A4").Text = "Mar"

```

```
sheet.Range("A5").Text = "Apr"
sheet.Range("A6").Text = "May"

' Random Data.
Dim r As Random = New Random
For i As Integer = 2 To 6
    For j As Integer = 2 To 3
        sheet.Range(i, j).Number = r.Next(0, 500)
    Next j
Next i

' Embedded Chart.
Dim chart As IChartShape = sheet.Charts.Add()

' Setting chart type.
chart.ChartType = ExcelChartType.Line

' Setting Chart Title.
chart.ChartTitle = "Product Sales comparison"

' Product A.
Dim productA As IChartSerie = chart.Series.Add("ProductA")
productA.Values = sheet.Range("B2:B6")
productA.CategoryLabels = sheet.Range("A2:A6")

' Product B.
Dim productB As IChartSerie = chart.Series.Add("ProductB")
productB.Values = sheet.Range("C2:C6")
productB.CategoryLabels = sheet.Range("A2:A6")
```

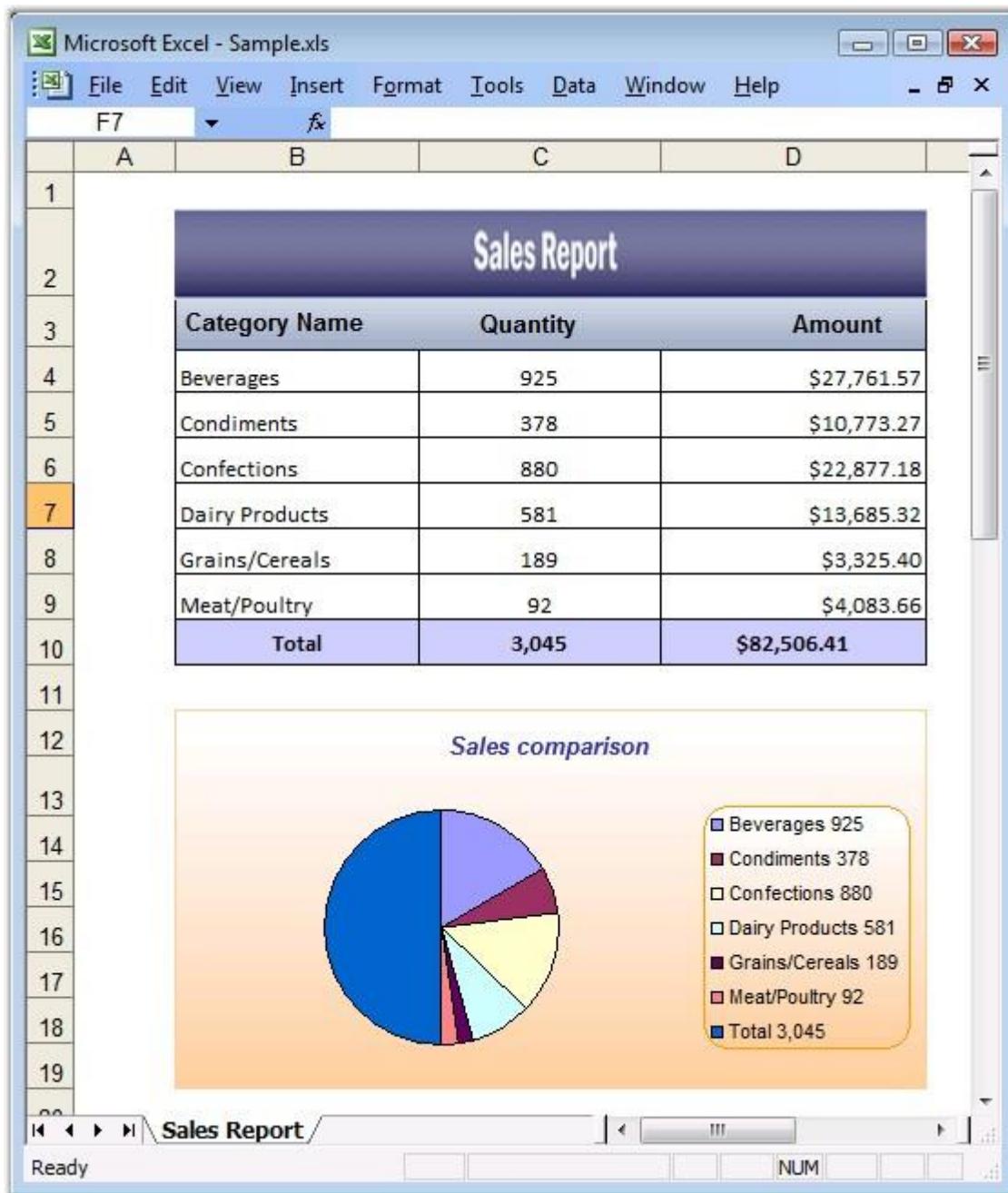


Figure 76: XlsIO with Embedded Chart

4.2.2.2 Chart Worksheet

The **IChart** interface represents the in-memory representation of the Chart Worksheet in an Excel workbook. Formatting is similar to the one discussed in the [Embedded Chart](#) in the previous section.

[C#]

```
// Entering the Data for the chart.
sheet.Range["A1"].Text = "Texas books Unit sales";
sheet.Range["A1:D1"].Merge();
sheet.Range["A1"].CellStyle.Font.Bold = true;

sheet.Range["B2"].Text = "Jan";
sheet.Range["C2"].Text = "Feb";
sheet.Range["D2"].Text = "Mar";

sheet.Range["A3"].Text = "Austin";
sheet.Range["A4"].Text = "Dallas";
sheet.Range["A5"].Text = "Houston";
sheet.Range["A6"].Text = "San Antonio";

sheet.Range["B3"].Number = 53.75;
sheet.Range["B4"].Number = 52.85;
sheet.Range["B5"].Number = 59.77;
sheet.Range["B6"].Number = 96.15;

sheet.Range["C3"].Number = 79.79;
sheet.Range["C4"].Number = 59.22;
sheet.Range["C5"].Number = 10.09;
sheet.Range["C6"].Number = 73.02;

sheet.Range["D3"].Number = 26.72;
sheet.Range["D4"].Number = 33.71;
sheet.Range["D5"].Number = 45.81;
sheet.Range["D6"].Number = 12.17;

// Adding a New chart to the Existing Worksheet.
IChart chart = workbook.Charts.Add();
chart.DataRange = sheet.Range["B3:D6"];
chart.Name = "ChartWorksheet";
chart.PrimaryCategoryAxis.Title = "City";
chart.PrimaryValueAxis.Title = "Sales (in Dollars)";
chart.ChartTitle = "Texas Books Unit Sales";

// Setting the Series Names in a Legend.
IChartSerie serieOne = chart.Series[0];
serieOne.Name = "Jan";
```

```
IChartSerie seriethree = chart.Series[2];
seriethree.Name = "March";
```

[VB.NET]

```
' Entering the Data for the chart.
sheet.Range("A1").Text = "Texas books Unit sales"
sheet.Range("A1:D1").Merge()
sheet.Range("A1").CellStyle.Font.Bold = True

sheet.Range("B2").Text = "Jan"
sheet.Range("C2").Text = "Feb"
sheet.Range("D2").Text = "Mar"

sheet.Range("A3").Text = "Austin"
sheet.Range("A4").Text = "Dallas"
sheet.Range("A5").Text = "Houston"
sheet.Range("A6").Text = "San Antonio"

sheet.Range("B3").Number = 53.75
sheet.Range("B4").Number = 52.85
sheet.Range("B5").Number = 59.77
sheet.Range("B6").Number = 96.15

sheet.Range("C3").Number = 79.79
sheet.Range("C4").Number = 59.22
sheet.Range("C5").Number = 10.09
sheet.Range("C6").Number = 73.02

sheet.Range("D3").Number = 26.72
sheet.Range("D4").Number = 33.71
sheet.Range("D5").Number = 45.81
sheet.Range("D6").Number = 12.17

' Adding a New chart to the Existing Worksheet.
Dim chart As IChart = workbook.Charts.Add()
chart.DataRange = sheet.Range("B3:D6")
chart.Name = "ChartWorksheet"
chart.PrimaryCategoryAxis.Title = "City"
chart.PrimaryValueAxis.Title = "Sales (in Dollars)"
chart.ChartTitle = "Texas Books Unit Sales"

' Setting the Serie Names in a Legend.
```

```
Dim serieOne As IChartSerie = chart.Series(0)
serieOne.Name = "Jan"
Dim serietwo As IChartSerie = chart.Series(1)
serietwo.Name = "Feb"
Dim seriethree As IChartSerie = chart.Series(2)
seriethree.Name = "March"
```



Figure 77: XlsIO with Chart Worksheet

4.2.2.3 Edit Charts

The properties of the Chart in an existing workbook can be opened and edited by using Essential XlsIO. Following code example illustrates how an existing workbook with a chart is opened, and how the chart properties are modified.

[C#]

```
// Opening the Existing Worksheet from a Workbook.  
IWorkbook workbook =  
    application.Workbooks.Open(@"..\..\..\..\..\Data>EditChartsTemplate.xls");  
  
// The first worksheet object in the worksheets collection is accessed.  
IWorksheet sheet = workbook.Worksheets[0];  
  
// Editing the Existing Chart.  
IChart chart = sheet.Charts[0];  
chart.ChartTitle = "Texas Books Unit Sales";  
chart.PrimaryCategoryAxis.Title = "City";  
chart.PrimaryValueAxis.Title = "Sales (in Dollars)";  
chart.Legend.Position = ExcelLegendPosition.Top;  
  
// Setting the Series Names in a Legend.  
IChartSerie serieOne = chart.Series[0];  
serieOne.Name = "Jan";  
IChartSerie seriетwo = chart.Series[1];  
serietwo.Name = "Feb";  
IChartSerie seriethree = chart.Series[2];  
seriethree.Name = "March";  
  
// Setting the Title Area text.  
IChartTextArea Area = chart.ChartTitleArea;  
Area.Bold = true;  
Area.Underline = ExcelUnderline.Single;  
  
// Setting the Minimum and Maximum Value for Value Axis.  
chart.PrimaryValueAxis.MinimumValue = 10;  
chart.PrimaryValueAxis.MaximumValue = 100;  
  
// Setting the Height of the chart.  
chart.Height = 1/10;  
  
// Setting the Width of the chart.  
chart.Width = 1/72;  
  
// Saving the workbook to disk.  
workbook.SaveAs("Sample.xls");  
  
// Closing the workbook.  
workbook.Close();
```

[VB.NET]

```
' Opening the Existing Worksheet from a Workbook.  
Dim workbook As IWorkbook =  
application.Workbooks.Open("../..\..\..\..\Data>EditChartsTemplate.xls")  
  
' The first worksheet object in the worksheets collection is accessed.  
Dim sheet As IWorksheet = workbook.Worksheets(0)  
  
' Editing the Existing Chart.  
Dim chart As IChart = sheet.Charts(0)  
chart.ChartTitle = "Texas Books Unit Sales"  
chart.PrimaryCategoryAxis.Title = "City"  
chart.PrimaryValueAxis.Title = "Sales (in Dollars)"  
chart.Legend.Position = ExcelLegendPosition.Top  
  
' Setting the Series Names in a Legend.  
Dim serieOne As IChartSerie = chart.Series(0)  
serieOne.Name = "Jan"  
Dim serietwo As IChartSerie = chart.Series(1)  
serietwo.Name = "Feb"  
Dim seriethree As IChartSerie = chart.Series(2)  
seriethree.Name = "March"  
  
' Setting the Title Area text.  
Dim Area As IChartTextArea = chart.ChartTitleArea  
Area.Bold = True  
Area.Underline = ExcelUnderline.Single  
  
' Setting the Minimum and Maximum Value for the Value Axis.  
chart.PrimaryValueAxis.MinimumValue = 10  
chart.PrimaryValueAxis.MaximumValue = 100  
  
' Setting the Height of the chart.  
chart.Height = 1 / 10  
  
' Setting the Width of the chart.  
chart.Width = 1 / 72  
  
' Saving the workbook to disk.  
workbook.SaveAs("Sample.xls")  
  
' Closing the workbook.  
workbook.Close()
```

4.2.2.3.1 Resizing and Positioning of Chart Elements

Chart elements such as chart title, legend, and plot area, can be positioned and resized easily as needed. To avoid spacing problems caused by the lengthy chart titles, plot area, or legends, the user can change the way that how these elements have to be positioned in the chart. You can also specify the exact position of the chart elements in the chart area.

The following code examples illustrate how to resize and position the chart elements.

[C#]

```
'Step 1: Instantiate the spreadsheet creation engine
ExcelEngine excelEngine = new ExcelEngine();

'Step 2: Instantiate the excel application object
IApplication application = excelEngine.Excel;
IWorkbook workbook =
application.Workbooks.Open(@"../../Data/Sample.xlsx",
ExcelOpenType.Automatic);

IWorksheet sheet = workbook.Worksheets[0];
IChart chart = sheet.Charts[0];

//Manually positioning chart title

//Edge: Specifies that the width or Height will be interpreted as right
or bottom of the chart element

//Factor: Specifies that the width or Height will be interpreted as the
width or height of the chart element

chart.PlotArea.Layout.LeftMode = LayoutModes.edge;
chart.PlotArea.Layout.TopMode = LayoutModes.edge;
//Value in points should not be a negative value if LayoutMode is Edge
//It can be a negative value if the LayoutMode is Factor.

chart.ChartTitleArea.Layout.Left = 1;
chart.ChartTitleArea.Layout.Top = 20;

//Manually positioning and resizing chart plot area

//Inner: Specifies that the plot area size will determine the size of
the plot area, without including the tick marks and axis labels

chart.PlotArea.Layout.LayoutTarget = LayoutTargets.inner;
chart.PlotArea.Layout.LeftMode = LayoutModes.edge;
chart.PlotArea.Layout.TopMode = LayoutModes.edge;
chart.PlotArea.Layout.Left = 50;
chart.PlotArea.Layout.Top = 75;
```

```

chart.PlotArea.Layout.Width = 300;
chart.PlotArea.Layout.Height = 200;
//Manually positioning and resizing chart legend
chart.Legend.Layout.LeftMode = LayoutModes.edge;
chart.Legend.Layout.TopMode = LayoutModes.edge;
chart.Legend.Layout.Left = 400;
chart.Legend.Layout.Top = 150;
chart.Legend.Layout.Width = 50;
chart.Legend.Layout.Height = 100;
//Saving workbook
workbook.Version = ExcelVersion.Excel2007;
string fileName = @"../../Output/SampleOutput.xlsx";
workbook.SaveAs(fileName);
workbook.Close();
excelEngine.Dispose();

```

[VB]

```

'Step 1: Instantiate the spreadsheet creation engine
Dim excelEngine As ExcelEngine = New ExcelEngine()
'Step 2: Instantiate the excel application object
Dim application As IApplication = excelEngine.Excel
Dim workbook As IWorkbook =
application.Workbooks.Open("../../Data/Sample.xlsx",
ExcelOpenType.Automatic)
Dim worksheet As IWorksheet = workbook.Worksheets(0)
Dim chart As IChart = worksheet.Charts(0)
'Manually positioning chart title
//Edge: Specifies that the Width or Height will be interpreted as the right
or bottom of the chart element
//Factor: Specifies that the Width or Height will be interpreted as the
width or height of the chart element
chart.PlotArea.Layout.LeftMode = LayoutModes.edge
chart.PlotArea.Layout.TopMode = LayoutModes.edge
//Value in points should not be negative value if LayoutMode is Edge
//It can be a negative value if LayoutMode is Factor.

```

```

chart.ChartTitleArea.Layout.Left = 1
chart.ChartTitleArea.Layout.Top = 20
'Manually positioning and resizing chart plot area
'Inner: Specifies that the plot area size shall determine the size of the
plot area without including the tick marks and axis labels
chart.PlotArea.Layout.LayoutTarget = LayoutTargets.inner
chart.PlotArea.Layout.LeftMode = LayoutModes.edge
chart.PlotArea.Layout.TopMode = LayoutModes.edge
'Floating point value between -1 and 1
chart.PlotArea.Layout.Left = 50
chart.PlotArea.Layout.Top = 75
chart.PlotArea.Layout.Width = 300
chart.PlotArea.Layout.Height = 200
'Manually positioning and resizing chart legend
chart.Legend.Layout.LeftMode = LayoutModes.edge
chart.Legend.Layout.TopMode = LayoutModes.edge
chart.Legend.Layout.Left = 400
chart.Legend.Layout.Top = 150
chart.Legend.Layout.Width = 50
chart.Legend.Layout.Height = 100
'Saving the workbook to disk
workbook.SaveAs(fileName)
'Close the workbook
workbook.Close()
'No exception will be thrown if there are unsaved workbooks
excelEngine.ThrowNotSavedOnDestroy = False
excelEngine.Dispose()

```

4.2.2.3.2 Rich-Text Formatting for Chart Elements

Chart titles and data labels that are not linked to the worksheet data can be edited directly on the chart. You can also use rich-text formatting for the chart elements to enhance their appearance.

The following code examples explain how to apply rich-text formatting to the chart elements.

[C#]

```
'Step 1: Instantiate the spreadsheet creation engine
ExcelEngine excelEngine = new ExcelEngine();

'Step 2: Instantiate the excel application object
IApplication application = excelEngine.Excel;
IWorkbook workbook = application.Workbooks.Open(@"../../Data/Sample.xlsx",
ExcelOpenType.Automatic);

IWorksheet sheet = workbook.Worksheets[0];
IChart chart = sheet.Charts[0];
chart.ChartTitle = "Title with a variable font style";
//Rich-text in chart title
//Create a new font with the following properties
IFont redFont = workbook.CreateFont();
redFont.Bold = true;
redFont.Italic = false;
redFont.Size = 18;
redFont.FontName = "Georgia";
redFont.Color = ExcelKnownColors.Red;
//SetFont(int startIndex, int endIndex, IFont font)
IChartRichTextString chartRTS = chart.ChartTitleArea.RichText;
chartRTS.SetFont(0, 12, redFont);
//Rich-Text in data label
//Create another font with the following properties
IFont greenFont = workbook.CreateFont();
greenFont.Bold = true;
greenFont.Italic = false;
greenFont.Size = 17;
greenFont.FontName = "Times New Roman";
greenFont.Color = ExcelKnownColors.Green;
//SetFont(int startIndex, int endIndex, IFont font)
chart.Series[0].DataPoints[0].DataLabels.RichText.SetFont(0, 0, greenFont);
//Create a third font with the following properties
IFont blueFont = workbook.CreateFont();
blueFont.Bold = true;
```

```

blueFont.Italic = true;
blueFont.Size = 10;
blueFont.FontName = "Calibri";
blueFont.Color = ExcelKnownColors.Blue;
//SetFont(int startIndex, int endIndex, IFont font)
chart.Series[0].DataPoints[0].DataLabels.RichText.SetFont(1, 2, blueFont);
//Save workbook
workbook.Version = ExcelVersion.Excel2007;
string fileName = @"../../Output/SampleOutput.xlsx";
workbook.SaveAs(fileName);
workbook.Close();
excelEngine.Dispose();

```

[VB]

```

'Step 1: Instantiate the spreadsheet creation engine.
Dim excelEngine As ExcelEngine = New ExcelEngine()
'Step 2 : Instantiate the excel application object.
Dim application As IApplication = excelEngine.Excel
Dim workbook As IWorkbook =
application.Workbooks.Open("../../Data/Sample.xlsx",
ExcelOpenType.Automatic)
Dim worksheet As IWorksheet = workbook.Worksheets(0)
Dim chart As IChart = worksheet.Charts(0)
chart.ChartTitle = "Title with a variable font style"
'Rich-text in chart title
'Create a new font with the following properties
Dim redFont As IFont = workbook.CreateFont()
redFont.Bold = True
redFont.Italic = False
redFont.Size = 18
redFont.FontName = "Georgia"
redFont.Color = ExcelKnownColors.Red
Dim chartRTS As IChartRichTextString = chart.ChartTitleArea.RichText
SetFont(int startIndex, int endIndex, IFont font)
chartRTS.SetFont(0, 12, redFont)

```

```

'Rich-text in data label

'Create a second font with the following properties
Dim greenFont As IFont = workbook.CreateFont()

greenFont.Bold = True
greenFont.Italic = False
greenFont.Size = 18
greenFont.FontName = "Georgia"
greenFont.Color = ExcelKnownColors.Red

'SetFont(int startIndex, int endIndex, IFont font)
chart.Series(0).DataPoints(0).DataLabels.RichText.SetFont(0, 0, greenFont)

'Rich-text in data label

'Create a third font with the following properties
Dim blueFont As IFont = workbook.CreateFont()

blueFont.Bold = True
blueFont.Italic = False
blueFont.Size = 18
blueFont.FontName = "Georgia"
blueFont.Color = ExcelKnownColors.Red

'SetFont(int startIndex, int endIndex, IFont font)
chart.Series(0).DataPoints(0).DataLabels.RichText.SetFont(0, 0, blueFont)

'Save the workbook to disk
workbook.SaveAs(fileName)

'Close the workbook
workbook.Close()

'No exception will be thrown if there are un-saved workbooks
excelEngine.ThrowNotSavedOnDestroy = False
excelEngine.Dispose()

```

4.2.2.3.3 3-D Chart Wall Settings

Essential XlsIO allows users to modify side wall, back wall, and floor settings of a 3-D chart. The following code example explains how to apply these settings to the 3-D chart.

[C#]

```
//Instantiate the spreadsheet creation engine
```

```
ExcelEngine excelEngine = new ExcelEngine();

//Create a new workbook (similar to creating a new workbook
in Excel)

//Open a workbook including data
IWorkbook workbook =
excelEngine.Excel.Workbooks.Open(@"EmbeddedChart.xlsx");

//The first worksheet object in the worksheet collection is
accessed.

IWorksheet sheet = workbook.Worksheets[0];
sheet.Name = "Sample";

//Add a new chart to the existing worksheet
IChartShape chart = workbook.Worksheets[0].Charts[0];

//Set chart series
IChartSerie serieOne = chart.Series[0];
IChartSerie serieTwo = chart.Series[1];

//Set fill type of chart back wall
chart.BackWall.Fill.FillType = ExcelFillType.Gradient;

//Set fill options for the back wall
chart.BackWall.Fill.GradientColorType =
ExcelGradientColor.TwoColor;
chart.BackWall.Fill.GradientStyle =
ExcelGradientStyle.Diagonal_Down;

//Set the foreground and background color of the back wall
chart.BackWall.Fill.ForeColor =
System.Drawing.Color.WhiteSmoke;
chart.BackWall.Fill.BackColor =
System.Drawing.Color.LightBlue;

//Set the border line color of the back wall
```

```
chart.BackWall.Border.LineColor =
System.Drawing.Color.Wheat;

//Set thickness of the back wall
chart.BackWall.Thickness = 10;

//Set fill type of side wall
chart.SideWall.Fill.FillType = ExcelFillType.SolidColor;

//Set the foreground and background colors of the side wall
chart.SideWall.Fill.BackColor = System.Drawing.Color.White;
chart.SideWall.Fill.ForeColor = System.Drawing.Color.White;

//Set border line color of the side wall
chart.SideWall.Border.LineColor =
System.Drawing.Color.Beige;

//Set fill type of floor
chart.Floor.Fill.FillType = ExcelFillType.Pattern;

//Set pattern type of the floor
chart.Floor.Fill.Pattern = ExcelGradientPattern.Pat_Divot;

//Set the foreground and background color of the floor
chart.Floor.Fill.ForeColor = System.Drawing.Color.Blue;
chart.Floor.Fill.BackColor = System.Drawing.Color.White;

//Set thickness of the floor
chart.Floor.Thickness = 3;

//Show value as data labels
serieOne.DataPoints.DefaultDataPoint.DataLabels.IsValue =
true;
serieTwo.DataPoints.DefaultDataPoint.DataLabels.IsValue =
true;
```

```

//Set embedded chart positions
chart.TopRow = 2;
chart.BottomRow = 30;
chart.LeftColumn = 5;
chart.RightColumn = 18;
serieTwo.Name = "Temperature, deg.F";

//Set chart legends
chart.Legend.Position = ExcelLegendPosition.Right;
chart.Legend.IsVerticalLegend = false;

//Save the workbook
workbook.SaveAs("Sample.xlsx");

```

[VB.NET]

```

'Instantiate the spreadsheet creation engine
Dim excelEngine As New ExcelEngine()

'Create a new workbook (similar to creating a new
workbook in Excel)
'Open a workbook including data
Dim workbook As IWorkbook =
excelEngine.Excel.Workbooks.Open("EmbeddedChart.xlsx")

'The first worksheet object in the worksheet collection is
accessed.
Dim sheet As IWorksheet = workbook.Worksheets(0)
sheet.Name = "Sample"

```

```
'Add a new chart to the existing worksheet
Dim chart As IChartShape = workbook.Worksheets(0).Charts(0)

' Set chart series
Dim serieOne As IChartSerie = chart.Series(0)
Dim serieTwo As IChartSerie = chart.Series(1)

' Set fill type of back wall
chart.BackWall.Fill.FillType = ExcelFillType.Gradient

' Set fill options for the back wall
chart.BackWall.Fill.GradientColorType =
ExcelGradientColor.TwoColor
chart.BackWall.Fill.GradientStyle =
ExcelGradientStyle.Diagonal_Down

'set the foreground and background color of the back wall
chart.BackWall.Fill.ForeColor =
System.Drawing.Color.WhiteSmoke
chart.BackWall.Fill.BackColor =
System.Drawing.Color.LightBlue

' Set border line color of the back wall
chart.BackWall.Border.LineColor =
System.Drawing.Color.Wheat

' Set thickness of the back wall
chart.BackWall.Thickness = 10

' Set fill type of side wall
chart.SideWall.Fill.FillType = ExcelFillType.SolidColor

' Set foreground and backcolor of the side wall
chart.SideWall.Fill.BackColor = System.Drawing.Color.White
chart.SideWall.Fill.ForeColor = System.Drawing.Color.White
```

```
'Set border line color of the side wall
chart.SideWall.Border.LineColor =
System.Drawing.Color.Beige

'Set fill type of floor
chart.Floor.Fill.FillType = ExcelFillType.Pattern

'Set pattern type of the floor
chart.Floor.Fill.Pattern = ExcelGradientPattern.Pat_Divot

'Set foreground and background color of the floor
chart.Floor.Fill.ForeColor = System.Drawing.Color.Blue
chart.Floor.Fill.BackColor = System.Drawing.Color.White

'Set thickness of the floor
chart.Floor.Thickness = 3

>Show value as data labels
serieOne.DataPoints.DefaultDataPoint.DataLabels.IsValue =
True
serieTwo.DataPoints.DefaultDataPoint.DataLabels.IsValue =
True

'Set embedded chart positions
chart.TopRow = 2
chart.BottomRow = 30
chart.LeftColumn = 5
chart.RightColumn = 18
serieTwo.Name = "Temperature, deg.F"

'Set chart legends
chart.Legend.Position = ExcelLegendPosition.Right
chart.Legend.IsVerticalLegend = False
```

```
' Save the workbook
workbook.SaveAs("Sample.xlsx")
```

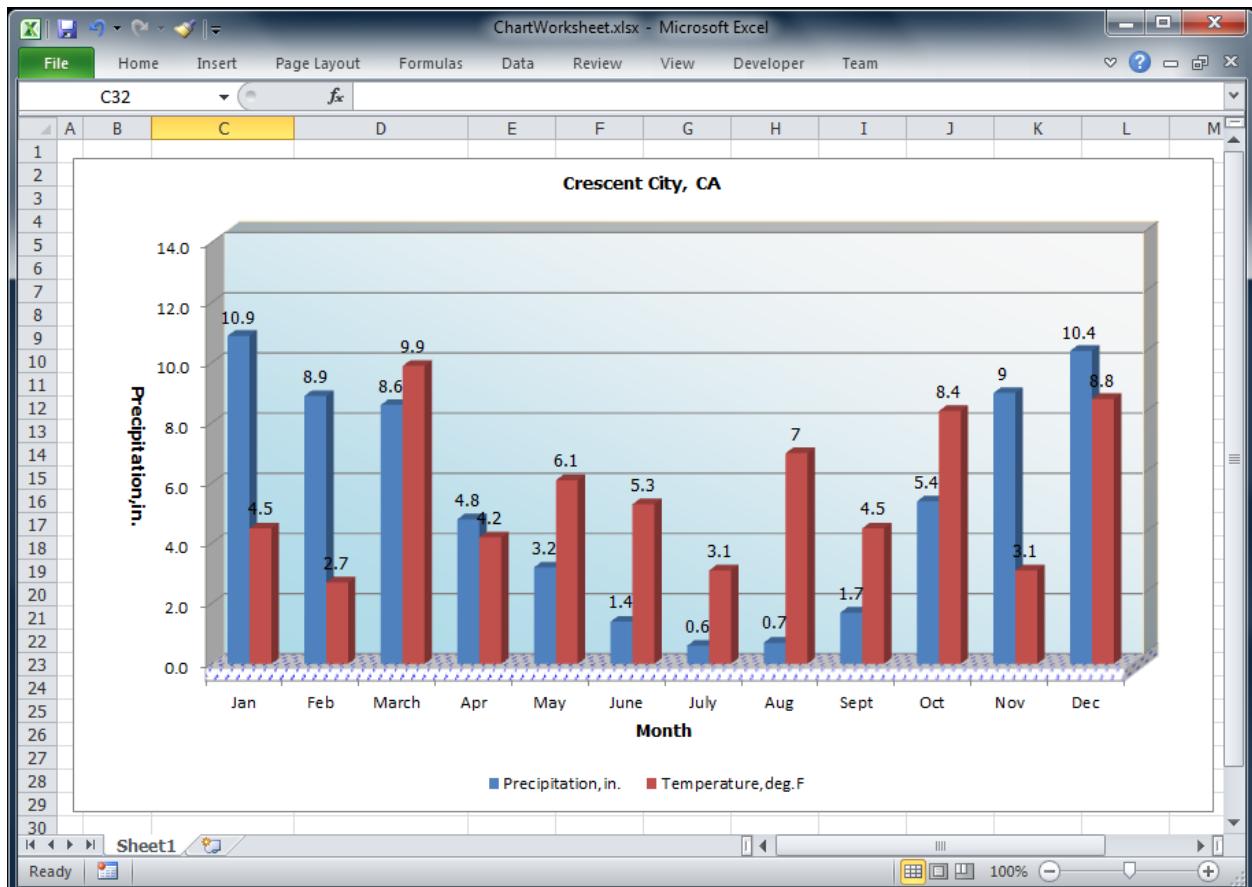


Figure 78: 3-D Chart Applied with Wall Settings

4.2.2.3.4 Filtering Chart Series and Categories

Essential XlsIO supports filtering chart series and categories for Excel 2013 files.

The following code samples explain how to perform filtering for series and categories in a chart.

[C#]

```
//Step 1: Instantiate the spreadsheet creation engine.
ExcelEngine excelEngine = new ExcelEngine();
```

```
//Step 2: Instantiate the Excel application object.  
IApplication application = excelEngine.Excel;  
application.DefaultVersion = ExcelVersion.Excel2010;  
  
//The first worksheet object is created.  
IWorkbook workbook = application.Workbooks.Create(1);  
IWorksheet sheet = workbook.Worksheets[0];  
  
//Insert the data in sheet-1.  
sheet.Range["B1"].Text = "Product-A";  
sheet.Range["C1"].Text = "Product-B";  
sheet.Range["D1"].Text = "Product-C";  
sheet.Range["E1"].Text = "Product-D";  
sheet.Range["A2"].Text = "Jan";  
sheet.Range["A3"].Text = "Feb";  
sheet.Range["A4"].Text = "Mar";  
sheet.Range["A5"].Text = "Apr";  
sheet.Range["A6"].Text = "May";  
sheet.Range["B2"].Number = 25;  
sheet.Range["B3"].Number = 20;  
sheet.Range["B4"].Number = 35;  
sheet.Range["B5"].Number = 67;  
sheet.Range["B6"].Number = 23;  
sheet.Range["C2"].Number = 35;  
sheet.Range["C3"].Number = 25;  
sheet.Range["C4"].Number = 14;  
sheet.Range["C5"].Number = 78;  
sheet.Range["C6"].Number = 45;  
sheet.Range["D2"].Number = 40;  
sheet.Range["D3"].Number = 55;  
sheet.Range["D4"].Number = 51;  
sheet.Range["D5"].Number = 89;  
sheet.Range["D6"].Number = 64;  
sheet.Range["E2"].Number = 67;
```

```
sheet.Range["E3"].Number = 44;
sheet.Range["E4"].Number = 23;
sheet.Range["E5"].Number = 53;
sheet.Range["E6"].Number = 55;

//Chart is added in sheet-1.
IChartShape chart = sheet.Charts.Add();

//Setting the chart DataRange.
chart.DataRange = sheet.Range["A1:E6"];

//Change the chart seriesInRows.
chart.IsSeriesInRows = false;

//Setting the Chart Type.
chart.ChartType = ExcelChartType.Column_Clustered_3D;

//Get the series and categories from the chart.
IChartSeries series = chart.Series;
IChartCategories categories = chart.Categories;

//Set the Backwall fill option.
chart.BackWall.Fill.FillType = ExcelFillType.Gradient;
chart.BackWall.Fill.GradientColorType = ExcelGradientColor.TwoColor;
chart.BackWall.Fill.GradientStyle = ExcelGradientStyle.Diagonal_Down;
chart.BackWall.Fill.ForeColor = System.Drawing.Color.WhiteSmoke;
chart.BackWall.Fill.BackColor = System.Drawing.Color.LightBlue;
chart.BackWall.Thickness = 10;

//Set the sidewall foreground and backcolor.
chart.SideWall.Fill.FillType = ExcelFillType.SolidColor;
chart.SideWall.Fill.BackColor = System.Drawing.Color.White;
chart.SideWall.Fill.ForeColor = System.Drawing.Color.White;
```

```
//Set floor fill option.  
chart.Floor.Fill.FillType = ExcelFillType.Pattern;  
chart.Floor.Fill.Pattern = ExcelGradientPattern.Pat_Divot;  
chart.Floor.Fill.ForeColor = System.Drawing.Color.Blue;  
chart.Floor.Fill.BackColor = System.Drawing.Color.White;  
chart.Floor.Thickness = 3;  
  
//First series and second category of the chart is filtered.  
series[0].IsFiltered = true;  
categories[1].IsFiltered = true;  
  
//Series and category names of the chart are filtered.  
chart.SeriesNameLevel = ExcelSeriesNameLevel.SeriesNameLevelNone;  
chart.CategoryLabelLevel =  
ExcelCategoriesLabelLevel.CategoriesLabelLevelNone;  
  
//Chart size setting.  
chart.LeftColumn = 8;  
chart.RightColumn = 16;  
chart.TopRow = 9;  
chart.BottomRow = 27;  
  
//Setting legend.  
chart.Legend.Position = ExcelLegendPosition.Right;  
chart.Legend.IsVerticalLegend = false;  
  
//Save the workbook.  
workbook.SaveAs("Sample.xlsx");
```

[VB]

```
'Step 1: Instantiate the spreadsheet creation engine.  
Dim excelEngine As New ExcelEngine()  
  
'Step 2: Instantiate the Excel application object.  
Dim application As IApplication = excelEngine.Excel  
application.DefaultVersion = ExcelVersion.Excel2010  
  
'The first worksheet object is created.  
Dim workbook As IWorkbook = application.Workbooks.Create(1)  
Dim sheet As IWorksheet = workbook.Worksheets(0)  
  
'Insert the data in sheet-1.  
sheet.Range("B1").Text = "Product-A"  
sheet.Range("C1").Text = "Product-B"  
sheet.Range("D1").Text = "Product-C"  
sheet.Range("E1").Text = "Product-D"  
sheet.Range("A2").Text = "Jan"  
sheet.Range("A3").Text = "Feb"  
sheet.Range("A4").Text = "Mar"  
sheet.Range("A5").Text = "Apr"  
sheet.Range("A6").Text = "May"  
sheet.Range("B2").Number = 25  
sheet.Range("B3").Number = 20  
sheet.Range("B4").Number = 35  
sheet.Range("B5").Number = 67  
sheet.Range("B6").Number = 23  
sheet.Range("C2").Number = 35  
sheet.Range("C3").Number = 25  
sheet.Range("C4").Number = 14  
sheet.Range("C5").Number = 78  
sheet.Range("C6").Number = 45  
sheet.Range("D2").Number = 40  
sheet.Range("D3").Number = 55  
sheet.Range("D4").Number = 51
```

```

sheet.Range("D5").Number = 89
sheet.Range("D6").Number = 64
sheet.Range("E2").Number = 67
sheet.Range("E3").Number = 44
sheet.Range("E4").Number = 23
sheet.Range("E5").Number = 53
sheet.Range("E6").Number = 55

'Chart is added in sheet-1.

Dim chart As IChartShape = sheet.Charts.Add()

'Setting the chart DataRange.
chart.DataRange = sheet.Range("A1:E6")
'Change the chart seriesInRows.
chart.IsSeriesInRows = False

'Setting the chart type.
chart.ChartType = ExcelChartType.Column_Clustered_3D

'Get the series and categories from the chart.
Dim series As IChartSeries = chart.Series
Dim categories As IChartCategories = chart.Categories

'Set the Backwall fill option.
chart.BackWall.Fill.FillType = ExcelFillType.Gradient
chart.BackWall.Fill.GradientColorType = ExcelGradientColor.TwoColor
chart.BackWall.Fill.GradientStyle = ExcelGradientStyle.Diagonal_Down
chart.BackWall.Fill.ForeColor = System.Drawing.Color.WhiteSmoke
chart.BackWall.Fill.BackColor = System.Drawing.Color.LightBlue
chart.BackWall.Thickness = 10

'Set the sidewall foreground and backcolor.
chart.SideWall.Fill.FillType = ExcelFillType.SolidColor
chart.SideWall.Fill.BackColor = System.Drawing.Color.White

```

```
chart.SideWall.Fill.ForeColor = System.Drawing.Color.White

' Set floor fill option.
chart.Floor.Fill.FillType = ExcelFillType.Pattern
chart.Floor.Fill.Pattern = ExcelGradientPattern.Pat_Divot
chart.Floor.Fill.ForeColor = System.Drawing.Color.Blue
chart.Floor.Fill.BackColor = System.Drawing.Color.White
chart.Floor.Thickness = 3

' First series and second category of the chart is filtered.
series(0).IsFiltered = True
categories(1).IsFiltered = True

' Series and category names of the chart are filtered.
chart.SeriesNameLevel = ExcelSeriesNameLevel.SeriesNameLevelNone
chart.CategoryLabelLevel = ExcelCategoriesLabelLevel.CategoriesLabelLevelNone

' Setting embedded chart position.
chart.LeftColumn = 8
chart.RightColumn = 16
chart.TopRow = 9
chart.BottomRow = 27

' Setting legend.
chart.Legend.Position = ExcelLegendPosition.Right
chart.Legend.IsVerticalLegend = False

' Save the workbook.
workbook.SaveAs("Sample.xlsx")
```

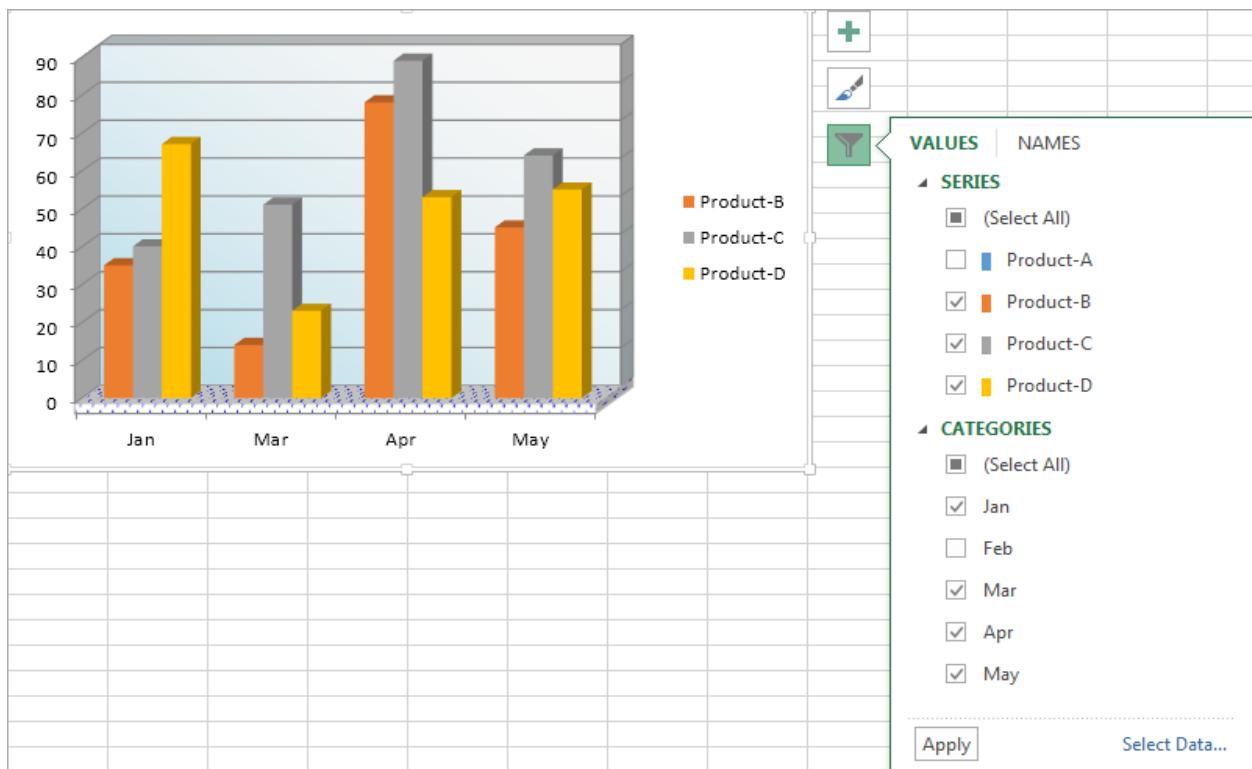


Figure 79: XlsIO with First Series and Second Category Filtered

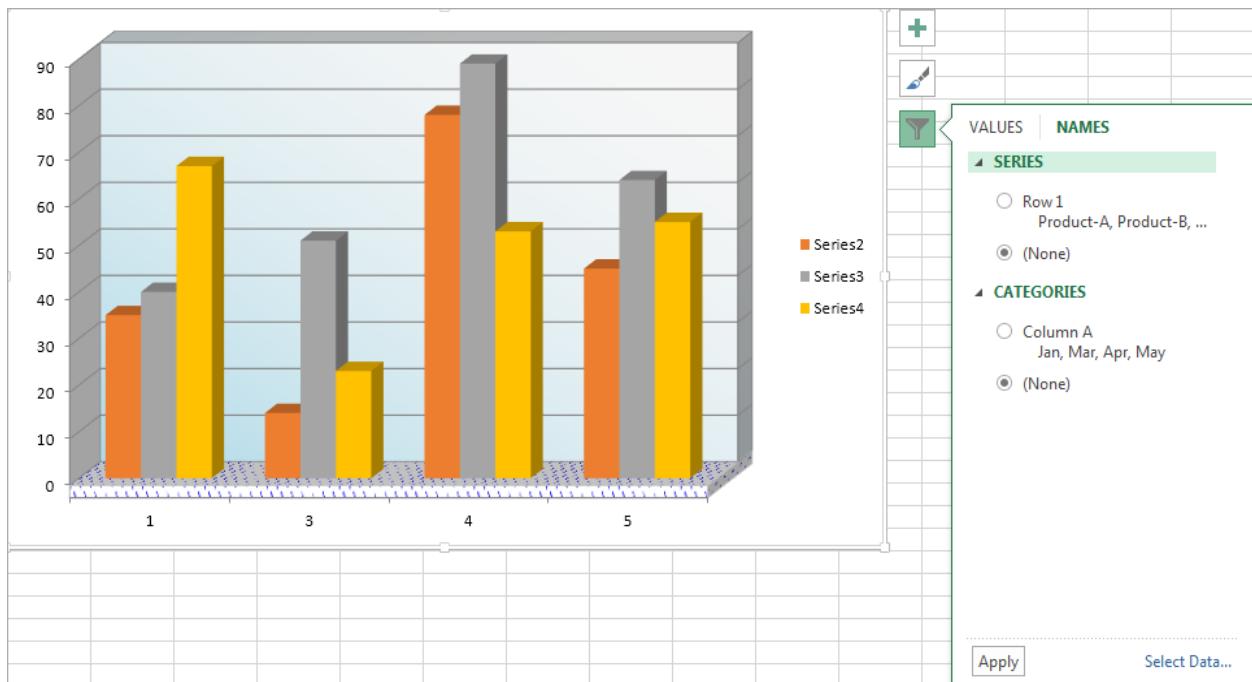


Figure 80: XlsIO with Series and Category Name Filtered

4.2.2.4 Sparklines

Sparklines Creation Using MS Excel 2010:

In MS Excel 2010, the Sparklines can be inserted by selecting any of the sparklines type from the Insert menu.

In MS Excel 2010, click Insert Menu. Select any of the Sparklines type.

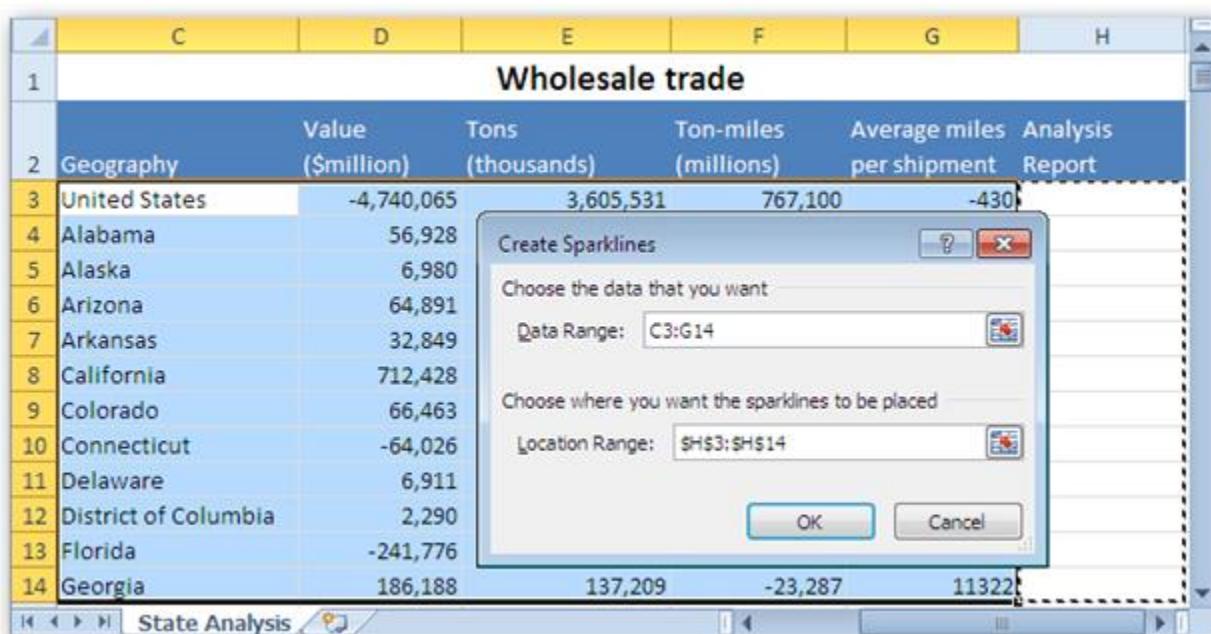


Figure 81: Create Sparklines Dialog Box

MS Excel 2010 allows you to select the range of data for the Sparklines creation. It also allows you to choose where the sparklines can be placed.



Figure 82: Sparklines Tool

The Sparklines appear once you select the data range and the location range. Now you can customize the appearance of Sparklines in terms of color, style etc. A group of Sparkline tools are available on the ribbon to change the high point, low point, color, edit the sparkline data etc.

Sparkline Creation Using XlsIO:

XlsIO provides support for creation of Sparklines by using simple APIs.

- **ISparklineGroups** interface caches the SparklineGroup that needs to be added to the Spreadsheet.
- **ISparklineGroup** represents Sparklines in object, and has properties that allows to customize it.
- **ISparklines** interface returns the collection of Sparkline present in a Worksheet.
- **ISparkline** represents a sparkline in the Sparklines. Currently, XlsIO supports all the three types of sparklines- Line, Column, Win/Loss which are supported in Excel 2010.

Following code example illustrates how to create Sparklines by using XlsIO.

[C#]

```
ISparklineGroup sparklineGroup = sheet.SparklineGroups.Add();

sparklineGroup.SparklineType = SparklineType.Line;

ISparklines sparklines = sparklineGroup.Add();

IRange dataRange = sheet.Range["D6:G17"];
IRange referenceRange = sheet.Range["H6:H17"];

sparklines.Add(dataRange, referenceRange);
```

[VB]

```
Dim sparklineGroup As ISparklineGroup = sheet.SparklineGroups.Add()
sparklineGroup.SparklineType = SparklineType.Line

Dim sparklines As ISparklines = sparklineGroup.Add()

Dim dataRange As IRange = sheet.Range("D6:G17")
Dim referenceRange As IRange = sheet.Range("H6:H17")

sparklines.Add(dataRange, referenceRange)
```

Sparklines Options by XlsIO:

MS Excel 2010 provides various options through Sparklines tool ribbon in order to customize the appearance of Sparklines. See [figure 2](#)

Type:

In MS Excel 2010, click Design and then Type in order to customize the Sparkline type for the current Sparklines. XlsIO provides an equivalent API to perform this with simple properties as follows.

[C#]

```
sparklineGroup.SparklineType = SparklineType.Line
```

Show:

In MS Excel 2010, click Design and then Show in order to customize the view of the Sparklines with high point, low point, first point, last point, negative point, markers. XlsIO provides an equivalent API to perform this with simple properties as follows.

Known Limitations: The Markers can be applied only for the line sparkline type.

[C#]

```
sparklineGroup.ShowFirstPoint = true;
sparklineGroup.ShowLastPoint = true;
sparklineGroup.ShowHighPoint = true;
sparklineGroup.ShowLowPoint = true;
sparklineGroup.ShowMarkers = true;
sparklineGroup.ShowNegativePoint = true;
```

Sparkline Color:

The appearance of the Sparklines can be customized by applying colors. Click Design and then select Style. Choose the Sparkline color option in order to customize the Sparklines. XlsIO provides an equivalent API to perform this with simple property as follows.

[C#]

```
sparklineGroup.SparklineColor = Color.Blue;
```

Marker Color:

The appearance of points in the sparklines can be customized by applying colors to it. Click Design and then select Style. Choose the Marker Color option to customize the appearance of points in the Sparklines. XlsIO provides an equivalent API to perform this with simple properties as follows.

[C#]

```
sparklineGroup.FirstPointColor = Color.Green;
sparklineGroup.LastPointColor = Color.DarkOrange;
sparklineGroup.HighPointColor = Color.DarkBlue;
sparklineGroup.LowPointColor = Color.DarkViolet;
sparklineGroup.MarkersColor = Color.Black;
```

Edit Group Data and Location:

MS Excel provides an option to edit the location and group data of an existing Sparklines, by which you can assign a new location or group data for an existing sparklines. XlsIO provides an equivalent API to perform this functionality.

[C#]

```
sparklines.RefreshRanges(dataRange, referenceRange);
```

Line Weight:

MS Excel 2010 provides an exclusive option to customize the Line Weight of the Line Sparkline type. XlsIO provides an API to perform this functionality.

[C#]

```
sparklineGroup.LineWeight = 1.0;
```

Known Limitations: The Line weight can be applied only for the line sparkline type.

Hidden and Empty Cell Settings:

Normally, in a sparkline group data there is a possibility of an empty cell or an hidden cell. MS Excel 2010 provides a dialog box to

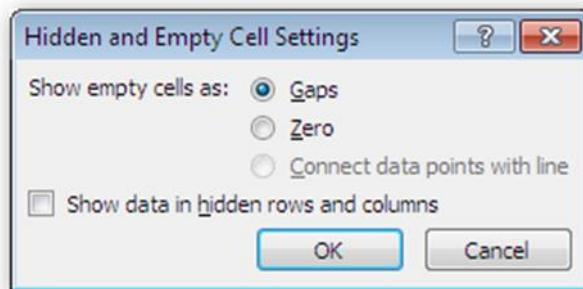


Figure 83: Hidden and Empty cells Settings

XlsIO provides a simple API to implement the above functionality. This is illustrated in the following code.

[C#]

```
sparklineGroup.DisplayEmptyCellsAs = SparklineEmptyCells.Gaps;  
sparklineGroup.DisplayHiddenRC = true;
```

Display Axis:

MS Excel 2010 provides an option to display the axis for the sparklines types. This is illustrated in the following code.

[C#]

```
sparklineGroup.DisplayAxis = true;
```

Plot Right to Left:

The plotting of Sparklines is done from left to right by default. There is an option available in the Sparkline tools to customize the plotting nature from right to left. XlsIO provides a simple API to perform this functionality.

[C#]

```
sparklineGroup.PlotRightToLeft = true;
```

Clear:

XlsIO provides an API to clear the selected Sparklines within the sparkline groups and also the selected sparklinegroup within the excel spreadsheet. This is illustrated in the following code.

[C#]

```
//Clears the sparkline group from the sheet.  
sheet.SparklineGroups.Remove(sparklineGroup);  
//Clears the Sparkline from the sparklines.  
sparklines.Remove(sparkline);
```

4.2.3 Links

A hyperlink is a convenient way to allow the user of a workbook to instantly access another place in the workbook, or another workbook, or a file associated with another application. A hyperlink can be inserted in a cell or a shape in Excel. Select the cell or shape, and select Hyperlink from the **Insert** menu, or right-click anywhere in the cell or shape, and then select **Hyperlink** from the pop-up menu. You can enter a cell reference in the current workbook, browse to another workbook, a different file, or a web page, and even enter an email address and subject line. You can also edit the text for a hyperlink in a cell.

Following is the **Insert Hyperlink** dialog box of MS Excel that allows to set various hyperlinks.

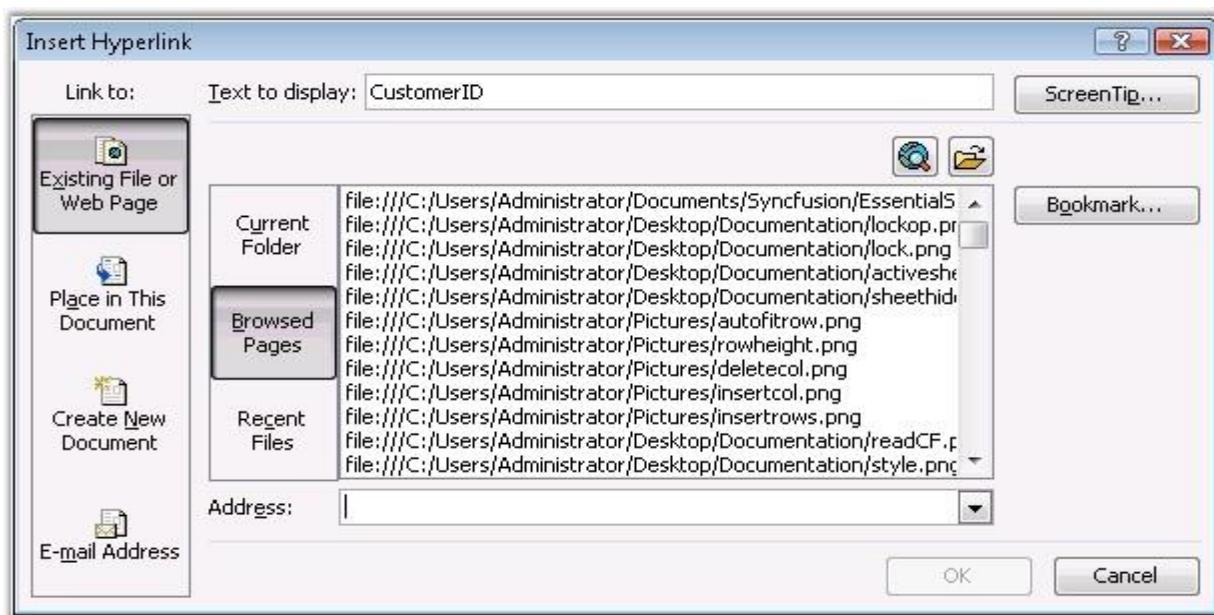


Figure 84: Inserting Hyperlink

XlsIO provides support to set the following types of hyperlinks with the **Type** and **Address** properties of the **IHyperlink** interface.

- Hyperlink to a Worksheet Range
- Hyperlink to Website
- Hyperlink to e-mail
- Hyperlink to external files

You can also set the text to be displayed in a hyperlink, and a tooltip that shows the purpose of the link, by using the **TextToDisplay** and **ScreenTip** properties.

Following code example illustrates how to insert various hyperlinks.

[C#]

```
// Creating a Hyperlink for a Website.
IHyperLink hyperlink = sheet.HyperLinks.Add(sheet.Range["C5"]);
hyperlink.Type = ExcelHyperLinkType.Url;
hyperlink.Address = "http://www.syncfusion.com";
hyperlink.ScreenTip = "To know more About SYNCFUSION PRODUCTS go through
this link";

// Creating a Hyperlink for e-mail.
IHyperLink hyperlink1 = sheet.HyperLinks.Add(sheet.Range["C7"]);
hyperlink1.Type = ExcelHyperLinkType.Url;
hyperlink1.Address = "mailto:Username@syncfusion.com";
hyperlink1.ScreenTip = "Send Mail";

// Creating a Hyperlink for Opening Files using type as File.
IHyperLink hyperlink2 = sheet.HyperLinks.Add(sheet.Range["C9"]);
hyperlink2.Type = ExcelHyperLinkType.File;
hyperlink2.Address = @"C:\Program files";
hyperlink2.ScreenTip = "File path";
hyperlink2.TextToDisplay = "Hyperlink for files using File as type";

// Creating a Hyperlink for Opening Files using type as Unc.
IHyperLink hyperlink3 = sheet.HyperLinks.Add(sheet.Range["C11"]);
hyperlink3.Type = ExcelHyperLinkType.Unc;
hyperlink3.Address = @"C:\Documents and Settings";
hyperlink3.ScreenTip = "Click here for files";
hyperlink3.TextToDisplay = "Hyperlink for files using Unc as type";

// Creating a Hyperlink to another cell using type as Workbook.
IHyperLink hyperlink4 = sheet.HyperLinks.Add(sheet.Range["C13"]);
hyperlink4.Type = ExcelHyperLinkType.Workbook;
hyperlink4.Address = "Sheet1!A15";
hyperlink4.ScreenTip = "Click here";
hyperlink4.TextToDisplay = "Hyperlink to cell A15";
```

[VB.NET]

```
' Creating a Hyperlink for a Website.
Dim hyperlink As IHyperLink = sheet.HyperLinks.Add(sheet.Range("C5"))
hyperlink.Type = ExcelHyperLinkType.Url
hyperlink.Address = "http://www.Syncfusion.com"
hyperlink.ScreenTip = "To know more About SYNCFUSION PRODUCTS go through
this link"
```

```
' Creating a Hyperlink for e-mail.  
Dim hyperlink1 As IHyperLink = sheet.HyperLinks.Add(sheet.Range("C7"))  
hyperlink1.Type = ExcelHyperLinkType.Url  
hyperlink1.Address = "mailto:Username@syncfusion.com"  
hyperlink1.ScreenTip = "Send Mail"  
  
' Creating a Hyperlink for Opening Files using type as File.  
Dim hyperlink2 As IHyperLink = sheet.HyperLinks.Add(sheet.Range("C9"))  
hyperlink2.Type = ExcelHyperLinkType.File  
hyperlink2.Address = "C:\Program files"  
hyperlink2.ScreenTip = "File path"  
hyperlink2.TextToDisplay = "Hyperlink for files using File as type"  
  
' Creating a Hyperlink for Opening Files using type as Unc.  
Dim hyperlink3 As IHyperLink = sheet.HyperLinks.Add(sheet.Range("C11"))  
hyperlink3.Type = ExcelHyperLinkType.Unc  
hyperlink3.Address = "C:\Documents and Settings"  
hyperlink3.ScreenTip = "Click here for files"  
hyperlink3.TextToDisplay = "Hyperlink for files using Unc as type"  
  
' Creating a Hyperlink to another cell using type as Workbook.  
Dim hyperlink4 As IHyperLink = sheet.HyperLinks.Add(sheet.Range("C13"))  
hyperlink4.Type = ExcelHyperLinkType.Workbook  
hyperlink4.Address = "Sheet1!A15"  
hyperlink4.ScreenTip = "Click here"  
hyperlink4.TextToDisplay = "Hyperlink to cell A15"
```

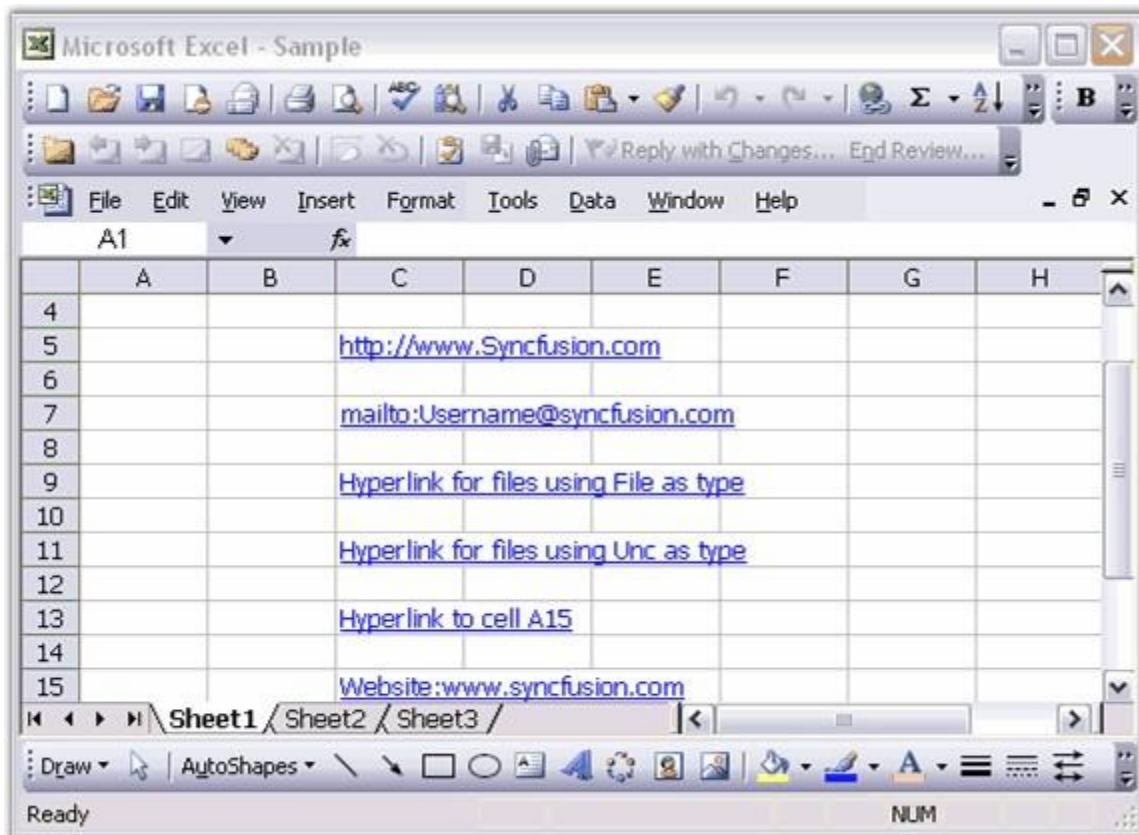


Figure 85: XlsIO with Hyperlinks

4.2.4 Header/Footer

Often, there is a need to include some information about your document at the top (the header), or the bottom (the footer) of each printed sheet. Spreadsheets often need several pages to print. It is important to put the right information on the header or footer, so you can tell which pages go together.

MS Excel provides an option to insert headers and footers, through the below handy dialog box from the **View** menu, to make this process as easy as possible.

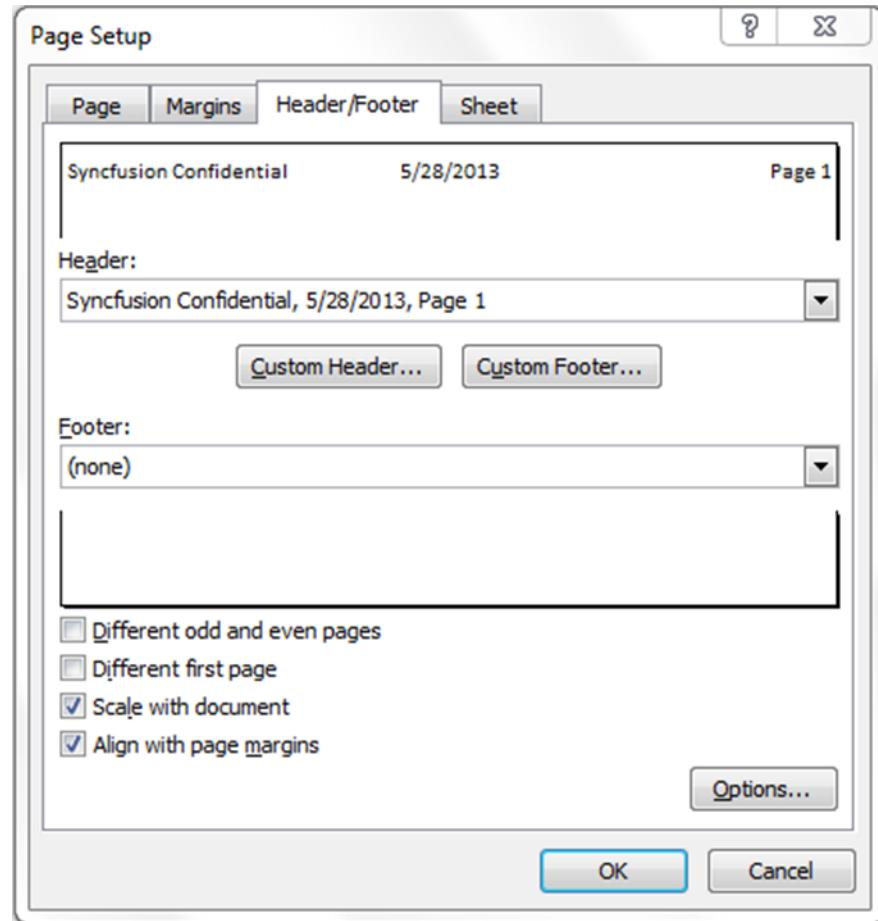


Figure 86: Page Setup Dialog Box to insert Headers and Footers

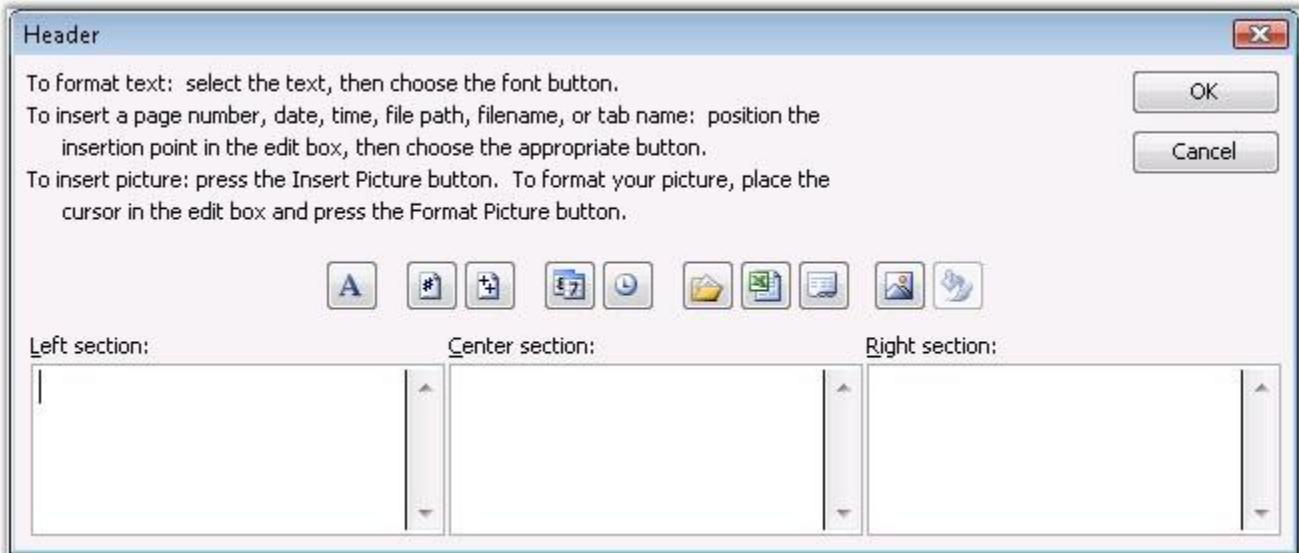


Figure 87: Header Dialog Box

Inserting Headers and Footers in XlsIO

You can insert headers and footers through XlsIO, with the properties in the **IPageSetup**. Headers and footers can also be inserted to a Chart Worksheet.

The string that the header/footer takes, is a script that you can use to format the header. Please refer to the following link for more information on formatting strings:
<http://support.microsoft.com/smartererror/default.aspx?spid=global&query=213618>.

Codes to Format Text	Description
&L	Left-aligns the characters that follow.
&C	Centers the characters that follow.
&R	Right-aligns the characters that follow.
&E	Turns double-underline printing on or off.
&X	Turns superscript printing on or off.
&Y	Turns subscript printing on or off.
&B	Turns bold printing on or off.
&I	Turns italic printing on or off.
&U	Turns underline printing on or off.
&S	Turns strikethrough printing on or off.
&"fontname"	Prints the characters that follow in the specified font. Be sure to include the quotation marks around the font name.
&n	Prints the characters that follow in the specified font size. Use a two-digit number to specify a size in points.

Codes to Insert Specific Data	Description
&D	Prints the current date.

&T	Prints the current time.
&F	Prints the name of the document.
&A	Prints the name of the workbook tab (the "sheet name").
&P	Prints the page number.
&P+number	Prints the page number plus number.
&P-number	Prints the page number minus number.
&&	Prints a single ampersand.
&N	Prints the total number of pages in the document.

Following code example illustrates how to insert images in the header.

```
[C#]

// Insert image in right header.
Image img = Image.FromFile(@"logo.jpg");

// Right Header Image.
sheet.PageSetup.RightHeaderImage = img;
sheet.PageSetup.RightHeader = "&G";
```

```
[VB.NET]

' Insert image in right header.
Dim img As Image = Image.FromFile("logo.jpg")

' Right Header Image.
sheet.PageSetup.RightHeaderImage = img
sheet.PageSetup.RightHeader = "&G"
```

 **Note:** XlsIO does not provide any option to get the page count. You can only insert the page count by using the format string, as illustrated in the following code sample.

```
[C#]
```

```
// Setting the page number in the Center Header.
sheet.PageSetup.CenterHeader = "&P";
```

[VB.NET]

```
' Setting the page number in the Center Header.
sheet.PageSetup.CenterHeader = "&P"
```

Header Footer options

Property	Description
DifferentOddAndEvenPagesHF	If true, the headers and footers on odd-numbered pages will be different from those on even-numbered pages.
DifferentFirstPageHF	When set to true, this property specifies a separate header and footer for the first page.
HFScaleWithDoc	If true, the headers and footers will use the same font size and scaling as in the worksheet. If false, the headers and footers will not shrink or expand with document scaling.
AlignHFWithPageMargins	If true, the header and footer margin is aligned with the left and right margins of the worksheet. If false, the header and footer margin is aligned with paper edges.

The code sample below illustrates the usage of properties meant for header and footer options.

[C#]

```
//Setting the header footer page setup options.

sheet.PageSetup.DifferentOddAndEvenPagesHF = false;
sheet.PageSetup.DifferentFirstPageHF = false;
sheet.PageSetup.HFScaleWithDoc = true;
sheet.PageSetup.AlignHFWithPageMargins = true;
```

[VB]

```
'Setting the header footer page setup options.
```

```

sheet.PageSetup.DifferentOddAndEvenPagesHF = False
sheet.PageSetup.DifferentFirstPageHF = False
sheet.PageSetup.HFScaleWithDoc = True
sheet.PageSetup.AlignHFWithPageMargins = True

```

4.2.5 Tables

In Excel, Tables can be inserted by selecting Table option from the Insert menu.

Table Creation By Using XlsIO

XlsIO provides support to read and write tables in a spreadsheet. The table is added as an **IListObject** to the worksheet. The input data to the table must be a range of data existing in the worksheet. **IListObject** returns the collection of tables in the worksheet.

[C#]

```

// Create Table
IListObject table1 = sheet.ListObjects.Create("Table1", sheet["A1:C8"]);

```

[VB .NET]

```

' Create Table
Dim table1 As IListObject = sheet.ListObjects.Create("Table1",
sheet("A1:C8"))

```

Total Row

You can add the "Total Row" to any table by accessing the **Table Columns**. Columns in the tables are accessed by using the index. It is possible to set "Totals Calculation" to the Total Row cells by using the **ExcelTotalsCalculation** enumerator. These cells will be updated once they are calculated.

[C#]

```

// Total Row
table1.Columns[0].TotalsRowLabel = "Total";
table1.Columns[1].TotalsCalculation = ExcelTotalsCalculation.Sum;
table1.Columns[2].TotalsCalculation = ExcelTotalsCalculation.Sum;

```

[VB .NET]

```
' Total Row
table1.Columns(0).TotalsRowLabel = "Total"
table1.Columns(1).TotalsCalculation = ExcelTotalsCalculation.Sum
table1.Columns(2).TotalsCalculation = ExcelTotalsCalculation.Sum
```

Formatting Table

You can apply built-in styles for the tables by using the **TableBuiltInStyles** enumerator of XlsIO.

[C#]

```
// Apply built-in style.
table1.BuiltInTableStyle = TableBuiltInStyles.TableStyleMedium9;
```

[VB .NET]

```
' Apply built-in style.
table1.BuiltInTableStyle = TableBuiltInStyles.TableStyleMedium9
```

Reading Existing Table

XlsIO provides support to read an existing table from the spreadsheet. It can be accessed from the sheet by using the "Table Index".

[C#]

```
IListObject table = sheet.ListObjects[0];
table.BuiltInTableStyle = TableBuiltInStyles.TableStyleMedium1;
```

[VB .NET]

```
Dim table As IListObject = sheet.ListObjects[0]
table.BuiltInTableStyle = TableBuiltInStyles.TableStyleMedium1
```

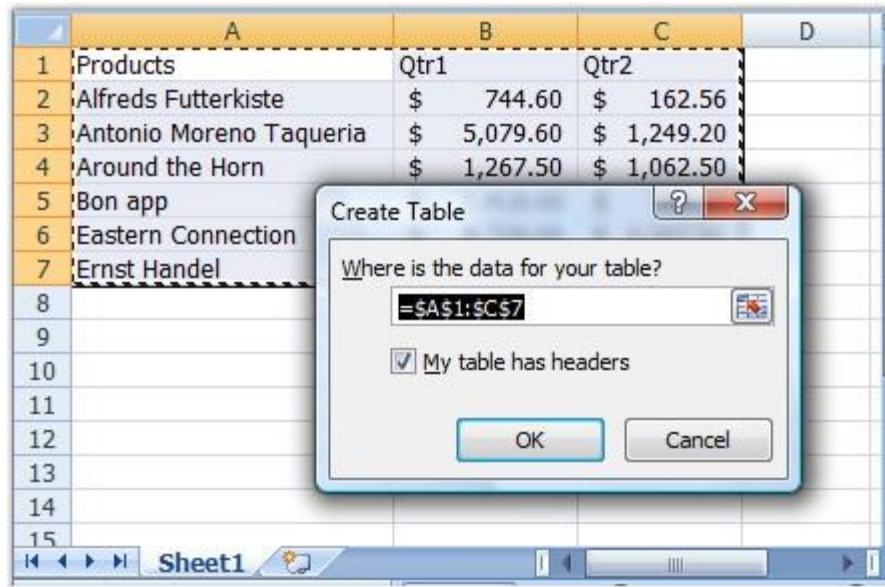


Figure 88: Table inserted by using Excel

4.2.6 Pivot Tables

Pivot Tables is an Excel feature that helps in summarizing large data. Instead of analyzing rows upon rows of records, a pivot table can aggregate your data and show a new perspective with few clicks. You can also move columns to rows or vice versa. You can create the table by defining which fields to view, and how the data should be displayed. Based on the field selections, Excel aggregates and organizes the data so that you can see a different view of your data.

Essential XlsIO supports the usage of Pivot Table. Creation and Manipulation of pivot table is supported in Excel 2007 format, and pivot table in existing document can be preserved for Excel 2003 format. The following topics illustrate this.

4.2.6.1 Excel 2003

XlsIO supports the usage of Pivot Tables in a Template file. It is even possible to dynamically refresh the data in a pivot table by using XlsIO. The following steps illustrate how to do this.

- Create the pivot table using MS Excel GUI.
- Specify the named range to be the data source of the pivot table.
- Make sure that the "Refresh On Open" option of the pivot table is selected.
- Dynamically refresh the data in the Named Range.

[C#]

```
// Change the range values that the Pivot Tables range refers to.
myWorkbook.Names["PivotRange"].RefersToRange = mySheet.Range["A1:D27"];
```

[VB.NET]

```
' Change the range values that the Pivot Tables range refers to.
Private myWorkbook.Names("PivotRange").RefersToRange =
mySheet.Range("A1:D27")
```

4.2.6.2 Excel 2007

Pivot Table Creation By Using MS Excel 2007

In Excel, Pivot table can be inserted by selecting the **PivotTable** option from the **Insert** menu.

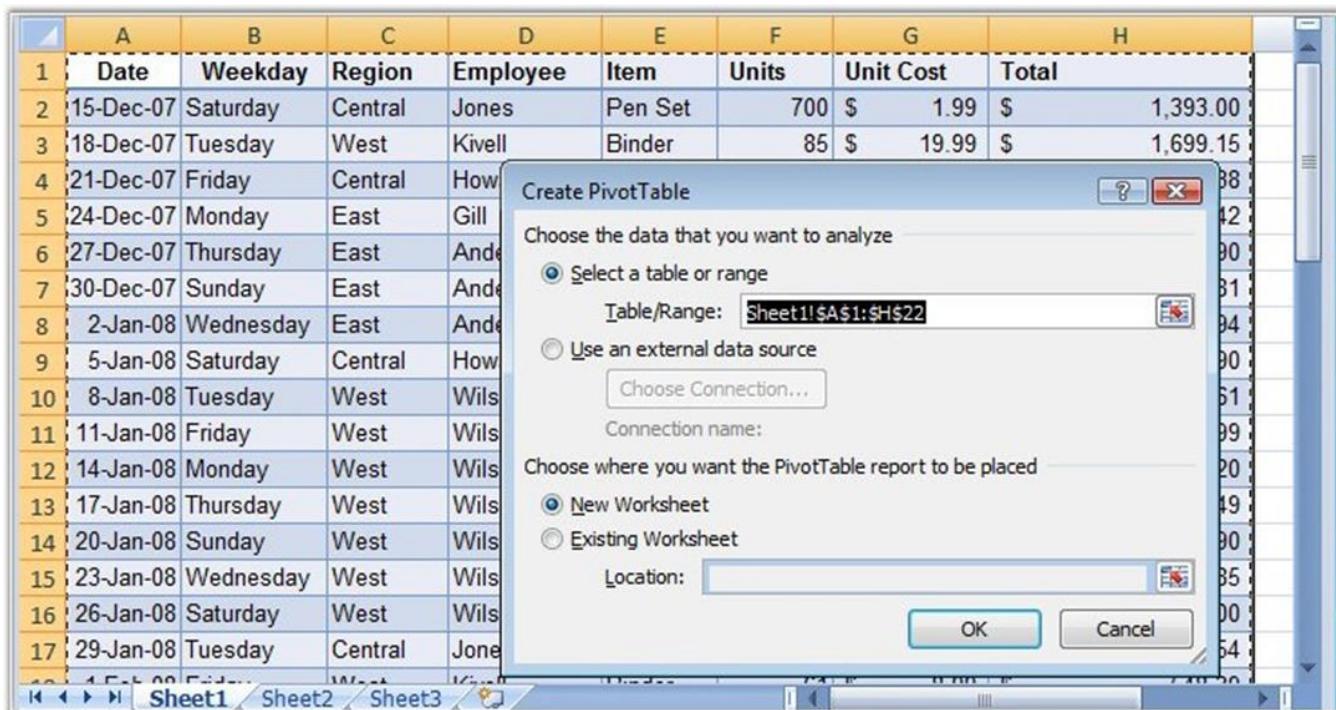


Figure 89: Create PivotTable Dialog Box

Excel automatically selects the entire range. However, it is possible to modify it if necessary. It also allows choosing where to place the PivotTable (New Worksheet is most commonly used to place the pivot table).

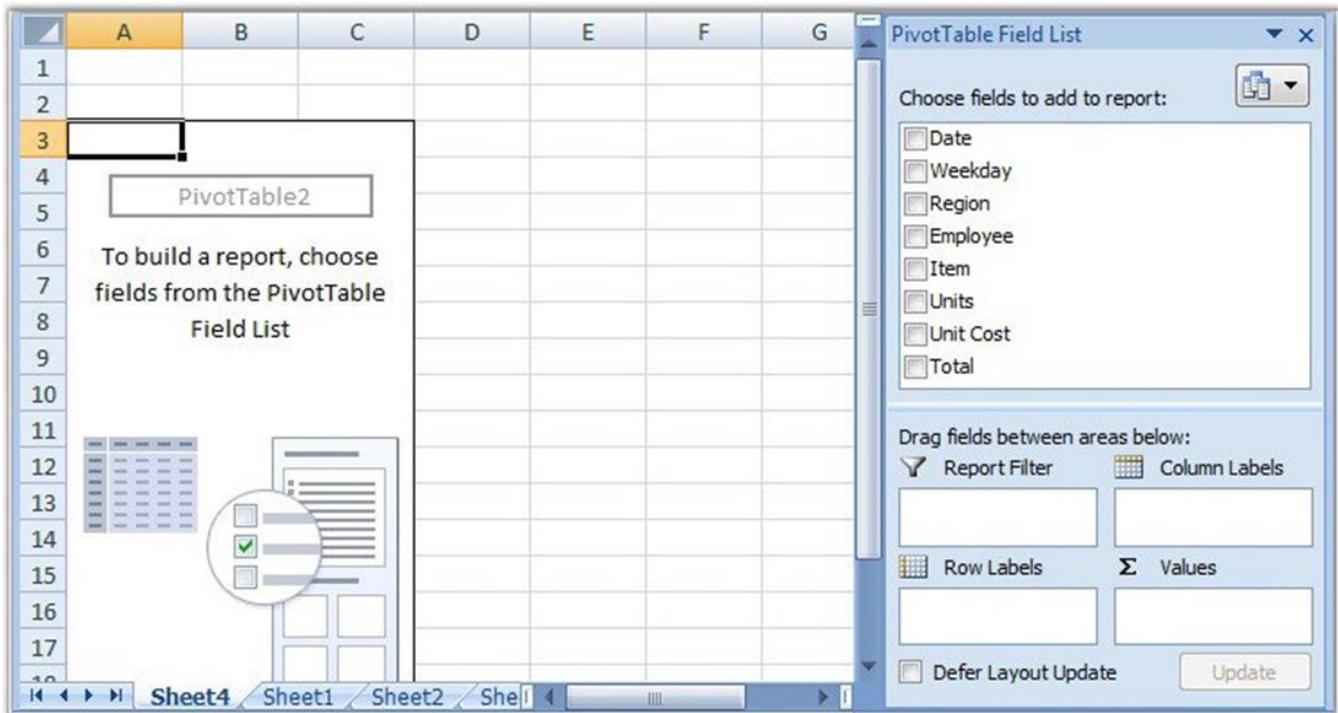


Figure 90: New Sheet to place the Pivot Table

Once you select a field, the Pivot Table appears. Now you need to populate it with data fields, which appear in the field list on the right. Fields can be dragged to the Pivot Table grid, to one of the defined areas.

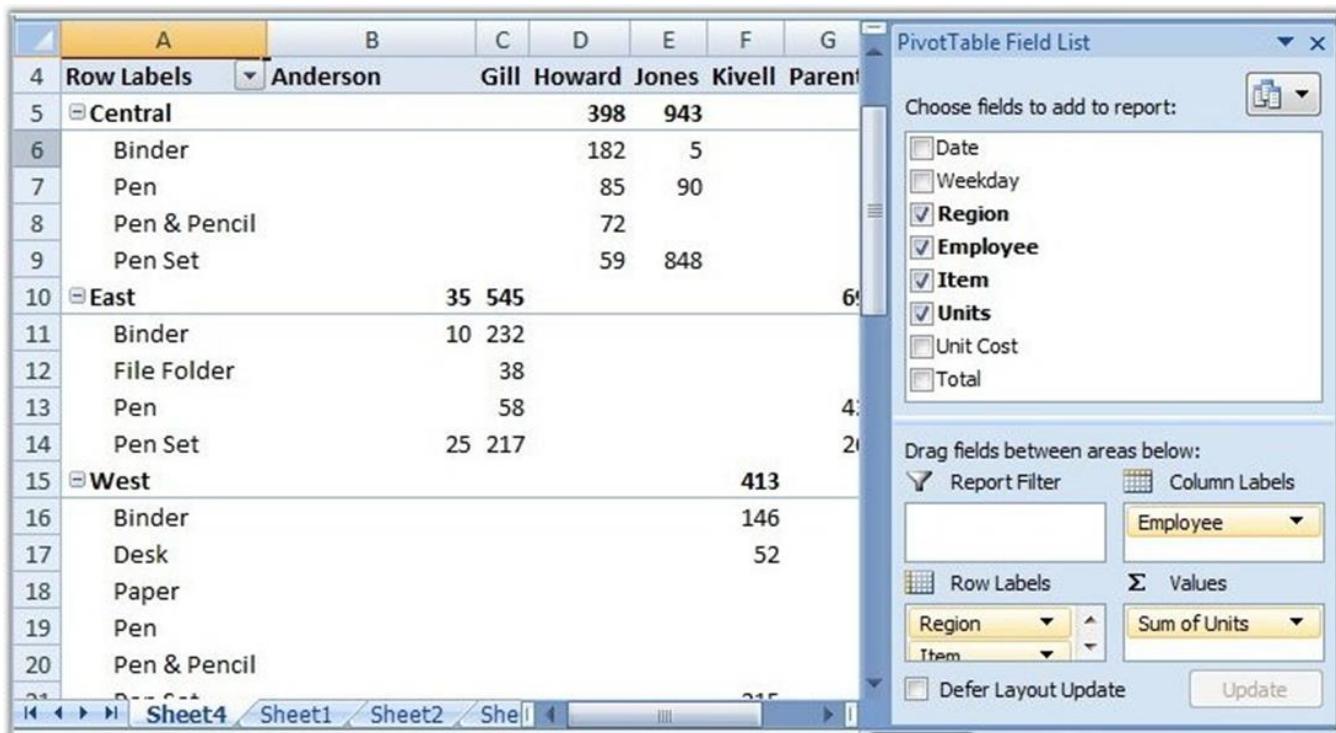


Figure 91: Adding fields to the Pivot table

To filter by a field, open its drop-down list and select the value by which to filter. The table now displays data only for the filtered criterion (in this case, the Central region).

You can also sort by a field by opening its drop-down list and selecting one of the sort orders.

PivotTable Creation Manipulation Using XlsIO

XlsIO provides support for creation and manipulation of Pivot Table by using simple APIs.

IPivotCache interface caches the data that needs to be summarized. **IPivotTable** represents a pivot table in object, and has properties that allow customizing it. **IPivotTable** interface returns the collection of Pivot Tables present in a worksheet. **IPivotField** represents the field in the pivot table. This includes row, column and data field axis. **IPivotDataFields** gets collection of data field.



Note: Pivot Table is currently not supported for .xls format.

Following code example illustrates how to create a pivot table by using XlsIO.

[C#]

```
IPivotCache cache = book.PivotCaches.Add(sheet["A1:D136"]);
```

```

IPivotTable pivotTable = sheet1.PivotTables.Add("PivotTable1", sheet1["A1"], cache);
pivotTable.Fields[0].Axis = PivotAxisTypes.Row;
pivotTable.Fields[1].Axis = PivotAxisTypes.Row;
pivotTable.Fields[3].Axis = PivotAxisTypes.Column;

IPivotField field = pivotTable.Fields[2];
pivotTable.DataFields.Add(field, "sum", PivotSubtotalTypes.Sum);

```

Properties

The following properties of the IPivotTable interface are used to fetch pivot table fields.

Table 1: IPivotTable Properties Table

Property	Description
Name	Gets or sets pivot table name
Location	Returns pivot table location
CacheIndex	Gets Index of the pivot Cache. Read-only
Fields	Gets collection of pivot fields. Read-only
DataFields	Gets IDataField collection of pivot table data fields. Read-only
ColumnFields	Returns the collection of Column field for the specified pivot table. Read-only
RowFields	Returns the collection of Row field for the specified pivot table. Read-only
PageFields	Returns the collection of page field for the specified pivot table. Read-only
CalculatedFields	Returns the collection of calculated fields of the specified pivot table. Read-only
ColumnsPerPage	Specifies the number of columns per page for this PivotTable that the filter area will occupy
RowsPerPage	Specifies the number of rows per page for this PivotTable that the filter area will occupy
Options	Represents the pivot table options. Read-only

The following properties of the `IPivotTableOption` interface are used to customize the settings of the pivot table.

Table 2: IPivotTableOption Properties Table

Property	Description
ShowAsteriskTotals	True, if an asterisk (*) is displayed next to each subtotal and grand total value in the specified PivotTable report
ColumnHeaderCaption	Specifies the string to be displayed in column header of pivot Table when in compact layout mode.
RowHeaderCaption	Specifies the string to be displayed in Row header of pivot table when in compact layout mode
ShowCustomSortList	Specifies a boolean value that indicates whether the "custom lists" option is offered when sorting this PivotTable
ShowFieldList	False, to disable the ability to display the field list for the PivotTable. If the field list was already being displayed it disappears
IsDataEditable	True, to disable the alert for when the user overwrites values in the data area of the PivotTable
EnableFieldProperties	True, if the PivotTable Field dialog box is available when you double-click the PivotTable field
Indent	Specifies the indentation increment for compact axis and can be used to set the Report Layout to Compact Form
ErrorString	Returns or sets the string displayed in cells that contain errors when the <code>DisplayErrorString</code> property is True
DisplayErrorString	True, if the PivotTable report displays a custom error string in cells that contain errors. The default value is False
MergeLabels	True, if the specified PivotTable report's outer-row item, column item, subtotal, and grand total labels use merged cells.
PageFieldWrapCount	Returns or sets the number of page fields in each column or row in the PivotTable report.

PageFieldsOrder	Returns or sets the order in which page fields are added to the PivotTable report's layout
DisplayNullString	True, if the PivotTable report displays a custom string in cells that contain null values. The default value is True.
NullString	Returns or sets the string displayed in cells that contain null values when the DisplayNullString property is True.
PreserveFormatting	True, if formatting is preserved when the report is refreshed or recalculated by operations such as pivoting, sorting, or changing page field items.
ShowToolips	True, if tooltips displayed for the pivot table cell.
DisplayFieldCaptions	Gets/sets value controlling whether or not filter buttons and PivotField captions for rows and columns are displayed in the grid.
PrintTitles	True, if the print titles for the worksheet are set based on the PivotTable report. False, if the print titles for the worksheet are used.
RowLayout	This property specifies the pivot table row layout settings.

Following are the properties of the IPivotCache interface.

Table 3: IPivotCache Properties Table

Property	Description
Index	Gets zero-based cache index. Read-only
SourceType	Specifies the pivot table cache source type. Read-only
SourceRange	Returns the data source for the PivotTable report. Read-only.

SubTotals

You can also insert various subtotal types for the pivot table through the **PivotSubtotalTypes** enum.

It is also possible to insert multiple subtotals for a field by using **Subtotal** property of IPivotField. This is demonstrated in the following code example.

[C#]

```
IPivotTable pivotTable = sheet1.PivotTables.Add("PivotTable1", sheet1["A1"], cache);
pivotTable.Fields[0].Axis = PivotAxisTypes.Row;
pivotTable.Fields[0].Subtotals = PivotSubtotalTypes.Sum |
PivotSubtotalTypes.Average | PivotSubtotalTypes.Max |
PivotSubtotalTypes.Min;
```

Pivot Table Options

Excel provides various options through the **PivotTableOptions** dialog box to customize the appearance of the pivot table.

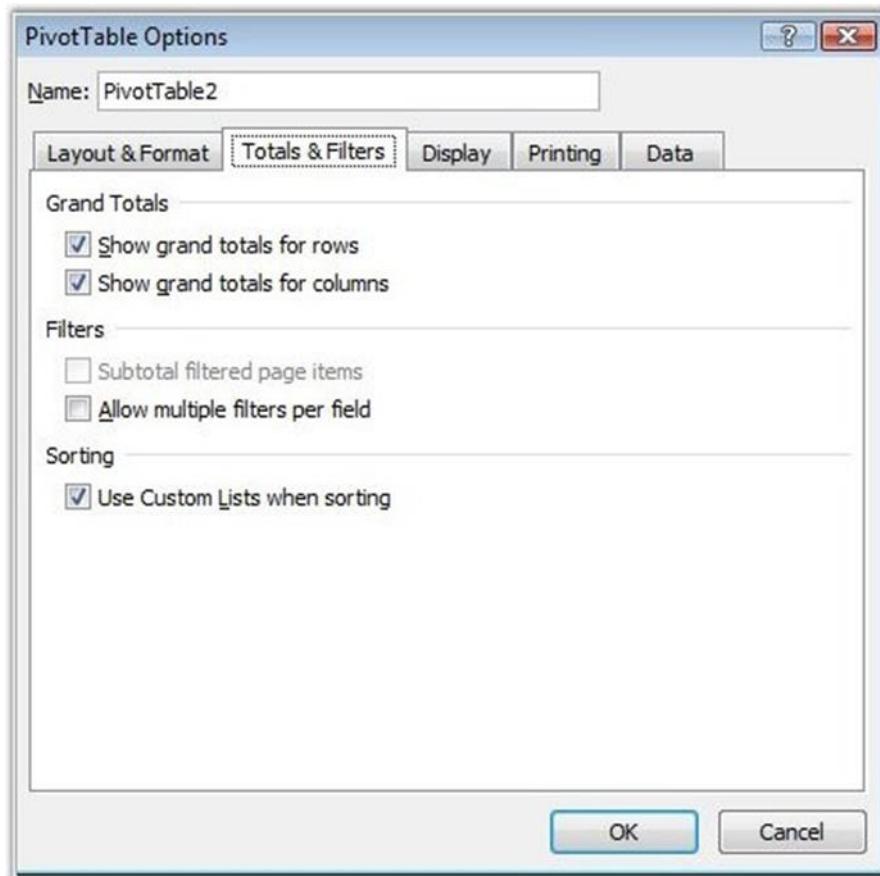


Figure 92: PivotTable Options Dialog Box

XlsIO supports the pivot table options using IPivotTableOptions interface to control various settings for the existing Pivot table. The following code sample illustrates the same.

```
[C#]
IPivotTable pivotTable = sheet.PivotTables[0];
IPivotTableOptions options = pivotTable.Options;
options.ShowFieldList = true;
options.ColumnHeaderCaption = "Sales Details";
options.ColumnHeaderCaption = "Customer Names";
options.ErrorString = "#ERROR#";
```

Show or Hide the Field List

In MS Excel, click the Field List button in the Design Tab. Show or Hide the pivot table field list pane in XlsIO,

```
[C#]
options.ShowFieldList = true;
```

Header Caption:

In MS Excel, the Field Header button is used to show or hide the pivot table caption. In XlsIO, to enable and disable the caption, use the **DisplayFieldCaption** property. Use the **RowHeaderCaption** and **ColumnHeaderCaption** properties to edit the respective pivot table headers.

```
[C#]
options.RowHeaderCaption = "Payment Dates";
options.ColumnHeaderCaption = "Payments";
```

Grand Total

You can display or hide the totals for the current Pivot Table report by selecting an option from **Design -> Layout-> Grand Totals**. XlsIO provides an equivalent API to perform with simple properties as follows.

```
[C#]
pivotTable.ColumnGrand = false;
pivotTable.RowGrand = true;
```

Show/Hide Collapse Button

You can also show/hide the **Collapse** button that appears in the fields of the pivot table, when there exists more than one item in a field. The following code example illustrates how to do this.

```
[C#]
pivotTable.ShowDrillIndicators = true;
```

Display Field Caption and Filter Option

It is also possible to show/hide the **Filter** button and field name in the pivot table by using the PivotTable Options dialog box in Excel. This is illustrated in the following code.

```
[C#]
pivotTable.DisplayFieldCaptions = true;
```

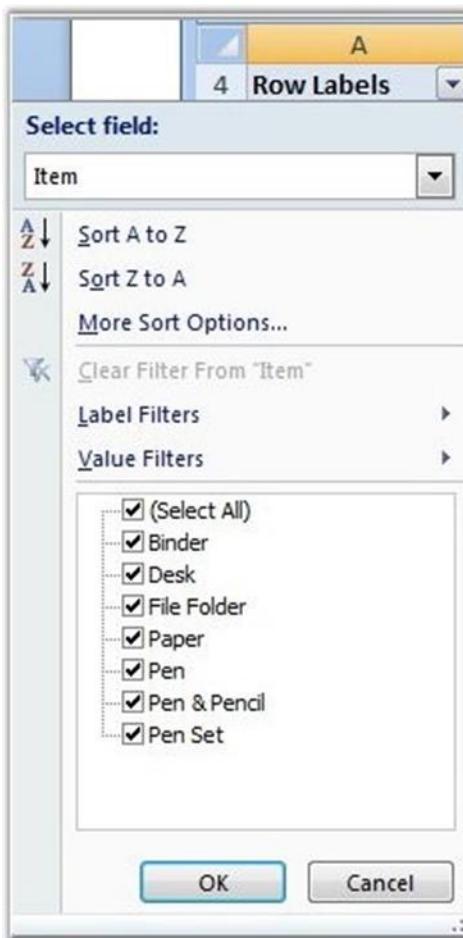


Figure 93: Field Captions

Repeating Row Label on Each Page

XlsIO allows setting the row label on each page, while printing, allowing users to view the header on each page.

[C#]

```
pivotTable.RepeatItemsOnEachPrintedPage = false;
```

Formatting the Pivot Table By using Excel 2007

Excel 2007 provides set of built-in styles that allow formatting the pivot table row and column header. When your cell pointer is inside the pivot table, you will have two new ribbon tabs under **PivotTable Tools heading - Options and Design**. On the Design ribbon, the Pivot Table Styles gallery offers 85 built-in formats for pivot tables.

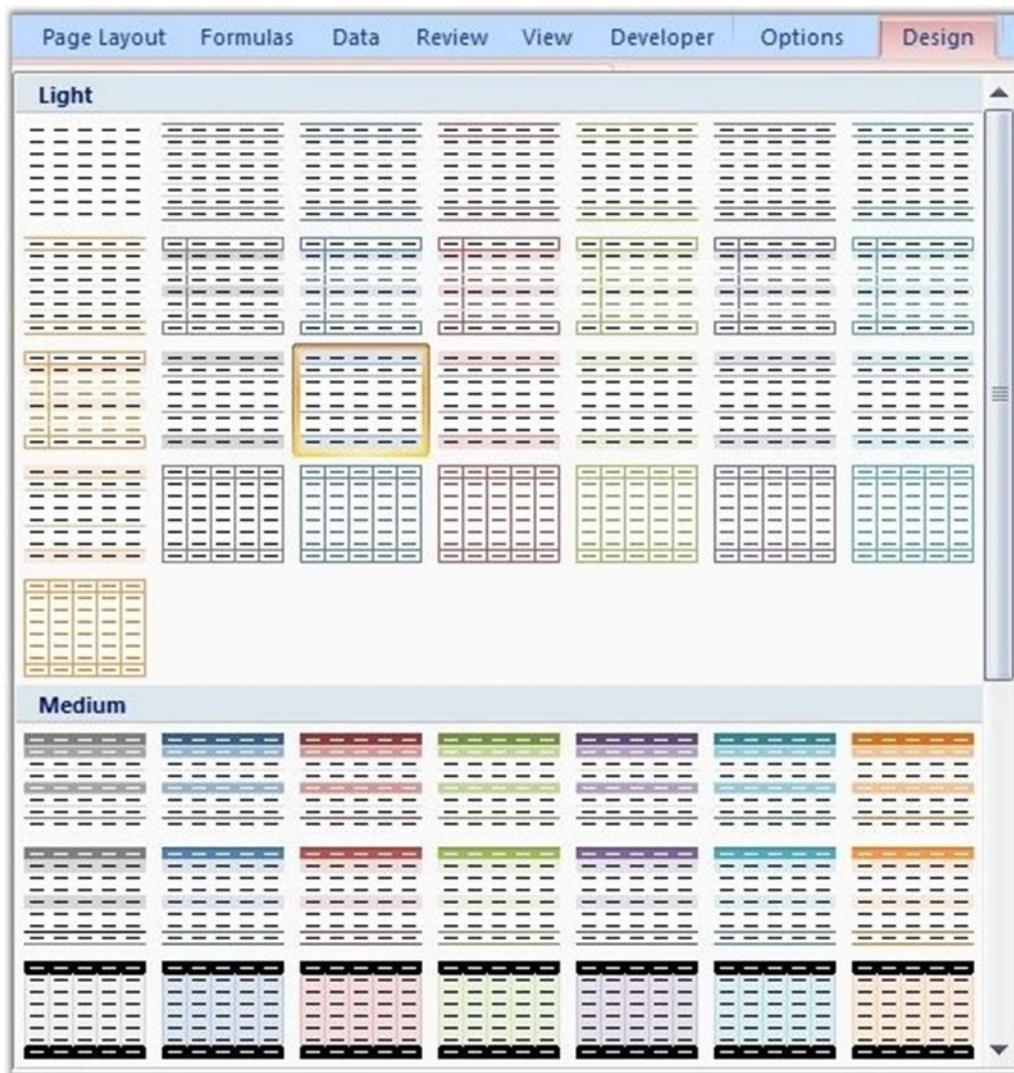


Figure 94: Pivot Table Styles Gallery

Formatting Pivot Table By Using XlsIO

XlsIO supports 85 built-in styles of Excel 2007, enabling users to create a table with rich formatting. This is done by using the **BuiltInStyle** property of IPivotTable as follows.

[C#]

```
pivotTable.BuiltInStyle = PivotBuiltInStyles.PivotStyleDark12;
```

Adding Calculated Field in the existing Pivot Table

Calculated fields are a special type of database field that perform calculations by using the contents of other fields in the pivot table with the given formula. The formula can contain operators and expressions as in other worksheet formulas. You can use constants and refer to data from the PivotTable. XlsIO supports to read and create the Calculated Fields in the existing pivot table. The following are MS Excel restriction when using the formula.

- Formula cannot contain cell references or defined names and
- Formula cannot contains Worksheet functions that require cell references
- Formula cannot use array functions.

In MS Excel, the Calculated Field can be added using the calculation option from the Option tab. In XlsIO, same can be achieved with following code sample.

[C#]

```
IPivotTable pivotTable = sheet.PivotTables[0];
IPivotField field = pivotTable.CalculatedFields.Add("Percent",
"Sales/Total*100");
```

The formula can be fetched from the formula property of the IPivotField.

[C#]

```
field.Formula = "Sales/Total*100";
```

Lay Out the Pivot Table as MS Excel 2007

We have provided support for drawing a pivot table similar to the MS Excel layout using Essential XlsIO. Previously, we let MS Excel lay out the pivot table for XlsIO. By using a layout method, we draw the pivot table layout using XlsIO. Now we can get any values of the pivot table using XlsIO dynamically, apply a filter to the pivot table, and can get the filtered values of the pivot table dynamically.

Methods

Method Name	Description
Layout	Method to lay out the pivot table using XlsIO like the MS Excel pivot table layout.

The following code example illustrates how to enable Essential XlsIO to lay out the pivot table like MS Excel.

[C#]

```

IPivotCache cache = book.PivotCaches.Add(sheet["A1:D136"]);
IPivotTable pivotTable = sheet1.PivotTables.Add("PivotTable1",
sheet1["A1"], cache);

IPivotField field = pivotTable.Fields[0];
Field.Axis = PivotAxisTypes.Page;
//Setting the Filter to Page field of Pivot table
IPivotFilter filter = field .PivotFilters.Add();
filter.Value1 = "East";

pivotTable.Fields[0].FilterValue="Binder";
pivotTable.Fields[1].Axis = PivotAxisTypes.Row;
pivotTable.Fields[3].Axis = PivotAxisTypes.Column;

IPivotField field = pivotTable.Fields[2];
pivotTable.DataFields.Add(field, "sum", PivotSubtotalTypes.Sum);
//The following code sample must be included to XlsIO layout the pivot
//table like MS Excel.
pivotTable.Layout();

```

[VB .Net]

```

Dim cache As IPivotCache = book.PivotCaches.Add(sheet("A1:D136"))
Dim pivotTable As IPivotTable = sheet1.PivotTables.Add("PivotTable1",
sheet1("A1"), cache)

Dim field As IPivotField = pivotTable.Fields(0)
Field.Axis = PivotAxisTypes.Page
'Setting the Filter to Page field of Pivot table
Dim filter As IPivotFilter = field.PivotFilters.Add()

```

```

filter.Value1 = "East"
pivotTable.Fields(1).Axis = PivotAxisTypes.Row
pivotTable.Fields(3).Axis = PivotAxisTypes.Column

Dim field As IPivotField = pivotTable.Fields(2)
pivotTable.DataFields.Add(field, "sum", PivotSubtotalTypes.Sum)

'The following code sample must be included to XlsIO layout the pivot
'table 'like MS Excel.

pivotTable.Layout()

```

Supported Elements:

1. Apply filter value to page filter of the pivot table.
2. Pivot table values can be accessed dynamically.

Apply Filter to Pivot Table

In Microsoft Excel, filtered data of a pivot table displays only the subset of data that meet the criteria we specified. The Excel has drop-down filter arrows for report/page filter fields, row fields, and column fields. This can be achieved in XlsIO using the **IPivotFilters** interface.

Page Field Filter

The page field filter filters the pivot table based on page field items. The following code example illustrates how to apply multiple filters to the page field items.

[C#]

```

//Step 1: Instantiate the spreadsheet creation engine.
ExcelEngine excelEngine = new ExcelEngine();
//Step 2: Instantiate the excel application object.
IApplication application = excelEngine.Excel;

//Set the default version as Excel 2010.
application.DefaultVersion = ExcelVersion.Excel2010;

IWorkbook workbook = application.Workbooks.Open("PivotCodeDate.xlsx");

```

```
// The first worksheet object in the worksheets collection is accessed.  
IWorksheet worksheet = workbook.Worksheets[0];  
  
//Access the worksheet to draw pivot table.  
IWorksheet pivotSheet = workbook.Worksheets[1];  
  
//Select the data to add in cache.  
IPivotCache cache = workbook.PivotCaches.Add(worksheet["A1:H50"]);  
  
//Insert the pivot table.  
IPivotTable pivotTable = pivotSheet.PivotTables.Add("PivotTable1",  
pivotSheet["A1"], cache);  
  
//Set field axis to page.  
pivotTable.Fields[4].Axis = PivotAxisTypes.Page;  
  
//Set field axis to row.  
pivotTable.Fields[2].Axis = PivotAxisTypes.Row;  
pivotTable.Fields[6].Axis = PivotAxisTypes.Row;  
  
//Set field axis to column.  
pivotTable.Fields[3].Axis = PivotAxisTypes.Column;  
  
IPivotField field = pivotSheet.PivotTables[0].Fields[5];  
pivotTable.DataFields.Add(field, "Sum of Units",  
PivotSubtotalTypes.Sum);  
  
//Apply page field filter.  
IPivotField pageField = pivotTable.Fields[4];  
  
//Select multiple items in page field to filter.  
pageField.Items[1].Visible = false;  
pageField.Items[2].Visible = false;
```

```
//Apply built-in style.  
pivotTable.BuiltInStyle = PivotBuiltInStyles.PivotStyleMedium2;  
  
//Activate the pivot worksheet.  
pivotSheet.Activate();  
  
//Save the workbook to disk.  
workbook.SaveAs("PivotTable.xlsx");  
  
//Close the workbook.  
workbook.Close();  
  
//No exception will be thrown if there are unsaved workbooks.  
excelEngine.ThrowNotSavedOnDestroy = false;  
excelEngine.Dispose();
```

[VB]

```
'Step 1: Instantiate the spreadsheet creation engine.  
Dim excelEngine As New ExcelEngine()  
'Step 2: Instantiate the excel application object.  
Dim application As IApplication = excelEngine.Excel  
  
'Set the default version as Excel 2010.  
application.DefaultVersion = ExcelVersion.Excel2010  
  
'Get the path of input file.  
Dim workbook As IWorkbook =  
application.Workbooks.Open("PivotCodeDate.xlsx")  
  
'The first worksheet object in the worksheets collection is accessed.  
Dim worksheet As IWorksheet = workbook.Worksheets(0)
```

```

'Access the worksheet to draw pivot table.

Dim pivotSheet As IWorksheet = workbook.Worksheets(1)

>Select the data to add in cache.

Dim cache As IPivotCache =
workbook.PivotCaches.Add(worksheet("A1:H50"))

Select multiple items in page field to filter.

pageField.Items(1).Visible = False
pageField.Items(2).Visible = False

'Apply built-in style.

pivotTable.BuiltInStyle = PivotBuiltInStyles.PivotStyleMedium2

'Activate the pivot worksheet.

```

```

pivotSheet.Activate()

'Save the workbook to disk.
workbook.SaveAs("PivotTable.xlsx")

'Close the workbook.
workbook.Close()

'No exception will be thrown if there are unsaved workbooks.
excelEngine.ThrowNotSavedOnDestroy = False
excelEngine.Dispose()

```

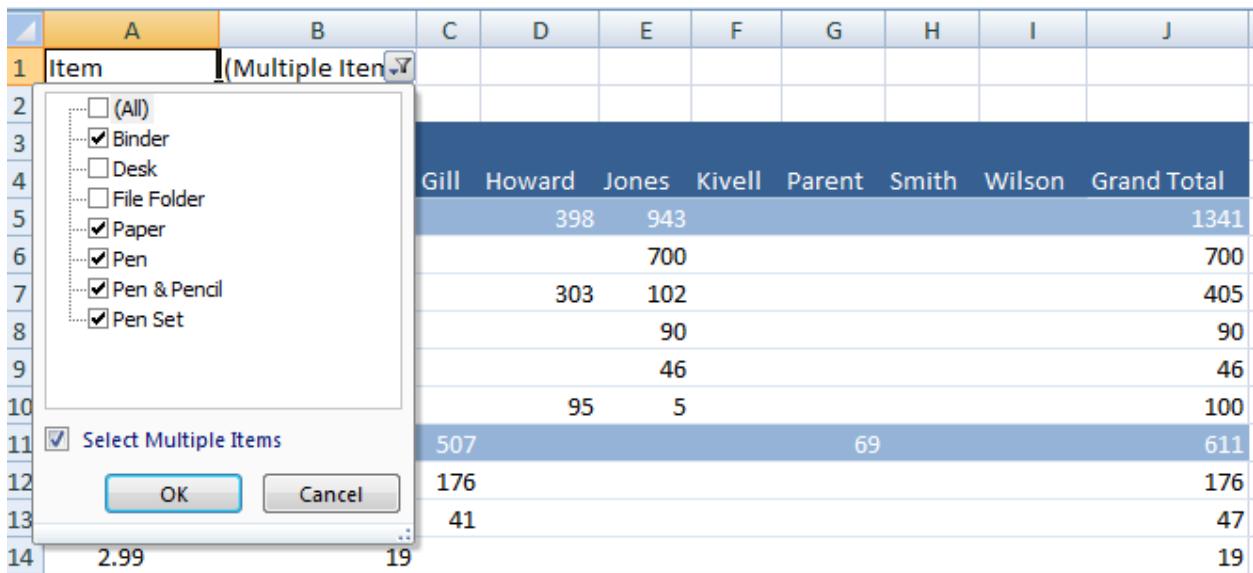


Figure 95: Applying multiple filter

Row Field and Column Field Filter

The row field and column field filter filters the pivot table based on labels, values and items of fields. The following code example illustrates how to apply this filter to a pivot table.

[C#]

```
//Step 1: Instantiate the spreadsheet creation engine.  
ExcelEngine excelEngine = new ExcelEngine();  
//Step 2: Instantiate the excel application object.  
IApplication application = excelEngine.Excel;  
  
//Set the default version as Excel 2010.  
application.DefaultVersion = ExcelVersion.Excel2010;  
  
IWorkbook workbook = application.Workbooks.Open("PivotCodeDate.xlsx");  
  
// The first worksheet object in the worksheets collection is accessed.  
IWorksheet worksheet = workbook.Worksheets[0];  
  
//Access the worksheet to draw pivot table.  
IWorksheet pivotSheet = workbook.Worksheets[1];  
  
//Select the data to add in cache.  
IPivotCache cache = workbook.PivotCaches.Add(worksheet["A1:H50"]);  
  
//Insert the pivot table.  
IPivotTable pivotTable = pivotSheet.PivotTables.Add("PivotTable1",  
pivotSheet["A1"], cache);  
pivotTable.Fields[4].Axis = PivotAxisTypes.Page;  
  
pivotTable.Fields[2].Axis = PivotAxisTypes.Row;  
pivotTable.Fields[6].Axis = PivotAxisTypes.Row;  
pivotTable.Fields[3].Axis = PivotAxisTypes.Column;  
  
IPivotField field = pivotSheet.PivotTables[0].Fields[5];  
pivotTable.DataFields.Add(field, "Sum of Units",  
PivotSubtotalTypes.Sum);  
  
//Apply row field filter.  
IPivotField rowField = pivotTable.Fields[2];  
//Applying Label based row field filter
```

```

rowField.PivotFilters.Add(PivotFilterType.CaptionEqual, null, "East",
null);

//Apply column field label based filter.
IPivotField colField = pivotTable.Fields[3];

colField.Items[0].Visible = false;
colField.Items[1].Visible = false;

//Apply built-in style.
pivotTable.BuiltInStyle = PivotBuiltInStyles.PivotStyleMedium2;

//Activate the pivot worksheet.
pivotSheet.Activate();

//Save the workbook to disk.
workbook.SaveAs("PivotTable.xlsx");

//Close the workbook.
workbook.Close();

//No exception will be thrown if there are unsaved workbooks.
excelEngine.ThrowNotSavedOnDestroy = false;
excelEngine.Dispose();

```

[VB]

```

'Step 1: Instantiate the spreadsheet creation engine.
Dim excelEngine As New ExcelEngine()
'Step 2: Instantiate the excel application object.
Dim application As IApplication = excelEngine.Excel

'Set the default version as Excel 2010.

```

```
application.DefaultVersion = ExcelVersion.Excel2010

Dim workbook As IWorkbook =
application.Workbooks.Open("PivotCodeDate.xlsx")

'The first worksheet object in the worksheets collection is accessed.
Dim worksheet As IWorksheet = workbook.Worksheets(0)

'Access the worksheet to draw pivot table.
Dim pivotSheet As IWorksheet = workbook.Worksheets(1)

'Select the data to add in cache.
Dim cache As IPivotCache =
workbook.PivotCaches.Add(worksheet("A1:H50"))

'Insert the pivot table.
Dim pivotTable As IPivotTable =
pivotSheet.PivotTables.Add("PivotTable1", pivotSheet("A1"), cache)
pivotTable.Fields(4).Axis = PivotAxisTypes.Page

pivotTable.Fields(2).Axis = PivotAxisTypes.Row
pivotTable.Fields(6).Axis = PivotAxisTypes.Row
pivotTable.Fields(3).Axis = PivotAxisTypes.Column

Dim field As IPivotField = pivotSheet.PivotTables(0).Fields(5)
pivotTable.DataFields.Add(field, "Sum of Units",
PivotSubtotalTypes.Sum)

'Apply row field filter.
Dim rowField As IPivotField = pivotTable.Fields(2)
'Applying Label based row field filter
rowField.PivotFilters.Add(PivotFilterType.CaptionEqual, Nothing,
"East", Nothing)

'Apply column field label based filter.
Dim colField As IPivotField = pivotTable.Fields(3)
```

```
colField.Items(0).Visible = False
colField.Items(1).Visible = False

'Apply built-in style.
pivotTable.BuiltInStyle = PivotBuiltInStyles.PivotStyleMedium2

'Activate the pivot worksheet.
pivotSheet.Activate()

'Save the workbook to disk.
workbook.SaveAs("PivotTable.xlsx")

'Close the workbook.
workbook.Close()

'No exception will be thrown if there are unsaved workbooks.
excelEngine.ThrowNotSavedOnDestroy = False
excelEngine.Dispose()
```

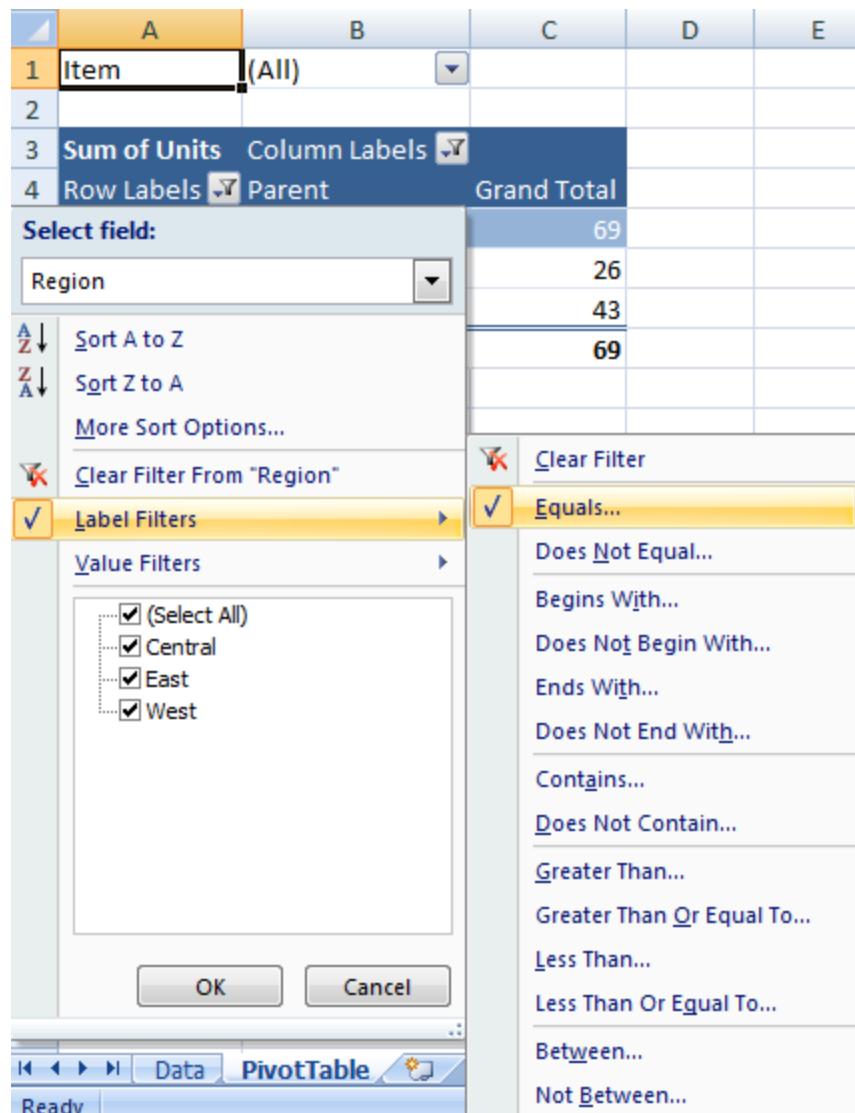


Figure 96: Applying Row Field Filter

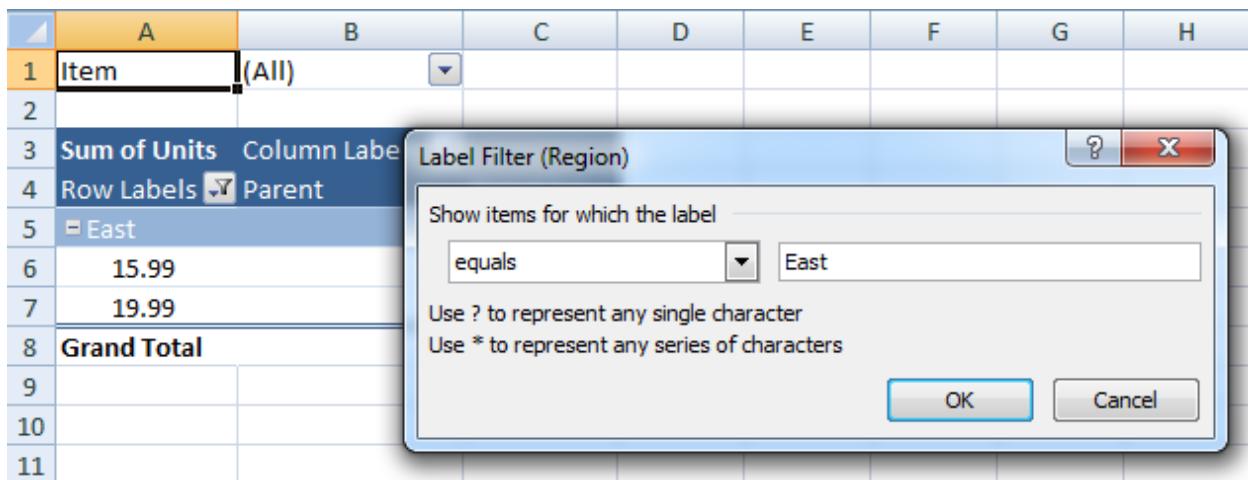


Figure 97: Setting Value to the Filter

4.2.7 PivotCharts

PivotCharts are interactive graphical representations of the PivotTable data that allows rapid analysis of the displayed data. PivotTable is an Excel feature that helps in summarizing large volume of data such as stock market report, Cash flow report etc. A PivotTable can aggregate data instead of analyzing rows upon large volume of records and with a few clicks the PivotChart allows rapid, dynamic, and flexible data analysis. The following sections describe the creation of PivotTables and PivotCharts.

PivotTable Creation

Kindly refer to the following topic for PivotTable creation:

[Excel 2007](#)

PivotChart Creation Using MS Excel 2010

In Microsoft Excel, the PivotChart can be created using the PivotChart Option from the Insert menu. Refer to Figure 98:PivotChart Creation.

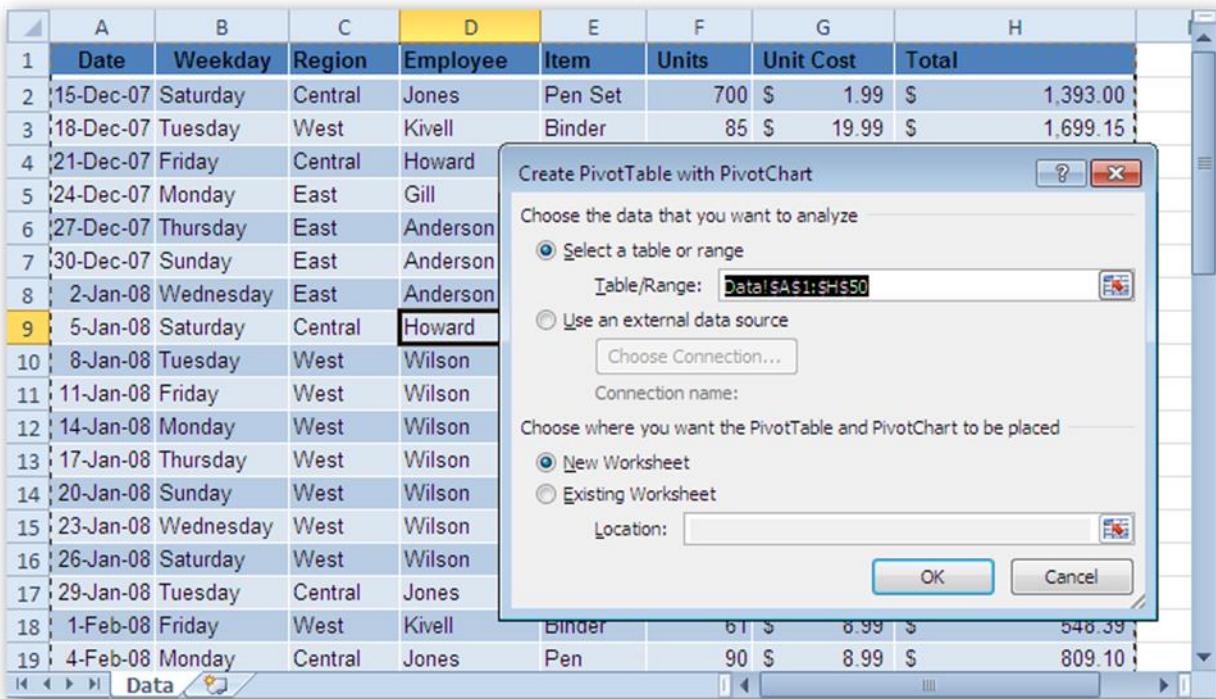


Figure 93: PivotChart Creation Dialog Box

By default, MS Excel selects the entire range. However, it is possible to modify the selected range if required. It also allows choosing the position of the PivotChart (New Worksheet is most commonly used to place the PivotChart).

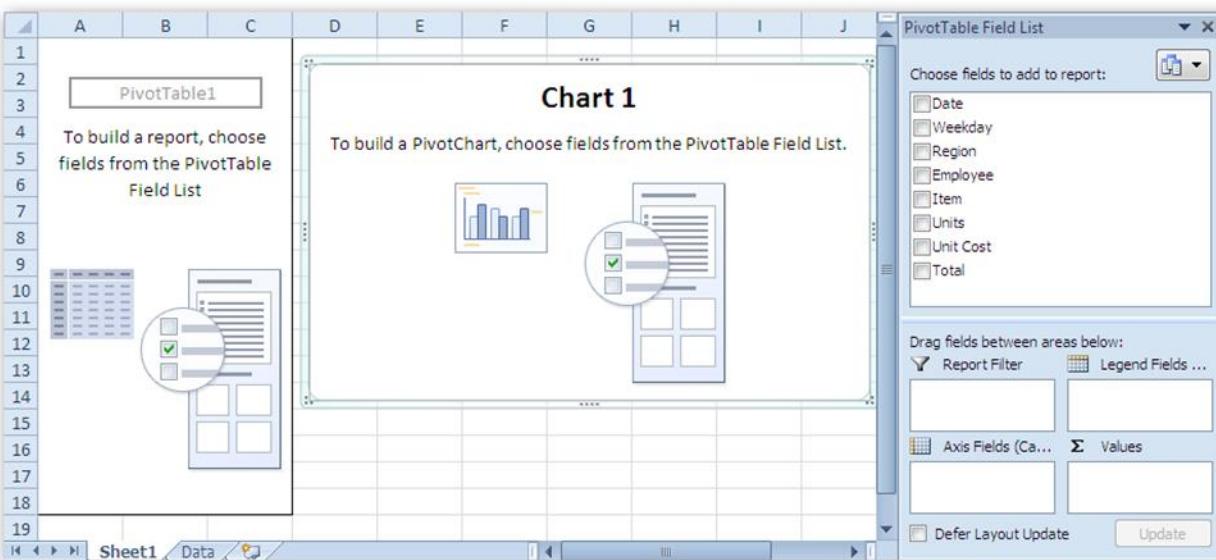


Figure 94: New Sheet to place the PivotTable and PivotChart

Once you select a field, the PivotTable and the PivotChart appear. PivotTable and PivotChart should be populated with data fields, which appear on the field list on the right. Fields can be dragged to the PivotTable grid, to one of the defined areas and also there is an option to move the created PivotChart to a separate sheet.

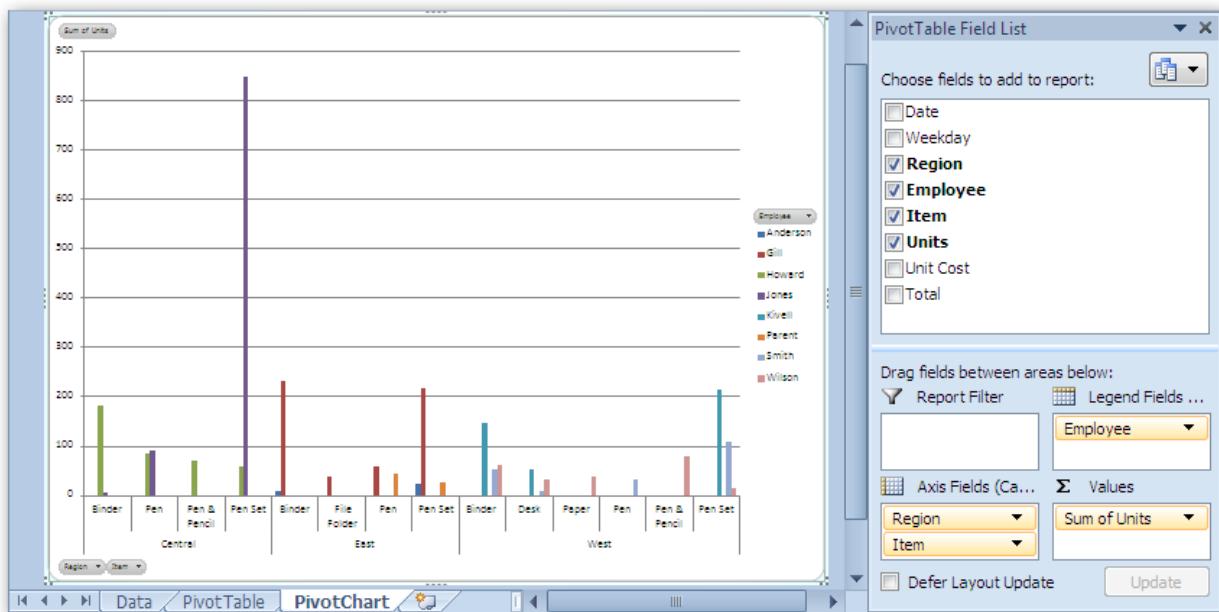


Figure 95:Adding fields to the PivotChart

PivotChart Creation Using XlsIO

XlsIO provides support for the creation of PivotTables and PivotCharts by using simple code samples. IPivotCache interface caches the data that need to be summarized when filtering. IPivotTable represents a PivotTable in object, and has properties that allow customizing it. IPivotTable interface returns the collection of PivotTables that are present in a worksheet. IPivotField represents the field in the PivotTable. This includes row, column and data field axes. IPivotDataFields get a collection of the data field and once the PivotTable is created, it is required to create a PivotChart by setting the PivotSource property of the IChart interface, which references the created PivotTable.

Note: PivotTable and PivotChart are currently not supported for .xls format.

The following code example illustrates the creation of a PivotTable and PivotChart by using XlsIO.

[C#]

```
// Insert the Pivotchart sheet to the workbook.
IChart pivotChartSheet = workbook.Charts.Add();

// Set the PivotSource for the Chart.
pivotChartSheet.PivotSource = pivotTable;

// Select the PivotChart type.
pivotChartSheet.PivotChartType = ExcelChartType.Column_Clustered;
```

[VB]

```
'Insert the Pivotchart sheet to the workbook
Dim pivotChartSheet As IChart = workbook.Charts.Add()

'Set the PivotSource for the Chart.
pivotChartSheet.PivotSource = pivotTable

>Select the PivotChart type.
pivotChartSheet.PivotChartType = ExcelChartType.Column_Clustered
```

PivotChart Options

The field buttons of the PivotChart can be displayed or hidden by selecting **PivotChart Tools->Analyze->Field Buttons**. XlsIO provides an equivalent API to perform simple properties as follows:

Note: These properties are exclusive for Excel 2010.

[C#]

```
pivotChartSheet.ShowAllFieldButtons = false;
pivotChartSheet.ShowAxisFieldButtons = false;
pivotChartSheet.ShowLegendFieldButtons = false;
pivotChartSheet.ShowReportFilterFieldButtons = false;
pivotChartSheet.ShowValueFieldButtons = false;
```

[VB]

```
pivotChartSheet.ShowAllFieldButtons = False
pivotChartSheet.ShowAxisFieldButtons = False
pivotChartSheet.ShowLegendFieldButtons = False
pivotChartSheet.ShowReportFilterFieldButtons = False
pivotChartSheet.ShowValueFieldButtons = False
```

Properties

Table 2: Properties Table

Property	Description	Type	Data Type	Reference links
PivotSource	The Pivot source will accept the object of the IPivotTable, which will be the source for the Pivot chart sheet. This property is responsible for the Pivot chart creation.	Public property	IPivotTable	NA
PivotChartType	The PivotChartType property decides the type for the created Pivot chart. It accepts	Public property	ExcelChartType	NA

Property	Description	Type	Data Type	Reference links
	all the chart types supported by MS Excel.			
ShowAllFieldButtons	It displays all the field buttons of the Pivot chart.	Public Property	True/false	NA
ShowAxisFieldButtons	It displays all the axis area fields as buttons in the Pivot chart with filtering options for the axis field values.	Public Property	True/false	NA
ShowLegendFieldButtons	It displays all the legend area fields as buttons in the Pivot chart with filtering options for the legend field values.	Public Property	True/False	NA
ShowReportFilterFieldButtons	It displays all the Report filter area fields as buttons in the Pivot chart with filtering options for the Report filter field values.	Public property	True/False	NA
ShowValueFieldButtons	It displays all the Values area fields as buttons in the Pivot chart with filtering options for the Values area fields.	Public property	True/False	NA

Sample Link

To understand this process, consider the sample project:

\EssentialStudio*.**.\Windows\XlsIO.Windows\Samples\2.0\Business Intelligence\Pivot Chart.

Note: It is mandatory to have Essential XlsIO installed. MS Excel is not required.

4.2.8 Form Controls

Essential XlsIO provides support to read and write the Text Box, Check Box and Combo Box controls. It enables to create forms which are very user friendly, and also enhance the appearance of the forms.

Note: Essential XlsIO provides support to read and write Form controls. Support for Active X Form controls is not yet available.

This section explains the usage of the following Form controls.

4.2.8.1 Text Box

Essential XlsIO can now read and write text boxes. The **ITextBoxShape** interface lets you add a new text box inside a worksheet. The **IFill** interface is used to customize the inner appearance of the textbox. **IShapeLineFormat** interface is used to modify the border. Various other properties like Horizontal and Vertical Alignment, Alternative Text, Text Rotation, and so on, are also supported.

[C#]

```
// Creates a new Text Box.
ITextBoxShape textbox = sheet.TextBoxes.AddTextBox(3, 7, 25, 100);
textbox.Text = "Essential XlsIO";

// Reads a Text Box.
ITextBoxShape shape1 = sheet.TextBoxes[0];
shape1.Name = "First TextBox";
shape1.Fill.ForeColor = Color.Gold;
shape1.Fill.BackColor = Color.Black;
shape1.Fill.Pattern = ExcelGradientPattern.Pat_90_Percent;
```

[VB .NET]

```
' Creates a new Text Box.  
Dim textbox As ITextBoxShape = sheet.TextBoxes.AddTextBox(3, 7, 25, 100)  
textbox.Text = "Essential XlsIO"  
  
' Reads a Text Box.  
ITextBoxShape shape1 = sheet.TextBoxes(0)  
shape1.Name = "First TextBox"  
shape1.Fill.ForeColor = Color.Gold  
shape1.Fill.BackColor = Color.Black  
shape1.Fill.Pattern = ExcelGradientPattern.Pat_90_Percent
```

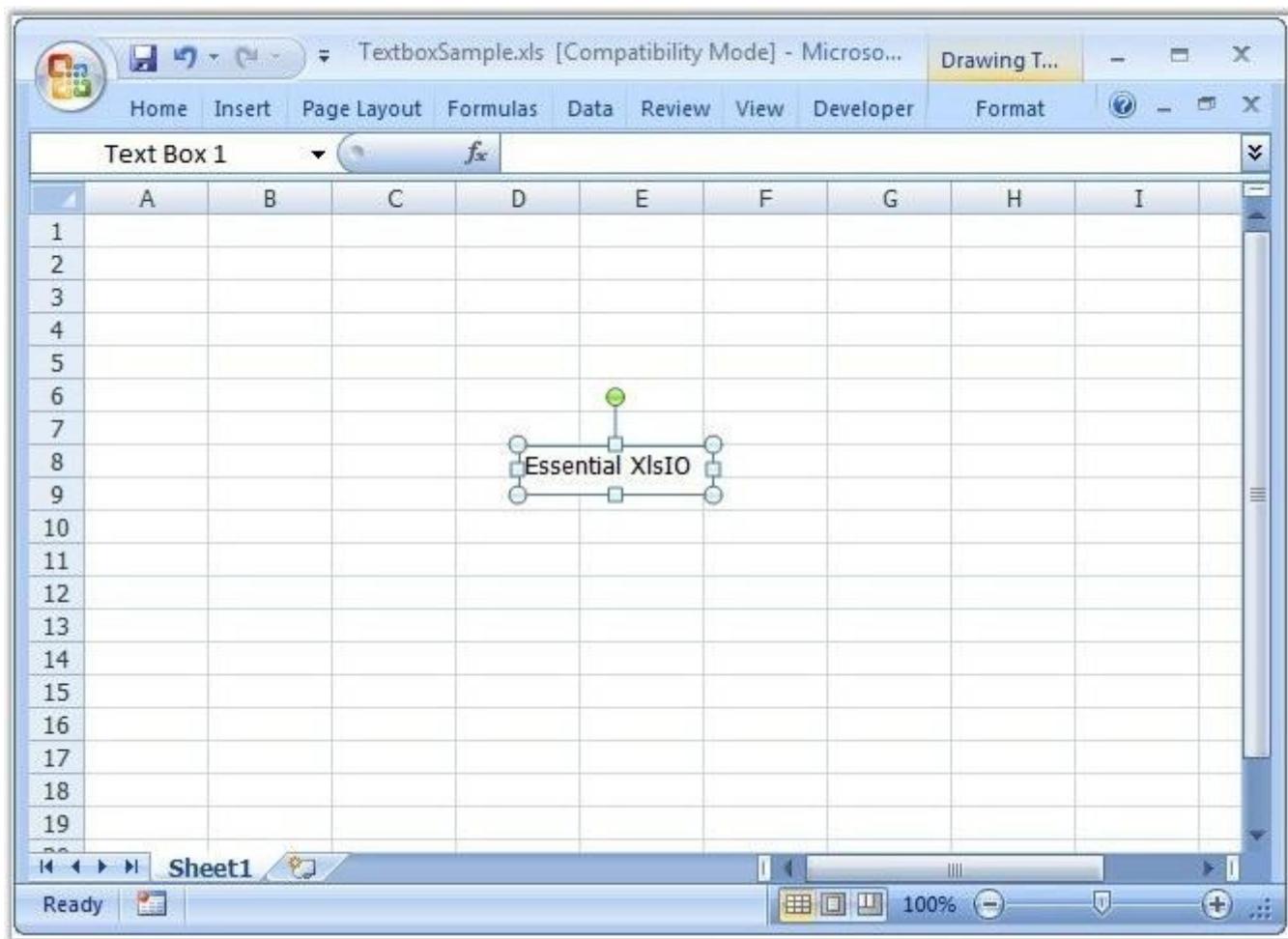


Figure 96: TextBox created using XlsIO

4.2.8.2 Check Box

Essential XlsIO supports reading and writing of check boxes. This can be done by using the **ICheckBoxShape** interface, which is used to add a check box inside a worksheet.

[C#]

```
//Create a check box with cell link.  
ICheckBoxShape chkBoxXlsIO = sheet.CheckBoxes.AddCheckBox(4, 4, 20,  
75);  
chkBoxXlsIO.Text = "XlsIO";  
chkBoxXlsIO.CheckState = ExcelCheckState.Checked;  
chkBoxXlsIO.LinkedCell = sheet["B4"];  
  
//Read a check box.  
ICheckBoxShape chkBoxXlsIO = sheet.CheckBoxes[0];  
chkBoxXlsIO.Name = "chkBoxXlsIO";
```

[VB .NET]

```
'Create a check box.  
Dim chkBoxXlsIO As ICheckBoxShape = sheet.CheckBoxes.AddCheckBox(5, 5,  
20, 75) chkBoxXlsIO.Text = "XlsIO"  
chkBoxXlsIO.CheckState = ExcelCheckState.Checked  
chkBoxXlsIO.LinkedCell = sheet ("B4")  
  
'Read a check box.  
Dim chkBoxXlsIO As ICheckBoxShape = sheet.CheckBoxes(0)  
chkBoxXlsIO.Name = "chkBoxXlsIO"
```

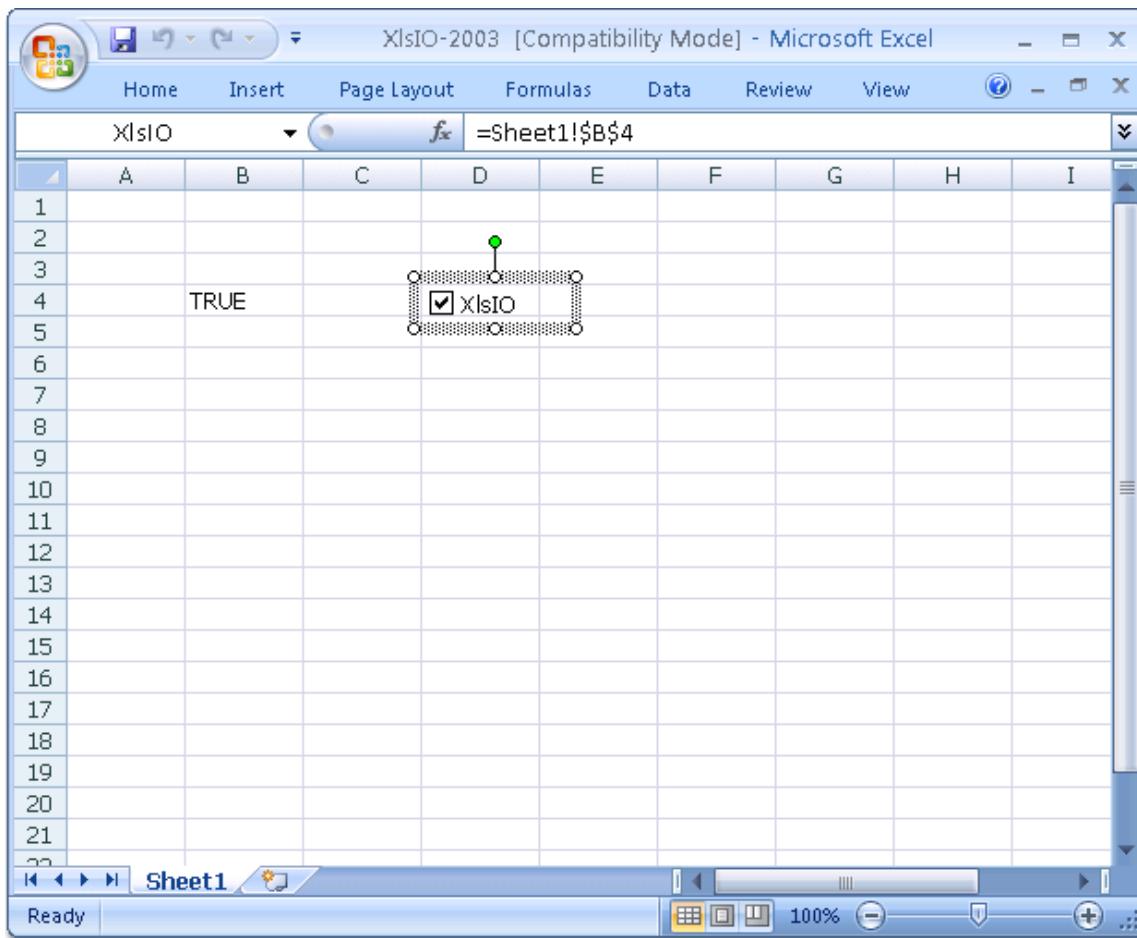


Figure 97: Check box with Linked Cell created using XlsIO

4.2.8.3 Combo Box

Essential XlsIO now provides support to read/write a Combo Box control. This is achieved by using the **IComboBoxShape** interface which is used to add a combo box inside a worksheet.

The following code example illustrates how to read/write a combo box control.

[C#]

```
// Create a Combo Box.
IComboBoxShape comboBox1 = sheet.ComboBoxes.AddComboBox(27, 5, 20, 100);

// Assign a value to the Combo Box.
comboBox1.ListFillRange = sheet["A1:A12"];
comboBox1.LinkedCell = sheet["C3"];
```

```
// Select an item.
comboBox1.SelectedIndex = 6;

// Read a Combo Box.
IComboBoxShape comboBox2 = sheet.ComboBoxes[0];
comboBox2.SelectedIndex = 3;
```

[VB.NET]

```
' Create a Combo Box.
Dim comboBox1 As IComboBoxShape = sheet.ComboBoxes.AddComboBox(27, 5, 20,
100)

' Assign a value to the Combo Box.
comboBox1.ListFillRange = sheet("A1:A12")
comboBox1.LinkedCell = sheet("C3")

' Select an item.
comboBox1.SelectedIndex = 6

' Read a Combo Box.
Dim comboBox2 As IComboBoxShape = sheet.ComboBoxes(0)
comboBox2.SelectedIndex = 3
```

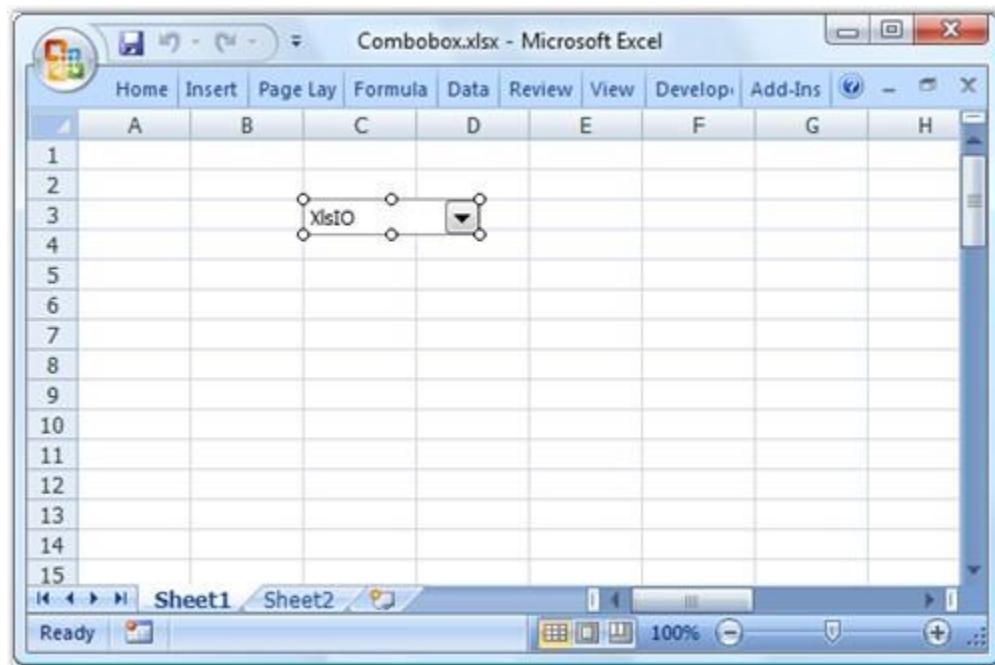


Figure 98: Combo Box control added to the Spreadsheet by using Essential XlsIO

4.2.8.4 Option Button

Essential XlsIO now provides support to read/write of Option Button control for **XLSX** format. This can be achieved by using the **IOptionButtonShape** interface which is used to add an option button inside a worksheet. The **IFill** interface is used to customize its appearance. **IShapeLineFormat** interface is used to modify the border. Various other text alignment properties are also supported.

The following code example illustrates how to read/write an option button control.

[C#]

```
ExcelEngine excelEngine = new ExcelEngine();
IApplication application = excelEngine.Excel;
application.DefaultVersion = ExcelVersion.Excel2007;
IWorkbook workbook = application.Workbooks.Create(1);
IWorksheet sheet = workbook.Worksheets[0];

// Create an Option Button.
IOptionButtonShape optionButton1 = sheet.OptionButtons.AddOptionButton(27,
5);

// Assign a value to the Option Button.
optionButton1.Text = "American Express";

// Format the control.
optionButton1.Fill.FillType = ExcelFillType.SolidColor;
optionButton1.Fill.ForeColor = Color.Yellow;

// Change the check state.
optionButton1.CheckState = ExcelCheckState.Checked;

// Save and close.
workbook.SaveAs("Sample.xlsx");
workbook.Close();
excelEngine.Dispose();

// Load the existing file.
excelEngine = new ExcelEngine();
application = excelEngine.Excel;
workbook = application.Workbooks.Open("Sample.xlsx",
ExcelOpenType.Automatic);
sheet = workbook.Worksheets[0];
```

```
// Read an Option Button.
IOptionButtonShape optionButton2 = sheet.OptionButtons[0];
optionButton2.CheckState = ExcelCheckState.Unchecked;

workbook.SaveAs("Unchecked.xlsx");
workbook.Close();
excelEngine.Dispose();
```

[VB .NET]

```
Dim excelEngine As New ExcelEngine()
Dim application As IApplication = excelEngine.Excel
application.DefaultVersion = ExcelVersion.Excel2007
Dim workbook As IWorkbook = application.Workbooks.Create(1)
Dim sheet As IWorksheet = workbook.Worksheets(0)

' Create an Option Button.
Dim optionButton1 As IOptionButtonShape =
sheet.OptionButtons.AddOptionButton(27, 5)

' Assign a value to the Option Button.
optionButton1.Text = "American Express"

' Format the control.
optionButton1.Fill.FillType = ExcelFillType.SolidColor
optionButton1.Fill.ForeColor = Color.Yellow

' Change the check state.
optionButton1.CheckState = ExcelCheckState.Checked

' Save and close.
workbook.SaveAs("Sample.xlsx")
workbook.Close()
excelEngine.Dispose()

' Load the existing file.
excelEngine = New ExcelEngine()
application = excelEngine.Excel
workbook = application.Workbooks.Open("Sample.xlsx",
ExcelOpenType.Automatic)
sheet = workbook.Worksheets(0)

' Read an Option Button.
Dim optionButton2 As IOptionButtonShape = sheet.OptionButtons(0)
```

```

optionButton2.CheckState = ExcelCheckState.Unchecked

workbook.SaveAs("Unchecked.xlsx")
workbook.Close()
excelEngine.Dispose()

```

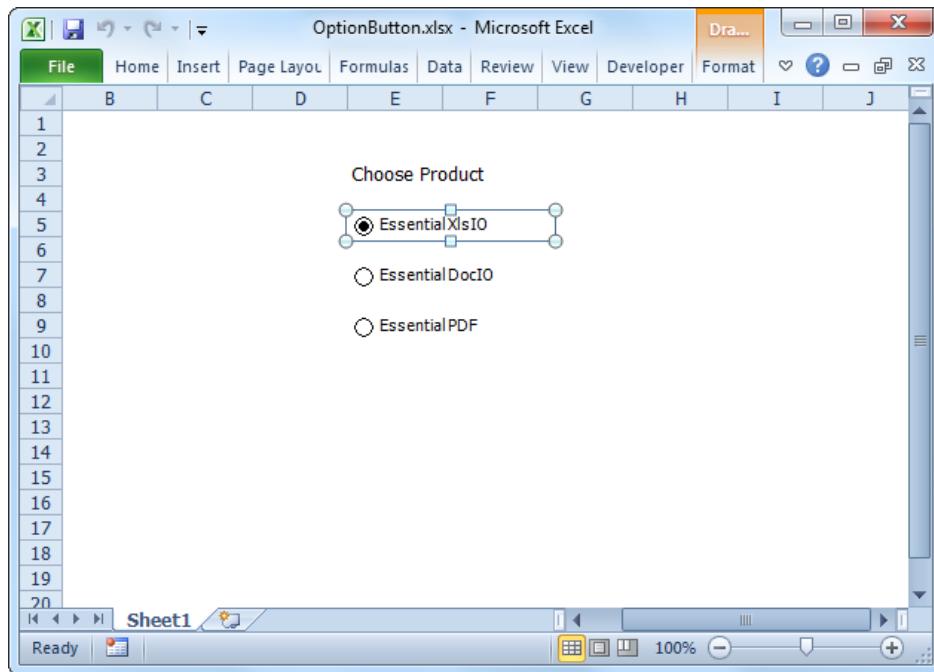


Figure 99: Option Button control added to the Spreadsheet by using Essential XlsIO

4.2.9 OLE Objects

Object Linking and Embedding (OLE) is one of the best known ways of inserting data into Microsoft Office documents. Though embedding or linking objects increase the size of the original document, they help improve the document readability by providing offline reading of documents (where the existing online links can be replaced). In order to read the content of the object, you may need the associated software to be installed in the machine. For example, a PDF file linked or embedded to an Excel file needs Adobe Reader in order to launch and read the PDF file.

Essential XlsIO supports read and write of OLE Objects in XLSX file format. Objects can either be linked or embedded in the Excel documents using **IOleObject** interface.



Note: Currently read and write functions for OLE Objects are supported in Windows, ASP.NET and WPF platforms only.

This section lists the following topics:

List of Properties

The following table lists the properties available.

Name of the Property	Type	Value Accepted	Description
DisplayAsIcon	Normal	Boolean	Gets or sets value indicating whether to display the OLE object as icon.
Location	Normal	IRange	Gets or sets the location of the OLE object in the sheet.
Picture	Normal	Image	Gets or sets the picture to display to represent the OLE object.
Shape	Normal	IPictureShape	Gets or sets picture shape object that defines look and position of the OLE Object inside the parent worksheet.
Size	Normal	System.Drawing.Size	Gets or sets the size of the OLE object.
OleObjectType	Normal	OleObjectType	Gets or sets the value indicating the type of OLE object.

Displaying an Object as Icon

The following set of code sample illustrates the condition when the property is set to True.

```
[C#]
oleObject1.DisplayAsIcon = true;
```

```
[VB .NET]
oleObject1.DisplayAsIcon = True
```

Run the code. The following output is generated.



Figure 100: Displayed as Icon

Setting the Location of an Object

The following set of code sample illustrates the condition when the location is set to K column, 8th cell.

[C#]

```
oleObject1.Location = sheet["K8"];
```

[VB .NET]

```
oleObject1.Location = sheet("K8")
```

Run the code. The following output is generated.

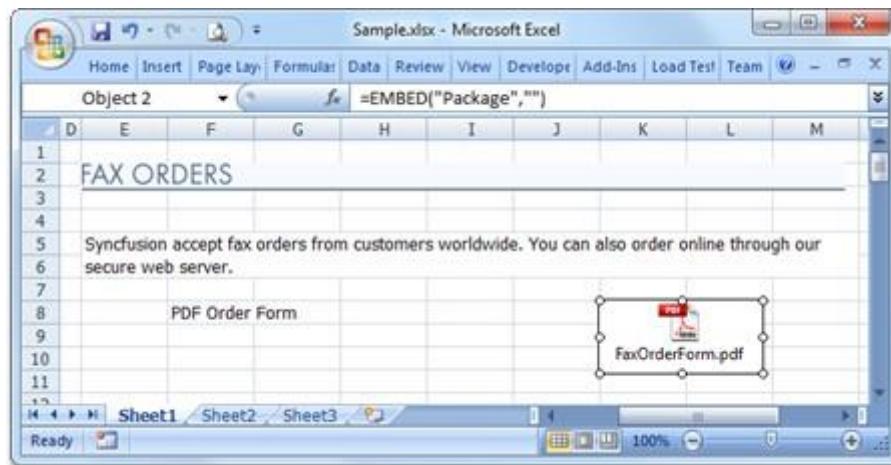


Figure 101: Location

Setting an Image an Object

The following set of code sample illustrates the condition when the property is set to any .png file image.

[C#]

```
oleObject1.Picture = Image.FromFile(@"image.png");
```

[VB .NET]

```
oleObject1.Picture = Image.FromFile("image.png").Picture = Image.FromFile("image.png")
```

Run the code. The following output is generated.



Figure 102: Word Icon Image

Setting Required Shape as an Object

The following set of code sample illustrates the condition when the property is set to any .png file image.

[C#]

```
oleObject1.Shape = sheet.Pictures[0];
```

[VB.NET]

```
oleObject1.Shape = sheet.Pictures(0)
```

Run the code. The following output is generated.



Figure 103: Word Icon Shape

Setting the Size of the Object

The following set of code sample illustrates the condition when the property is set to (30,30).

[C#]

```
oleObject1.Size = new Size(30, 30);
```

[VB.NET]

```
oleObject1.Size = New Size(30, 30)
```

Run the code. The following output is generated.



Figure 104: Size-Displayed Icon

OLE Objects and Linking Types

XlsIO supports two types of association of objects:

- Objects can be linked to the program
- Objects can be embedded in the program

1. Linked Objects

Linked objects remain as separate files. The linked object would expect the object to be in the same location as created, when the file is opened in another machine.

Linking an OLE Object to an Excel document

The following sample code illustrates how to linking an OLE Object to an Excel document.

```
[C#]

Image image = Image.FromFile(@"..\..\image.png");

// Select the object to insert, image for the display icon and the type of
// the OleObject field.
IOleObject ole2 = sheet.OleObjects.Add(@"..\..\Document.docx", image,
OleLinkType.Link);

// Set the location
ole2.Location = sheet[5, 12];

// Set the size
```

```
ole2.Size = new Size(30, 30);
```

[VB .NET]

```
Dim image As Image = Image.FromFile("../..\image.png")

' Select the object to insert, image for the display icon and the type of
the OleObject field.
Dim ole2 As IOleObject = sheet.OleObjects.Add("../..\Document.docx", image,
OleLinkType.Link)

' Set the location
ole2.Location = sheet(5, 12)

' Set the size
ole2.Size = New Size(30, 30)
```

Run the code. The following output is generated.

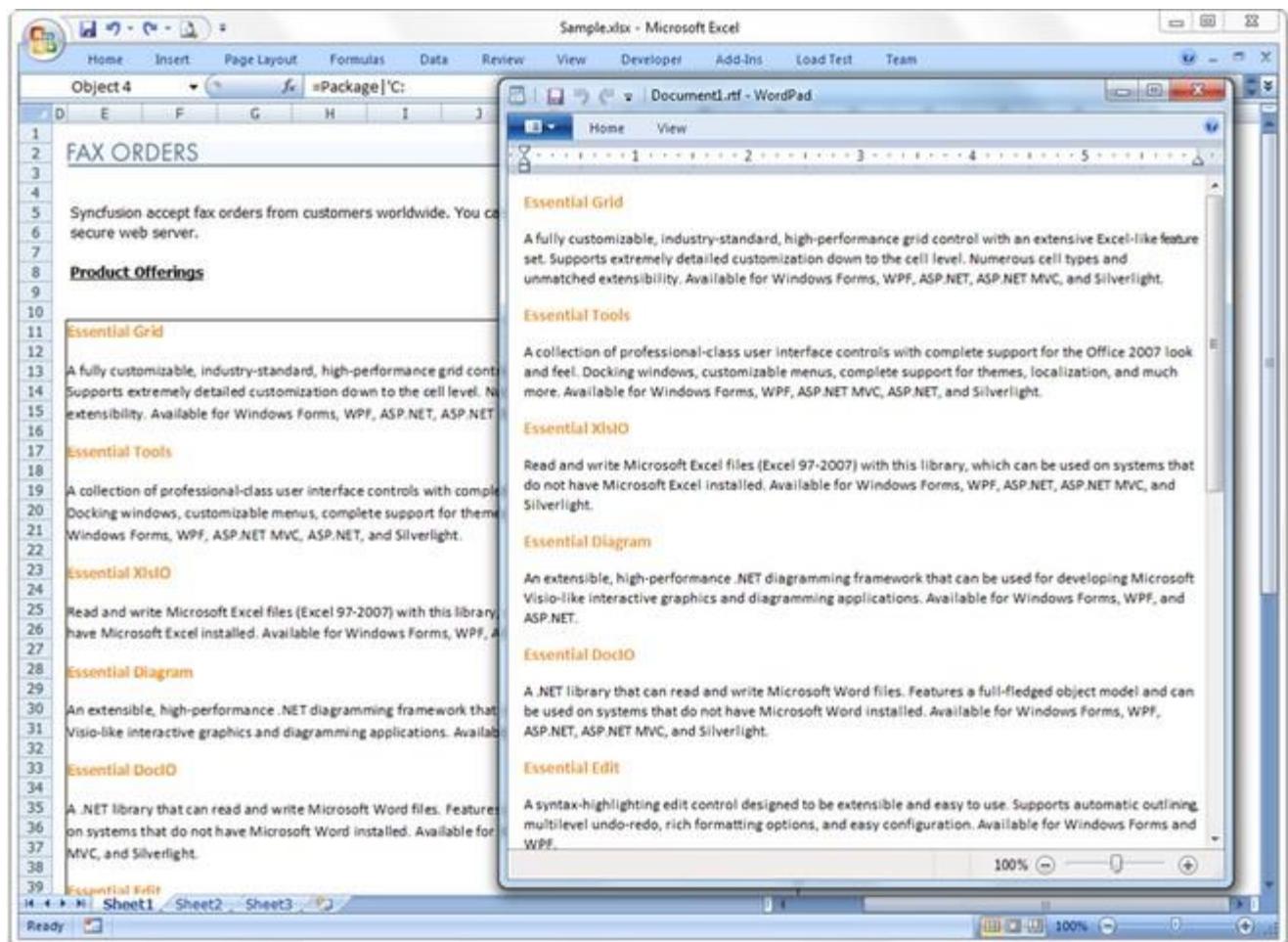


Figure 105: Linked Object

1. Embedded Objects

Embedded objects are stored in the document. When the file is opened in another machine, the embedded object can be viewed without having access to the original data.

Embedding an OLE Object in an Excel document

The following sample code illustrates how to embed an OLE Object to an Excel document.

[C#]

```
Image image = Image.FromFile(@"..\..\image.png");
// Select the object to insert, image for the display icon and the type of
```

```
the OLEObject field.  
IOleObject ole1 = sheet.OleObjects.Add(@"..\..\Test.pptx", image,  
OleLinkType.Embed);  
  
// Set the location  
ole1.Location = sheet[5, 2];  
  
// Set the size  
ole1.Size = new Size(30, 30);
```

[VB.NET]

```
Dim image As Image = Image.FromFile("../..\image.png")  
  
' Select the object to insert, image for the display icon and the type of  
' the OLEObject field.  
Dim ole1 As IOleObject = sheet.OleObjects.Add("../..\Test.pptx", image,  
OleLinkType.Embed)  
  
' Set the location  
ole1.Location = sheet(5, 2)  
  
' Set the size  
ole1.Size = New Size(30, 30)
```

Run the code. The following output is generated.

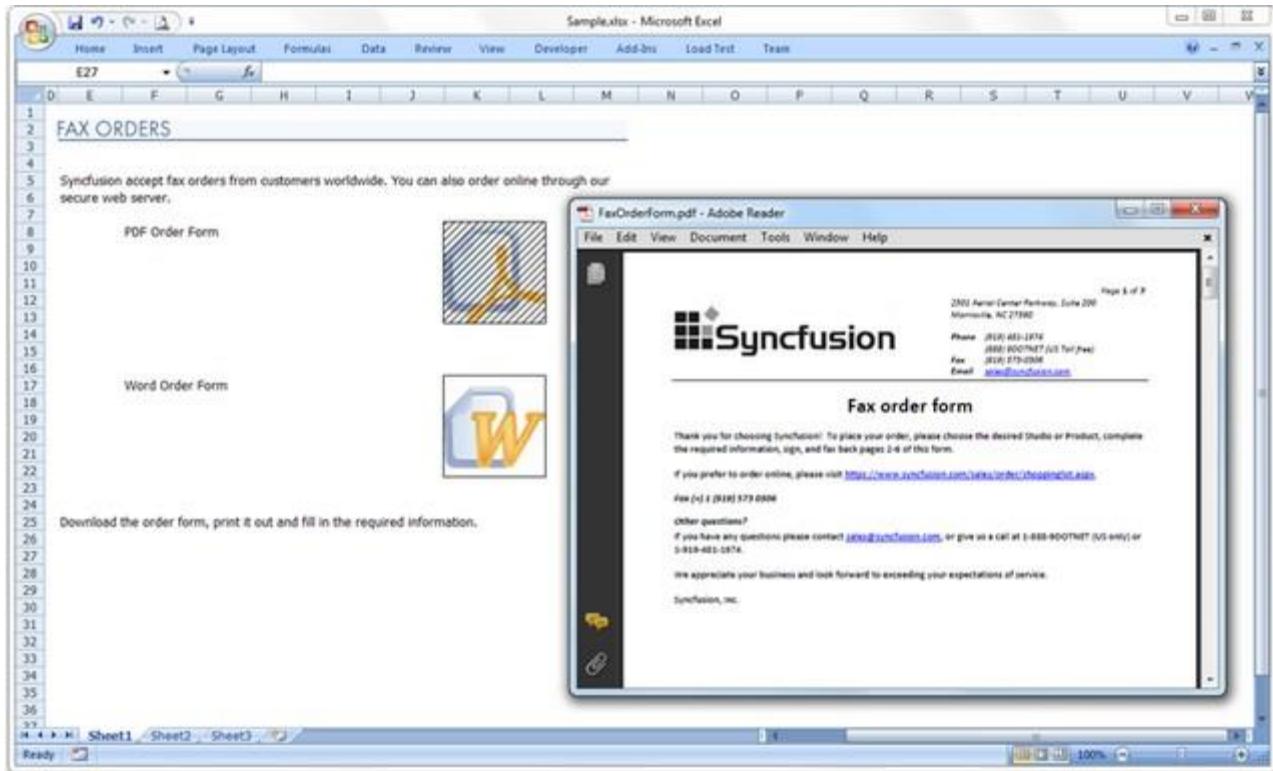


Figure 106: Embedded Object

Setting the Type of the OLE Object

The following code samples illustrate the condition when the property is set to `AdobeAcrobatDocument`.

[C#]

```
oleObject1.OleObjectType = OleObjectType.AdobeAcrobatDocument;
```

[VB .NET]

```
oleObject1.OleObjectType = OleObjectType.AdobeAcrobatDocument
```

4.3 Page Layout

Excel also provides support to control the layout of a particular page. It provides various customization options to customize the page margin, orientation, size, and so on.

Following topics explain the various ways your spreadsheet fits onto paper, and how it can be controlled by using XlsIO.

4.3.1 Page Setup

In MS Excel, the way the spreadsheet fits onto paper can be controlled through the **Page Setup** dialog box. You can select the size and orientation of the paper, the width of the margins, what goes into the header and footer of each page, and the order of printing cells for sheets that will take several pieces of paper.

Note: Though the sample code uses sheet object, it is possible to read/write page setup options for chart worksheet and embedded chart using **IChartPageSetup** interface.

There may also be a need to change the first page number, which starts with '1', by default. This can be done through the page number customization options provided by the Page Setup dialog box.

[C#]

```
sheet.PageSetup.AutoFirstPageNumber = false;
sheet.PageSetup.FirstPageNumber = 2;
```

[VB .NET]

```
sheet.PageSetup.AutoFirstPageNumber = false;
sheet.PageSetup.FirstPageNumber = 2;
```

Following topics explain how various other page setup options can be set by using XlsIO.

4.3.1.1 Margins

Page margins are the blank spaces between the worksheet data and the edges of the printed page, and hence provide better readability. Page margins can be used for items such as headers, footers and page numbers.

Excel allows to set the page margin through the Page Setup dialog box. Note that the page margins that you define in a given worksheet, are stored with that particular worksheet, when you save the workbook. You cannot change the default page margins for new workbooks.

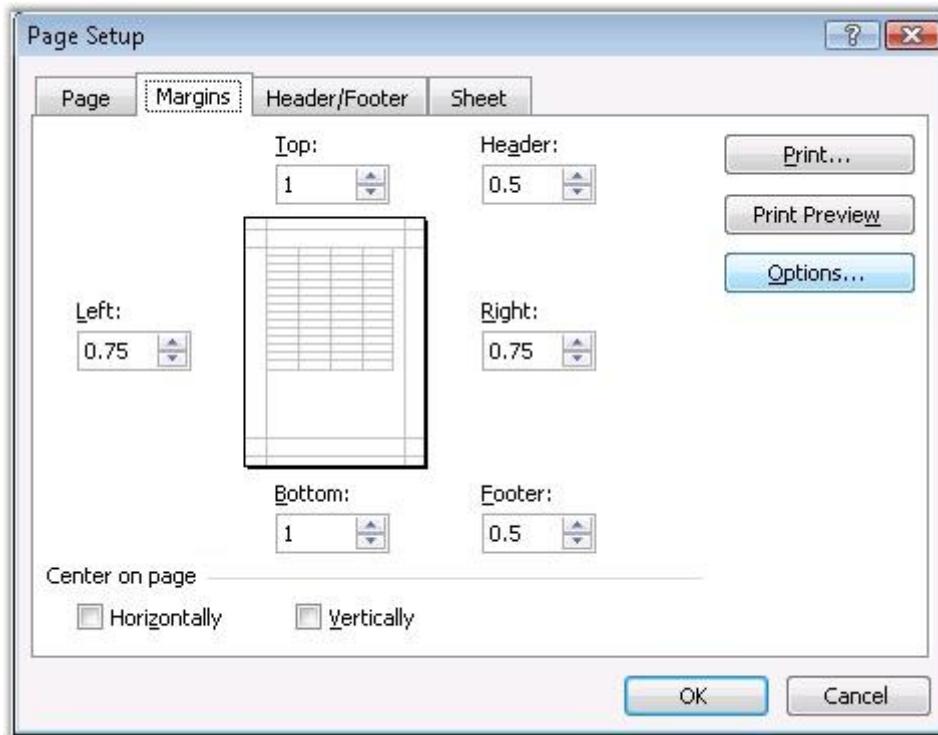


Figure 107: Page Setup - Margins

XlsIO has APIs to define the margins in a sheet through the properties of `IPageSetup`. It sets the value in terms of inches.

Following code example illustrates how to set the margin.

[C#]

```
// Page Setup Using Margins.
sheet.PageSetup.LeftMargin = 2;
sheet.PageSetup.RightMargin = 2;
sheet.PageSetup.TopMargin = 2;
sheet.PageSetup.BottomMargin = 2;
```

[VB .NET]

```
' Page Setup Using Margins.
sheet.PageSetup.LeftMargin = 2
sheet.PageSetup.RightMargin = 2
sheet.PageSetup.TopMargin = 2
sheet.PageSetup.BottomMargin = 2
```

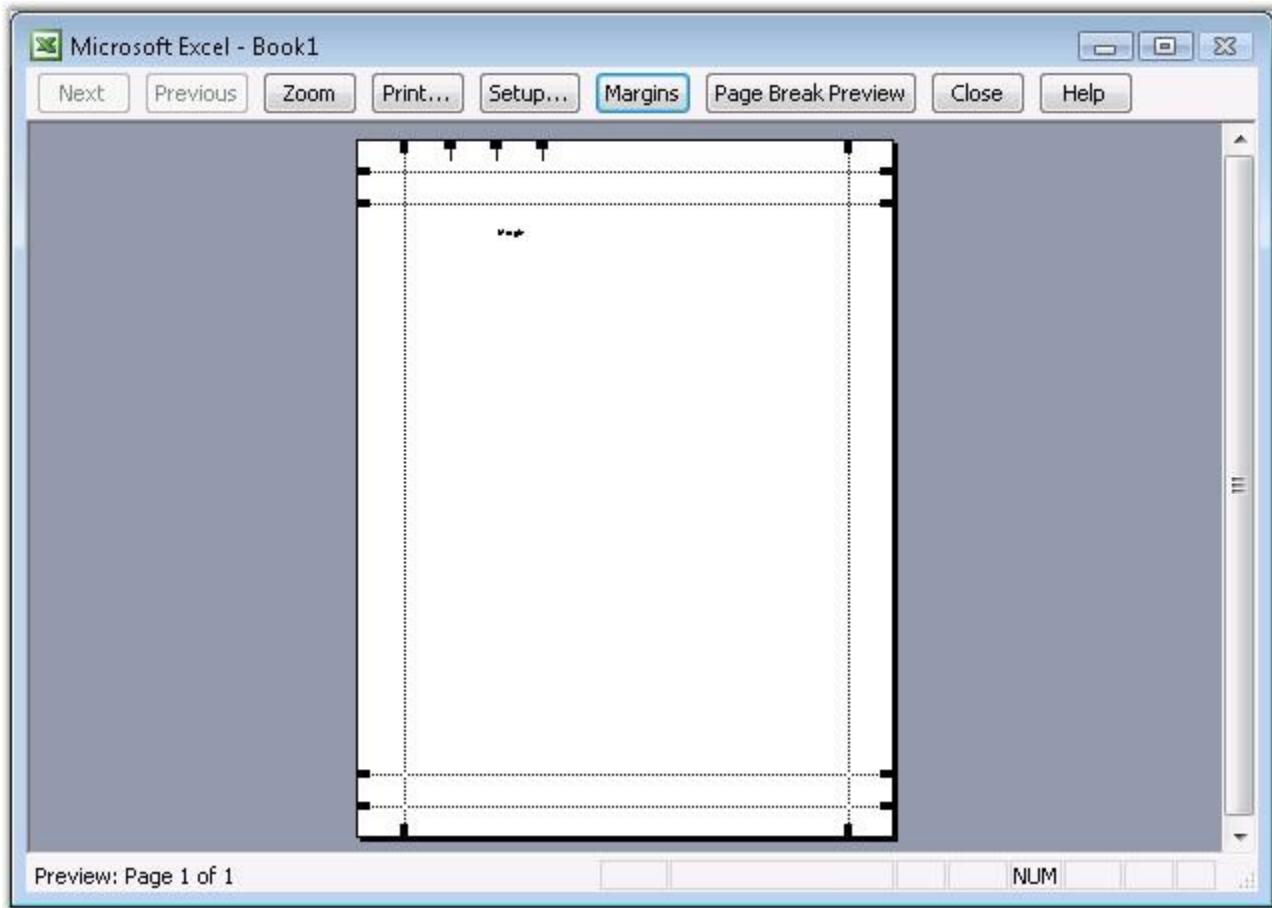


Figure 108: XlsIO with Margins

4.3.1.2 Orientation

While creating small worksheets, it is not necessary to change the direction/orientation of the pages, but some worksheets and charts require the width of the pages to be greater than its length.

Similar to the landscape painting, whose width is greater than the length, Landscape page orientation enables you to fit wider items on a page. A page, whose width is greater than the length, is called a Portrait orientation, like portraits of people.

Excel allows to change the orientation of the page, through the Page Setup dialog box. It allows to change the orientation to Landscape or Portrait.

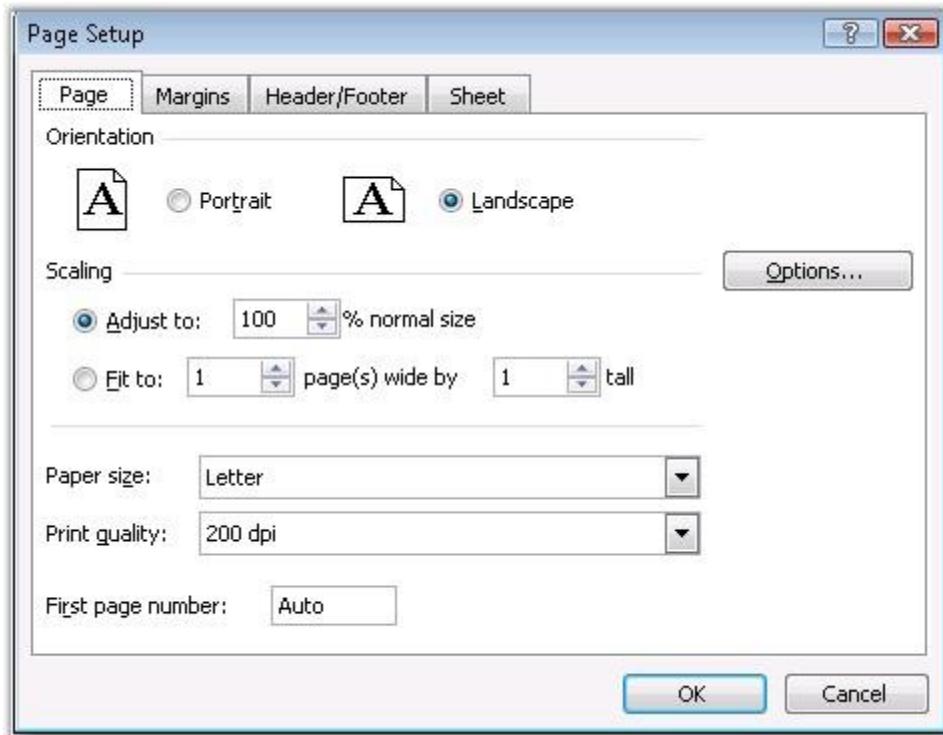


Figure 109: Page Setup - Page Orientation

XlsIO defines the orientation through the **Orientation** property of **IPageSetup**. Following code example illustrates how to set the page orientation.

[C#]

```
// Setting the Page Orientation as Portrait or Landscape.
sheet.PageSetup.Orientation = ExcelPageOrientation.Landscape;
```

[VB.NET]

```
' Setting the Page Orientation as Portrait or Landscape.
sheet.PageSetup.Orientation = ExcelPageOrientation.Landscape
```

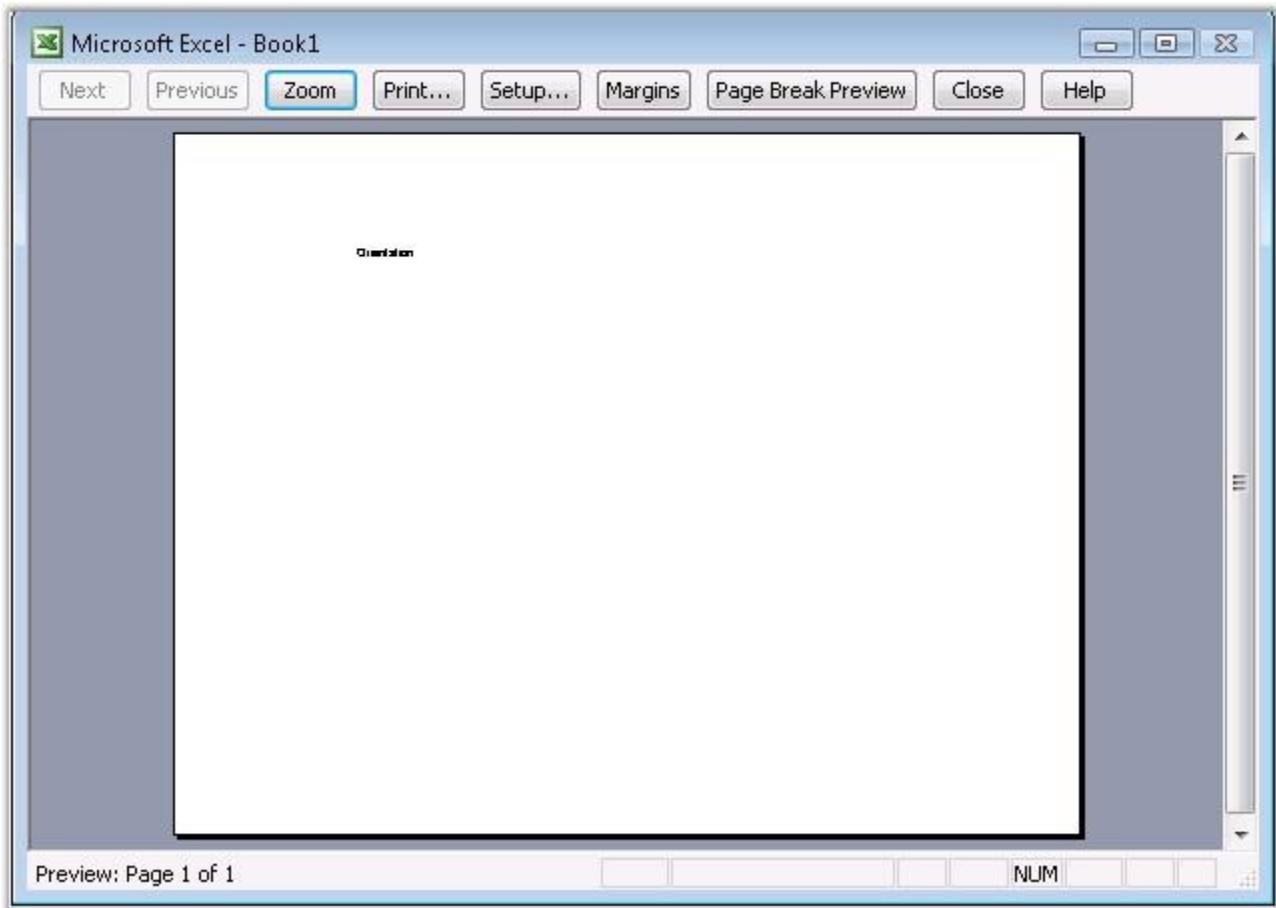


Figure 110: XlsIO with Page Orientation Set

4.3.1.3 Paper Size

In order to fit information on a page or change the appearance of the page, you may want to customize your page layout. One better option is to change the paper size of the worksheet, as per the need.

The default paper size in Excel is 8 1/2" x 11" sheets, but it can be changed through the **Page Setup** dialog box. XlsIO allows to change the paper size through the **PaperSize** property.

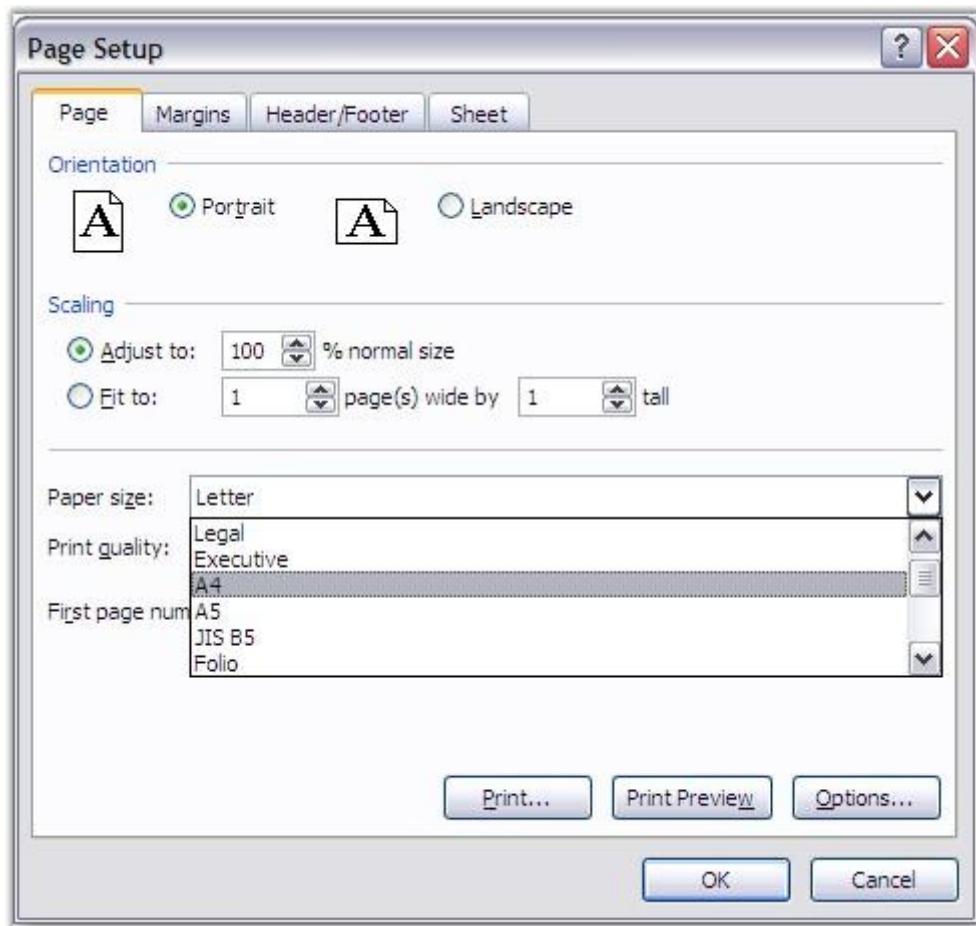


Figure 111: Page Setup - Page [Paper Size]

Following code example illustrates how to set the paper size in XlsIO.

[C#]

```
// Setting the Paper Type.
sheet.PageSetup.PaperSize = ExcelPaperSize.PaperA3;
```

[VB .NET]

```
' Setting the Paper Type.
sheet.PageSetup.PaperSize = ExcelPaperSize.PaperA3
```

4.3.1.4 Breaks

Page Breaks are dividers that break a worksheet into separate pages for printing. To print a worksheet with the exact number of pages that you want, you can adjust the page breaks in the worksheet before you print it. Excel inserts automatic page breaks, based on the paper size, margin settings, scaling options, and the positions of any manual page breaks that you insert, and it also allows to insert/remove breaks at preferred locations.

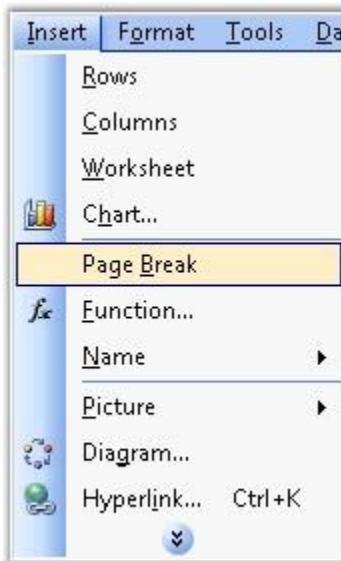


Figure 112: Insert menu -> Page Break

XlsIO provides support for inserting/removing Horizontal and Vertical page breaks in a worksheet by using the **IHPagbreak** and **IVPagebreak** interfaces.



Note: By default, page breaks are not shown in the Normal view. However, you can view them by inserting new page breaks.

```
[C#]
// Entering text into the cells.
sheet.Range["A1:M20"].Text = "PageBreak";

// Giving Horizontal Page Breaks.
sheet.HPageBreaks.Add(sheet.Range["A5"]);
sheet.HPageBreaks.Add(sheet.Range["A10"]);
sheet.HPageBreaks.Add(sheet.Range["A15"]);

// Giving Vertical Page Breaks.
sheet.VPageBreaks.Add(sheet.Range["B5"]);
sheet.VPageBreaks.Add(sheet.Range["E10"]);
sheet.VPageBreaks.Add(sheet.Range["K15"]);
```

[VB.NET]

```
' Entering text into the cells.  
sheet.Range("A1:M20").Text = "PageBreak"  
  
' Giving Horizontal Page Breaks.  
sheet.HPageBreaks.Add(sheet.Range("A5"))  
sheet.HPageBreaks.Add(sheet.Range("A10"))  
sheet.HPageBreaks.Add(sheet.Range("A15"))  
  
' Giving Vertical Page Breaks.  
sheet.VPageBreaks.Add(sheet.Range("B5"))  
sheet.VPageBreaks.Add(sheet.Range("E10"))  
sheet.VPageBreaks.Add(sheet.Range("K15"))
```

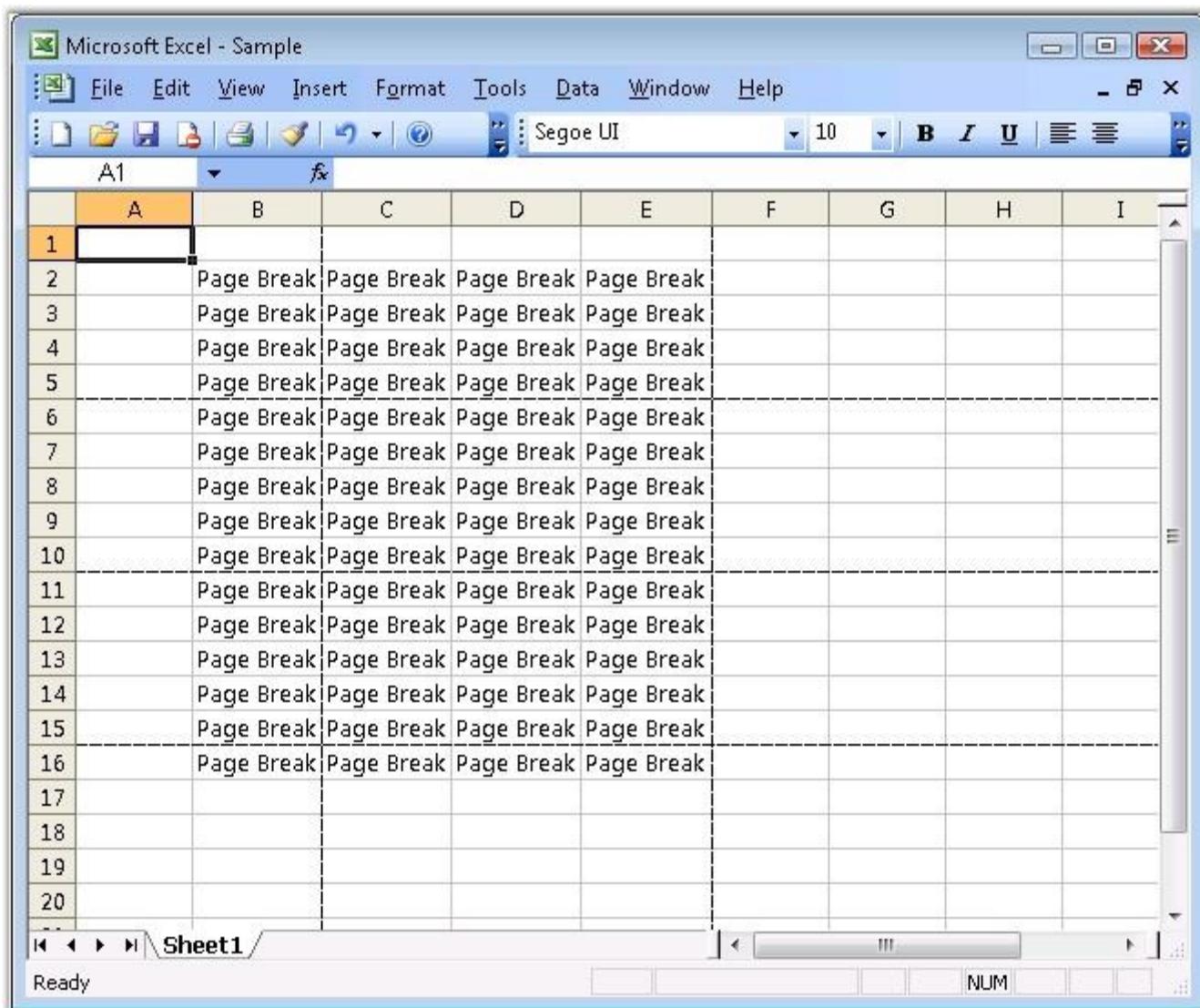


Figure 113: XlsIO with Page Breaks

You can also display or hide page breaks in the normal view by using the **DisplayPageBreaks** property of **IWorksheet**.

4.3.1.5 Background

MS Excel enables to set the background for the worksheet with an image, which covers the entire worksheet. Depending upon the image size and type, the background graphic may either be stretched across your worksheet or tiled.

 **Note:** The sheet backgrounds may tremendously increase the file size of the workbooks.

Background images which are set this way, cannot be printed. To set a Watermark that can be printed, you can make use of Headers and Footers. This can be viewed only through the **Print Preview** option, and it is not visible in the Normal view.

XlsIO provides support for inserting background images through the **BackgroundImage** property of **IPageSetup**.

Following code example illustrates how to insert a background image.

[C#]

```
// Setting the Paper Type.  
sheet.PageSetup.BackgroundImage = image;
```

[VB.NET]

```
' Setting the Paper Type.  
sheet.PageSetup.BackgroundImage = image
```

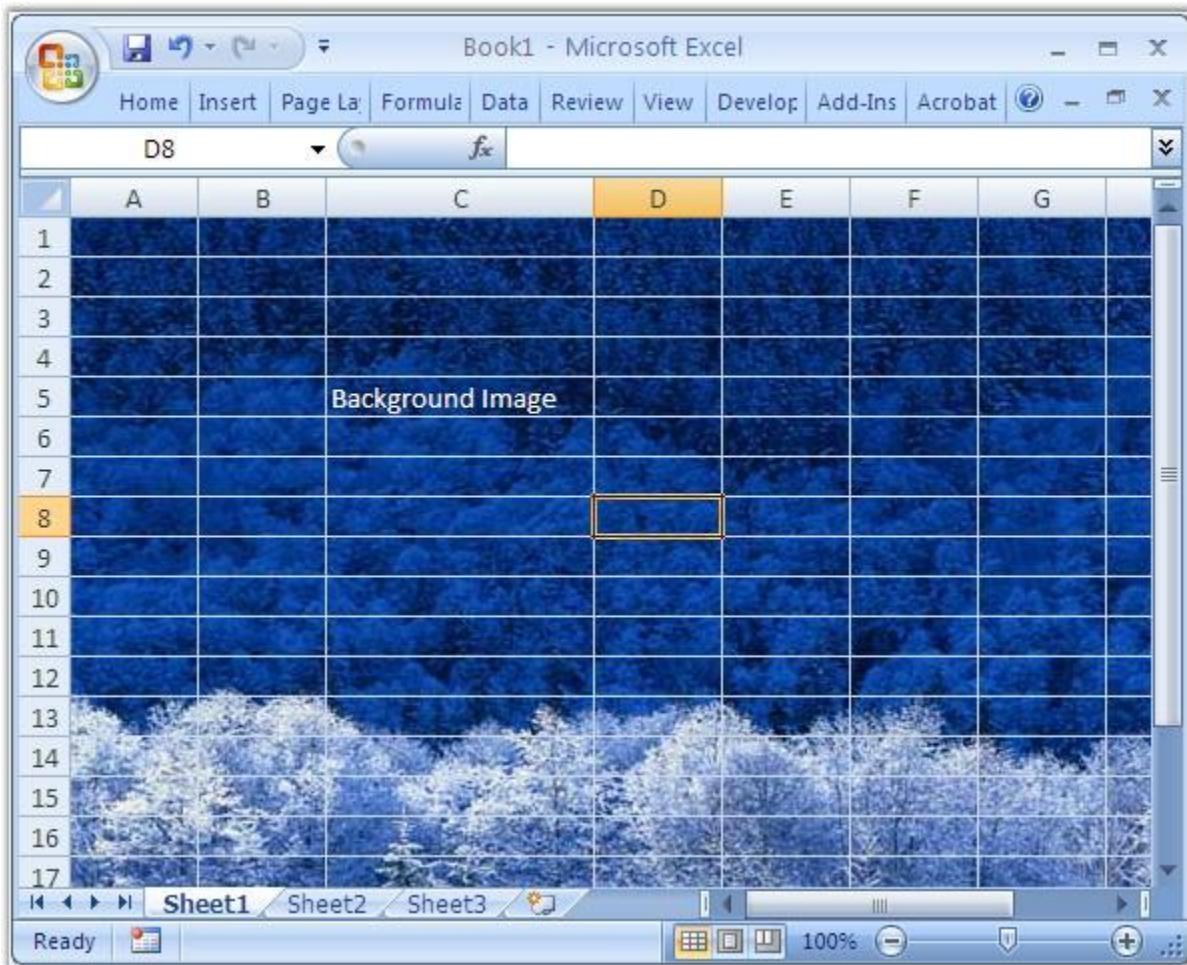


Figure 114: Setting Background Image Using XlsIO

4.3.2 Scale To Fit

Scaling lets you specify a certain percentage to reduce or enlarge your worksheet. The "Fit to" Page feature allows you to force the worksheet to print on a specific number of pages, without having to calculate the percentage yourself.

When you need to print a worksheet that is too large to fit on the page, without making the font very small, you can use the Orientation and Scaling features.

Excel enables this feature through the Page Setup dialog box.

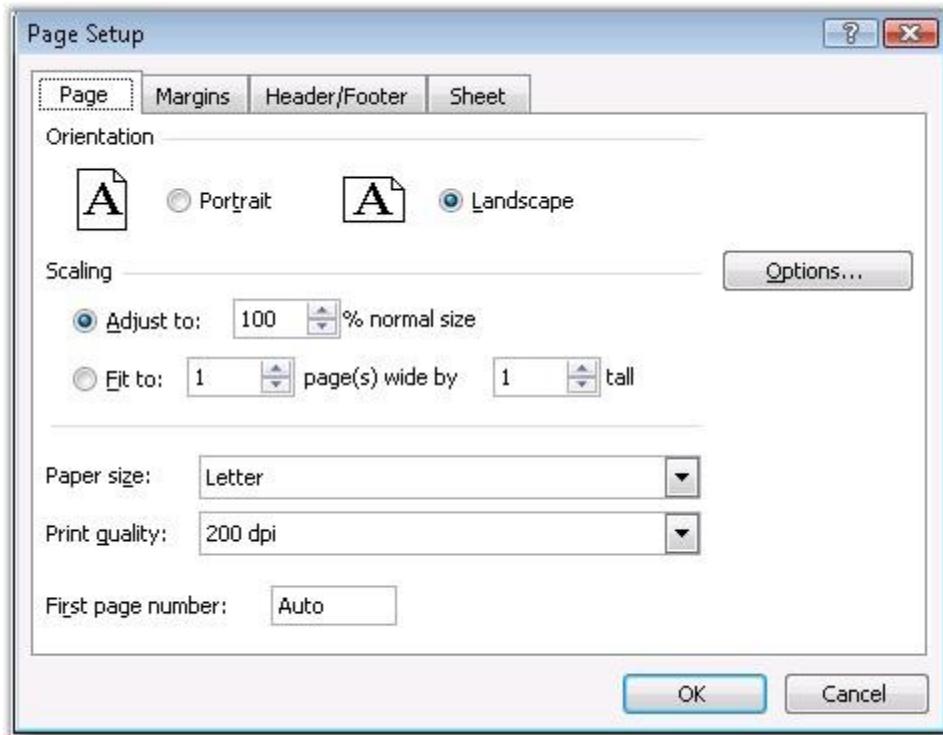


Figure 115: Page Setup - Page Scaling

XlsIO allows to scale the pages length and width wise, while printing. The following code example illustrates this.

[C#]

```
sheet.PageSetup.FitToPagesTall = 2;
sheet.PageSetup.FitToPagesWide = 3;
```

[VB .NET]

```
sheet.PageSetup.FitToPagesTall = 2;
sheet.PageSetup.FitToPagesWide = 3;
```

4.3.3 Sheet Options

Spreadsheets can be large. The **Sheet** tab in the **Page Setup** dialog box can be used to print only the cells required by the user. MS Excel provides various options for customizing the sheet, for printing through the Sheet tab.

This section explains the various print settings that can be applied to a spreadsheet through the XlsIO's APIs.

4.3.3.1 Print settings

MS Excel enables to customize the print settings through the following options.

- Print Area
- Print Titles
- Printing Options
- Page Order

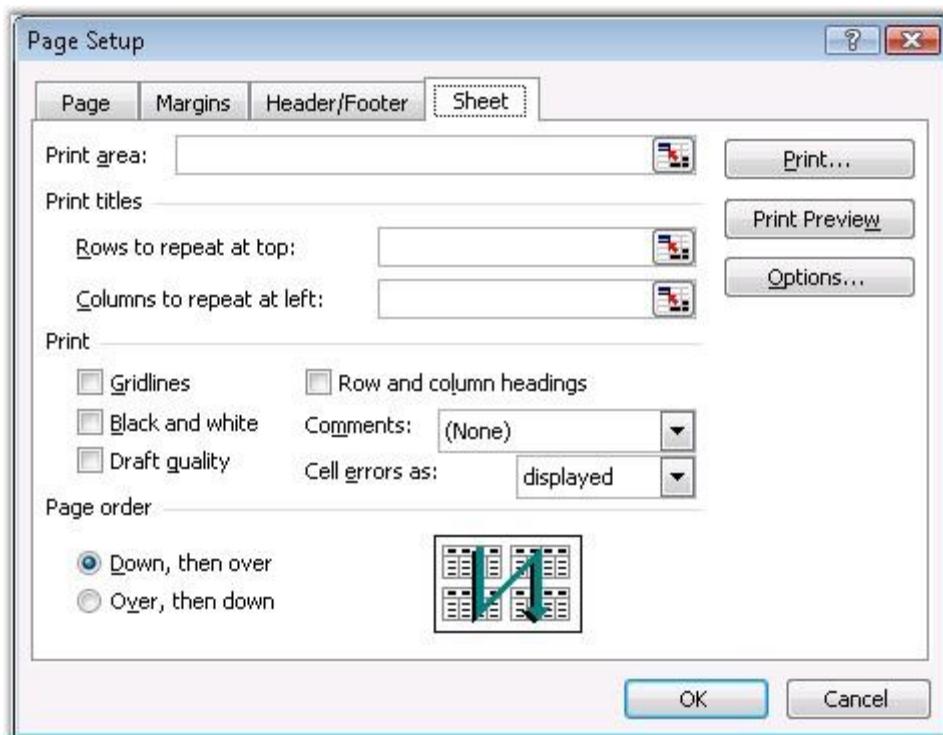


Figure 116: Page Setup - Sheet

This section explains the XlsIO's support for setting these options through simple APIs.

Print Area

The Print Area specifies the range of cells to be printed. You can set the printing range through the **PrintArea** property.

[C#]

```
//Sets printing range
sheet.PageSetup.PrintArea = "$G$7:$K$9,$G$11,$H$12,$I$13,$J$14";
```

[VB .NET]

```
'Sets printing range
sheet.PageSetup.PrintArea = "$G$7:$K$9,$G$11,$H$12,$I$13,$J$14";
```

Print Titles

MS Excel provides an option to repeat rows and columns, so that the labels will be displayed on every page that it takes to print the sheet. This can be selected through the Sheet tab of the Page Setup dialog box.

XlsIO allows setting these titles through the APIs discussed in the following code.

[C#]

```
sheet.PageSetup.PrintTitleColumns = "$B$1:$C$65536";
sheet.PageSetup.PrintTitleRows = "";
```

[VB .NET]

```
sheet.PageSetup.PrintTitleColumns = "$B$1:$C$65536"
sheet.PageSetup.PrintTitleRows = ""
```

Print Options

There are other settings that can be used to customize the Print options. They are as follows.

Grid Lines

These are the gray lines that separate the cells. Checking the box will enable them to print. These can be enabled/disabled through XlsIO by using the PrintGridlines property of IPageSetup interface.

Headings

Row and column headings are the row numbers and the column letters. Checking the box enables them to print in MS Excel. XlsIO has the option to enable/disable headings through the **PrintHeadings** property of **IPageSetup**. Remember, headings are not the same as the labels created.

Color

Excel allows to set the colors for printing. You can print a sheet without colors by using the **BlackAndWhite** property of the **IPageSetup** interface in XlsIO.

Quality

Excel provides options to toggle the quality by using the "DraftQuality" option. Draft quality is a fast, but not a crisp print quality. XlsIO allows to enable this option through the **Draft** property of the **IPageSetup** interface. You can also set the print quality that controls the dpi by using the **PrintQuality** property.

Comments

Comments are little notes that you can attach to cells. They can be printed all together at the end of the sheet, or within the sheet, or not at all. This can be set through XlsIO by using the **PrintNotes** property.

Page Order

Excel allows to set the page order in which the sections of a worksheet should be printed, when it does not fit on one paper page. The default option is **DownThenOver**.

The other option, **OverThenDown**, prints the cells across the top of the sheet, first, and then moves down to print the next set of rows.

XlsIO allows to set the print direction as illustrated in the following code example.

[C#]

```
// Set direction of printing.
workbook.Worksheets[0].PageSetup.Order = ExcelOrder.DownThenOver;
```

[VB .NET]

```
' Set direction of printing.  
workbook.Worksheets[0].PageSetup.Order = ExcelOrder.DownThenOver
```

4.4 Formulas

Formulas are entries in Excel that have an equation, which calculates the value to display. A typical formula might contain cells, constants, and even functions. Essential XlsIO has advanced support for working with Formulas.

Following topics elaborate on the various types of formulas, and their support by XlsIO.

4.4.1 Function Library

Excel supports various built-in functions that make large calculations in large sheets easier.

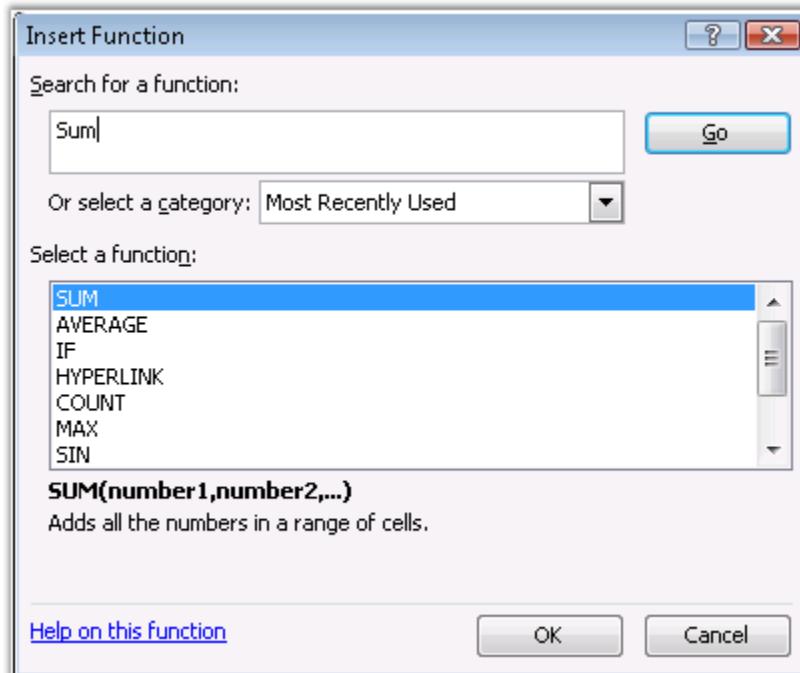


Figure 117: Insert Function Dialog Box in Excel

XlsIO provides support for reading and writing around 520+ predefined Excel functions.

Formula Writing

You can enter formulas in a spreadsheet by using the **Formula** property. Following code example illustrates the built-in function of Excel by using XlsIO APIs.

[C#]

```
// Excel Functions
sheet.Range["A22"].Text = "ABS(ABS(-A3))";
sheet.Range["B22"].Formula = "ABS(ABS(-A3))";

sheet.Range["A23"].Text = "ABS(ABS(-100))";
sheet.Range["B23"].Formula = "ABS(ABS(-100))";

sheet.Range["A24"].Text = "-A3";
sheet.Range["B24"].Formula = "-A3";

sheet.Range["A25"].Text = "ACOS(A8)";
sheet.Range["B25"].Formula = "ACOS(A8)";

sheet.Range["A26"].Text = "ADDRESS(1,1)";
sheet.Range["B26"].Formula = "ADDRESS(1,1)";

sheet.Range["A27"].Text = "ADDRESS(1,1,2)";
sheet.Range["B27"].Formula = "ADDRESS(1,1,2)";

sheet.Range["A28"].Text = "ADDRESS(1,1,3)";
sheet.Range["B28"].Formula = "ADDRESS(1,1,3)";

sheet.Range["A29"].Text = "ADDRESS(1,1,4)";
sheet.Range["B29"].Formula = "ADDRESS(1,1,4)";

sheet.Range["A30"].Text = "ASIN(A8)";
sheet.Range["B30"].Formula = "ASIN(A8)";

sheet.Range["A31"].Text = "ATAN(A8)";
sheet.Range["B31"].Formula = "ATAN(A8)";

sheet.Range["A32"].Text = "ATANH(A8);
```

```

sheet.Range["B32"].Formula = "ATANH(A8)";

sheet.Range["A33"].Text = "BETADIST(A8,A8,A8)";
sheet.Range["B33"].Formula = "BETADIST(A8,A8,A8)";

sheet.Range["A34"].Text = "BETAINV(A8,A8,A8)";
sheet.Range["B34"].Formula = "BETAINV(A8,A8,A8)";

sheet.Range["A35"].Text = "BINOMDIST(A4,A3,A8,A6)";
sheet.Range["B35"].Formula = "BINOMDIST(A4,A3,A8,A6)";

sheet.Range["A36"].Text = "CEILING(A3,A4)";
sheet.Range["B36"].Formula = "CEILING(A3,A4)";

sheet.Range["A37"].Text = "CELL(B3,A4)";
sheet.Range["B37"].Formula = "CELL(B3,A4)";

sheet.Range["A38"].Text = "CHAR(65)";
sheet.Range["B38"].Formula = "CHAR(65)";

sheet.Range["A39"].Text = "CHIDIST(A3,A4)";
sheet.Range["B39"].Formula = "CHIDIST(A3,A4)";

sheet.Range["A40"].Text = "CHIINV(A8,A4)";
sheet.Range["B40"].Formula = "CHIINV(A8,A4)";

sheet.Range["A41"].Text = "CHITEST(A3:A8,A13:F18)";
sheet.Range["B41"].Formula = "CHITEST(A3:A8,A13:F18)";

sheet.Range["A42"].Text =
"CHITEST({150,100,200,300,500,0.3},{95,155,195,305,495,0.7})";
sheet.Range["B42"].Formula =
"CHITEST({150,100,200,300,500,0.3},{95,155,195,305,495,0.7})";

sheet.Range["A43"].Text = "CONFIDENCE(A8,A4,A5)";
sheet.Range["B43"].Formula = "CONFIDENCE(A8,A4,A5)";

sheet.Range["A44"].Text = "CORREL(A3:A8,A13:A18)";
sheet.Range["B44"].Formula = "CORREL(A3:A8,A13:A18)";

sheet.Range["A45"].Text =
"CORREL({150,100,200,300,500,0.3},{95,155,195,305,495,0.7})";
sheet.Range["B45"].Formula =
"CORREL({150,100,200,300,500,0.3},{95,155,195,305,495,0.7})";

sheet.Range["A46"].Text = "CRITBINOM(A3,A8,A8)";

```

```

sheet.Range["B46"].Formula = "CRITBINOM(A3,A8,A8)";
sheet.Range["A47"].Text = "DATEVALUE(B8)";
sheet.Range["B47"].Formula = "DATEVALUE(B8)";

sheet.Range["A48"].Text = "DAYS360(A3,A4)";
sheet.Range["B48"].Formula = "DAYS360(A3,A4)";

sheet.Range["A49"].Text = "DOLLAR(A3)";
sheet.Range["B49"].Formula = "DOLLAR(A3)";

sheet.Range["A50"].Text = "FIND(B4,B7)";
sheet.Range["B50"].Formula = "FIND(B4,B7)";

sheet.Range["A51"].Text = "FINDB(B4,B7)";
sheet.Range["B51"].Formula = "FINDB(B4,B7)";

sheet.Range["A52"].Text = "FINV(A8,A4,A5)";
sheet.Range["B52"].Formula = "FINV(A8,A4,A5)";

sheet.Range["A53"].Text = "FISHER(A8)";
sheet.Range["B53"].Formula = "FISHER(A8)";

sheet.Range["A54"].Text = "FTEST(A3:A8,A13:A18)";
sheet.Range["B54"].Formula = "FTEST(A3:A8,A13:A18)";

sheet.Range["A55"].Text =
"FTEST({150,100,200,300,500,0.3},{95,155,195,305,495,0.7})";
sheet.Range["B55"].Formula =
"FTEST({150,100,200,300,500,0.3},{95,155,195,305,495,0.7})";

sheet.Range["A56"].Text = "FV(A8,A4,A5)";
sheet.Range["B56"].Formula = "FV(A8,A4,A5)";

sheet.Range["A57"].Text = "GAMMAINV(A8,A4,A5)";
sheet.Range["B57"].Formula = "GAMMAINV(A8,A4,A5)";

sheet.Range["A58"].Text = "HYPGEOMDIST(A4,A3,A5,A6)";
sheet.Range["B58"].Formula = "HYPGEOMDIST(A4,A3,A5,A6)";

sheet.Range["A59"].Text = "INDEX(A3,1)";
sheet.Range["B59"].Formula = "INDEX(A3,1)";

sheet.Range["A60"].Text = "INDEX({150,100,200,300,500,0.3},3)";
sheet.Range["B60"].Formula = "INDEX({150,100,200,300,500,0.3},3)";

```

```

sheet.Range["A61"].Text = "INDIRECT(B5)";
sheet.Range["B61"].Formula = "INDIRECT(B5)";

sheet.Range["A62"].Text = "INFO(B6)";
sheet.Range["B62"].Formula = "INFO(B6)";

sheet.Range["A63"].Text = "INTERCEPT(A3:A8,A13:A18)";
sheet.Range["B63"].Formula = "INTERCEPT(A3:A8,A13:A18)";

sheet.Range["A64"].Text =
"INTERCEPT({150,100,200,300,500,0.3},{95,155,195,305,495,0.7})";
sheet.Range["B64"].Formula =
"INTERCEPT({150,100,200,300,500,0.3},{95,155,195,305,495,0.7})";

sheet.Range["A65"].Text = "IPMT(A18,3,A5,A6)";
sheet.Range["B65"].Formula = "IPMT(A18,3,A5,A6)";

sheet.Range["A66"].Text = "IRR(A9:A12)";
sheet.Range["B66"].Formula = "IRR(A9:A12)";

sheet.Range["A67"].Text = "IRR({-100,100,200,150})";
sheet.Range["B67"].Formula = "IRR({-100,100,200,150})";

sheet.Range["A68"].Text = "KURT(A3:A8)";
sheet.Range["B68"].Formula = "KURT(A3:A8)";

sheet.Range["A69"].Text = "KURT({150,100,200,300,500,0.3})";
sheet.Range["B69"].Formula = "KURT({150,100,200,300,500,0.3})";

sheet.Range["A70"].Text = "LARGE(A13:A18,3)";
sheet.Range["B70"].Formula = "LARGE(A13:A18,3)";

sheet.Range["A71"].Text = "LARGE({95,155,195,305,495,0.7},3)";
sheet.Range["B71"].Formula = "LARGE({95,155,195,305,495,0.7},3)";

sheet.Range["A72"].Text = "LOGEST({10,20,30},{10,20,30})";
sheet.Range["B72"].Formula = "LOGEST({10,20,30},{10,20,30})";

sheet.Range["A73"].Text = "LOGNORMDIST({10,20,30},A4,A5)";
sheet.Range["B73"].Formula = "LOGNORMDIST({10,20,30},A4,A5)";

sheet.Range["A74"].Text = "MAX({10,20,30;5,15,35;6,16,36})";
sheet.Range["B74"].Formula = "MAX({10,20,30;5,15,35;6,16,36})";

sheet.Range["A75"].Text = "MAXA({10,20,30;5,15,35;6,16,36})";
sheet.Range["B75"].Formula = "MAXA({10,20,30;5,15,35;6,16,36})";

```

```

sheet.Range["A76"].Text = "MID(B6,A19,A19)";
sheet.Range["B76"].Formula = "MID(B6,A19,A19)";

sheet.Range["A77"].Text = "MID(\"Test string\",A19,A19*A19)";
sheet.Range["B77"].Formula = "MID(\"Test string\",A19,A19*A19)";

sheet.Range["A78"].Text = "MIDB(\"Test string\",A19,A19*A19)";
sheet.Range["B78"].Formula = "MIDB(\"Test string\",A19,A19*A19)";

sheet.Range["A79"].Text = "LOGINV(A8,A8,A8)";
sheet.Range["B79"].Formula = "LOGINV(A8,A8,A8)";

sheet.Range["A80"].Text = "LOOKUP(A3,{1,2,3,100})";
sheet.Range["B80"].Formula = "LOOKUP(A3,{1,2,3,100})";

sheet.Range["A81"].Text = "LOOKUP(A3,A3:A8)";
sheet.Range["B81"].Formula = "LOOKUP(A3,A3:A8)";

sheet.Range["A82"].Text = "LOOKUP(A3,A3:A8,A13:A18)";
sheet.Range["B82"].Formula = "LOOKUP(A3,A3:A8,A13:A18)";

sheet.Range["A83"].Text = "MATCH(A1,{1,2,3,4,5,100,200,300})";
sheet.Range["B83"].Formula = "MATCH(A1,{1,2,3,4,5,100,200,300})";

sheet.Range["A84"].Text = "MIRR(A9:A12,1,3)";
sheet.Range["B84"].Formula = "MIRR(A9:A12,1,3)";

sheet.Range["A85"].Text = "MIRR({-100,100,200,150},1,3)";
sheet.Range["B85"].Formula = "MIRR({-100,100,200,150},1,3)";

sheet.Range["A86"].Text = "MATCH(A3,A3:A8)";
sheet.Range["B86"].Formula = "MATCH(A3,A3:A8)";

sheet.Range["A87"].Text = "MDETERM({3,6,1;1,1,0;3,10,1})";
sheet.Range["B87"].Formula = "MDETERM({3,6,1;1,1,0;3,10,1})";

sheet.Range["A88"].Text = "MEDIAN({10,20,40,10,21})";
sheet.Range["B88"].Formula = "MEDIAN({10,20,40,10,21})";

sheet.Range["A89"].Text = "MIN({10,20,30;5,15,35;6,16,36})";
sheet.Range["B89"].Formula = "MIN({10,20,30;5,15,35;6,16,36})";

sheet.Range["A90"].Text = "MINA({10,20,30;5,15,35;6,16,36})";
sheet.Range["B90"].Formula = "MINA({10,20,30;5,15,35;6,16,36})";

```

```

sheet.Range["A91"].Text = "MODE(A3:A4)";
sheet.Range["B91"].Formula = "MODE(A3:A4)";

sheet.Range["A92"].Text = "NEGBINOMDIST(A3,A4,A8)";
sheet.Range["B92"].Formula = "NEGBINOMDIST(A3,A4,A8)";

sheet.Range["A93"].Text = "NORMINV(A8,A4,A5)";
sheet.Range["B93"].Formula = "NORMINV(A8,A4,A5)";

sheet.Range["A94"].Text = "NORMSINV(A8)";
sheet.Range["B94"].Formula = "NORMSINV(A8)";

sheet.Range["A95"].Text = "NPER(A3,A4,A5)";
sheet.Range["B95"].Formula = "NPER(A3,A4,A5)";

sheet.Range["A96"].Text = "NPV(A3,A4)";
sheet.Range["B96"].Formula = "NPV(A3,A4)";

sheet.Range["A97"].Text = "PEARSON(A3:A8,A13:A18)";
sheet.Range["B97"].Formula = "PEARSON(A3:A8,A13:A18)";

sheet.Range["A98"].Text = "PERCENTILE(A3:A8,A18)";
sheet.Range["B98"].Formula = "PERCENTILE(A3:A8,A18)";

sheet.Range["A99"].Text = "PERCENTRANK(A3:A8,A3)";
sheet.Range["B99"].Formula = "PERCENTRANK(A3:A8,A3)";

sheet.Range["A100"].Text = "PERMUT(A3,2)";
sheet.Range["B100"].Formula = "PERMUT(A3,2)";

sheet.Range["A101"].Text = "PMT(A3,A4,A5)";
sheet.Range["B101"].Formula = "PMT(A3,A4,A5)";

sheet.Range["A102"].Text = "PPMT(A8,A4,A5,A6)";
sheet.Range["B102"].Formula = "PPMT(A8,A4,A5,A6)";

sheet.Range["A103"].Text = "PROB(A3:A4,A8:A18,A3)";
sheet.Range["B103"].Formula = "PROB(A3:A4,A8:A18,A3)";

sheet.Range["A104"].Text = "PRODUCT({150,2,3,4,5,20})";
sheet.Range["B104"].Formula = "PRODUCT({150,2,3,4,5,20})";

sheet.Range["A105"].Text = "PV(A3,A4,A5)";
sheet.Range["B105"].Formula = "PV(A3,A4,A5)";

sheet.Range["A106"].Text = "QUARTILE(A3:A7,A8)";

```

```

sheet.Range["B106"].Formula = "QUARTILE(A3:A7,A8)";

sheet.Range["A107"].Text = "RATE(A19,-A3,A4)";
sheet.Range["B107"].Formula = "RATE(A19,-A3,A4)";

sheet.Range["A108"].Text = "RANK(A3,A3:A8)";
sheet.Range["B108"].Formula = "RANK(A3,A3:A8)";

sheet.Range["A109"].Text = "RSQ(A3:A8,A18:A18)";
sheet.Range["B109"].Formula = "RSQ(A3:A8,A18:A18)";

sheet.Range["A110"].Text = "SEARCH(B4,B7)";
sheet.Range["B110"].Formula = "SEARCH(B4,B7)";

sheet.Range["A111"].Text = "SEARCHB(B4,B7)";
sheet.Range["B111"].Formula = "SEARCHB(B4,B7)";

sheet.Range["A112"].Text = "SKEW(A3:A8)";
sheet.Range["B112"].Formula = "SKEW(A3:A8)";

sheet.Range["A113"].Text = "SLOPE(A3:A8,A13:A18)";
sheet.Range["B113"].Formula = "SLOPE(A3:A8,A13:A18)";

sheet.Range["A114"].Text = "SMALL(A3:A8,3)";
sheet.Range["B114"].Formula = "SMALL(A3:A8,3)";

sheet.Range["A115"].Text = "STDEV(A3:A8)";
sheet.Range["B115"].Formula = "STDEV(A3:A8)";

sheet.Range["A116"].Text = "STDEVA(A3:A8)";
sheet.Range["B116"].Formula = "STDEVA(A3:A8)";

sheet.Range["A117"].Text = "STDEVP(A3:A8)";
sheet.Range["B117"].Formula = "STDEVP(A3:A8)";

sheet.Range["A118"].Text = "STDEVPA(A3:A8)";
sheet.Range["B118"].Formula = "STDEVPA(A3:A8)";

sheet.Range["A119"].Text = "STEYX(A3:A8,A13:A18)";
sheet.Range["B119"].Formula = "STEYX(A3:A8,A13:A18)";

sheet.Range["A120"].Text = "SUBSTITUTE(B3,B4,\\"Test\\")";
sheet.Range["B120"].Formula = "SUBSTITUTE(B3,B4,\\"Test\\")";

sheet.Range["A121"].Text = "SUBTOTAL(A19,A3:A8)";
sheet.Range["B121"].Formula = "SUBTOTAL(A19,A3:A8)";

```

```

sheet.Range["A122"].Text = "SUM(A3:A8,A13:A18)";
sheet.Range["B122"].Formula = "SUM(A3:A8,A13:A18)";

sheet.Range["A123"].Text = "SUMIF(A3:A8, >300, A13:A18)";
sheet.Range["B123"].Formula = "SUMIF(A3:A8, >300, A13:A18)";

sheet.Range["A124"].Text = "SUMPRODUCT(A3:A8,A13:A18)";
sheet.Range["B124"].Formula = "SUMPRODUCT(A3:A8,A13:A18)";

sheet.Range["A125"].Text = "SUMSQ(A3:A8)";
sheet.Range["B125"].Formula = "SUMSQ(A3:A8)";

sheet.Range["A126"].Text = "SUMX2MY2(A3:A8,A13:A18)";
sheet.Range["B126"].Formula = "SUMX2MY2(A3:A8,A13:A18)";

sheet.Range["A127"].Text = "SUMX2PY2(A3:A8,A13:A18)";
sheet.Range["B127"].Formula = "SUMX2PY2(A3:A8,A13:A18)";

sheet.Range["A128"].Text = "SUMXMY2(A3:A8,A13:A18)";
sheet.Range["B128"].Formula = "SUMXMY2(A3:A8,A13:A18)";

sheet.Range["A129"].Text = "SYD(A3,A4,A5,A19)";
sheet.Range["B129"].Formula = "SYD(A3,A4,A5,A19)";

sheet.Range["A130"].Text = "TDIST(A3,1,A19)";
sheet.Range["B130"].Formula = "TDIST(A3,1,A19)";

sheet.Range["A131"].Text = "TIMEVALUE(B10)";
sheet.Range["B131"].Formula = "TIMEVALUE(B10)";

sheet.Range["A132"].Text = "TINV(A8,A4)";
sheet.Range["B132"].Formula = "TINV(A8,A4)";

sheet.Range["A133"].Text = "TRANSPOSE({150,2,3,4,5,20})";
sheet.Range["B133"].Formula = "TRANSPOSE({150,2,3,4,5,20})";

sheet.Range["A134"].Text = "TREND({150,2,3,4,5,20},{110,21,6,1,3,50})";
sheet.Range["B134"].Formula = "TREND({150,2,3,4,5,20},{110,21,6,1,3,50})";

sheet.Range["A135"].Text = "TRIMMEAN(A3:A8,A18)";
sheet.Range["B135"].Formula = "TRIMMEAN(A3:A8,A18)";

sheet.Range["A136"].Text = "TTEST(A3:A8,A13:A18,1,1)";
sheet.Range["B136"].Formula = "TTEST(A3:A8,A13:A18,1,1)";

```

```

sheet.Range["A137"].Text = "TYPE({150,2,3,4,5,20})";
sheet.Range["B137"].Formula = "TYPE({150,2,3,4,5,20})";

sheet.Range["A138"].Text = "UPPER(B7)";
sheet.Range["B138"].Formula = "UPPER(B7)";

sheet.Range["A139"].Text = "VAR(A3:A8)";
sheet.Range["B139"].Formula = "VAR(A3:A8)";

sheet.Range["A140"].Text = "VARA(A3:A8)";
sheet.Range["B140"].Formula = "VARA(A3:A8)";

sheet.Range["A141"].Text = "VARPA(A3:A8)";
sheet.Range["B141"].Formula = "VARPA(A3:A8)";

sheet.Range["A142"].Text = "VDB(A3,A4,A5,0,1)";
sheet.Range["B142"].Formula = "VDB(A3,A4,A5,0,1)";

sheet.Range["A143"].Text = "ZTEST(A3:A8,4)";
sheet.Range["B143"].Formula = "ZTEST(A3:A8,4)";

sheet.Range["A144"].Text = "ZTEST({150,100,200,300,500,0.3},4)";
sheet.Range["B144"].Formula = "ZTEST({150,100,200,300,500,0.3},4)";

```

C1	A	B	C
1	110	200	310
$=A1+B1$			

Figure 118: Formula Settings

Note that formula separators vary for each culture/regional settings, and there will be exceptions in such cases. This can be overcome by setting the separators by using the **SetSeparators** method of IWorkbook. Following code example illustrates how to change the formula separators through XlsIO.

```

[C#]

workbook.SetSeparators(";", ", ", ",");

```

- Sheet1 and Sheet2 are the default names of the worksheets.

```

[VB .NET]

```

```
workbook.SetSeparators(";", " ", ")")
```

- In addition to being able to access values in the same worksheet, you can also access values across worksheets. Assume that the "B" is present on the second worksheet, then use the following code for calculation.

[C#]

```
sheet.Range["C2:C4"].Formula = "=SUM(Sheet2!B2:B4,Sheet1!A2:A4)";
```

- Sheet1 and Sheet2 are the default names of the worksheets.

[VB .NET]

```
sheet.Range("C2:C4").Formula = "=SUM(sheet2!B2:B4,sheet1!A2:A4)"
```

You can also read the Formula Text and the Computed Value of the formula in the cell.

Following code example illustrates how to read the formulas and computed values.

[C#]

```
// Read computed Formula Value.  
this.txtFormulaNumber.Text = sheet.Range["C1"].FormulaNumberValue.ToString();  
  
// Read Formula.  
this.txtFormula.Text = sheet.Range["C1"].Formula;
```

[VB .NET]

```
' Read computed Formula Value.  
Me.txtFormulaNumber.Text = sheet.Range("C1").FormulaNumberValue.ToString()  
  
' Read Formula.  
Me.txtFormula.Text = sheet.Range("C1").Formula
```

You can also get the Formula values as bool, date, and number type. Note that XlsIO can only read already computed formulas, and cannot compute. Please refer to the [Calculation Engine](#) for more information on dynamic formula computation.

Following properties of the **IRange** interface are used to fetch formulas, computed values, and to check if there exists a formula in the cell.

Property	Description
Formula	Returns or sets the object's formula in A1-style notation and in the language of the macro. Read/write Variant.
FormulaArray	Represents array-entered formula. Visit http://www.cpearson.com/excel/array.htm for more information.
FormulaArrayR1C1	Returns or sets the formula array for the range by using R1C1-style notation.
FormulaBoolValue	Returns the calculated value of the formula as a boolean.
FormulaDateTime	Gets/sets formula DateTime value contained by this cell. DateTime.MinValue if not all cells of the range have same DateTime value.
FormulaErrorValue	Returns the calculated value of the formula as a string.
FormulaHidden	True if the formula will be hidden when the worksheet is protected; False if at least part of formula in the range is not hidden.
FormulaNumberValue	Gets/sets number value evaluated by formula.
FormulaR1C1	Returns or sets the formula for the range by using R1C1-style notation.
FormulaStringValue	Gets/sets string value evaluated by formula.
HasDataValidation	Indicates whether specified range object has data validation. If Range is not single cell, then returns True only if all cells have data validation. This is a Read-Only property.
HasDateTime	Indicates whether the range contains DateTime value. This is a Read-Only property.
HasExternalFormula	Indicates if current range has external formula. This is a Read-Only property.

HasFormula	True if all cells in the range contain formulas; False if at least one of the cells in the range doesn't contain a formula. This is a Read-Only property.
HasFormulaArray	Indicates whether range contains array-entered formula. This is a Read-Only property.
HasFormulaBoolValue	Indicates if current range has formula bool value. This is a Read-Only property.
HasFormulaDateTime	Indicates if current range has formula value formatted as DateTime. This is a Read-Only property.
HasFormulaErrorValue	Indicates if current range has formula error value. This is a Read-Only property.
HasNumber	Indicates whether the range contains number. This is a Read-Only property.
HasRichText	Indicates whether cell contains formatted rich text string.
HasString	Indicates whether the range contains String. This is a Read-Only property.
HasStyle	Indicates whether range has default style. False means default style. This is a Read-Only property.
IgnoreErrorOptions	Represents various ignore error options in Excel.

4.4.1.1 Array Formula

Array Formula is a special type of formula in Excel. It works with an array or series of data values, rather than a single data value. XlsIO supports the usage of Array formula through the `FormulaArray` property.

Following code example explains how an array of values, from Named Range, is used for computation. For more details on Named Ranges, please refer [Defined Names](#).

[C#]

```
// Insert Array Formula.
sheet.Range["A1:D1"].FormulaArray = "{1,2,3,4}";
sheet.Names.Add("ArrayRange", sheet.Range["A1:D1"]);
sheet.Range["A2:D2"].FormulaArray = "ArrayRange+100";
```

[VB.NET]

```
' Insert Array Formula.
sheet.Range("A1:D1").FormulaArray = "{1,2,3,4}"
sheet.Names.Add("ArrayRange", sheet.Range("A1:D1"))
sheet.Range("A2:D2").FormulaArray = "ArrayRange+100"
```

	C	D	E
B	10	20	30
	110	120	130
fx (=ArrayRange+100)			140

Figure 119: XlsIO with Array Formula

See Also[External Formula](#)

4.4.1.2 External Formula

Essential XlsIO allows you to insert/preserve formulas that refer values in other worksheets/workbooks. Note that XlsIO can only write/preserve formulas. You cannot update/refresh the calculated values in Excel, which should be refreshed by MS Excel.

Following code illustrates the insertion of a formula that refers to a value in another workbook.

[C#]

```
// Write external Formula Value.
sheet.Range["C1"].Formula = "[One.xls]Sheet1!$A$1*5";
```

[VB.NET]

```
' Write external Formula Value.
sheet.Range["C1"].Formula = "[One.xls]Sheet1!$A$1*5";
```



Note: Enable automatic updation of links in Excel, to view the result for the preceding code.

See Also

[Array Formula](#)

4.4.2 Calculation

This section explains various options that enable users to customize the formula calculations according to their needs. It also demonstrates how XlsIO can be linked with Essential Calculate that acts as an Excel engine, to calculate Formula values. Following topics elaborate on these options of Excel.

4.4.2.1 Calculation Options

Excel has a range of options allowing you to control the way it calculates. To do this, open **Tools** menu, select **Options**, and click **Calculation** tab in the Options dialog box. By default, when one value is changed, the other cell that depends on this, immediately gets changed. This behavior can be changed by using the following Modes of Calculations.

- Automatic
- Automatic except for Tables
- Manual

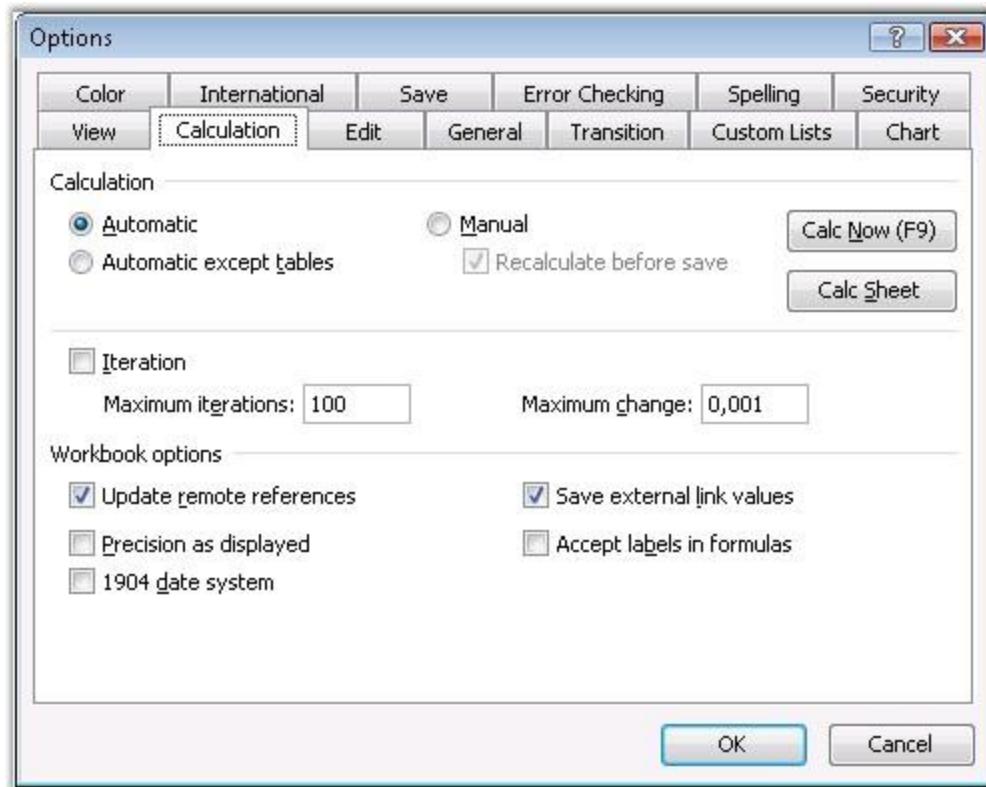


Figure 120: Options Dialog Box- Calculation

Automatic Calculation

In the Automatic Calculation mode, Excel automatically recalculates all open workbooks at each and every change and whenever you open a workbook.

Usually when you open a workbook in Automatic mode, and recalculation is done, you will not be able to see the recalculation, because the changes will not be reflected until the workbook is saved. An exception is when you open a workbook in Excel 2000 that was saved by using Excel 97, or open a workbook by using Excel 2002/2003 saved by using Excel 2000. This is because Excel's calculation engines are different, and also because a Full calculation is done.

Manual Calculation

In the Manual Calculation mode, Excel will only recalculate all open workbooks, when you request it by pressing F9 or CTRL-ALT-F9, or when you Save a workbook. For workbooks taking more than a fraction of a second to recalculate, it is usually better to set the Calculation to "Manual".

Excel tells you when the workbook needs recalculation, by showing Calculate in the status bar.

Automatic Except Tables

Excel's Data Tables feature is designed to perform multiple calculations of the workbook, each driven by different values in the table. So using "Automatic except Tables" will stop Excel from automatically triggering multiple calculations at each calculation, but will still calculate all dependent formulae, except tables.

XlsIO provides support for all the above modes of calculation. Following code example illustrates how to set the calculation mode.

[C#]

```
IWorkbook workbook = application.Workbooks.Create();
workbook.CalculationOptions.CalculationMode = ExcelCalculationMode.Manual;
```

[VB .NET]

```
Dim workbook as IWorkbook
workbook = application.Workbooks.Create()
workbook.CalculationOptions.CalculationMode = ExcelCalculationMode.Manual
```

There are other options that Excel provides to customize the calculation further.

Recalculate Before Save

In Manual mode, this option controls whether Excel will recalculate the workbook as part of the Save process. The default value is set to **True**. You can control this through XlsIO by using the **RecalcOnSave** property of **ICalculationOptions** interface.

Iteration

If you have intentional circular references in your workbook, these settings allow you to control the maximum number of times the workbook will be recalculated (iterations), and the convergence criteria (maximum change: when to stop). The default value should be set to False, so that Excel does not try to solve accidental circular references. XlsIO allows to control these iterations as follows.

[C#]

```
IWorkbook workbook = application.Workbooks.Create();
```

```
workbook.CalculationOptions.IterationEnabled = true;
workbook.CalculationOptions.MaximumIteration = 99;
workbook.CalculationOptions.MaximumChange = 40;
```

[VB.NET]

```
Dim workbook as IWorkbook
workbook = application.Workbooks.Create()
workbook.CalculationOptions.IterationEnabled = true;
workbook.CalculationOptions.MaximumIteration = 99;
workbook.CalculationOptions.MaximumChange = 40;
```

4.4.2.2 Calculation Engine

Essential XlsIO has a calculation engine of its own and can compute the values of the formula entered during runtime. Essential Calculate is now [from v9.1.x.x] integrated with Essential XlsIO, and thus makes it possible to get the values of the formula entered by using XlsIO during runtime, without any additional references or packages. Currently there are over 180+ functions that are supported by this Calculate engine, which cover all common usage scenarios.

Sample Link

To understand this process, consider the sample project:

{Drive:}\Program Files\EssentialStudio*.**\Windows\XlsIO.Windows\Samples\2.0\Data Management\Compute All Formulas.

4.4.2.2.1 Adding Calculation Engine to an Application

Enable Formula Calculations:

Essential XlsIO includes support for enabling the calculations of Essential Calculate supported formulas that are added at runtime to the worksheet and the computed value will be set to the "CalculatedValue" property associated to the "IRange" object. The following code sample illustrates how to enable the sheet formula calculations.

[C#]

```
IWorksheet sheet = workbook.Worksheets[0];
//Formula calculation is enabled for the sheet.
sheet.EnableSheetCalculations();
string computedValue = sheet.Range["C1"].CalculatedValue;
```

Disable Formula Calculations:

Essential XlsIO will be able to disable the calculations of Essential Calculate supported formulas that are added at runtime to the worksheet. The following code sample illustrates how to disable the sheet formula calculations.

[C#]

```
IWorksheet sheet = workbook.Worksheets[0];
//Formula calculation is enabled for the sheet.
sheet.DisableSheetCalculations();
```

Here are some code samples, to evaluate some formulas entered by using Essential XlsIO during runtime. The XlsIO computed value is identical to the values computed by using MS Excel.

[C#]

```
//Inserting sample text into the first cell of the first worksheet.
sheet.Range["A1"].Number = 10.99;
sheet.Range["B1"].Number = 10;
sheet.Range["C1"].Formula = "A1+B1";
sheet.Range["D1"].Formula = "AVERAGE(A1:B1)";

//Formula calculation is enabled for the sheet.
sheet.EnableSheetCalculations();

Console.WriteLine(sheet.Range["C1"].CalculatedValue.ToString(),
sheet.Range["C1"].Formula);
Console.WriteLine(sheet.Range["D1"].CalculatedValue.ToString(), sheet.Range["D1"].Formula);

//Add more data
sheet.Range["A2"].Number = 11.99;
sheet.Range["B2"].Number = 11;
sheet.Range["C2"].Formula = "A2+B2";
sheet.Range["D2"].Formula = "AVERAGE(A2:B2)";

Console.WriteLine(sheet.Range["C2"].CalculatedValue.ToString(), sheet.Range["C2"].Formula);
```

```
Console.WriteLine(sheet.Range["D2"].CalculatedValue.ToString(), sheet.Range["D2"].Formula);
```

[VB.NET]

```
' Inserting sample text into the first cell of the first worksheet.  
sheet.Range("A1").Number = 10.99  
sheet.Range("B1").Number = 10  
sheet.Range("C1").Formula = "A1+B1"  
sheet.Range("D1").Formula = "AVERAGE(A1:B1)"  
  
' Formula calculation is enabled for the sheet.  
sheet.EnableSheetCalculations()  
Console.WriteLine(sheet.Range("C1").FormulaNumberValue.ToString(), sheet.Range("C1").Formula)  
Console.WriteLine(sheet.Range("D1").FormulaNumberValue.ToString(), sheet.Range("D1").Formula)  
  
' Add more data.  
sheet.Range("A2").Number = 11.99  
sheet.Range("B2").Number = 11  
sheet.Range("C2").Formula = "A2+B2"  
sheet.Range("D2").Formula = "AVERAGE(A2:B2)"  
  
Console.WriteLine(sheet.Range("C2").FormulaNumberValue.ToString(), sheet.Range("C2").Formula)  
Console.WriteLine(sheet.Range("D2").FormulaNumberValue.ToString(), sheet.Range("D2").Formula)
```

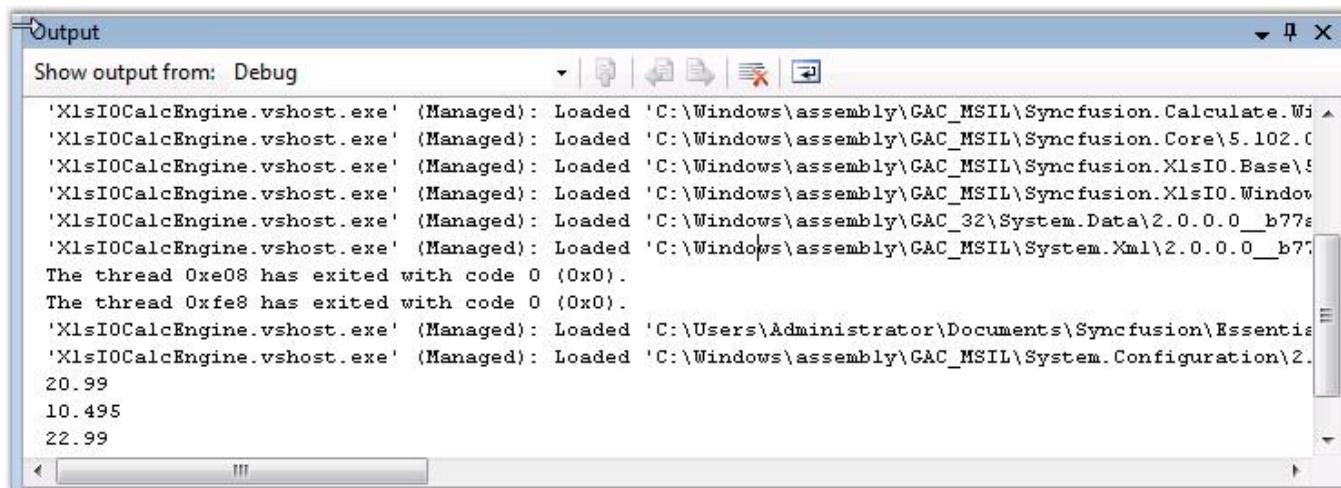


Figure 121: Output Box in Visual Studio

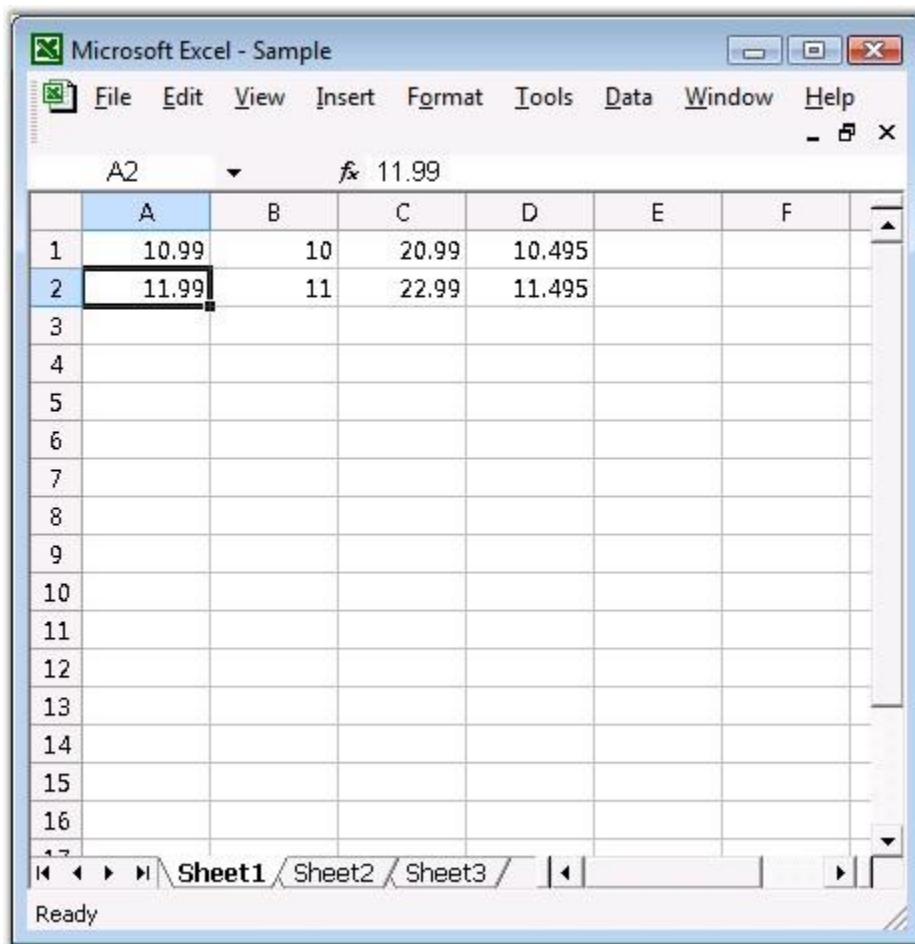


Figure 122: XlsIO with XlsIO CalcEngine

**Note:**

- 1 *In order to use the Essential XlsIO's Calculate engine, you have to add the following namespace:*
 - *using Syncfusion.Calculate*
- 2 *Do not add reference to Syncfusion.Calculate.Base. It will throw conflict errors as these are already integrated with XlsIO from Version 7.2.X.X.*
- 3 *Only the formulas that are supported by Calculate engine can be calculated at runtime using Essential XlsIO.*

4.4.3 Defined Names

Named Ranges is a powerful feature in Excel, which makes it possible to assign a name to a group of cells. XlsIO has APIs for inserting new named ranges into workbooks, and also to read existing named ranges. Named Ranges are mainly used in formulae.

It enables users with better readability, and at a glance we can predict what it is, and what we're adding up with the meaningful range names.

To create named ranges, open **Insert menu**, point to **Name** and select **Define**. Names are created at workbook-level, by default, but you can create a sheet-level name, by entering the sheet name followed by an exclamation mark (!), followed by the name of the range. For instance, to create a named range "x" on Sheet1, open **Insert menu**, point to **Name**, select **Define**, and enter the name as 'Sheet1!x'.

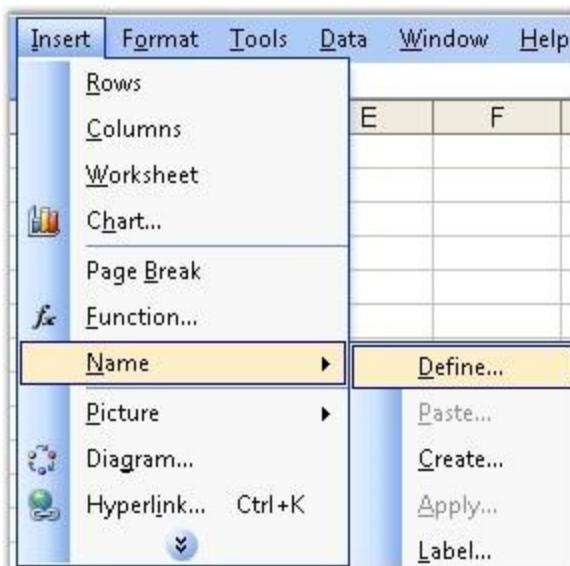


Figure 123: Inserting Defined Names using Insert Menu

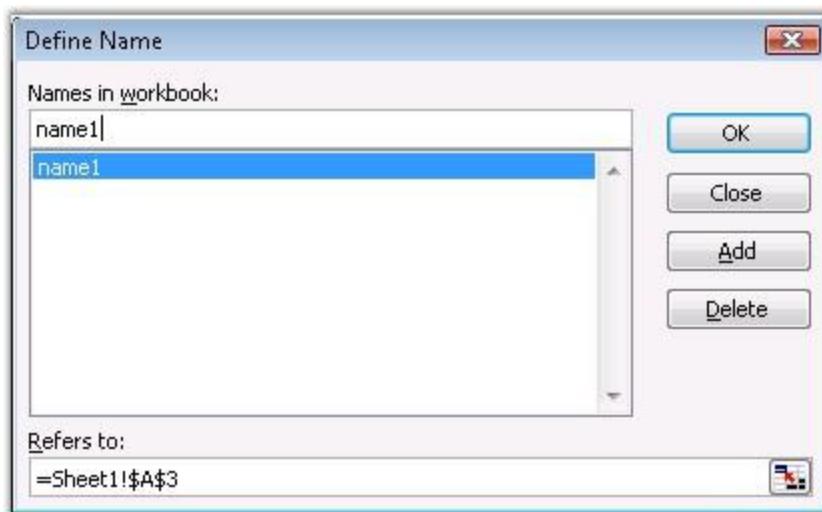


Figure 124: Define Name Dialog Box

You can create named ranges in spreadsheets through XlsIO with the **IName** interface. Range for the name is specified through **RefersToRange** property, from which you can access the Range text and other information. This can be specified for workbooks and worksheets.

Following code example illustrates how to create named ranges and use it in formulas.

[C#]

```
// The first worksheet object in the worksheets collection is accessed.  
IWorksheet sheet = workbook.Worksheets[0];  
  
// Defining the Range and using it in the Formula.  
IName lname1 = workbook.Names.Add("One");  
lname1.RefersToRange = sheet.Range["C3"];  
  
IName lname2 = workbook.Names.Add("Two");  
lname2.RefersToRange = sheet.Range["D3"];  
  
// Formula using Defined Names.  
sheet.Range["E3"].Formula = "=SUM(One, Two)";
```

[VB.NET]

```
' The first worksheet object in the worksheets collection is accessed.  
Dim sheet As IWorksheet = workbook.Worksheets(0)  
  
' Defining the Range and using it in the Formula.  
Dim lname1 As IName = workbook.Names.Add("One")  
lname1.RefersToRange = sheet.Range("C3")  
  
Dim lname2 As IName = workbook.Names.Add("Two")  
lname2.RefersToRange = sheet.Range("D3")  
  
' Formula using Defined Names.  
sheet.Range("E3").Formula = "=SUM(One, Two)"
```

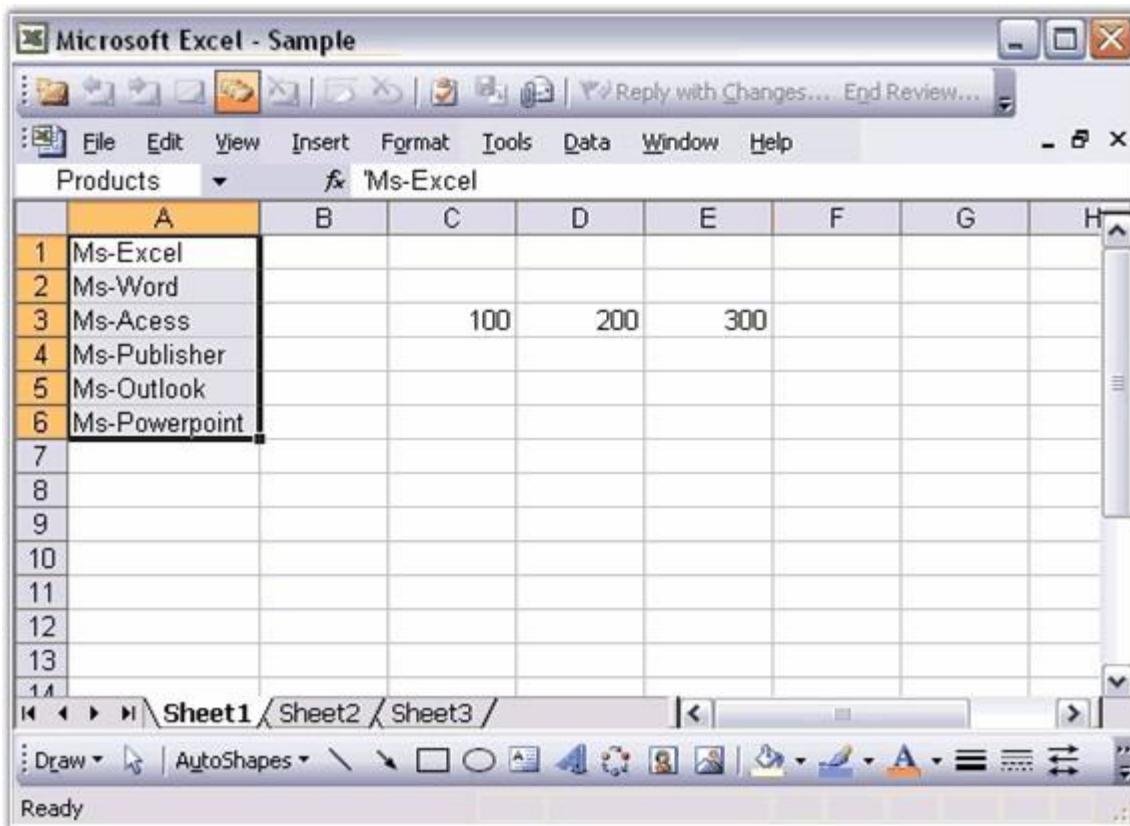


Figure 125: XlsIO with Named Ranges

Following code example illustrates how to get/set sheet-level named ranges.

[C#]

```
IWorksheet sheet = workbook.Worksheets[0];

// Add a name.
IName lname1 = sheet.Names.Add("CellName");
lname1.RefersToRange = sheet.Range("C3");

// Get name.
Console.WriteLine(sheet.Names["CellName"].Value);
```

[VB.NET]

```
Dim sheet As IWorksheet = workbook.Worksheets(0)

' Add a name.
```

```

Dim lname1 As IName = sheet.Names.Add("CellName")
lname1.RefersToRange = sheet.Range("C3")

' Get name.
Console.WriteLine(sheet.Names("CellName").Value)

```

You can get/read all the names from a worksheet or workbook just by enumerating the INames collection as follows.

[C#]

```

for (int i = 0; i <= workbook.Names.Count; i++)
{
    Console.WriteLine(workbook.Names[i].Name.ToString());
    Console.WriteLine(workbook.Names[i].RefersToRange.Address.ToString());
    Console.WriteLine(workbook.Names[i].Name.Value.ToString());
}

```

[VB.NET]

```

Dim i As Integer
For i = 0 To workbook.Names.Count Step i + 1
    Console.WriteLine(workbook.Names(i).Name.ToString())
    Console.WriteLine(workbook.Names(i).RefersToRange.Address.ToString())
    Console.WriteLine(workbook.Names(i).Name.Value.ToString())
Next

```

You can also delete a name in the workbook/worksheet by using the **Delete** method of IName. Note that deleting the cell, does not delete the name from the **Name** collection.

Following table lists the properties of **IName**.

Property	Description
Index	Returns the index number of the Name object within the collection. This is a Read-Only property.
IsLocal	Indicates whether name is local.
Name	Returns or sets the name of the object. Read/Write String.
NameLocal	Returns or sets the name of the object, in the language of the user. Read/Write

Property	Description
	String for Name.
RefersToRange	Gets/sets the Range associated with the Name object.
Value	For the Name object, a string containing the formula that the name is defined to is referred. The string is in A1-style notation in the language of the macro, without an equal sign.
ValueR1C1	Gets named range Value in R1C1 style. This is a Read-Only property.
Visible	Determines whether the object is visible. Read/Write Boolean.
Scope	Returns the scope of the name range.

Scope of Named Range

The scope of the named range can be accessed as follows.

[C#]

```
IName name = workbook.Names.Add("Name1");
name.RefersToRange = sheet.Range["A1"];
Console.WriteLine(name.Scope);
```

[VB .NET]

```
Dim name As IName = workbook.Names.Add("Name1")
name.RefersToRange = sheet.Range("A1")
Console.WriteLine(name.Scope)
```

4.4.4 Formula Auditing

Excel has an option to find the quickest way to identify any cell that contains an error on the active worksheet, and ignore the error that is showed with green indicator, through the Error Checking dialog box. This dialog box provides various options to get information on the error, how a formula is evaluated, its trace, and an option to ignore the error by changing its data type.

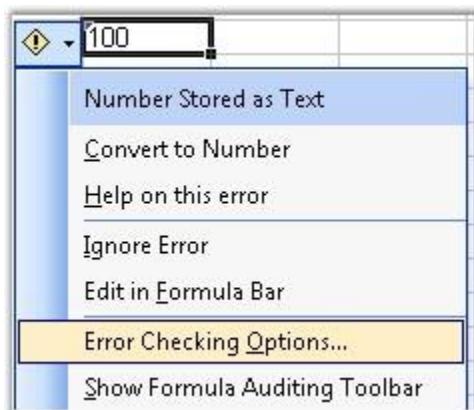


Figure 126: : Error Checking

Excel has the following set of rules that can be enabled or disabled, to show/hide warnings with green indicators.

- **Evaluates to Error Value**-This rule treats cells containing formulas that result in an error, and displays a warning.
- **Text Date**-This rule treats formulas that contain text formatted cells with years represented as 2-digits, as an error, and displays a warning while checking for errors.
- **Number stored as Text**-This rule treats numbers formatted as text or preceded by an apostrophe, as an error, and displays a warning.
- **Inconsistent Formula in Region**-This rule treats a formula in a region of your worksheet that differs from the other formulas in the same region, as an error, and displays a warning.
- **Formula omits Cells in Region**-This rule treats formulas that omit certain cells in a region, as an error, and displays a warning.
- **Unlocked Cells containing Formulas**-This rule treats an unlocked cell containing a formula, as an error, and displays a warning when checking for errors.
- **Formulas referring to Empty Cells**-This rule treats formulas that refer to empty cells, as an error, and displays a warning.

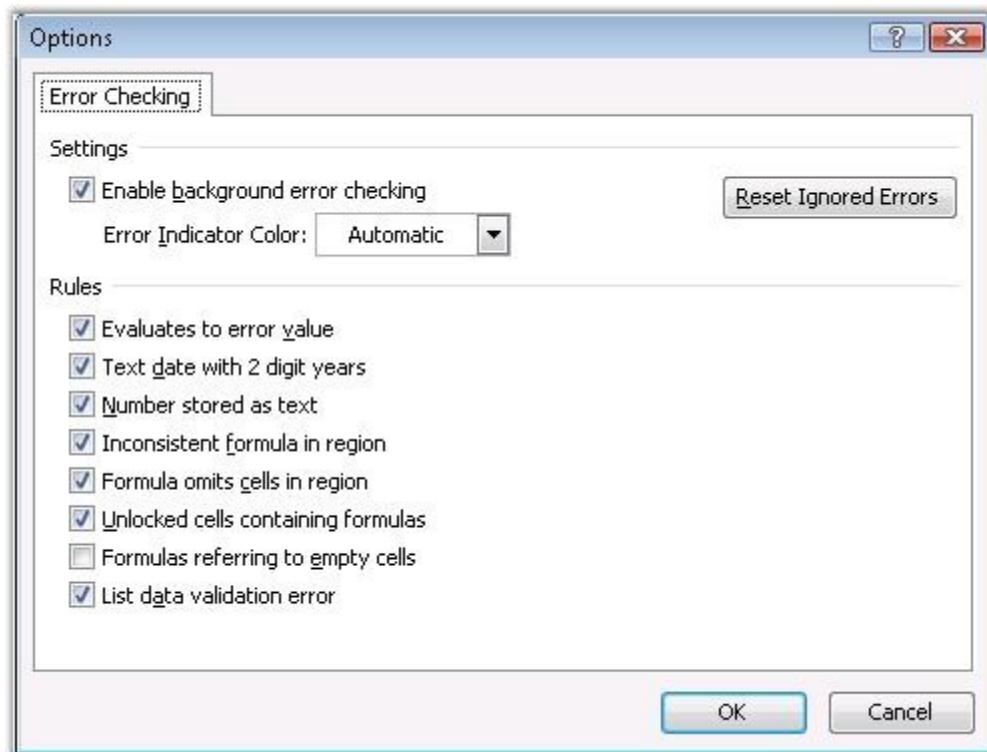


Figure 127: Options Dialog Box - Error Checking

XlsIO provides all the above options to ignore errors, and remove the green indicators. This can be done through the **IgnoreErrorOptions** property of the IRange interface.

Following are the values that can be set for the IgnoreError option, through the **ExcelIgnoreError** enumerator.

Member name	Description
None	Represents None flag of excel ignore error indicator.
EvaluateToError	Represents EvaluateToError flag of excel ignore error indicator.
EmptyCellReferences	Represents EmptyCellReferences flag of excel ignore error indicator.
NumberAsText	Represents NumberAsText flag of excel ignore error indicator.
OmittedCells	Represents OmittedCells flag of excel ignore error indicator.
InconsistentFormula	Represents InconsistentFormula flag of excel ignore error indicator.
TextDate	Represents TextDate flag of excel ignore error indicator.

UnlockedFormulaCells	Represents UnlockedFormulaCells flag of excel ignore error indicator.
All	Represents All flag of excel ignore error indicator.

Following code example illustrates how to ignore or set an error indicator.

[C#]

```
// Sets warning if number is entered as text.
sheet.Range["A2:D2"].IgnoreErrorOptions = ExcelIgnoreError.NumberAsText;

// Ignores all the error warnings.
sheet.Range["A3"].IgnoreErrorOptions = ExcelIgnoreError.None;
```

[VB.NET]

```
' Sets warning if number is entered as text.
sheet.Range["A2:D2"].IgnoreErrorOptions = ExcelIgnoreError.NumberAsText

' Ignores all the error warnings.
sheet.Range["A3"].IgnoreErrorOptions = ExcelIgnoreError.None
```

4.5 Data

XlsIO has advanced support to work with data in a worksheet and here are some key functionalities that makes it easier.

4.5.1 Filter

MS Excel **AutoFilter** feature literally makes filtering out unwanted data in a data list, as easy as clicking a button. When the cell pointer is located within any cell in your data list, open the **Data** menu, point to **Filter**, and select **AutoFilter**. Once this is done, the program adds drop-down buttons to each of the field names in the top row of the list. This feature is specifically used in large spreadsheets, when the user wants to look for particular data, based on some criteria.

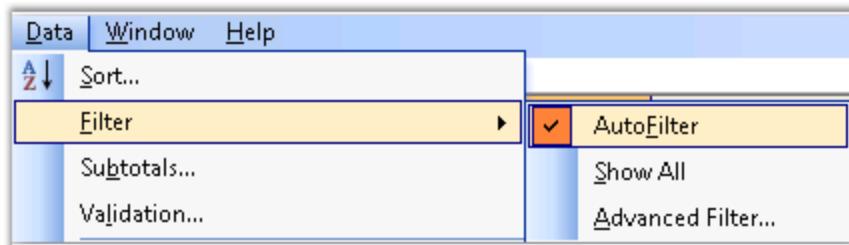


Figure 128: AutoFilter from Data Menu

AutoFilters in Essential XlsIO

Essential XlsIO also comes with APIs for reading and writing AutoFilters in a worksheet. You can specify the range of data that needs to be viewed through the **FilterRange** property. Following code example illustrates writing AutoFilters.

[C#]

```
// Creating an AutoFilter in the first worksheet. Specifying the AutoFilter
// range.
sheet.AutoFilters.FilterRange = sheet.Range["A1:B7"];
```

[VB.NET]

```
' Creating an AutoFilter in the first worksheet. Specifying the AutoFilter
// range.
sheet.AutoFilters.FilterRange = sheet.Range("A1:B7")
```

XlsIO also provides options to set the built-in conditions for filters by using various properties of **IAutoFilter**. Following code example illustrates various conditions based on which data can be filtered.

[C#]

```
IAutoFilter filter = sheet1.AutoFilters[0];
filter.IsTop = true;
filter.IsTop10 = true;
filter.Top10Number = 5;
```

[VB.NET]

```
Dim filter As IAutoFilter = sheet1.AutoFilters(0)
```

```
filter.IsTop = True
filter.IsTop10 = True
filter.Top10Number = 5
```

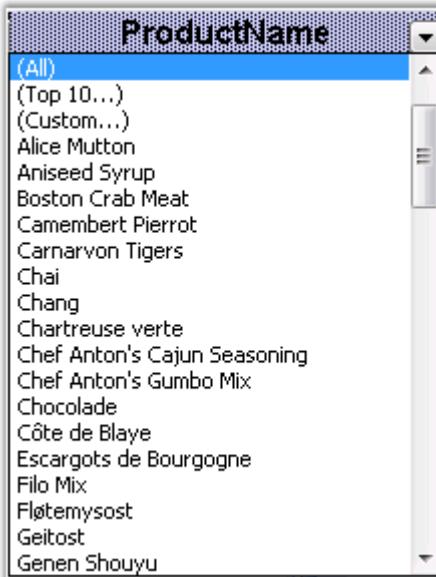


Figure 129: Writing Autofilters with XlsIO

See Also

[Data Validation](#), [Importing and Exporting](#), [Template Markers](#), [Grouping and Ungrouping](#)

4.5.2 Data Validation

The Data Validation feature available in MS Excel dynamically validates the data that is entered into a cell. The validation rules are specified in the Data Validation **Settings** tab. Data Validation in MS Excel is achieved by selecting the "Validation" item in the **Data** menu. Excel has various validation types for each **data type**, and options to show the error box for invalid data through the **Data Validation** dialog box, which is shown in the following screen shot.

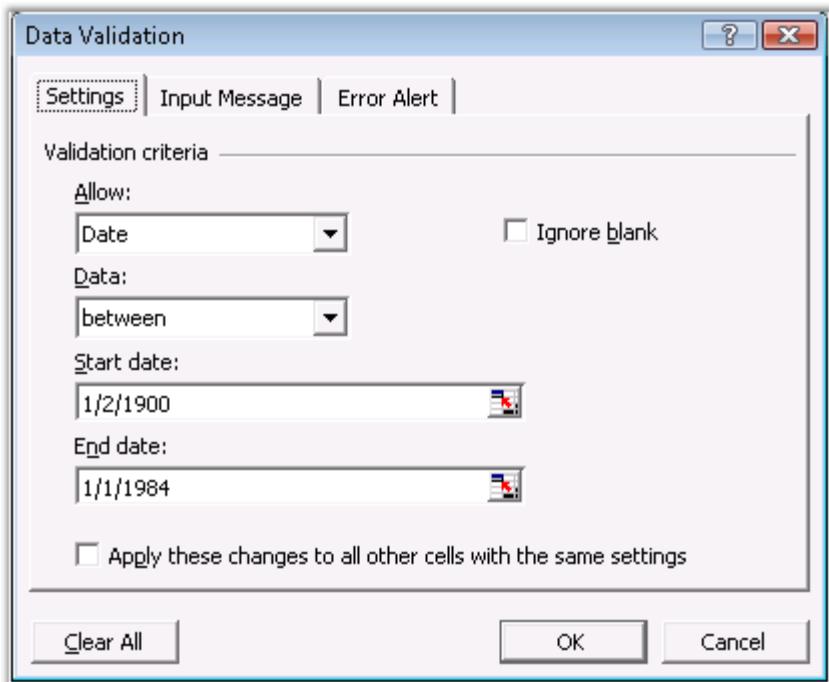


Figure 130: Data Validation Settings

Data Validation in Essential XlsIO

Essential XlsIO, equivalent to the MS Excel, is built with APIs to read and write data validation in a worksheet by using the **IDataValidation** class. Following are some validation types that XlsIO supports.

- Text Length Validation
- Time Validation
- List Validation
- Number Validation
- Date Validation
- Custom Validation

An article which describes data validation is available in the following path:
http://www.syncfusion.com:91/products/xlsio/backoffice/Articles/data_validation.aspx.

[C#]

```
// Data validation to list the values in the first cell.
IDataValidation validation = sheet.Range["A1"].DataValidation;
sheet.Range["A1"].Text = "Data validation list";
validation.ListOfValues = new string[] { "ListItem1", "ListItem2",
"ListItem3" };
```

```
validation.PromptBoxText = "Data Validation list";
validation.IsPromptBoxVisible = true;
validation.ShowPromptBox = true;
```

[VB.NET]

```
' Data validation to list the values in the first cell.
Dim validation As IDataValidation = sheet.Range("A1").DataValidation
sheet.Range("A1").Text = "Data validation list"
validation.ListOfValues = New String() {"ListItem1", "ListItem2", "ListItem3"}
validation.PromptBoxText = "Data Validation list"
validation.IsPromptBoxVisible = True
validation.ShowPromptBox = True
```

The following screen shots illustrates the error alert settings through the **Data Validation** dialog box in MS Excel.

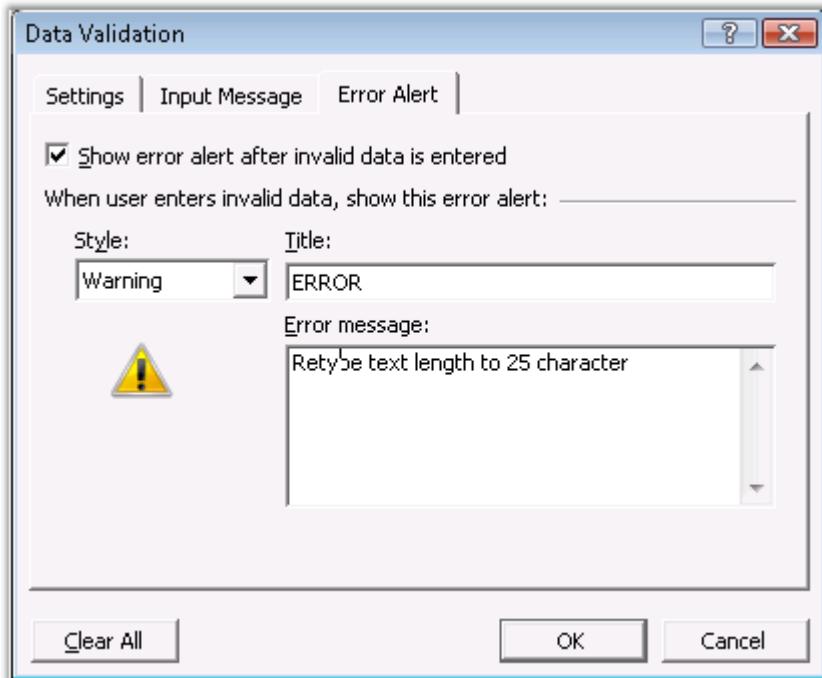


Figure 131: Error Alert Options in MS Excel

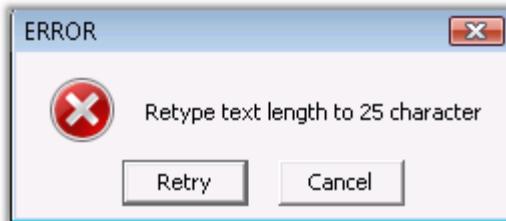


Figure 132: Error box

XlsIO has numerous validation rules and features which are demonstrated in the following code example. **AllowType** property sets the type of validation, **CompareOperator** sets the validation criteria, and **ShowErrorBox** shows the error box with an error message.

[C#]

```
// Data Validation for Numbers.
IDataValidation validation1 = sheet.Range["A3"].DataValidation;
sheet.Range["A3"].Text = "Enter a Number";
validation1.AllowType = ExcelDataType.Integer;
validation1.CompareOperator = ExcelDataValidationComparisonOperator.Between;
validation1.FirstFormula = "0";
validation1.SecondFormula = "10";
validation1.ShowErrorBox = true;
validation1.ErrorBoxText = "Enter Value between 0 to 10";
validation1.ErrorBoxTitle = "ERROR";
validation1.PromptBoxText = "Data Validation using Condition for Numbers";
validation1.ShowPromptBox = true;

// Data Validation for Date.
IDataValidation validation2 = sheet.Range["A5"].DataValidation;
sheet.Range["A5"].Text = "Enter the Date";
validation2.AllowType = ExcelDataType.Date;
validation2.CompareOperator = ExcelDataValidationComparisonOperator.Between;
validation2.FirstDateTime = new DateTime(2003, 5, 10);
validation2.SecondDateTime = new DateTime(2004, 5, 10);
validation2.ShowErrorBox = true;
validation2.ErrorBoxText = "Enter Value between 10/5/2003 to 10/5/2004";
validation2.ErrorBoxTitle = "ERROR";
validation2.PromptBoxText = "Data Validation using Condition for Date";
validation2.ShowPromptBox = true;
```

[VB.NET]

```
' Data Validation for Numbers.
Dim validation1 As IDataValidation = sheet.Range("A3").DataValidation
sheet.Range("A3").Text = "Enter a Number"
```

```

validation1.AllowType = ExcelDataType.Integer
validation1.CompareOperator = ExcelDataValidationComparisonOperator.Between
validation1.FirstFormula = "0"
validation1.SecondFormula = "10"
validation1.ShowErrorBox = True
validation1.ErrorBoxText = "Enter Value between 0 to 10"
validation1.ErrorBoxTitle = "ERROR"
validation1.PromptBoxText = "Data Validation using Condition for Numbers"
validation1.ShowPromptBox = True

' Data Validation for Date.
Dim validation2 As IDataValidation = sheet.Range("A5").DataValidation
sheet.Range("A5").Text = "Enter the Date"
validation2.AllowType = ExcelDataType.Date
validation2.CompareOperator = ExcelDataValidationComparisonOperator.Between
validation2.FirstDateTime = New DateTime(2003,5,10)
validation2.SecondDateTime = New DateTime(2004,5,10)
validation2.ShowErrorBox = True
validation2.ErrorBoxText = "Enter Value between 10/5/2003 to 10/5/2004"
validation2.ErrorBoxTitle = "ERROR"
validation2.PromptBoxText = "Data Validation using Condition for Date"
validation2.ShowPromptBox = True

```

Reading the Existing Data Validation Settings

You can also read the Data Validation settings in an existing workbook. The following code example illustrates this.

[C#]

```

// Reading the Data Validation list.
this.comboBox1.Items.AddRange(sheet.Range[ "A1"
].DataValidation.ListOfValues);

```

[VB .NET]

```

' Reading the Data Validation list.
Me.comboBox1.Items.AddRange(sheet.Range("A1").DataValidation.ListOfValues)

```

See Also

[AutoFilters](#), [Importing and Exporting](#), [Template Markers](#), [Grouping and Ungrouping](#)

4.5.3 Data Sorting for a Given Range of Cells in the Worksheet

This feature allows sorting any range of cells dynamically at runtime, without any dependency on MS Excel. You can sort a given range of cells in a worksheet by applying the sort feature to a range, which changes the related data within the range. The user can sort the range in the following ways:

- Sorting by Cell Values
- Sorting by Font Color
- Sorting by Cell Color



Note: Presently we don't support sorting based on cell icon, parsing and serialization of the sorting details.

4.5.3.1 Properties, Methods and Events tables

IDataSort Members:

IDataSort members represent the sort of the range.

Properties

Name	Description
IsCaseSensitive	Indicates whether or not to perform case sensitive sort
HasHeader	Indicates whether the range has header
Orientation	Represents the sort orientation
SortFields	Represents the SortFields Collection
SortRange	Represents the sort range
Algorithm	Represents the algorithm to sort

Methods

Name	Description
Sort()	Sorts the range, based on the sort fields.

ISortField Members:

ISortField members contain all the sort information.

Properties

Name	Description
Key	Represents the column to be sorted
SortOn	Represents on which the range should be sorted.
Order	Represents the sort order
Color	Represents the color to sort. Throws exception when SortOn type is "Values".

ISortFields Members:

ISortFields members represent the collection Sort Field.

Properties

Name	Description
Count	Represents the field count

Methods

Name	Description
Add	Adds the SortField in the collection
Remove	Removes the SortField in the collection

4.5.3.2 Sorting Data by Cell Values

This is used to sort a range of cells dynamically, at runtime. This is explained in the following code samples:

Sorted the column 'ID' in Descending order

	A	B	C	D	E	F	G	H
1								
2								
3								
4	ID	Name	DOJ	Salary				
5	59653	Francisco Chang	1/14/2001	\$29,221.00				
6	58893	Yoshi Tannamuri	6/3/2006	\$50,066.00				
7	57376	José Pedro Freyre	5/21/2010	\$1,178.00				
8	51815	Philip Cramer	5/28/2004	\$39,236.00				
9	51479	Manuel Pereira	2/17/2008	\$58,125.00				
10	50704	Christina Berglund	3/10/2011	\$26,379.00				
11	50585	Felipe Izquierdo	5/20/2009	\$52,881.00				
12	50585	Giovanni Rovelli	5/25/2006	\$19,425.00				
13	49634	Diego Roel	6/19/2010	\$9,628.00				
14	48421	Paolo Accorti	10/16/2006	\$55,078.00				
15	48414	Antonio Moreno	1/8/2000	\$41,592.00				
16	48264	Patricio Simpson	3/24/2000	\$29,579.00				
17	47610	Yoshi Latimer	6/10/2003	\$35,999.00				
18	46666	Patricia McKenna	11/15/2003	\$42,272.00				
19	44257	André Fonseca	3/9/2005	\$15,269.00				
20	44032	Thomas Hardy	2/4/2011	\$41,399.00				
21	43366	Eduardo Saavedra	9/16/2007	\$12,099.00				
22	42255	Renate Messner	10/30/2004	\$24,962.00				
23	39352	Ana Trujillo	2/6/2012	\$30,834.00				
24	34574	Carine Schmitt	2/3/2002	\$12,047.00				
25	29601	Carlos González	5/24/2012	\$12,160.00				
26	29454	Carlos Hernández	4/15/2001	\$33,181.00				
27	28785	Lino Rodríguez	10/3/2002	\$7,677.00				

Column "ID" is Sorted in Descending order.

C#

```
//Creates the data sorter
IDataSort sorter = book.CreateDataSorter();

//Range to sort
sorter.SortRange = range;

//Adds the sort field with the column index, sort based on and order by attribute
ISortField sortField = sorter.SortFields.Add(0, SortOn.Values, OrderBy.Ascending);

//Adds another sort field
ISortField sortField2 = sorter.SortFields.Add(1, SortOn.Values,
OrderBy.Ascending);

//Sort based on the sort Field attribute
sorter.Sort();
```

VB.Net

```
''Creates the Data sorter
Dim sorter As IDataSort = book.CreateDataSorter()

''Specifies the sort range
sorter.SortRange = range

Dim field As ISortField

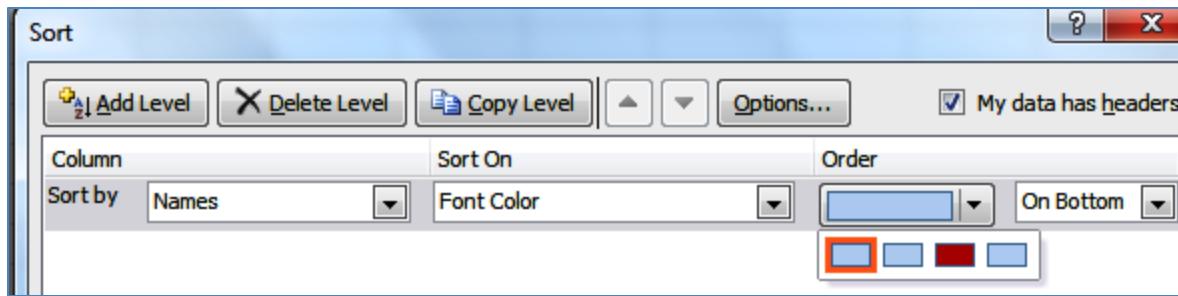
'' Adds the sort field with column index, sort based on and order by attribute
field = sorter.SortFields.Add(2, SortOn.Values, OrderBy.OnTop)

'' Adds the second sort field
field = sorter.SortFields.Add(2,SortOn.Values,OrderBy.OnTop)

'' Sorts the data with the sort field attribute
sorter.Sort()
```

4.5.3.3 Sorting by Font Color

With this feature, MS Excel moves the text that is applied with the selected color to the specified location (bottom or top) of the sorting range.



Sorting by Font Color

This is explained in the following code samples:

C#

```
//Creates the data sorter
IDataSort sorter = book.CreateDataSorter();
//Range to sort
sorter.SortRange = range;
//Creates the sort field with the column index, sort based on and order by
attribute
ISortField sortField = sorter.SortFields.Add(2, SortOn.FontColor, OrderBy.OnTop);
//Specifies the color to sort the data
sortField.Color = Color.Red;
//Sort based on the sort Field attribute
sorter.Sort();
```

VB

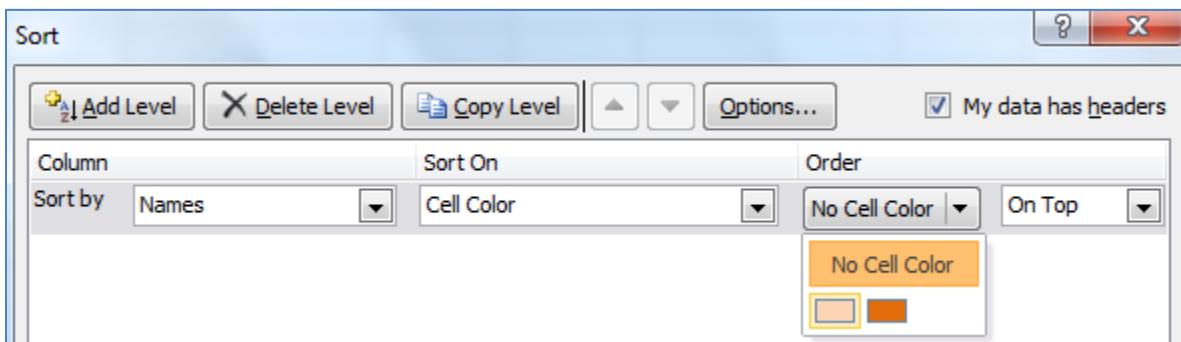
```
'Creates the Data sorter
Dim sorter As IDataSort = book.CreateDataSorter()
'Specifies the sort range
sorter.SortRange = range

Dim field As ISortField
' Adds the sort field with column index, sort based on and order by attribute
field = sorter.SortFields.Add(2, SortOn.FontColor, OrderBy.OnTop)
```

```
''Sorts the data based on this color
field.Color = Color.Red
'' Sorts the data with the sort field attribute.
sorter.Sort()
```

4.5.3.4 Sorting by Cell Color

With this feature, MS Excel moves the cell text and color to the specified location (bottom or top) of the sorting range.



Sorting by Cell Color

This is explained in the following code samples:

C#

```
//Creates the data sorter
IDataSort sorter = book.CreateDataSorter();
//Range to sort
sorter.SortRange = range;

//Creates the sort field with the column index, sort based on and order by
attribute
ISortField sortField = sorter.SortFields.Add(2, SortOn.CellColor,
OrderBy.OnTop);

//Specifies the color to sort the data
sortField.Color = Color.Red;
//Sort based on the sort field attribute
```

```
sorter.Sort();
```

VB

```
''Creates the Data sorter
Dim sorter As IDataSort = book.CreateDataSorter()
''Specifies the sort range.
sorter.SortRange = range

Dim field As ISortField
'' Adds the sort field with column index, sort based on and order by attribute
field = sorter.SortFields.Add(2, SortOn.CellColor, OrderBy.OnTop)
''Sorts the data based on this color
field.Color = Color.Red
'' Sorts the data with the sort field attribute
sorter.Sort()
```

4.5.4 Import/Export

XlsIO has some helper methods that enable working with the ADO.NET data sources very easily.

Importing

It only takes one line of code to import an ADO.NET data table into a worksheet. There are similar methods for working with other data sources like **Data View**, **Data Column**, **Arrays**, and so on. A data table from another source can be imported inside a worksheet by using the **ImportDataTable** method. It has an option to select the record range (row start, row end, col start and col end). It also allows preserving the data type in the data source.

Exporting

Similarly, it is simple to export the sheet data to a data table by using the **ExportDataTable** method of **IWorksheet**. This method allows to select the various data table options such as include column names, export formula calculated values, styles and types, through the **ExcelExportDataTableOption** enumeration. It has the following values.

Member Name	Description
None	No datatable exports flags.
ColumnNames	Represents the ColumnNames datatable export flag.
ComputedFormulaValues	Represents the ComputedFormulaValues datatable export flag.
DetectColumnTypes	Indicates that XlsIO should try to detect column types.
DefaultStyleColumnTypes	When DetectColumnTypes is set and this flag is set too, it means that default column style must be used to detect style, if this flag is not set, but DetectColumnTypes is set, then first cell in the column will be used to detect column type.

 For more information on this feature, see: <http://www.syncfusion.com/products/reporting-edition/xlsio/features#export-options>

The following image illustrates the import and export of the data from/to a grid or database.

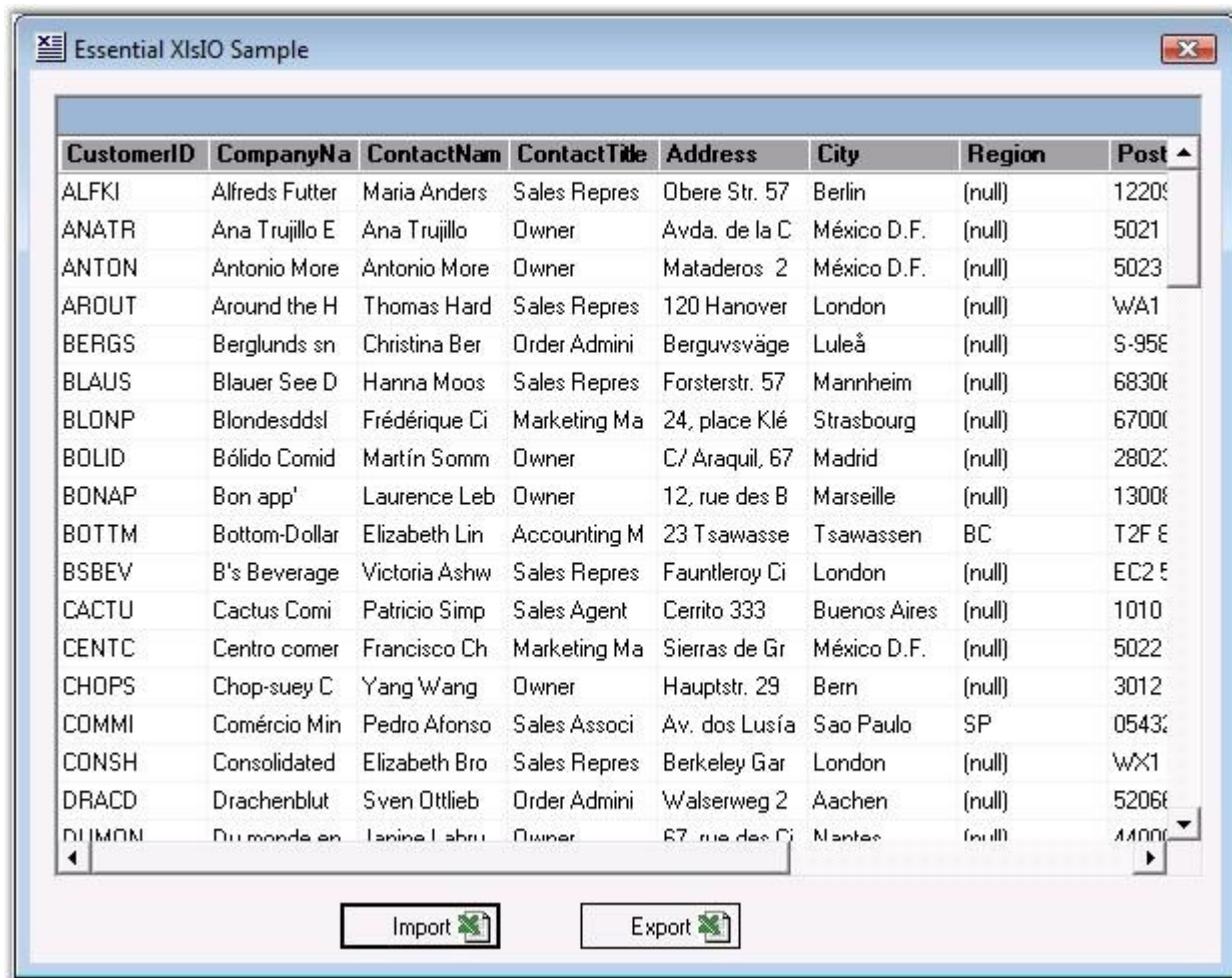


Figure 133: Import/Export Data

Following code example illustrates how to export a data table from a sheet to a grid, and then import a data table from a data grid to a sheet.

```
[C#]

// Read data from the spreadsheet.
DataTable customersTable =
sheet.ExportDataTable(sheet.UsedRange, ExcelExportDataTableOptions.ColumnNames);
this.dataGridView1.DataSource = customersTable;

// Export DataTable.
if(this.dataGridView1.DataSource != null)
{
```

```

sheet.ImportDataTable((DataTable)this.dataGrid1.DataSource,true,1,1,-1,-1);
}
else
{
    MessageBox.Show("There is no datatable to export, Please import a
datatable first","Error");
    return;
}

```

[VB .NET]

```

' Read data from the spreadsheet.
Dim customersTable As DataTable =
sheet.ExportDataTable(sheet.UsedRange,ExcelExportDataTableOptions.ColumnNames)
Me.dataGrid1.DataSource = customersTable

' Export DataTable.
If Not Me.dataGrid1.DataSource Is Nothing Then
    sheet.ImportDataTable(CType(Me.dataGrid1.DataSource,
DataTable),True,1,1,-1,-1)
Else
    MessageBox.Show("There is no datatable to export, Please import a
datatable first","Error")
    Return
End If

```

ImportDataTable has few overloads which can be used to enable some of the customization options. For more details, see XlsIO class reference in [Online documentation](#).

It enables importing data from named ranges, allows to show/hide field names in the columns, import only a particular range of records in a data table, and allows to preserve the data types in a data table.

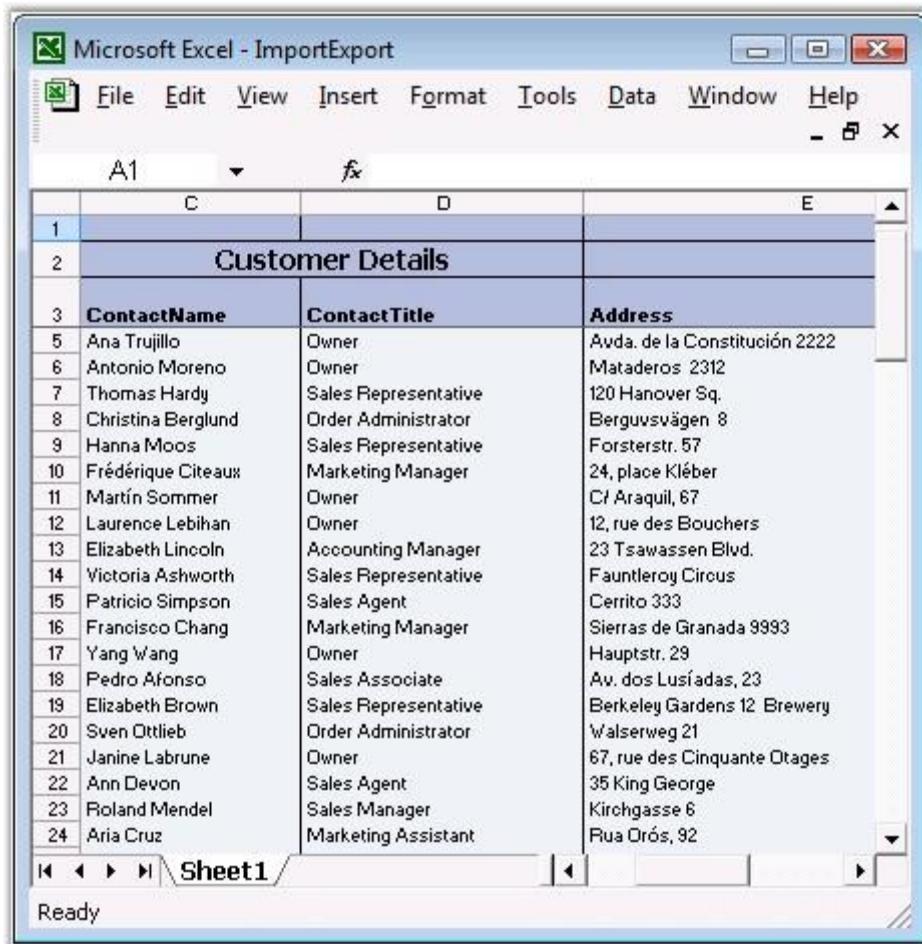


Figure 134: MS Excel file generated from the Grid data by using XlsIO

Exporting Excel to PDF

Essential XlsIO allows exporting an Excel document into PDF format. Use the Convert method of the ExcelToPdfConverter class to convert the Excel spreadsheet and save the PDF output.



Note: You need to have both **Essential PDF** and **Essential XlsIO** installed in your system since **Syncfusion.ExceltoPDFConverter.Base.dll** is conditionally shipped when both **XlsIO.Base** and **Pdf.Base** is installed.

```
[C#]
```

```
//Open the Excel document you want to convert
ExcelToPdfConverter converter = new ExcelToPdfConverter("Sample.xlsx");
```

```
//Initialize the PDF document
PdfDocument pdfDoc = new PdfDocument();

//Initialize the ExcelToPdfconverterSettings
ExcelToPdfConverterSettings settings = new ExcelToPdfConverterSettings();

// Set the Layout Options for the output Pdf page.
settings.LayoutOptions = LayoutOptions.FitSheetOnOnePage;

//Assign the PDF document to the TemplateDocument property of
//ExcelToPdfConverterSettings
settings.TemplateDocument = pdfDoc;
settings.DisplayGridLines = GridLinesDisplayStyle.Invisible;

//Convert the Excel document to PDF document
pdfDoc = converter.Convert(settings);

//Save the PDF file
pdfDoc.Save("ExceltoPdf.pdf");
```

[VB.NET]

```
'Open the Excel document you want to convert
Dim converter As New ExcelToPdfConverter("Sample.xlsx")

'Initialize the PDF document
Dim pdfDoc As New PdfDocument()

'Initialize the ExcelToPdfconverterSettings
Dim settings As New ExcelToPdfConverterSettings()

'Set the Layout Options for the output PDF page.
settings.LayoutOptions = LayoutOptions.FitSheetOnOnePage

'Assign the PDF document to the TemplateDocument property of
```

```
ExcelToPdfConverterSettings  
settings.TemplateDocument = pdfDoc  
settings.DisplayGridLines = GridLinesDisplayStyle.Invisible  
  
'Convert the Excel document to PDF document  
pdfDoc = converter.Convert(settings)  
  
'Save the PDF file  
pdfDoc.Save("ExceltoPdf.pdf")
```

Supported Elements

This feature provides support for the following elements:

- Styles
- Character formatting
- Headers and footers
- Images
- Text boxes
- Hyperlinks
- Document properties
- Comments
- Encryption
- TableStyle Support
- Text Rotations
- Excel Page Setup Options
- Unicode Support
- Background Images
- Printing Titles when Converting the Excel to PDF
- Page Break Support
- Print Area Support
- Print Order Support
- Unicode in Headers and Footers

Styles

This feature supports almost all the styles supported by Excel 2007.

Character formatting

This feature supports almost all character formatting. The supported character formatting features are:

- Character fonts
- Font size
- Font styles (bold, italic, underline, and strikethrough)
- Subscript and superscript
- Text highlighting
- Indents, tabs, and spaces
- Line spacing
- Left, right, and center justification
- Line breaks within the cell

Headers and Footers

Page headers and footers are supported and can contain images, text, and page number fields.

Images

The images present in the document are supported along with their corresponding positions and sizes.

Known Limitations - Images placed inside a shape will not be preserved in the generated PDF document.

Text Box

The text value present in the text box will be rendered as text at its actual position in the generated PDF document.

Hyperlinks

The hyperlinks present in the Excel documents will also be preserved in the generated PDF document.

Document Properties

Document properties present in the Excel documents will also be preserved in the generated PDF document.

Table Styles Support

Built-In Table styles present in the Excel documents will also be preserved in the generated PDF document.

Text Rotations

Rotated text present in the Excel sheet cell will be preserved in the generated PDF document.

Excel sheet Page Setup options

The Page setup option of the input Excel sheet will be preserved in the generated PDF document. The following are the Excel page setup options that are preserved.

- Orientation
- Center On Page

Unicode Support

The other language and unicode present in the input Excel document will be preserved in the generated PDF document.

Background Images

The Background image present in the Excel document will be preserved in the generated PDF document.

Note: The image will be get tiled based on the size of the output pdf document.

Comments

Comments present in the Excel document cells will also be preserved in the generated PDF document.

Encryption

An encrypted Excel document will also be preserved and generated as an encrypted PDF document by passing the password for the encrypted Excel document.

Unsupported Elements

The following list contains unsupported elements that presently will not be preserved in the generated PDF document.

- Shapes and auto-shapes
- Charts
- Grouping columns
- OLE Objects
- Text rotations
- Background images

Printing Titles when Converting the Excel to PDF

Title rows and columns in the Excel sheet can be printed on the PDF page, using this feature. By setting the print titles for rows and columns in the Excel sheet, the same will get reflected in the PDF when converting the Excel to PDF.

Page Break Support

Manually inserted page breaks that are available in the Excel document, will be included while laying out the PDF document.
nPrint Order Support.

Print Order Support

The Print order enabled in the Excel document will be considered while laying out the PDF page.
The following are the page order options that are supported:

- Down Then Over
- Over Then Down

Note: It considers the Print Area and Page breaks while laying out, based on Print Order.

Print Area Support

Print Area available in the Excel document will be considered while laying out the PDF document.
Both, Row Index Only[1: 20] and Column Index Only[A:D] support have also been included in
Unicode in Headers and Footers.

Unicode in Headers and Footers

The other language and unicode present in the headers and footers will be preserved in the
generated PDF document.

For More Information Refer:

[AutoFilters](#), [Validating Data](#), [Template Markers](#), [Grouping and Ungrouping](#), [Print Settings](#).

4.5.4.1 Binding Data Objects to Template Markers

Support is provided for binding business objects to template markers by binding the list of
custom class objects to template markers. It also supports header names, images and
enumeration type in business objects.

The following are its sub-features:

- Representing the HeaderName using C# Attributes
- Representing the NumberFormat using C# Attributes
- Template Marker with the Class name

4.5.4.1.1 Use Case Scenario

This feature helps users to bind business objects to template markers.

4.5.4.1.2 Feature Summary

The following are the key features:

- Representing the HeaderName using C# Attributes – The Template Marker can apply the
header name of the property (to be displayed) as the column name.

- Representing the NumberFormat using C# Attributes – the Template Marker can apply the number format, based on the property's NumberFormat attribute.
- Template Marker with the Class name - The Template Maker can bind all the properties of the business objects dynamically.

4.5.4.1.3 Attributes

Attributes	Description	Type	Data Type
HeaderName	Header name of the column/row.	Normal	String
NumberFormat	Number Format for the column/row	Normal	String

4.5.4.1.4 Binding Business Objects to Template Markers

The Marker Syntax with business objects is shown below:

Sales Person Name	Sold
%Sales.SalesPerson;insert:copystyles	%Sales.Sold

Figure 99

The following code sample illustrates the binding of data from a business object, to a marker.

```
[C#]
//Definition of the business objects
class Sales
{
    private string m_salesPerson;
    private int m_sold;

    public string SalesPerson
    {
        get
        {
            return m_salesPerson;
        }
    }
}
```

```

        set
        {
            m_salesPerson = value;
        }
    }

    public int Sold
    {
        get
        {
            return m_sold ;
        }
        set
        {
            m_sold = value;
        }
    }

    public Customer(string name,int sold)
    {
        this.m_salesPerson = name;
        this.m_sold = sold;
    }
}

// Creating Template Marker Processor
// Northwind Customers Table
ITemplateMarkersProcessor marker =
workbook.CreateTemplateMarkersProcessor();
marker.AddVariable("Sales", arrSalesPerson);

// Processing the markers in the template
marker.ApplyMarkers();

```

Template Marker representing the HeaderName and NumberFormat

The following code example illustrates the binding of data from a business object to a marker, with the NumberFormat and HeaderName arguments.

Sales Person Name	Sold
%Sales.SalesPerson;insert:copyStyles	%Sales.Sold

Figure 100

```
[C#]
//Definition of the business objects with the marker argument
class Sales
{
    private string m_salesPerson;
    private int m_sold;

    public string SalesPerson
    {
        get
        {
            return m_salesPerson;
        }
        set
        {
            m_salesPerson = value;
        }
    }
    [TemplateMarkerAttributes("Sold","$#,###")]
    public int Sold
    {
        get
        {
            return m_sold ;
        }
        set
        {
            m_sold = value;
        }
    }
    public Sales()
    {
    }
    public Sales(string name,int sold)
    {
        this.m_salesPerson = name;
        this.m_sold = sold;
    }
}
```

```

        }
    }

// Creating the Template Marker Processor
// Northwind Customers Table
ITemplateMarkersProcessor marker =
workbook.CreateTemplateMarkersProcessor();
marker.AddVariable("Sales", arrSalesPerson);

// Processing the markers in the template
marker.ApplyMarkers();

```

Output:

Sales Person Name	Sold
Andy Bernard	\$45,000
Jim Halpert	\$34,000
Karen Filippelli	\$75,000
Phyllis Lapin	\$56,500
Stanley Hudson	\$46,500
BernardShah	\$48,000
Patricia McKenna	\$34,000
Antonio Moreno Taquería	\$90,000
Thomas Hardy	\$56,500
Christina Berglund	\$46,500
Hanna Moos	\$45,000
Frédérique Citeaux	\$34,000
Martín Sommer	\$75,000
Laurence Lebihan	\$56,500
Elizabeth Lincoln	\$25,000
Victoria Ashworth	\$45,000
Patricio Simpson	\$34,000
Marketing Manager	\$75,000
YangWang	\$56,500
Pedro Afonso	\$46,500
Elizabeth Brown	\$45,000

Figure 101

Template Marker with the Class name

The following is the Marker syntax with the Class Name:

%Sales;insert:copystyles

Figure 102

```
[C#]
//Definition of the business objects with the marker argument
class Sales
{
    private string m_salesPerson;
    private int m_sold;
    [TemplateMarkerAttributes("Sales Person Name")]
    public string SalesPerson
    {
        get
        {
            return m_salesPerson;
        }
        set
        {
            m_salesPerson = value;
        }
    }
    [TemplateMarkerAttributes("Sold","$#,###")]
    public int Sold
    {
        get
        {
            return m_sold ;
        }
        set
        {
            m_sold = value;
        }
    }
    public Sales()
    {
    }
    public Sales(string name,int sold)
    {
        this.m_salesPerson = name;
        this.m_sold = sold;
    }
}
```

```

// Creating Template Marker Processor
// Northwind Customers Table
ITemplateMarkersProcessor marker =
workbook.CreateTemplateMarkersProcessor();
marker.AddVariable("Sales", arrSalesPerson);

// Processing the markers in the template
marker.ApplyMarkers();

```

Output

Sales Person Name	Sold
Andy Bernard	\$45,000
Jim Halpert	\$34,000
Karen Fillippelli	\$75,000
Phyllis Lapin	\$56,500
Stanley Hudson	\$46,500
BernardShah	\$48,000
Patricia McKenna	\$34,000
Antonio Moreno Taquería	\$90,000
Thomas Hardy	\$56,500
Christina Berglund	\$46,500
Hanna Moos	\$45,000
Frédérique Citeaux	\$34,000
Martín Sommer	\$75,000
Laurence Lebihan	\$56,500
Elizabeth Lincoln	\$25,000
Victoria Ashworth	\$45,000
Patricio Simpson	\$34,000
Marketing Manager	\$75,000
YangWang	\$56,500
Pedro Afonso	\$46,500
Elizabeth Brown	\$45,000
Sven Ottlieb	\$34,000
Janine Labrune	\$75,000
Ann Devon	\$88,000
Roland Mendel	\$76,500

Figure 103

4.5.5 Template Markers

This is another variant of the Template based approach, but the difference is that the end-user places special markers in the template spreadsheet, which gets replaced along with the data during runtime. The main advantage of this approach is that the end-user gets the flexibility of designing the Excel report.

Cells in the worksheet can be filled with single data or with multiple records. Format of these data can be changed by using the arguments of the markers.

Marker Syntax

Each marker starts with some prefix, by default it is "%" character, and followed by the variable name and properties. There could be several arguments after the variable, which are delimited by some character, by default it is semicolon (;).

Company Name	Contact Name
%Customers.CompanyName;insert:copyStyles	%Customers.ContactName

Figure 135: Template Marker

Source

XlsIO can be used to bind various data sources to these markers. This includes data sources such as **Data Table**, **Data Set**, **Data Reader**, **Data View**, **Array**, **Variable** and **Formulas**.

Arguments

You can specify the following arguments in the marker to customize the worksheet.

- **Horizontal**-This argument specifies the horizontal direction of the data import for complex variables.
- **Vertical**-This argument specifies the vertical direction of the data import for complex variables.
- **Insert**-This argument inserts new rows or columns, depending on the direction argument for each new cell. Note that by default, the rows cannot be added.
- **insert:copyStyles**-This argument copies styles from the above row or left column.
- **jump:[cell reference in R1C1 notation]**-This argument binds the data to the cell at the specified reference. Cell reference addresses can be relative or absolute.
- **copyrange:[top-left cell reference in R1C1]:[bottom-right cell reference in R1C1]**-Copies the specified cells after each cell import.

Here is the sample after dynamically filling the data during runtime.

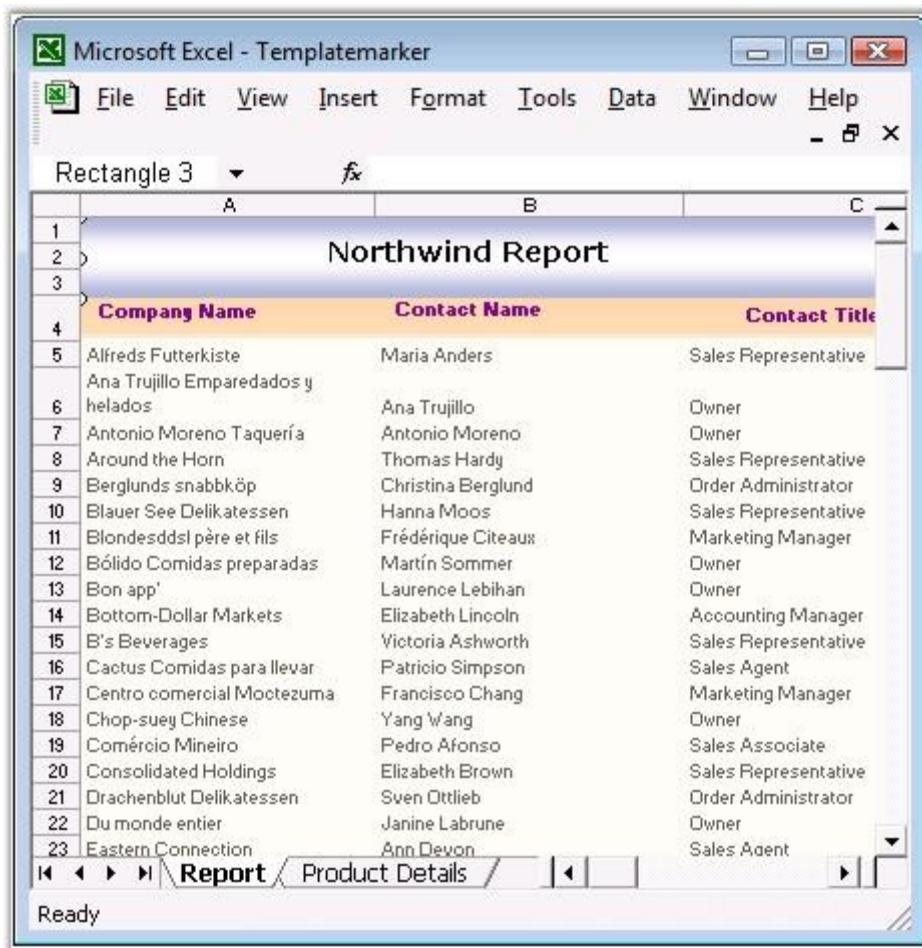


Figure 136: Template showing Markers replaced by Data using XlsIO

The unique advantage of this approach is that the end-user can have customized reports without modifying the source code of the report generating application.

Following code example illustrates how to bind the data from a data table, array and formula, to a marker.

[C#]

```
// Create Template Marker Processor.
// Northwind Customers Table
ITemplateMarkersProcessor marker =
workbook.CreateTemplateMarkersProcessor();
marker.AddVariable("Customers", northwindDt);
```

```

// Insert Array Horizontally.
string[] names = new string[] { "Mickey", "Donald", "Tom", "Jerry" };
string[] descriptions = new string[] { "Mouse", "Duck", "Cat",
"Mouse" };
marker.AddVariable("Names", names);
marker.AddVariable("Descriptions", descriptions);

// Stretch Formula
marker.AddVariable("NumbersTable", numbersDt);

// Process the markers in the template.
marker.ApplyMarkers();

```

[VB.NET]

```

' Create Template Marker Processor.
' Northwind Customers Table
Dim marker As ITemplateMarkersProcessor =
workbook.CreateTemplateMarkersProcessor()
marker.AddVariable("Customers", northwindDt)

' Insert Array Horizontally.
Dim names As String() = New String() {"Mickey", "Donald", "Tom",
"Jerry"}
Dim descriptions As String() = New String() {"Mouse", "Duck", "Cat",
"Mouse"}
marker.AddVariable("Names", names)
marker.AddVariable("Descriptions", descriptions)

' Stretch Formula
marker.AddVariable("NumbersTable", numbersDt)

' Process the markers in the template.
marker.ApplyMarkers()

```

Here, `CreateTemplateMarkerProcessor` returns the `ITemplateMarkersProcessor` interface, which creates and manipulates the marker data. `ApplyMarkers` method of `ITemplateMarkersProcessor` is the special method that processes the markers in the template.

You can also specify the marker by using the following code.

[C#]

```
// Insert Simple marker.
sheet.Range["B2"].Text = "%Marker";

// Insert marker which gets value of Author name.
sheet.Range["C2"].Text = "%Marker2.Worksheet.Workbook.Author";

// Insert marker which gets cell address.
sheet.Range["H2"].Text = "%ArrayProperty.Cells.Address";

ITemplateMarkersProcessor marker =
workbook.CreateTemplateMarkersProcessor();
marker.AddVariable("Marker", "First test of markers");
marker.AddVariable("Marker2", sheet.Range["B2"]);
marker.AddVariable("ArrayProperty", sheet.Range["B2:G2"]);

// Process the markers in the template.
marker.ApplyMarkers();
```

[VB.NET]

```
' Insert Simple marker.
sheet.Range("B2").Text = "%Marker"

' Insert marker which gets value of Author name.
sheet.Range("C2").Text = "%Marker2.Worksheet.Workbook.Author"

' Insert marker which gets cell address.
sheet.Range("H2").Text = "%ArrayProperty.Cells.Address"

Dim marker As ITemplateMarkersProcessor =
workbook.CreateTemplateMarkersProcessor()
marker.AddVariable("Marker", "First test of markers")
marker.AddVariable("Marker2", sheet.Range("B2"))
marker.AddVariable("ArrayProperty", sheet.Range("B2:G2"))

' Process the markers in the template.
marker.ApplyMarkers()
```

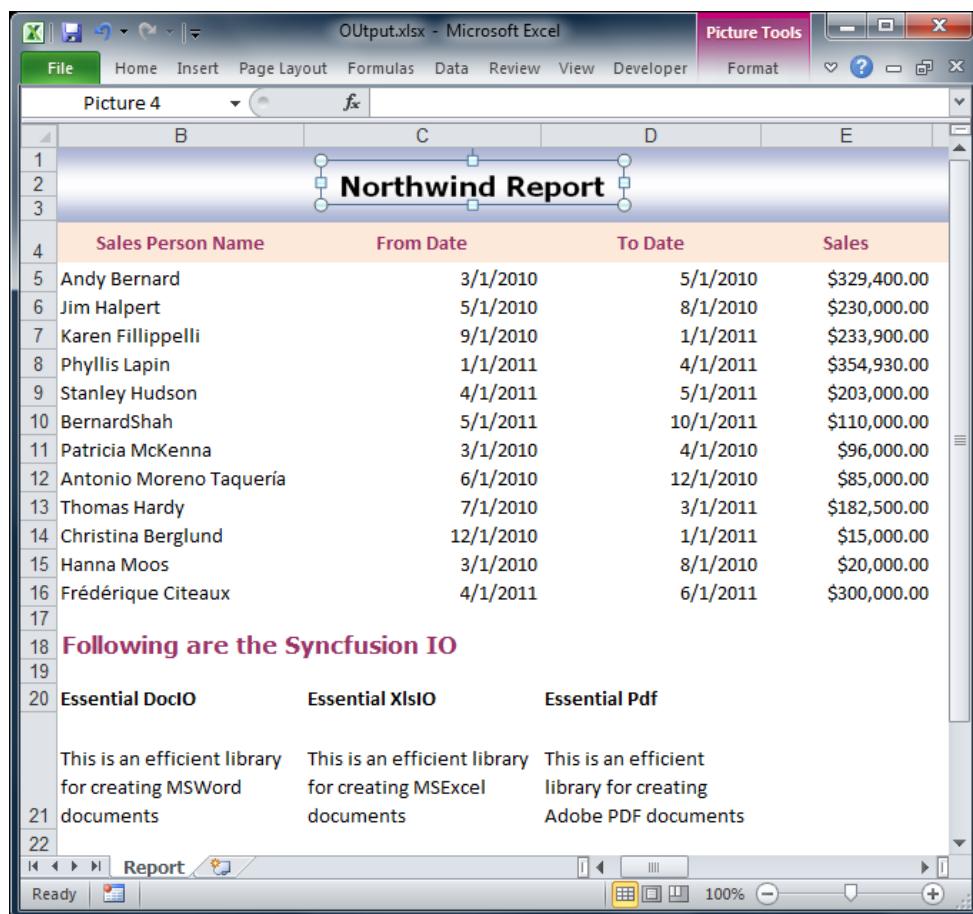
You can also create charts from the data that is bound at runtime by using the marker.

 Refer [How to Create Template Markers using XlsIO](#) for more details:

Detect Data Type and Number Formats

XlsIO now supports detecting the data type and applying the number format to the Template marker.

The following is the sample after dynamically detecting and applying data type and number format.



The screenshot shows a Microsoft Excel spreadsheet titled "Output.xlsx - Microsoft Excel". The main content is a table titled "Northwind Report" with the following data:

	Sales Person Name	From Date	To Date	Sales
5	Andy Bernard	3/1/2010	5/1/2010	\$329,400.00
6	Jim Halpert	5/1/2010	8/1/2010	\$230,000.00
7	Karen Fillippelli	9/1/2010	1/1/2011	\$233,900.00
8	Phyllis Lapin	1/1/2011	4/1/2011	\$354,930.00
9	Stanley Hudson	4/1/2011	5/1/2011	\$203,000.00
10	BernardShah	5/1/2011	10/1/2011	\$110,000.00
11	Patricia McKenna	3/1/2010	4/1/2010	\$96,000.00
12	Antonio Moreno Taquería	6/1/2010	12/1/2010	\$85,000.00
13	Thomas Hardy	7/1/2010	3/1/2011	\$182,500.00
14	Christina Berglund	12/1/2010	1/1/2011	\$15,000.00
15	Hanna Moos	3/1/2010	8/1/2010	\$20,000.00
16	Frédérique Citeaux	4/1/2011	6/1/2011	\$300,000.00

Below the table, there are three text boxes containing the following text:

- 20 Essential DocIO
- 20 Essential XlsIO
- 20 Essential Pdf

Each text box contains the same text: "This is an efficient library for creating MSWord documents", "This is an efficient library for creating MSExcel documents", and "This is an efficient library for creating Adobe PDF documents".

[C#]

```
//Create Template Marker Processor
```

```
ITemplateMarkersProcessor marker = workbook.CreateTemplateMarkersProcessor();
//Northwind customers table
marker.AddVariable("Customers", northwindDt,
VariableTypeAction.DetectNumberFormat);
//Process the markers and detect the number format along with the data type in the
template.
marker.ApplyMarkers();
```

[VB.NET]

```
'Create Template Marker Processor
Dim marker As ITemplateMarkersProcessor =
workbook.CreateTemplateMarkersProcessor()

'Northwind customers table
marker.AddVariable("Customers", northwindDt,
VariableTypeAction.DetectNumberFormat)

'Process the markers and detect the number format along with the data type in the
template.
marker.ApplyMarkers()
```

The following table gives the list of enumerations available:

Enum	Description
DetectDataType	Detects the DataType of the marker variable
DetectNumberFormat	Detects both the NumberFormat and DataType of the marker variable
None	Represents the 'None' action

Template Marker with Conditional Formatting

XlsIO allows the CreateConditionalFormat method in the ITemplateMarkerProcessor to dynamically apply the conditional format. It then creates or applies the conditional format to the template marker range dynamically.

Here is the sample for dynamically applied conditional format to data, during runtime.

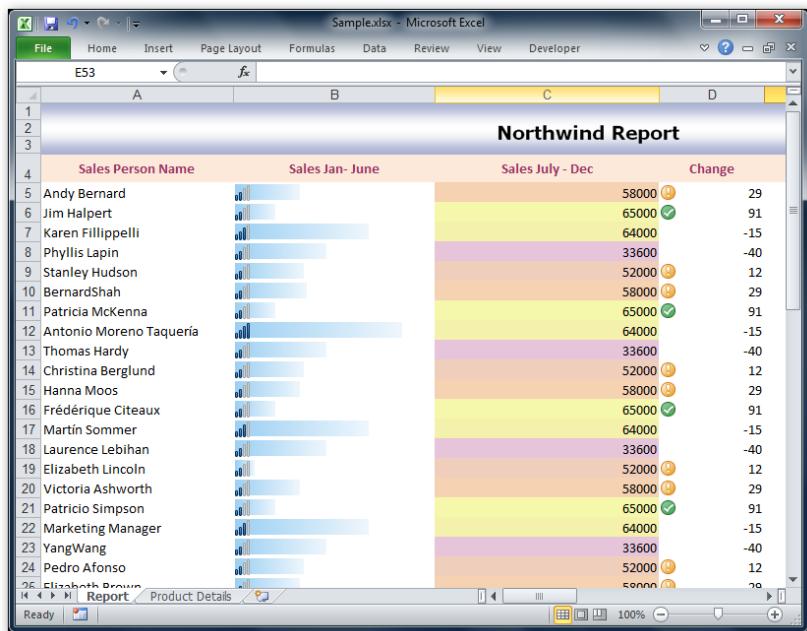


Figure 104: Dynamically applied conditional format to data (during runtime)

The following code sample illustrates how to create or apply conditional format to the Marker.

[C#]

```
ITemplateMarkersProcessor marker =
workbook.CreateTemplateMarkersProcessor();
IConditionalFormats conditions =
marker.CreateConditionalFormats(sheet["D3"]);
IConditionalFormat condition = conditions.AddCondition();
condition.FormatType = ExcelCFType.IconSet;
condition.IconSet.IconSet = ExcelIconsetType.ThreeFlags;
```

[VB]

```
marker = workbook.CreateTemplateMarkersProcessor()  
Dim conditions As IConditionalFormats =  
marker.CreateConditionalFormats(sheet("D3"))  
Dim condition As IConditionalFormat = conditions.AddCondition()  
condition.FormatType = ExcelCFType.IconSet  
condition.IconSet.IconSet = ExcelIconsetType.ThreeFlags
```

For More Information Refer:

[AutoFilters](#), [Validating Data](#), [Template Markers](#), [Grouping and Ungrouping](#)

4.5.6 Outlines

Microsoft Excel has grouping and outlining features, which allows you to group large quantities of data. You can group/ungroup a range of rows and columns. To do this, go to the **Data** menu, point to **Group and Outline**, and select **Group/UnGroup** in Excel.

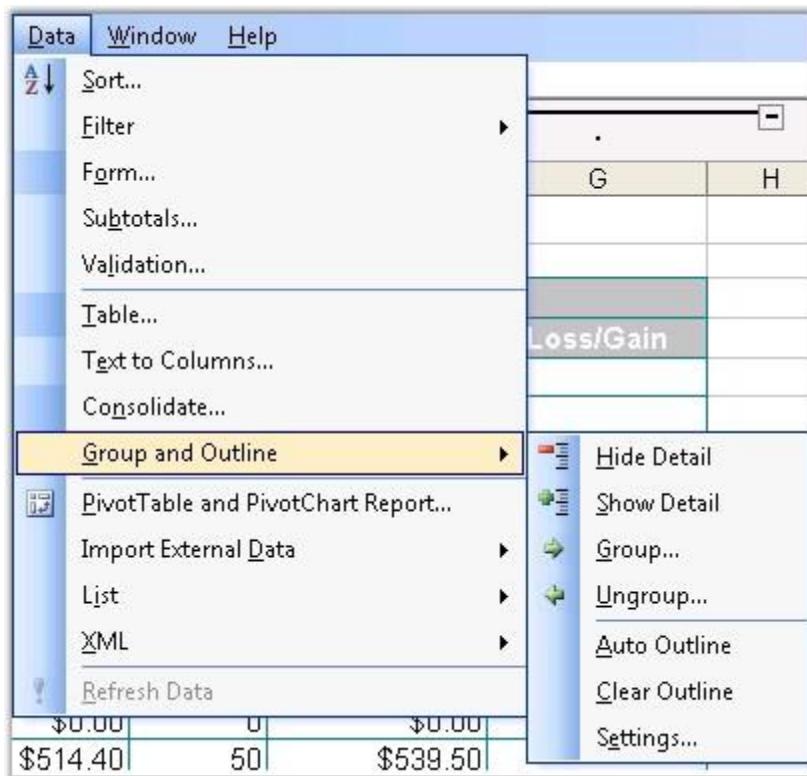


Figure 137: Grouping from Data Menu

Grouping and Ungrouping in Essential XlsIO

Essential XlsIO provides support to group and ungroup rows and columns by using the **Group** and **UnGroup** methods of **IRange**. You can also collapse or expand groups through one of its overload.

[C#]

```
// Grouping by Rows.
sheet.Range["A1:A3"].Group(ExcelGroupBy.ByRows, true);
sheet.Range["A4:A6"].Group(ExcelGroupBy.ByRows);

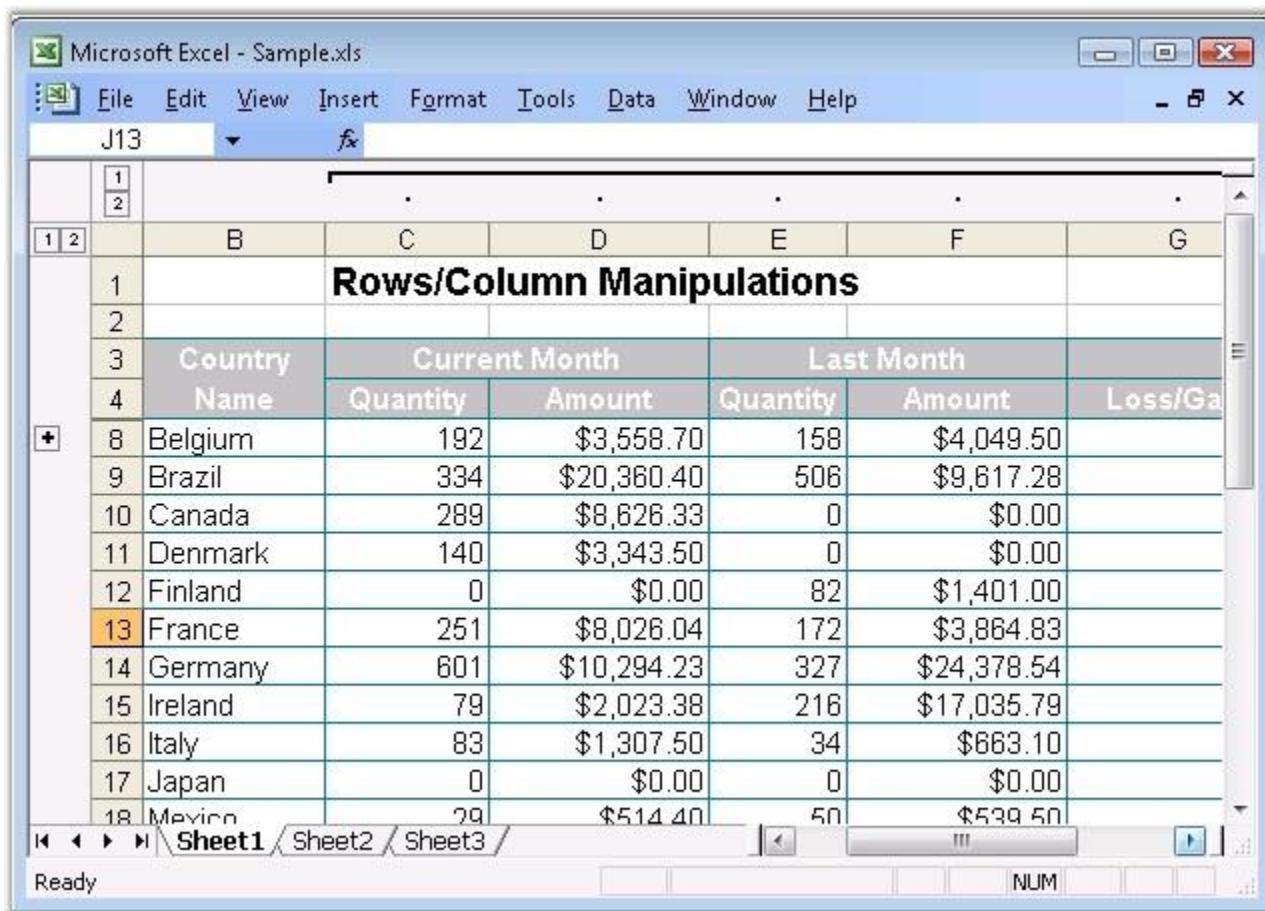
// Grouping by Columns.
sheet.Range["A1:B1"].Group(ExcelGroupBy.ByColumns, true);
sheet.Range["C1:F1"].Group(ExcelGroupBy.ByColumns);

// UnGroup by Rows
sheet.Range["A1:A3"].UnGroup(ExcelGroupBy.ByRows);
```

```
//Ungroup by columns  
sheet.Range["C1:F1"].UnGroup(ExcelGroupBy.ByColumns);
```

[VB.NET]

```
' Grouping by Rows.  
sheet.Range("A1:A3").Group(ExcelGroupBy.ByRows, True)  
sheet.Range("A4:A6").Group(ExcelGroupBy.ByRows)  
  
' Grouping by Columns.  
sheet.Range("A1:B1").Group(ExcelGroupBy.ByColumns, True)  
sheet.Range("C1:F1").Group(ExcelGroupBy.ByColumns)  
  
' UnGroup by Rows  
sheet.Range("A1:A3").UnGroup(ExcelGroupBy.ByRows)  
  
' Ungroup by columns  
sheet.Range("C1:F1").UnGroup(ExcelGroupBy.ByColumns)
```



The screenshot shows a Microsoft Excel window titled "Microsoft Excel - Sample.xls". The menu bar includes File, Edit, View, Insert, Format, Tools, Data, Window, and Help. The formula bar shows "J13" and "fx". The worksheet contains a table with the following data:

	B	C	D	E	F	G
1	Rows/Column Manipulations					
2						
3	Country	Current Month		Last Month		
	Name	Quantity	Amount	Quantity	Amount	Loss/Ga
8	Belgium	192	\$3,558.70	158	\$4,049.50	
9	Brazil	334	\$20,360.40	506	\$9,617.28	
10	Canada	289	\$8,626.33	0	\$0.00	
11	Denmark	140	\$3,343.50	0	\$0.00	
12	Finland	0	\$0.00	82	\$1,401.00	
13	France	251	\$8,026.04	172	\$3,864.83	
14	Germany	601	\$10,294.23	327	\$24,378.54	
15	Ireland	79	\$2,023.38	216	\$17,035.79	
16	Italy	83	\$1,307.50	34	\$663.10	
17	Japan	0	\$0.00	0	\$0.00	
18	Mexico	29	\$514.40	50	\$539.50	

Figure 138: Grouping in XlsIO

Excel has options to customize the Grouping settings through the **Settings** dialog box. You can show the Summary details row below or right of the column, by using the options provided in the Settings dialog box.

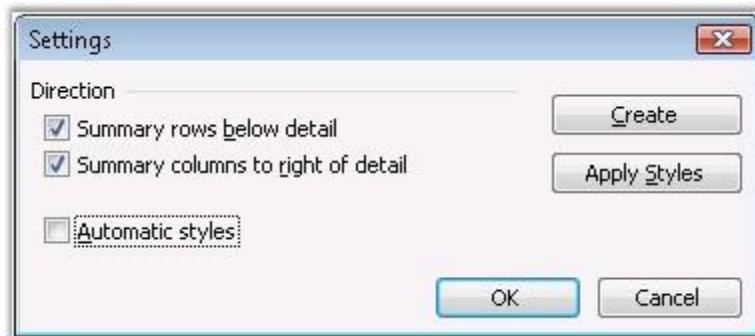


Figure 139: Grouping Settings Dialog in MS Excel

In **XlsIO**, this is set by using the **IsSummaryRowBelow** and **IsSummaryColumnRight** properties of **IPageSetup** class.

Following screenshot shows the disabled state of summary settings.

	1	2	A	B	C	D	E
1	1	2					
2							
3	Country Name	Current Month					Last
4		Quantity	Amount		Quantity		
5	Argentina	129	\$3,645.80		47		
6	Australia	0	\$0.00		0		
7	Austria	271	\$6,221.50		346		
8	Belgium	192	\$3,558.70		158		
9	Brazil	334	\$20,360.40		506		

Figure 140: Summary Settings Disabled

XlsIO also has options to check the existence of a group, and the level at which it exists. This can be done through the **IsGroupedByColumn/IsGroupedByRow** and **RowGroupLevel/ColumnGroupLevel** properties of IRange.

Expand/Collapse Groups

Essential XlsIO supports Expand and Collapse features for existing groups. Expand group comes with overloads that will allow to expand the entire parent including child groups. The Expand and Collapse features are available for both Column and Row groups.

```
[C#]

// Expand group with flag set to expand parent.
worksheet.Range["A11:A19"].ExpandGroup(ExcelGroupBy.ByRows,
ExpandCollapseFlags.ExpandParent);

// Collapse group.
worksheet.Range["A61:A114"].CollapseGroup(ExcelGroupBy.ByRows);
```

[VB .NET]

```
' Expand group with flag set to expand parent.
worksheet.Range("A11:A19").ExpandGroup(ExcelGroupBy.ByRows,
ExpandCollapseFlags.ExpandParent)

' Collapse group.
worksheet.Range("A61:A114").CollapseGroup(ExcelGroupBy.ByRows)
```

For More Information Refer:

[AutoFilters](#), [Validating Data](#), [Template Markers](#), [Grouping and Ungrouping](#)

4.6 Review

This section explains how the following features can be implemented by using XlsIO.

- **Comments**-This topic explains how comments can be inserted and formatted using the XlsIO's APIs.
- **Changes**-This topic explains various protection levels of Excel supported by XlsIO.

4.6.1 Comments

Microsoft Excel has the ability to insert Comments in cells. Comments enable a user to get additional information about a cell, such as, what the value represents. You can insert and format comments through the Insert menu in Excel. You can also format the comments inserted through the Format Comment dialog box.



Figure 141: Format Comment Dialog Box

XlsIO has APIs for inserting both **Regular** and **Rich Text** comments by using the **ICommentShape** interface. It has various properties to format the comments. Following code example illustrates how to insert comments.

```
[C#]
// Insert Comments.
// Adding comments to a cell.
sheet.Range["A1"].AddComment().Text = "Regular Comment";

// Sets author of the comment.
sheet.Range["A1"].AddComment().Author = "Syncfusion";

// Add Rich Text Comments.
IRange range = sheet.Range["A2"];
range.AddComment().RichText.Text = "RichText";
IRichTextString rtf = range.Comment.RichText;
```

```
// Formatting first 4 characters.
IFont redFont = workbook.CreateFont();
redFont.Bold = true;
rtfSetFont(0, 3, redFont);
```

[VB.NET]

```
' Insert Comments.
' Adding comments to a cell.
sheet.Range("A1").AddComment().Text= "Regular Comment"

' Sets author of the comment.
sheet.Range("A1").AddComment().Author = "Syncfusion"

' Add Rich Text Comments.
Dim range As IRange = sheet.Range("A2")
range.AddComment().RichText.Text = "RichText"
Dim rtf As IRichTextString = range.Comment.RichText

' Formatting first 4 characters.
Dim redFont As IFont = workbook.CreateFont()
redFont.Bold = True
rtfSetFont(0, 3, redFont)
```

A	B	C	D
		Administrator: This is comment with Gradient effect.	

Figure 142: XlsIO with Comments Inserted

It is also possible to read the Rich Text string formatting. The following code example illustrates how rich text comments from a cell are read by using XlsIO, and then displayed in a RichTextBox.

[C#]

```
// Read plain text comment.
this.txtPlainText.Text = sheet.Range["A1"].Comment.Text;
```

```
// Read Rich Text Comment.
this.richTextBox1.Rtf = sheet.Range["A2"].Comment.RichText.RtfText;
```

[VB.NET]

```
' Read plain text comment.
Me.txtPlainText.Text = sheet.Range("A1").Comment.Text

' Read Rich Text Comment.
Me.richTextBox1.Rtf = sheet.Range("A2").Comment.RichText.RtfText
```



Figure 143: Reading Rich Text Comments

You can also fill the comments with various types of fills by using the **IFill** interface. Following code example illustrates how to fill the comment shape with a TwoColor gradient.

[C#]

```
shape.Fill.TwoColorGradient();
shape.Fill.GradientStyle = ExcelGradientStyle.Horizontal;
shape.Fill.GradientColorType = ExcelGradientColor.TwoColor;
shape.Fill.ForeColorIndex = ExcelKnownColors.Red;
shape.Fill.BackColorIndex = ExcelKnownColors.White;
```

[VB.NET]

```
shape.Fill.TwoColorGradient()
```

```

shape.Fill.GradientStyle = ExcelGradientStyle.Horizontal
shape.Fill.GradientColorType = ExcelGradientColor.TwoColor
shape.Fill.ForeColorIndex = ExcelKnownColors.Red
shape.Fill.BackColorIndex = ExcelKnownColors.White

```

XlsIO also provides options to resize the comment size, and move/size with cell by using the **IsMoveWithCell** and **IsSizeWithCell** properties. You can also autofit the size of the comment by using the **AutoFit** property.



Note: Currently it is not possible to insert preformatted RTF tags in Excel by using XlsIO.

4.6.2 Changes

Excel provides various options that can be used to protect a workbook or worksheet, or just a cell.

This section explains how the following features of Excel can be implemented by using XlsIO.

- Workbook Protection-This section explains how a workbook can be protected by using the Windows and Structure option of Excel.
- Worksheet Protection-This section explains how a worksheet can be protected with or without a password. It also explains how a particular element modification can be prevented.
- Edit Range-This section explains how to lock/unlock a particular range in a protected worksheet.

4.6.2.1 Workbook Protection

MS Excel provides the creator of a workbook, the ability to protect the Structure and Windows of a workbook with a password. It includes the following options.

- Protecting Structure

Worksheets and chart sheets in a workbook with protection, cannot be moved, deleted, hidden, unhidden, or renamed, and new sheets cannot be inserted.

- Protecting Windows

Windows in a workbook with protection, cannot be moved, resized, hidden, unhidden, or closed. Windows in a workbook with protection, are sized and positioned the same way, each time the workbook is opened. This can be done by selecting **Protection** option from the **Tools** menu in Excel.

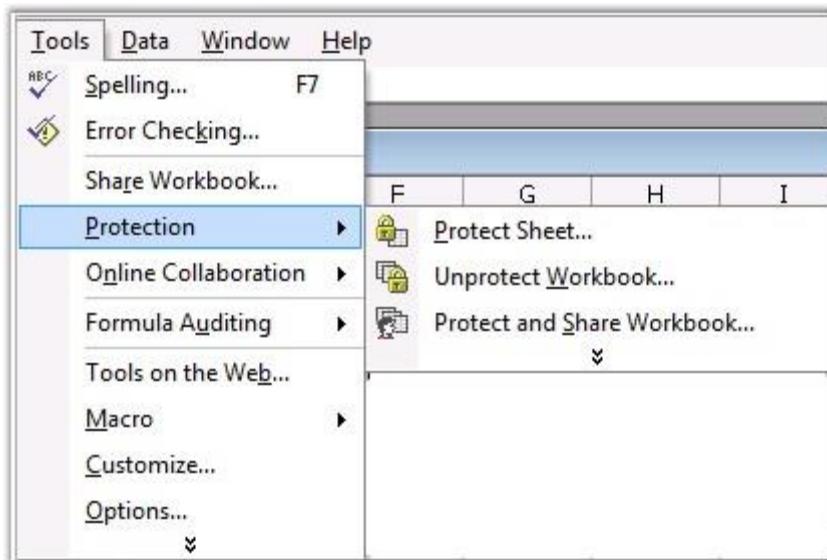


Figure 144: Tools menu - Protection



Figure 145: Protect Workbook Dialog Box

Protect method of **IWorkbook** interface provides options to protect and unprotect documents with password in XlsIO. You can also set/reset the Window and Structure option in this method.

Following code example illustrates how to protect a workbook with a password.

[C#]

```
// Protect Workbook.  
workbook.Protect(isProtectWindow, isProtectContent, "syncfusion");  
  
// Unprotect workbook.  
// Opening a Existing(Protected) Workbook.  
IWorkbook workbook = application.Workbooks.Open(@"ProtectedWorkbook.xls");  
  
// Unprotecting( unlocking) Workbook by using the Password.  
workbook.Unprotect("syncfusion");
```

[VB.NET]

```
' Protect Workbook.  
workbook.Protect(isProtectWindow, isProtectContent, "syncfusion")  
  
' Unprotect workbook.  
' Opening a Existing(Protected) Workbook.  
Dim workbook As IWorkbook =  
application.Workbooks.Open("ProtectedWorkbook.xls")  
  
' Unprotecting (unlocking) Workbook by using the Password.  
workbook.Unprotect("syncfusion");
```

Following illustration shows a protected document with the sheet max/close/min button disabled, also no sheets can be added/removed to the document.

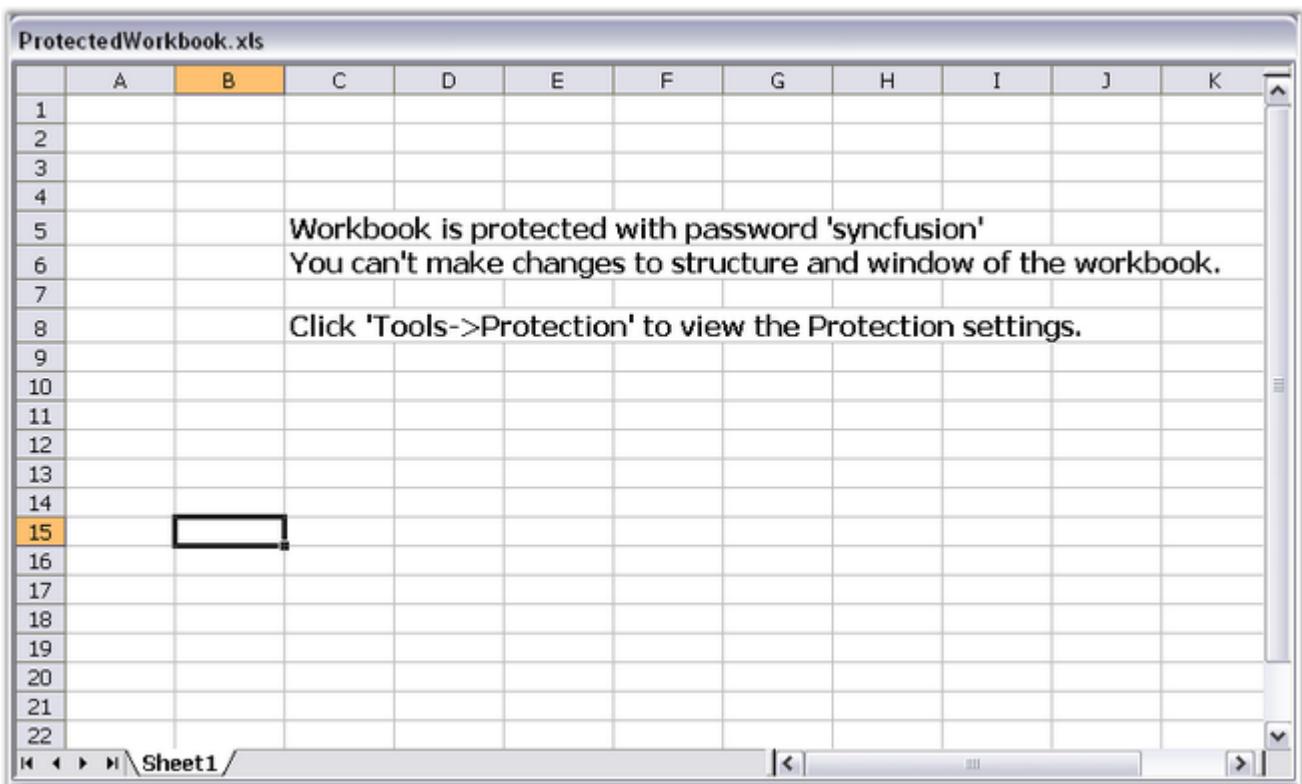


Figure 146: Protected Workbook

UnProtecting the Workbook

You can unprotect or remove protection for a document by entering the password in the **Unprotect Workbook** dialog box in MS Excel, as shown in the following screen shot.



Figure 147: Unprotecting Workbook by entering Correct Password

XlsIO also provides support to unprotect a workbook with password by using the **UnProtect** method.

[C#]

```
// Unprotect workbook.
// Opening an Existing (Protected) Workbook.
IWorkbook workbook = application.Workbooks.Open(@"ProtectedWorkbook.xls");

// Unprotecting (unlocking) Workbook by using the Password.
workbook.Unprotect("syncfusion");
```

[VB .NET]

```
' Unprotect workbook.
' Opening an Existing (Protected) Workbook.
Dim workbook As IWorkbook =
application.Workbooks.Open("ProtectedWorkbook.xls")

' Unprotecting (unlocking) Workbook by using the Password.
workbook.Unprotect("syncfusion");
```

4.6.2.2 Worksheet Protection

When you share an Excel file, so that others can collaborate on the data, you can prevent any user from making changes to specific worksheet or workbook elements, by protecting certain parts of the file.

Excel allows to protect a worksheet, and provides an option to specify the elements, users will be allowed to change, when you protect a worksheet. You can do this, by opening the **Tools** menu, and then clicking **Protection** option.

WorkSheet Protection XlsIO

XlsIO provides support for protecting and unprotecting elements in worksheets by using the **Protect** method of **IWorksheet**. By using the **ExcelSheetProtection** enumerator, you can set the elements that need protection.

Following code example illustrates how to protect a worksheet with a password. It also restricts formatting columns in the worksheet.

[C#]

```
// Protecting the Worksheet by using a Password.
sheet.Protect("syncfusion", ExcelSheetProtection.FormattingColumns);
```

[VB .NET]

```
' Protecting the Worksheet by using a Password.
sheet.Protect("syncfusion", ExcelSheetProtection.FormattingColumns)
```

You can also unprotect the worksheet by using the **Unprotect** method of XlsIO. It allows the user to remove the restriction added to worksheet elements.

Following code example illustrates how to remove worksheet protection.

[C#]

```
// Unprotecting (unlocking) the Worksheet using the Password.
sheet.Unprotect("syncfusion");
```

[VB .NET]

```
' Unprotecting (unlocking) the Worksheet using the Password.
sheet.Unprotect("syncfusion")
```

Chart Sheet Protection

Essential XlsIO now provides support to protect or unprotect a chart sheet.

a) Protecting a Chart Sheet

XlsIO provides options to protect chart sheets by using the **Protect** method. This method allows you to protect selected elements in a worksheet, so that they cannot be modified.

By using the **ExcelSheetProtection** enumeration, you can set the elements that need protection.

Following sample code illustrates protection of chart sheet (with password).

- This default call will protect the chart for Contents and Objects.
- You can also choose protection by using the overload.

The code mentioned below will choose default enumerations "Contents" and "Objects". The password chosen in the code sample below is "syncfusion".

[C#]

```
// Protect chart sheet.
chart.Protect("syncfusion");
```

[VB.NET]

```
' Protect chart sheet.
chart.Protect("syncfusion")
```

The protection can also be performed by using the enumerations in the code as shown below.

[C#]

```
// Protect chart sheet.
chart.Protect("syncfusion", ExcelSheetProtection.Content);
```

[VB.NET]

```
' Protect chart sheet.
chart.Protect("syncfusion", ExcelSheetProtection.Content)
```

The chart sheet is protected. The content in the sheet cannot be edited.

b) Removing protection of a Chart Sheet

You can remove the protection of a protected chart sheet by using the **Unprotect** method. Following code sample illustrates this.

[C#]

```
// Unprotect chart sheet
chart.Unprotect("syncfusion");
```

[VB.NET]

```
' Unprotect chart sheet.
chart.Unprotect("syncfusion")
```

The protection of the chart sheet is removed.

4.6.2.3 Edit Range

Excel also allows the user to set the permission for a range of cells, for modification, when the worksheet is protected. This can be enabled by using the **Protection** tab in the **Format Cell** dialog box. This option enables the user to remove the restriction for a range of cells, and enter/modify values in it.

XlsIO allows to unlock a range of cells by using the **Locked** property of **IRange**.

For more details, see [Lock Cells](#).

4.7 View

Excel provides multiple options to set the view for workbooks/worksheets, to allow users to get a clear perspective of large worksheets.

This section explains how the below features can be implemented through XlsIO.

- Window-This topic explains how various settings can affect the window of the Excel spreadsheet.
- Zoom-This topic explains how to apply zooming by using XlsIO.
- Show/Hide-This topic explains how to show/hide grid lines and headings in the worksheet.
- Macros-This topic explains the support provided by XlsIO with respect to VBA Macros.

4.7.1 Window

This section elaborates on the following features of Excel that control the display of worksheets.

4.7.1.1 FreezePane

It is difficult to read and understand very large spreadsheets. When you scroll too far to the right or down, you will not be able to view the headings that are located at the top and at the left side of the worksheet. Without the headings, its hard to keep track of the columns or rows of data, you are currently viewing.

Excel features Freeze Panes to avoid this problem. This feature can be enabled by selecting Freeze option from the Window menu. It allows you to "freeze" certain areas or panes of the spreadsheet, so that they remain visible at all times while scrolling to the right or bottom. Headings make it easier to read the data in the spreadsheet.

XlsIO provides support for the freeze panes functionality through the `FreezePanes` method of `IRange`.

[C#]

```
// Applying Freeze Pane to the sheet by specifying a cell.  
sheet.Range["B2"].FreezePanes();
```

[VB .NET]

```
' Applying Freeze Pane to the sheet by specifying a cell.  
sheet.Range("B2").FreezePanes()
```

	B	C	D	E
1				
2		Jan	Feb	March
60	E0060	10029	5678	45678
61	E0061	10040	5678	45678
62	E0062	10040	5678	45678
63	E0063	10040	5678	45678
64	E0064	10040	5678	45678
65	E0065	10040	5678	45678
66	E0066	10040	5678	45678
67	E0067	10040	5678	45678
68	E0068	10040	5678	45678
69	E0069	10040	5678	45678
70	E0070	10040	5678	45678
71	E0071	10040	5678	45678
72	E0072	10040	5678	45678
73	E0073	10040	5678	45678
74	E0074	10040	5678	45678
75	E0075	10040	5678	45678

Figure 148: XlsIO with Freeze Pane

XlsIO also allows you to scroll to the first row in the bottom pane and first column in the right pane. It helps you to navigate to the top row while opening a spreadsheet with large number of rows/columns. Note that this works only with the sheet that has the freeze panes.

[C#]

```
// Sets first visible row in the bottom pane.
sheet.FirstVisibleRow = 2;

// Sets first visible column in the right pane.
sheet.FirstVisibleColumn = 2;
```

[VB .NET]

```
' Sets first visible row in the bottom pane.
sheet.FirstVisibleRow = 2

' Sets first visible column in the right pane.
sheet.FirstVisibleColumn = 2
```



Note: *FirstVisibleColumn* and *FirstVisibleRow* indexes are "zero-based".

4.7.1.2 Split Pane

A very handy feature of Excel is its ability to view more than one copy of your worksheet, and scroll through each pane of your worksheet independently. You can do this by using a feature called Split Panes, which can be used to split your worksheet both horizontally and vertically. This is enabled in MS Excel by selecting the Split option from the Window menu.

While using Split Panes, the panes of your worksheet work simultaneously. If you make a change in one, it will simultaneously appear in the other.

XlsIO provides support for splitting the window through the *HorizontalSplit* and *VerticalSplit* properties. Following code example illustrates this.

[C#]

```
IWorksheet sheet = book.Worksheets[0];  
  
sheet.FirstVisibleColumn = 5;  
sheet.FirstVisibleRow = 11;  
sheet.VerticalSplit = 110;  
sheet.HorizontalSplit = 100;  
sheet.ActivePane = 1;  
  
book.SaveAs(WORKSHEETS_PANE);
```

[VB.NET]

```
Dim sheet As IWorksheet = book.Worksheets(0)  
  
sheet.FirstVisibleColumn = 5  
sheet.FirstVisibleRow = 11  
sheet.VerticalSplit = 110  
sheet.HorizontalSplit = 100  
sheet.ActivePane = 1  
  
book.SaveAs(WORKSHEETS_PANE)
```

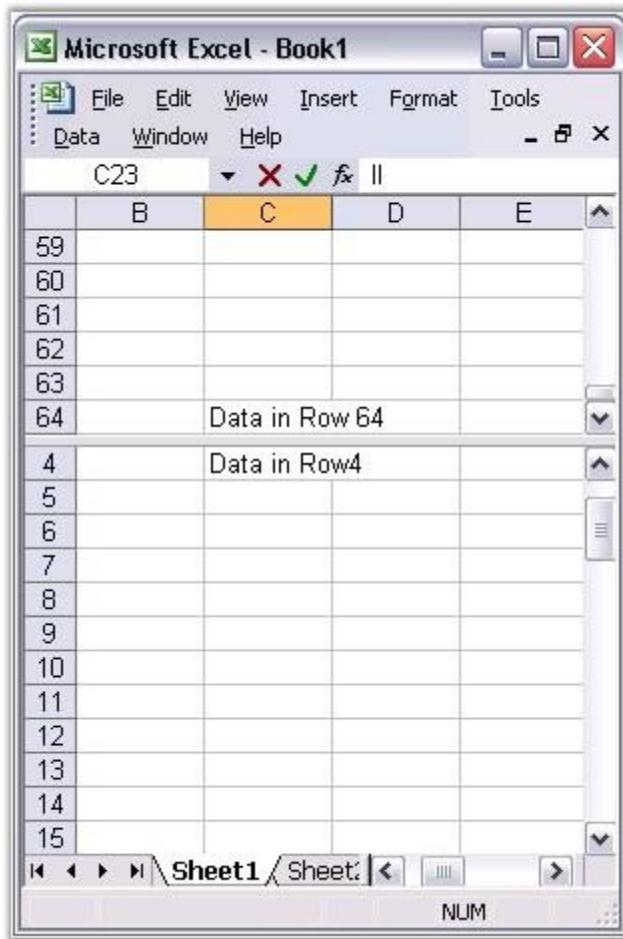


Figure 149: XlsIO with Freeze Pane

4.7.2 Zooming

Zooming feature controls the current document that appears on the screen, no matter how big or small it is. This enables reading the charts and figures in your Microsoft Excel spreadsheet without finding any difficulty.

Excel allows zooming the worksheet/range of cells to fit into the window. Default value of Excel zooming is 100 percent, and can be zoomed till 400 percent. Minimum Zooming is 10 percent.

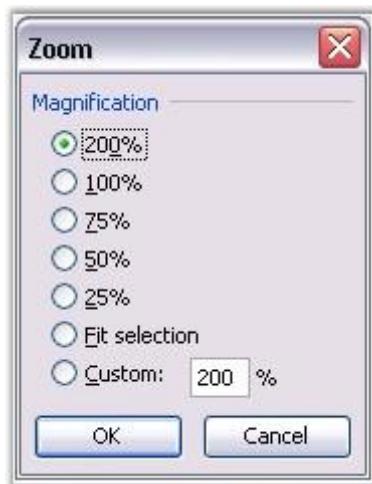


Figure 150: Zoom dialog box in Excel

Zooming in XlsIO

XlsIO allows to zoom a worksheet by using the **Zoom** property of **IWorksheet**. It returns or sets the display size of the window as a percentage (100 equals normal size, 200 equals double size, and so on).

Following code example illustrates how to zoom a worksheet to 150 percent.

[C#]

```
// Zooming to 150 percent.
sheet.Zoom = 150;
```

[VB]

```
// Zooming to 150 percent
sheet.Zoom = 150
```

4.7.3 Show/Hide Worksheet Elements

This topic describes how to show/hide the elements in a worksheet and workbook.

- Grid Lines
- Headings
- Scroll Bar

- Sheet Tabs

Grid Lines

Some users may find it easier to work with your worksheet applications, if they cannot see the grid lines. Excel provides options to show/hide grid lines in the worksheet. This is done by accessing the GridLines option by opening the **Tools** menu, pointing to **Option**, and then selecting **GridLines**.

XlsIO provides support for this feature through the **IsGridLine** property of **IWorksheet**. Color for the grid line can also be set through the **GridLineColor** property of **IWorksheet**.

[C#]

```
// Hides grid line.  
sheet.IsGridLinesVisible = false;
```

[VB]

```
' Hides grid line.  
sheet.IsGridLinesVisible = False
```

Headings

Headings are the display labels in worksheets that enable users to find out the cell number with ease. You can show/hide these headings by using the **IsRowColumnHeadersVisible** property of **IWorksheet**.

[C#]

```
sheet.IsRowColumnHeadersVisible = false;
```

[VB]

```
sheet.IsRowColumnHeadersVisible = False
```

	A	B
1	Column 1	Column 2
2	Column 3	Column 4
3	Column 5	Column 6
4	Column 7	Column 8
5	Column 9	Column 10
6	Column 11	Column 12
7	Column 13	Column 14
8	Column 15	Column 16
9	Column 17	Column 18
10	Column 19	Column 20
11	Column 21	Column 22

Figure 151: Row/Column Headers

Scroll Bar

You may allow the users to view a particular worksheet, but hide the content in the last part of the worksheet from them. This can be done by hiding the scrollbars, by turning off either scrollbar checkbox, in the View tab of the Options dialog box.

XlsIO allows to control the visibility of these horizontal and vertical scrollbars in a workbook by using the IsHScrollBarVisible and IsVScrollBarVisible properties of IWorkbook as follows.

[C#]

```
//Hides Horizontal scroll bar and show the vertical scroll bar
workbook.IsHScrollBarVisible = false;
workbook.IsVScrollBarVisible = true;
```

[VB]

```
'Hides Horizontal scroll bar and show the vertical scroll bar
workbook.IsHScrollBarVisible = False
workbook.IsVScrollBarVisible = True
```

Sheet Tabs

Excel allows to show/hide the workbook tabs, to prevent users from switching between sheets through sheet tabs, and focus their attention on a particular sheet.

XlsIO provides an option to hide the workbook tabs by using the **IWorkbook.DisplayWorkbookTabs** property. XlsIO also provides an option to get the current tab that is displayed by using the **DisplayedTab** property of IWorkbook.

[C#]

```
workbook.DisplayWorkbookTabs = false;
```

[VB]

```
workbook.DisplayWorkbookTabs = False
```

4.7.4 Macros

XlsIO supports the usage of Macros in the Template file. A macro is created by using the MS Excel GUI, and then saved. The spreadsheet is then opened by using XlsIO, and saved to retain the macro.

1. Create a macro by using MS Excel and save the file.
2. Open the saved file with XlsIO.
3. XlsIO preserves the macro during the Save process.

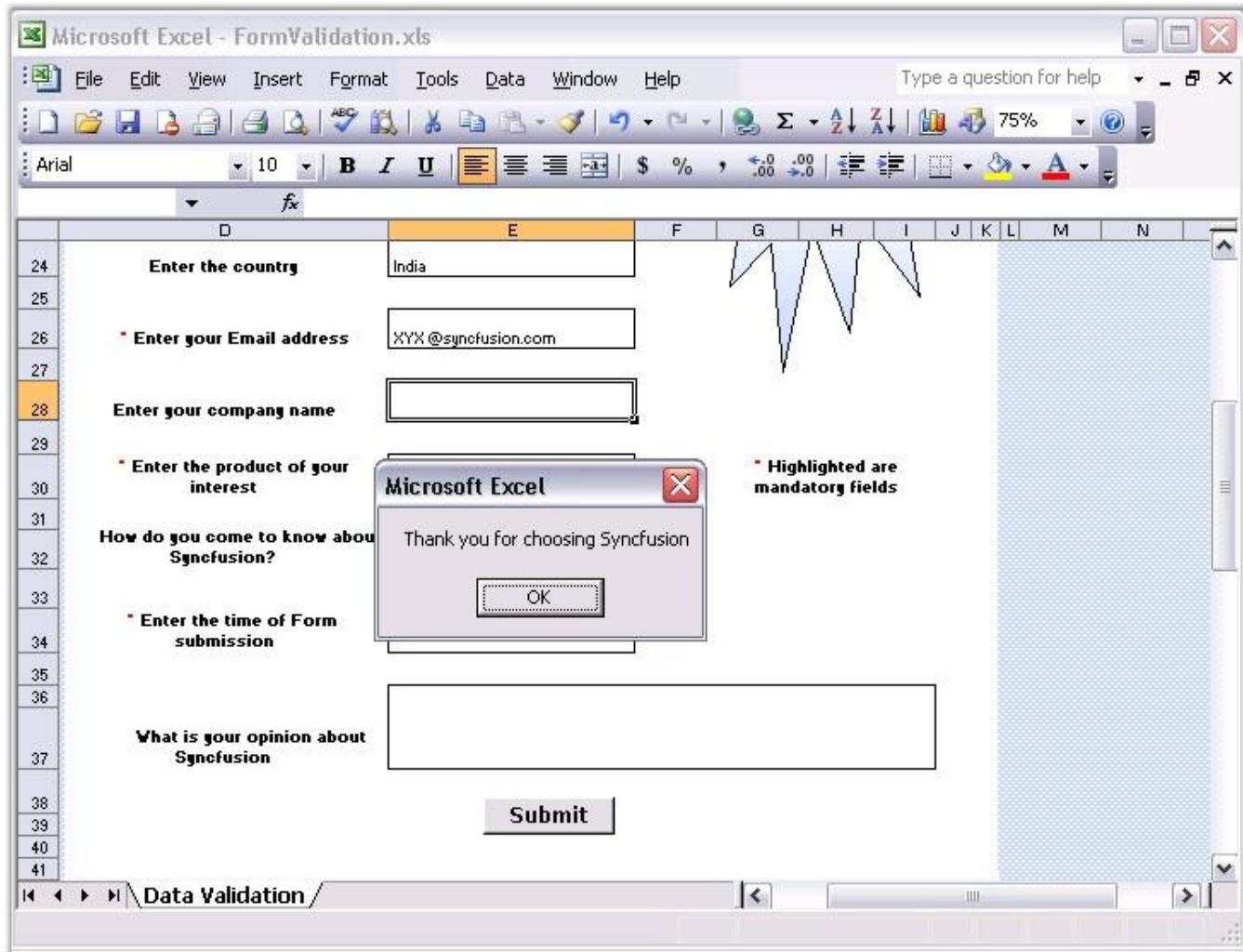


Figure 152: XlsIO with VBA in Template

XlsIO also provides support for disabling macros in the template workbook as follows. This will just ignore the macros in the template, and work normally as if there were no macros in the template.

[C#]

```
workbook.DisableMacrosStart = true;
```

[VB .NET]

```
workbook.DisableMacrosStart = True
```

4.8 Prepare

Excel provides various features to prepare a document for sharing and distribution. These features help the users to add details on their report and share them in a secured way.

This section elaborates on the following features that play a vital role in preparing an Excel spreadsheet for distribution.

- Document Properties-This topic explains the various document properties that can be added to a workbook through XlsIO.
- Encryption and Decryption-This topic explains the workbook encryption and decryption with password through XlsIO.

4.8.1 Document Properties

Document Properties are named values that provide information about the document, such as the date and time at which the document was last saved, the last user to modify the document, and so on. Document Properties are either built into the document, or are custom user-defined properties.

You can read, and manually add or modify some Built-In properties, and all Custom properties, by selecting the properties from the **File** menu. Built-in properties can be automatically updated properties such as **LastSaveDate**, or preset properties such as **Keywords**.

XlsIO allows to read and write Built-In and Custom properties through the **IBuiltinDocumentProperties** and **ICustomDocumentProperties**.

The following code example illustrates how to set the spreadsheet's Built-In and Custom properties.

[C#]

```
// Setting Built-in Document Properties.
IBuiltinDocumentProperties builtInProperites =
book.BuiltinDocumentProperties;
builtInProperites.Author = "Essential XlsIO";
builtInProperites.Comments = "This document was generated using Essential
XlsIO";
builtInProperites.ByteCount = 120;
builtInProperites.LastSaveDate = new DateTime( 1950, 1, 2, 3, 4, 5, 6 );
builtInProperites.Manager = "Manager";

// Setting Custom Properties.
```

```
ICustomDocumentProperties customProperites =
workbook.CustomDocumentProperties;
customProperites[ "Author" ].Text = ""Essential XlsIO"";
customProperites[ "Comments" ].Text = "XlsIO support Custom document
properties";
customProperites[ "Double" ].Double = 120.2;
```

[VB.NET]

```
' Setting Built-in Document Properties.
Dim builtInProperites As IBuiltInDocumentProperties =
book.BuiltInDocumentProperties
builtInProperites.Author = "Essential XlsIO"
builtInProperites.Comments = "This document was generated using Essential
XlsIO"
builtInProperites.ByteCount = 120
builtInProperites.LastSaveDate = New DateTime(1950, 1, 2, 3, 4, 5, 6)
builtInProperites.Manager = "Manager"

' Setting Custom Properties.
Dim customProperites As ICustomDocumentProperties =
workbook.CustomDocumentProperties
customProperites( "Author" ).Text = "Essential XlsIO"
customProperites( "Comments" ).Text = "XlsIO support Custom document
properties"
customProperites( "Double" ).Double = 120.2
```



Figure 153: XlsIO with Document Properties

4.8.2 Encryption and Decryption

This section illustrates encryption and decryption in XlsIO.

Encryption

Encryption is a method for protecting the workbook data, based on a password, which converts it into the form that cannot be understood, and restricts anonymous users from accessing the data in the document.

A password for encrypting the workbook can be set in MS Excel through the **Security** tab of the **Options** dialog box (Tools menu, Options command).

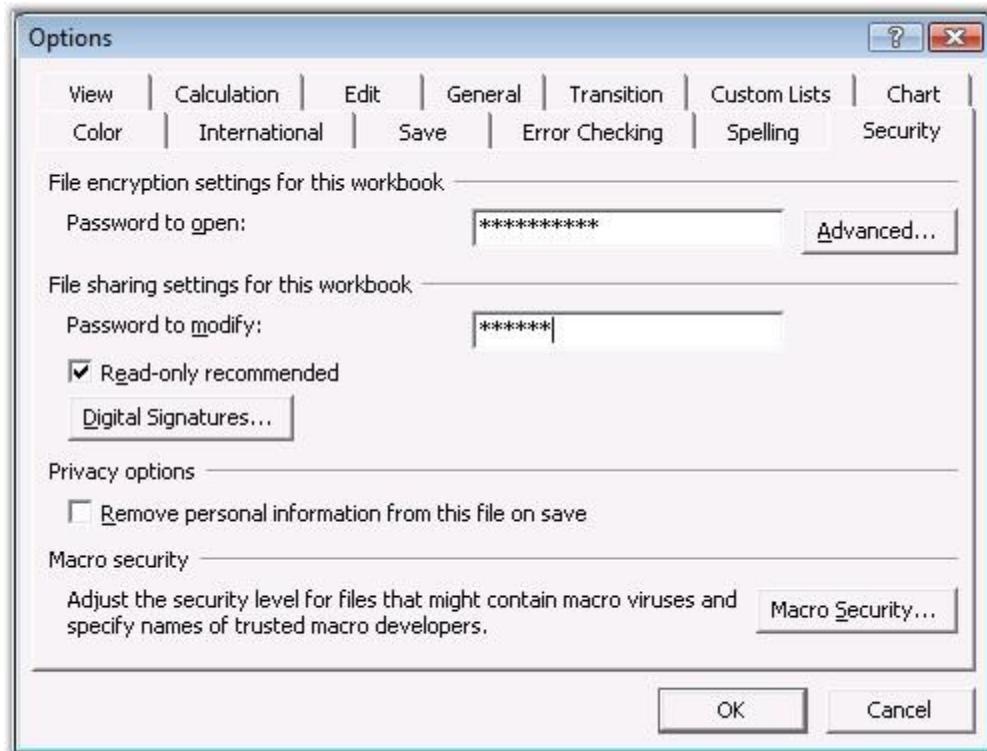


Figure 154: Options Dialog Box - Security

There are two separate passwords that the users must type to encrypt the document.

1. **Password To Open**-This password is encrypted to help protect your data from unauthorized access.



Figure 155: Entering Password to Open

2. **Password To Modify**-This password is not encrypted, and is only meant to give specific users permission to edit workbook data and save changes to the file.

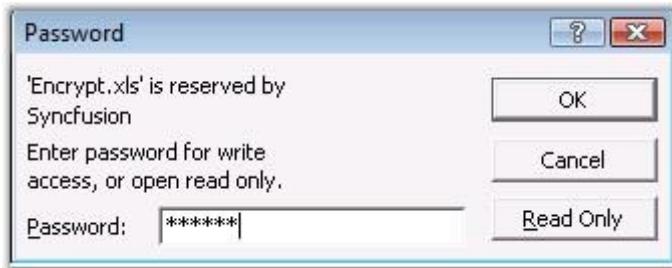


Figure 156: Entering Password to Modify



Note: Password protection of a workbook file is different from the workbook structure and window protection that you can set in the Protect Workbook dialog box.

Read-Only Recommended - This option will prompt read-only recommendation, when users open the file, and remind them that the data is important and should not be changed. This can be set with or without requiring a password to open the file.

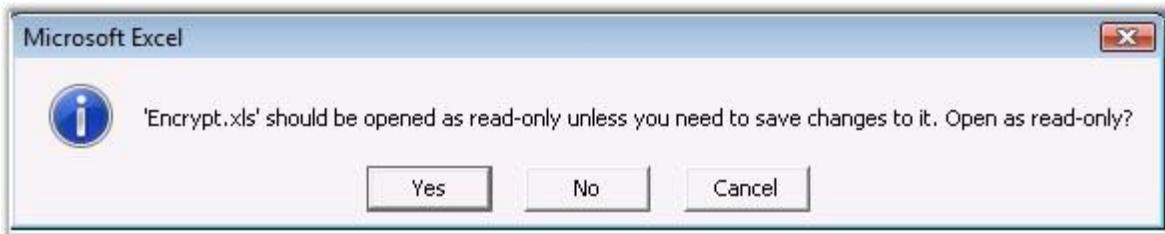


Figure 157: Prompting for Read-Only

XlsIO allows to set encryption with all the above options through the **IWorkbook** interface. You can set the password for encryption through the **PasswordToOpen** property.

Following code example illustrates how to encrypt an Excel workbook with password to open, modify, and set the read-only option.

[C#]

```
// Encryption:  
// Encrypt the workbook with password.  
workbook.PasswordToOpen = "syncfusion";  
  
// Set the password to modify the workbook.  
workbook.SetWriteProtectionPassword("modify");  
  
// Set the workbook as read-only.  
workbook.ReadOnlyRecommended = true;  
  
// Decryption:  
// Opening the encrypted workbook.  
IWorkbook workbook = application.Workbooks.Open("FileName.xls",  
ExcelParseOptions.Default, true, "syncfusion");
```

[VB.NET]

```
' Encryption:  
' Encrypt the workbook with password.  
workbook.PasswordToOpen = "syncfusion"  
  
' Set the password to modify the workbook.  
workbook.SetWriteProtectionPassword("modify")  
  
' Set the workbook as read-only.  
workbook.ReadOnlyRecommended= true  
  
' Decryption:  
' Opening the encrypted workbook.  
Dim workbook As Syncfusion.XlsIO.IWorkbook =  
Application.Workbooks.Open("FileName.xls", ExcelParseOptions.Default, True,  
"syncfusion")
```



Note: Essential XlsIO supports default encryption of the type "Office97-2000 compatible", and does not support weak and strong encryption types.

Decryption

Decryption is the process of converting encrypted data back into its original form, so that the data can be read from the workbook. You can decrypt the workbook with the encrypted password. Note that XlsIO cannot open workbooks without knowing the password.

[C#]

```
// Decryption:  
// Opening the encrypted workbook.  
IWorkbook workbook = application.Workbooks.Open("FileName.xls",  
ExcelParseOptions.Default, true, "password");
```

[VB .NET]

```
' Decryption:  
' Opening the encrypted workbook.  
Dim workbook As Syncfusion.XlsIO.IWorkbook =  
Application.Workbooks.Open("FileName.xls", ExcelParseOptions.Default, True,  
"password")
```

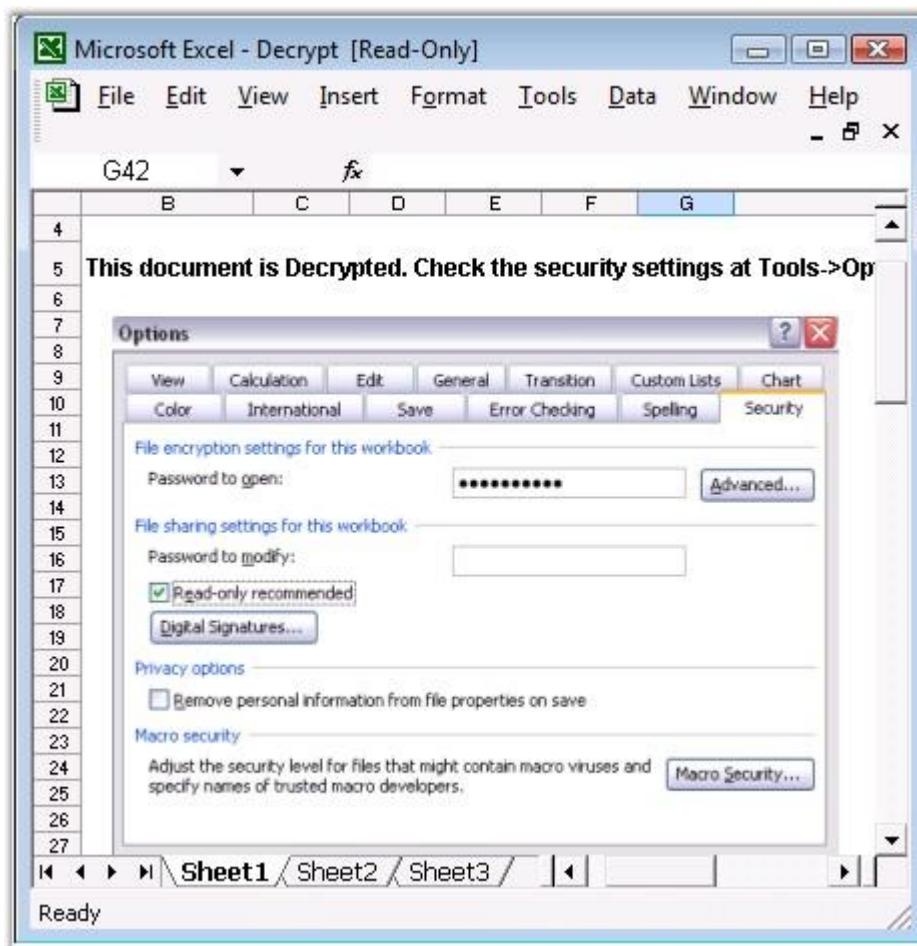


Figure 158: Decrypting the Document

4.8.2.1 Encryption and Decryption for Excel 2010

Now, support is provided for parsing and serializing the encrypted files in Excel 2010, with passwords. The type of encryption used in Excel 2010 is Agile encryption and MS Excel 2007 with SP2, can open MS Excel 2010 encrypted files.

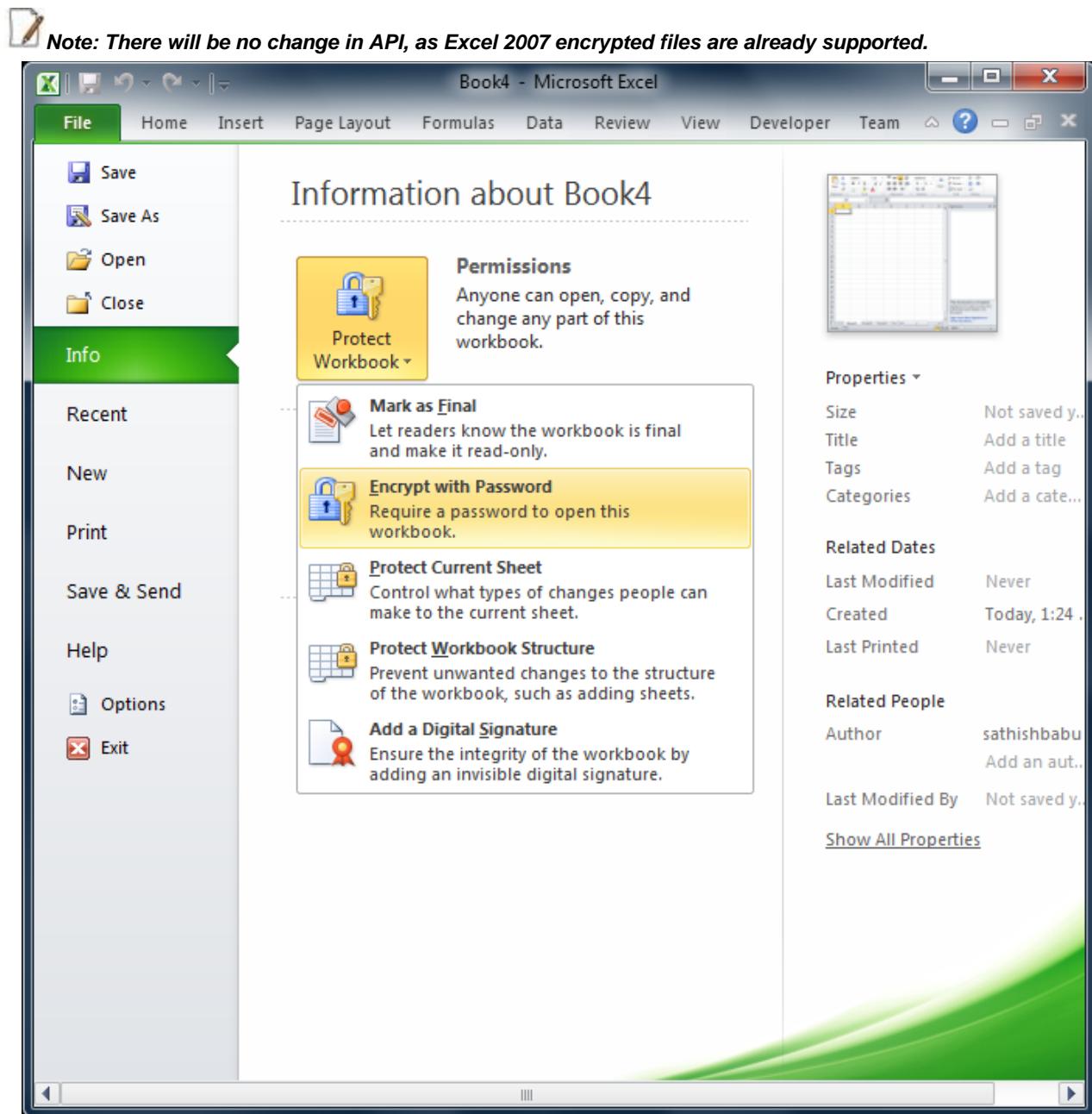


Figure 159: Encrypting a file in MS Excel 2010

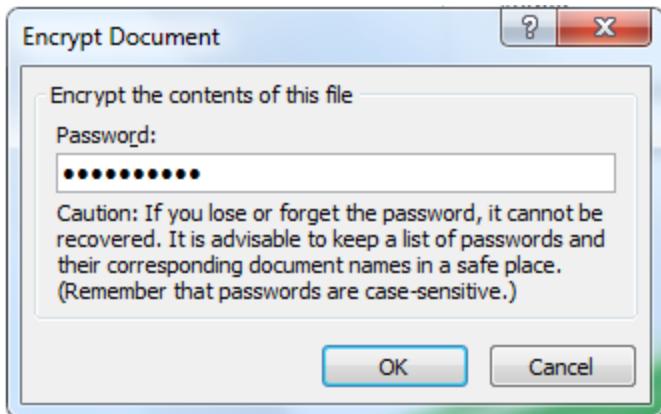


Figure 160: Encrypting a file with password in MS Excel 2010

4.9 Add-Ins

An Excel Add-In is a file (usually with a .xla or .xll extension) that Excel loads when it starts up. The file contains code (VBA in the case of a .xla Add-In) that adds additional functionality to Excel, usually in the form of new functions.

Add-Ins provide an excellent way of increasing the power of Excel, and they are the ideal vehicle for distributing your custom functions. Excel is shipped with a variety of Add-Ins ready for you to load and start using, and many third-party Add-Ins are also available. You can select these functions through the Add-Ins dialog box.



Figure 161: Add-Ins

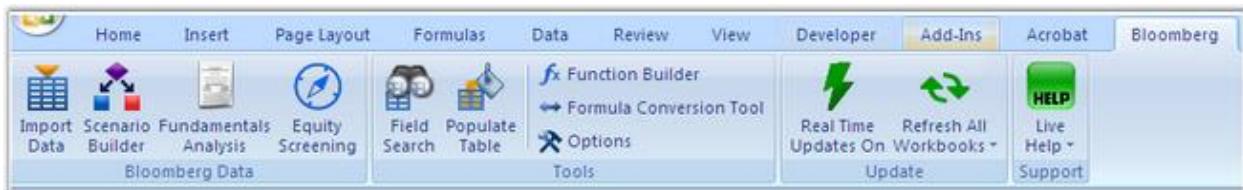


Figure 162: Add-Ins in Excel

XlsIO provides support for Excel and custom Add-Ins. They can be accessed by first registering, and then using the Add-In's custom functions through XlsIO formulas.

Following code example illustrates how to register and use Add-Ins.

[C#]

```
IAddInFunctions unknownFunctions = workbook.AddInFunctions;
unknownFunctions.Add("c:\blp\api\dde\blp.xla", "blp");

// Use the Function.
sheet.Range[ "A3" ].Formula = "blp(A1+" CORP\",\"PX_LAST\")";
```

[VB.NET]

```
Dim unknownFunctions As IAddInFunctions = workbook.AddInFunctions  
unknownFunctions.Add("c:\blp\api\dde\blp.xla", "blp")  
  
' Use the Function.  
sheet.Range("A3").Formula = "blp(A1"" CORP"", ""PX_LAST"")"
```



Note: If you move the file to another computer, or distribute it, the workbook will expect to find the same Add-In, in the same place, on their computers. But, if the Add-In is moved or deleted from the computer, the workbook won't be able to find it, and your code won't work. Make sure that the Add-In is accessed by locating the .xla file through the Tools menu (Tools -> Addins -> Browse).

5 Frequently Asked Questions

This section contains questions that are frequently asked pertaining to Essential XlsIO and answers for the corresponding questions. This section is divided into two bases on the level of the user:

- **Common**-This section contains most common questions and answers for beginners of XlsIO.
- **Advanced**-This section contains various questions and answers at the Advanced level.

5.1 Common

This section contains most common questions and answers for beginners of XlsIO.

5.1.1 How to change the grid line color of the Excel sheet?

You can change the grid line color of the Excel worksheet by using the ExcelKnownColors property. The following code example illustrates this.

[C#]

```
sheet.GridLineColor = ExcelKnownColors.Blue;
```

[VB .NET]

```
sheet.GridLineColor = ExcelKnownColors.Blue
```

5.1.2 How to copy and paste the values of the cells that contain only formulas?

You can copy and paste the values of the cells that contain only formulas by setting ExcelCopyRangeOptions of the CopyTo method to None. The following code example illustrates this.

[C#]

```
IRange src = sheet1.Range["A3"];
IRange dest = sheet1.Range["B1"];
src.CopyTo(dest, ExcelCopyRangeOptions.None);
```

[VB.NET]

```
Dim src As IRange = sheet1.Range("A3")
Dim dest As IRange = sheet1.Range("B1")
src.CopyTo(dest, ExcelCopyRangeOptions.None)
```

5.1.3 How to copy a range from one workbook to another?

The Range and CopyTo methods include overloads for copying the Source Worksheet range to the Destination Worksheet range. The following code example illustrates how to copy a range from one workbook to another workbook.

[C#]

```
// The first worksheet object in the worksheets collection in the Source
Workbook is accessed.
IWorksheet SourceWorksheet = SourceWorkbook.Worksheets[0];

// The first worksheet object in the worksheets collection in the
Destination Workbook is accessed.
IWorksheet DestinationWorksheet = DestinationWorkbook.Worksheets[0];

// Assigning an object to the range of cells (90 rows) both for source and
destination.
IRange source = SourceWorksheet.Range[1, 1, 90, 100];
IRange des = DestinationWorksheet.Range[1, 1, 90, 100];

// Copying (90 rows) from Source to Destination worksheet.
source.CopyTo(des);
```

[VB.NET]

```
' The first worksheet object in the worksheets collection in the Source
Workbook is accessed.
Dim SourceWorksheet As Syncfusion.XlsIO.IWorksheet =
SourceWorkbook.Worksheets(0)

' The first worksheet object in the worksheets collection in the Destination
```

```

Workbook is accessed.

Dim DestinationWorksheet As Syncfusion.XlsIO.IWorksheet =
DestinationWorkbook.Worksheets(0)

' Assigning an object to the range of cells (90 rows) both for source and
destination.

Dim source As Syncfusion.XlsIO.IRange = SourceWorksheet.Range(1, 1, 90, 100)
Dim des As Syncfusion.XlsIO.IRange = DestinationWorksheet.Range(1, 1, 90,
100)

' Copying (90 rows) from Source to Destination worksheet.
source.CopyTo(des)

```

5.1.4 How to ignore the green error marker in worksheets?

You can ignore the error marker that appears in cells, when there exists data that are of different formats. This can be done by using the ExcelIgnoreError enumerator that provides various options to ignore the error marker.

[C#]

```
// Ignore Error Options.
sheet.Range["B3"].IgnoreErrorOptions = ExcelIgnoreError.All;
```

[VB.NET]

```
' Ignore Error Options.
sheet.Range("B3").IgnoreErrorOptions = ExcelIgnoreError.All
```

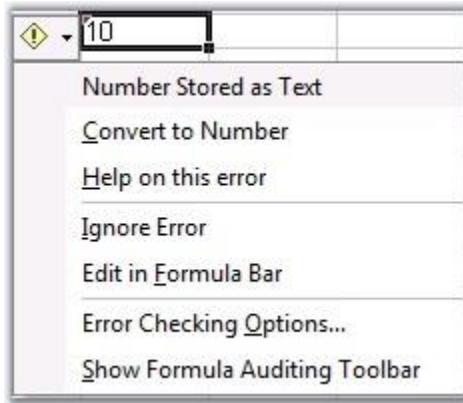


Figure 163: To ignore error

5.1.5 How to merge several Excel files to a single file?

XlsIO provides support to merge several Excel files to a single file. The following code example illustrates how to do this.

[C#]

```
// Merging worksheets.  
destinationWorkbook.Worksheets.AddCopy(sourceWorkbook.Worksheets);
```

[VB.NET]

```
' Merging worksheets.  
destinationWorkbook.Worksheets.AddCopy(sourceWorkbook.Worksheets)
```

5.1.6 How to open an Excel file from Stream?

XlsIO provides support for opening a template spreadsheet that is stored as a stream. The following code example illustrates this.

[C#]

```
// Opening a File from a Stream.  
FileStream fs = new  
FileStream(@"..\..\..\..\..\Data\OpenFromStreamTemplate.xls", FileMode.Open,  
FileAccess.ReadWrite, FileShare.ReadWrite);  
fs.Seek(0, SeekOrigin.Begin);  
IWorkbook workbook = application.Workbooks.Open(fs);
```

[VB.NET]

```
' Opening a File from a Stream.  
Dim fs As FileStream = New  
FileStream("../..\..\..\..\..\Data\OpenFromStreamTemplate.xls", FileMode.Open,  
FileAccess.ReadWrite, FileShare.ReadWrite)  
fs.Seek(0, SeekOrigin.Begin)  
Dim workbook As IWorkbook = application.Workbooks.Open(fs)
```

5.1.7 How to open an existing Xlsx workbook and save it as Xlsx?

You can open and save an existing Excel 2007 file to the .xlsx format by using XlsIO. The following code example illustrates how to do this.

[C#]

```
// Open an existing Excel 2007 file.
IWorkbook workbook =
excelEngine.Excel.Workbooks.Open(@"..\..\..\Data\Excel2007.xlsx",
ExcelOpenType.Automatic);

// Select the version to be saved.
workbook.Version = ExcelVersion.Excel2007;

// Save it as "Excel 2007" format.
workbook.SaveAs("Sample.xlsx");
```

[VB.NET]

```
' Open an existing Excel 2007 file.
Dim workbook As IWorkbook =
excelEngine.Excel.Workbooks.Open("../..\..\Data\Excel2007.xlsx",
ExcelOpenType.Automatic)

' Select the version to be saved.
workbook.Version = ExcelVersion.Excel2007

' Save it as "Excel 2007" format.
workbook.SaveAs("Sample.xlsx")
```



Note: You need to change the Excel Version, if you want to save to another version.

5.1.8 How to protect certain cells in a spreadsheet?

All the cells in an Excel spreadsheet have a Locked property, which determines if the cell will be editable when the worksheet is protected. All the cells are set to "Locked", by default. Hence when a worksheet is protected, all the cells in the worksheet get protected, by default.

However, there is often a need to protect only certain cells in a worksheet. In this scenario, you need to protect a worksheet, and set the `IsLocked` property to false for the cells that need to be made editable.

[C#]

```
// Sample data
sheetOne.Range["A1:K20"].Text = "Locked";

// A1:A10 will not be protected.
sheetOne.Range["A1:A10"].CellStyle.Locked = false;
sheetOne.Range["A1:A10"].Text = "UnLocked";
sheetOne.Protect("syncfusion", ExcelSheetProtection.FormattingColumns);
```

[VB .NET]

```
' Sample data
Private sheetOne.Range("A1:K20").Text = "Locked"

' A1:A10 will not be protected.
Private sheetOne.Range("A1:A10").CellStyle.Locked = False
Private sheetOne.Range("A1:A10").Text = "UnLocked"
sheetOne.Protect("syncfusion", ExcelSheetProtection.FormattingColumns)
```



Note: Locking/Unlocking cells in an unprotected worksheet has no effect.

5.1.9 How to save a file to stream?

XlsIO provides support to save a spreadsheet to a .NET stream. The following code example illustrates this.

[C#]

```
// Save the workbook to stream.
FileStream fs = new FileStream("FileStreamSample.xls", FileMode.Create,
FileAccess.ReadWrite, FileShare.ReadWrite);
workbook.SaveAs(fs);
workbook.Close();
```

[VB .NET]

```
' Save the workbook to stream.
Dim fs As FileStream = New FileStream("FileStreamSample.xls",
FileMode.Create, FileAccess.ReadWrite, FileShare.ReadWrite)
workbook.SaveAs(fs)
workbook.Close()
```

5.1.10 How to set a line break inside a cell?

In order to set a line break inside a cell, you have to enable Text Wrapping for the cell, and then break the text. The following code example illustrates how to do this.

[C#]

```
sheet.Range["A1"].CellStyle.WrapText = true;
sheet.Range["A1"].Text = String.Format("Hello\nworld");
```

[VB .NET]

```
sheet.Range("A1").CellStyle.WrapText = True
sheet.Range("A1").Text = String.Format("Hello" & Constants.vbLf & "world")
```

5.1.11 How to set or format a Header/Footer?

The string that the header/footer takes is a script that you can use to format the header/footer. For more information on formatting the string, see <http://support.microsoft.com/?kbid=213618>.

[C#]

```
mySheet.PageSetup.CenterHeader = @"<Gothic,bold>Center Header Text";
```

[VB .NET]

```
mySheet.PageSetup.CenterHeader = @"<Gothic,bold>Center Header Text"
```

5.1.12 How to set options to print Titles?

Printing Title Rows

The following code illustrates printing Title Rows.

[C#]

```
// Print Rows 1 to 3.  
sheet.PageSetup.PrintTitleRows = "$A$1:$IV$3";
```

[VB.NET]

```
' Print Rows 1 to 3.  
sheet.PageSetup.PrintTitleRows = "$A$1:$IV$3"
```

Printing Title Columns

The following code illustrates printing Title Columns.

[C#]

```
// Print Columns 1 to 3.  
sheet.PageSetup.PrintTitleColumns = "$A$1:$C$65536";
```

[VB.NET]

```
' Print Columns 1 to 3.  
sheet.PageSetup.PrintTitleColumns = "$A$1:$C$65536"
```

For information on Print settings, refer to section [4.3.3.1 Print Settings](#).

5.1.13 How to unfreeze the rows and columns in XlsIO?

You can unfreeze rows and columns in XlsIO by using the RemovePanes method. The following code example illustrates this.

[C#]

```
sheet.Range[8, 1].FreezePanes();
```

```
sheet.RemovePanes();
```

[VB .NET]

```
sheet.Range(8, 1).FreezePanes()
sheet.RemovePanes()
```

5.1.14 What is the maximum range of Rows and Columns?

XlsIO has support for 16,384 columns by 1,048,576 rows in Excel 2007 xlsx format, and 256 columns by 65,536 rows in Excel 97 to 2003 format. Note that this is the worksheet size of MS Excel itself.

For more information, see <http://office.microsoft.com/en-us/excel/HA100778231033.aspx>.

5.1.15 How to use Named Ranges with XlsIO?

The **NamedRanges** collection belongs to the workbook, and not to the worksheet. If you define two named ranges with the same name, then the named range that is defined last will replace the previous named range.

[C#]

```
// Looping through the Named Ranges in a spreadsheet.
foreach (IName name in mySheet.Names)
{
    MessageBox.Show(name.Name.ToString());
}

// There is already a named range called "One", I am changing the address
// that it points to.
mySheet.Names["One"].RefersToRange = mySheet.Range["B6"];

// Named ranges are added to the workbook collection in both the methods
// mentioned below.
// Adding the named Range to the workbook.
myWorkbook.Names.Add("TestRangeBook", mySheet.Range["A5"]);

// Adding the named Range to the workbook. Internally named range is added
// to the workbook names coll.
```

```
mySheet.Names.Add("TestRangeSheet", mySheet.Range["A5"]);

// Referencing from the sheet.
mySheet.Range["TestRangeSheet"].Number = 100;
```

[VB.NET]

```
' Looping through the Named Ranges in a spreadsheet.
Dim name As Syncfusion.XlsIO.IName
For Each name In mySheet.Names
    MessageBox.Show(name.Name.ToString())
Next name

' There is already a named range called "One", I am changing the address
that it points to.
mySheet.Names("One").RefersToRange = mySheet.Range("B6")

' Named ranges are added to the workbook collection in both the methods
mentioned below.
' Adding the named Range to the workbook.
myWorkbook.Names.Add("TestRangeBook", mySheet.Range("A5"))

' Adding the named Range to the workbook. Internally named range is added to
the workbook names coll.
mySheet.Names.Add("TestRangeSheet", mySheet.Range("A5"))

' Referencing from the sheet.
mySheet.Range("TestRangeSheet").Number = 100
```

5.1.16 How to create and open Excel Template files by using XlsIO?

Creating Excel Template Files

You can create either XLT or XLTX Excel Template files by saving a file with the **ExcelSaveType** property of the **SaveAs** method. The **ExcelSaveType** property must be set to **SaveAsTemplate** to create a template file of the existing file. The following code example illustrates this.

[C#]

```
// Save as XLT.
workbook.Version = ExcelVersion.Excel97to2003;
```

```

workbook.SaveAs("Sample.xlt", ExcelSaveType.SaveAsTemplate);

// Save as XLTX.
workbook.Version = ExcelVersion.Excel2007;
workbook.SaveAs("Sample.xltx", ExcelSaveType.SaveAsTemplate);

```

[VB.NET]

```

' Save as XLT.
workbook.Version = ExcelVersion.Excel97to2003
workbook.SaveAs("Sample.xlt",ExcelSaveType.SaveAsTemplate)

' Save as XLTX.
workbook.Version = ExcelVersion.Excel2007
workbook.SaveAs("Sample.xltx",ExcelSaveType.SaveAsTemplate)

```

Opening Excel Template Files

An Excel Template file is opened in the same way a document is opened. The following code example illustrates how to open a template file.

[C#]

```
workbook = application.Workbooks.Open(fileName, ExcelOpenType.Automatic);
```

[VB.NET]

```
workbook = application.Workbooks.Open(fileName, ExcelOpenType.Automatic)
```

5.1.17 How to open an Excel 2007 Macro Enabled Template?

XlsIO now provides support to open and save an Excel 2007 Macro Enabled Template to XLSM (Excel 2007 Macro Enabled Document) format. The following code example illustrates this.

[C#]

```

// Open an existing XLTM file.
workbook = application.Workbooks.Open(@"Template.xltm",
ExcelOpenType.Automatic);

// Save the file as XLSM.
workbook.Version = ExcelVersion.Excel2007;

```

```
workbook.SaveAs("Sample.xlsxm", ExcelSaveType.SaveAsTemplate);
```

[VB.NET]

```
' Open an existing XLTM file.  
workbook = application.Workbooks.Open("MacroTemplate.xltm",  
ExcelOpenType.Automatic)  
  
' Save the file as XLSM.  
workbook.Version = ExcelVersion.Excel2007  
workbook.SaveAs("Sample.xlsxm", ExcelSaveType.SaveAsTemplate)
```

5.1.18 How to Create Template Markers Using XlsIO?

Report created in Excel provides ordered and rich look for large datasets. This article focuses on creating an Excel report using template markers. A template marker is a special marker symbol created in an Excel template which will be bound the required user data. Essential XlsIO allows you to create and bind the template markers to data from various sources, such as data table, variables and arrays. This allows the user to control the data formats for the data bound to the template document.

How Does it Work?

Markers are applied in the template to the required cells. This will include the data source name and field name of interest. During data binding a search is conducted for the data source name and the field name in the Excel workbook and the corresponding data from the data source is bound to the marker. Cells in the worksheet can be filled with single data source or with multiple records. Format of these data can be changed using the arguments of the markers.

What is the Syntax of the Markers in Template?

Each marker starts with some prefix (by default it is “%” character) and followed by the variable name and properties. There could be several arguments after a variable which are delimited by some character (by default it is semicolon “;”.)

Company Name	Contact Name
%Customers.CompanyName;insert:copyStyles	%Customers.ContactName

Figure 164: Marker Syntax

What are the Various Sources of Binding Data to Markers?

Essential XlsIO allows data binding from following data sources.

1. Data Source

This includes data tables, datasets, data readers and data views. A data source can be used to bind large number records to the template document. This will add the rows for each record and fields to be bound are identified through the field name in the template.

Syntax: %DataSource.FieldName

2. Variable Name

This option will allow you to bind a single data stored in a variable to the marker in the template.

Syntax: %VariableName

3. Variable Array

This option will allow you to bind array of data stored in an array to the marker in the template.

Syntax: %VariableArray

4. Formulas

This option will allow you to create formulas for each row when multiple records comprising formula in the cells are bound to the marker. If a cell contains formula, by default it will be stretched to the rows/columns for any of the above sources of data binding.

Product ID	Quantity	Unit Price	Price
%NumbersTable.Column0	%NumbersTable.Column1	bersTable.Column2;copyran	=B5+C5

Figure 165: Formulas

What are the Various Arguments for the Marker?

The following arguments can be used with the marker to control the formatting while binding the data:

- **Horizontal**-This argument specifies the horizontal direction of the data import for complex variables.
- **Vertical**-This argument specifies the vertical direction of the data import for complex variables.
- **Insert**-This argument inserts new row or column depending on the direction argument for each new cell. By default, the rows can't be added.
- **Insert:copystyles**-This argument copies style from the row above or left column.

- **jump:[cell reference in R1C1 notation]**-This argument binds the data to the cell at the specified reference. Cell reference address can be relative or absolute.
- **copyrange:[top-left cell reference in R1C1]:[bottom-right cell reference in R1C1]**-Copies the specified cells after each cell import.

Following code sample illustrates processing and binding the marker with data.

```
// Create marker processor
ITemplateMarkersProcessor marker = workbook.CreateTemplateMarkersProcessor();

// Bind the data from the data table
marker.AddVariable("Customers", northwindDt);

// Process the markers in the template
marker.ApplyMarkers();
```

Here, **CreateTemplateMarkerProcessor** returns **ITemplateMarkersProcessor** interface which creates and manipulates the marker data. **ApplyMarkers** method of **ITemplateMarkersProcessor** is the special method that processes the markers in the template.

Here is a screen shot after binding data with marker variable.

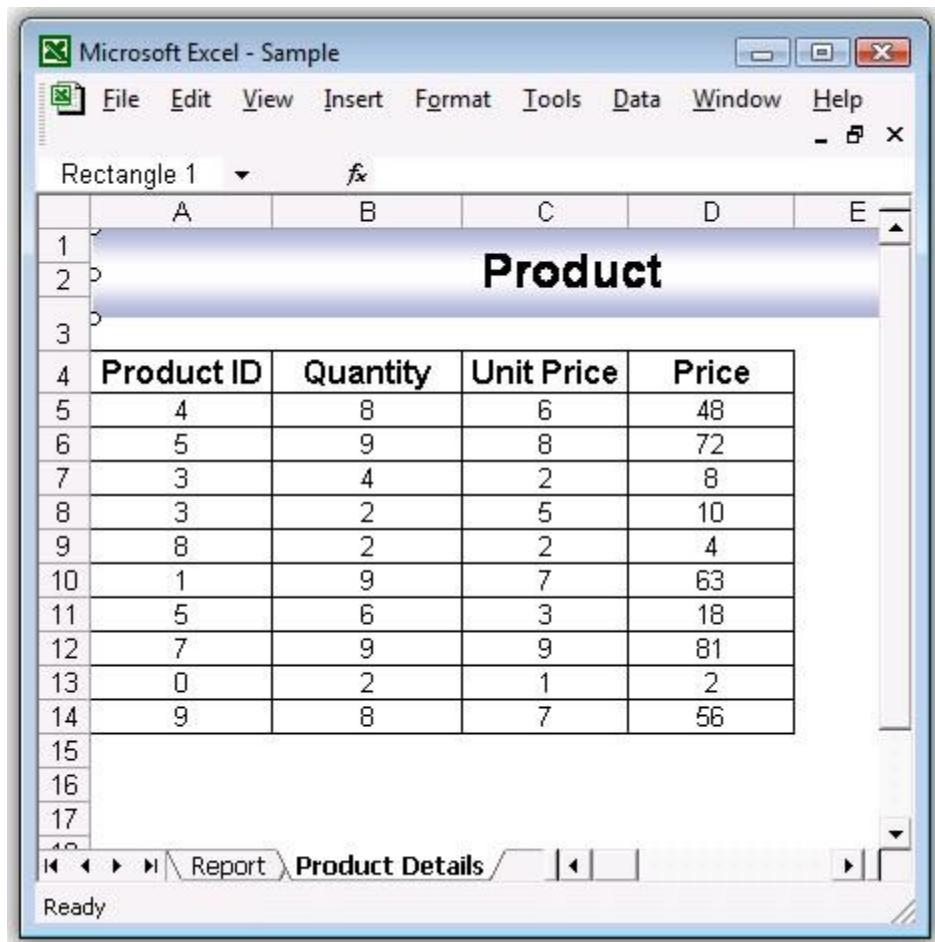


Figure 166: Marker Syntax

Here is a screen shot after binding the data with the marker that retains the formula.

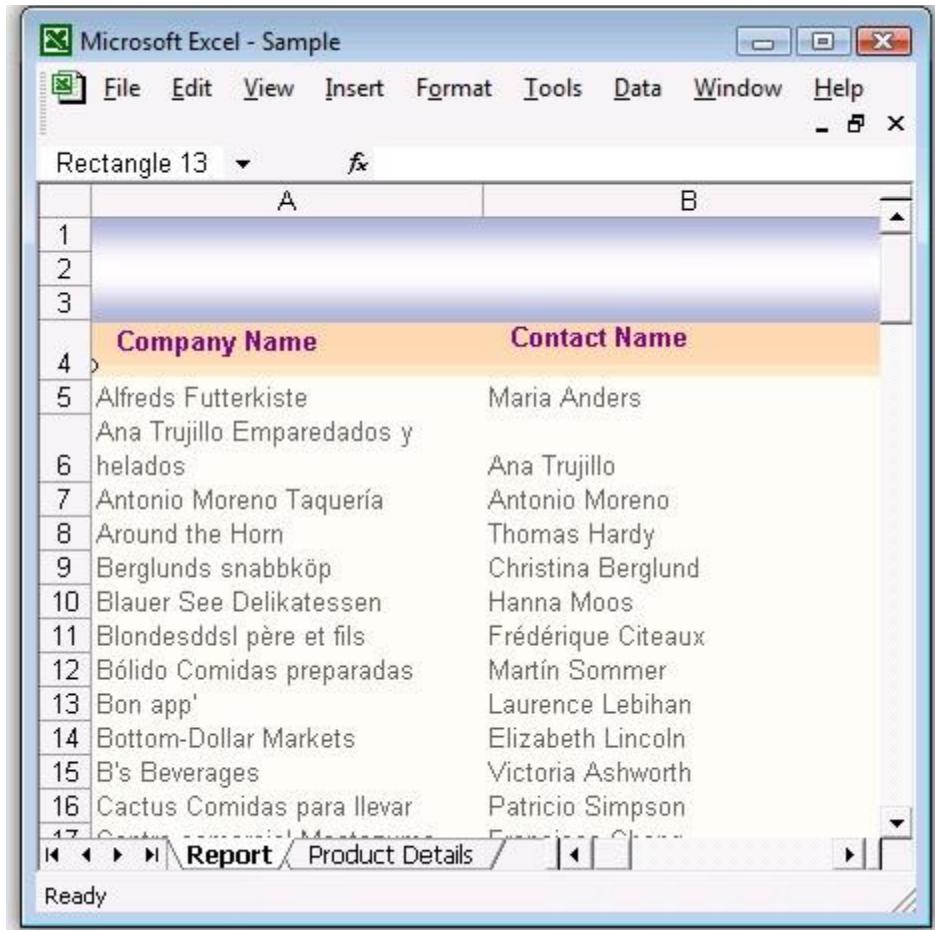


Figure 167: Marker Syntax

Summary

This article demonstrates about how Essential XlsIO can be used to generate rich reports in Excel format using template markers. You can create and format user specific information in a large report with few lines of coding and great performance.

5.1.19 How to add chart labels to scatter points?

The following code illustrates adding chart labels to the scatter points of the chart.

```
[C#]

//Get the chart from the charts collection
IChart chart = sheet.Charts[0];
```

```
//Get the first series from the Series collection
IChartSerie serieOne = chart.Series[0];

//Set the Series name to the Data Labels through Data Points
serieOne.DataPoints[0].DataLabels.IsSeriesName = true;

//Set the Value to the Data Labels through Data Points
serieOne.DataPoints[0].DataLabels.IsValue = true;
```

[VB]

```
'Get the chart from the charts collection
Dim chart As IChart = sheet.Charts(0)

'Get the first series from the Series collection
Dim serieOne As IChartSerie = chart.Series(0)

'Set the Series name to the Data Labels through Data Points
serieOne.DataPoints(0).DataLabels.IsSeriesName = True

'Set the Value to the Data Labels through Data Points
serieOne.DataPoints(0).DataLabels.IsValue = True
```

5.2 Advanced

This section contains various questions and answers for the Advanced-level users.

5.2.1 How to create a Chart with a discontinuous range?

The following code example illustrates creating a chart with discontiguous data ranges.

[C#]

```

// Entering the data for the chart.
sheet.Range["A1"].Text = "Texas books Unit sales";
sheet.Range["A1:D1"].Merge();
sheet.Range["A1"].CellStyle.Font.Bold = true;

sheet.Range["B2"].Text = "Jan";
sheet.Range["C2"].Text = "Feb";
sheet.Range["D2"].Text = "Mar";

sheet.Range["A3"].Text = "Austin";
sheet.Range["A4"].Text = "Dallas";
sheet.Range["A5"].Text = "Houston";
sheet.Range["A6"].Text = "San Antonio";

sheet.Range["B3"].Number = 53.75;
sheet.Range["B4"].Number = 52.85;
sheet.Range["B5"].Number = 59.77;
sheet.Range["B6"].Number = 96.15;

sheet.Range["C3"].Number = 79.79;
sheet.Range["C4"].Number = 59.22;
sheet.Range["C5"].Number = 10.09;
sheet.Range["C6"].Number = 73.02;

sheet.Range["D3"].Number = 26.72;
sheet.Range["D4"].Number = 33.71;
sheet.Range["D5"].Number = 45.81;
sheet.Range["D6"].Number = 12.17;

sheet.Range["F1"].Number = 26.72;
sheet.Range["F2"].Number = 33.71;

sheet.Range["F3"].Number = 45.81;
sheet.Range["F4"].Number = 12.17;

// Discontiguous range.
IRanges rangesOne = sheet.CreateRangesCollection();
rangesOne.Add(sheet.Range["B3:B6"]);
rangesOne.Add(sheet.Range["F1:F2"]);

IRanges rangesTwo = sheet.CreateRangesCollection();
rangesTwo.Add(sheet.Range["D3:D6"]);
rangesTwo.Add(sheet.Range["F3:F4"]);

// Adding a New (Embedded chart)to the Worksheet.
IChartShape shape = sheet.Charts.Add();

```

```

shape.PrimaryCategoryAxis.Title = "City";
shape.PrimaryValueAxis.Title = "Sales (in Dollars)";
shape.ChartTitle = "Texas Books Unit Sales";

// Setting the Series Names in a Legend.
IChartSerie serieOne = shape.Series.Add();
serieOne.Name = "Jan";
serieOne.Values = rangesOne;

IChartSerie serietwo = shape.Series.Add();
serietwo.Name = "March";
serietwo.Values = rangesTwo;

// Setting the (Rows & Columns) Property for the Embedded chart.
shape.BottomRow = 40;
shape.TopRow = 10;
shape.LeftColumn = 3;
shape.RightColumn = 15;

```

[VB .NET]

```

' Entering the data for the chart.
sheet.Range("A1").Text = "Texas books Unit sales"
sheet.Range("A1:D1").Merge()
sheet.Range("A1").CellStyle.Font.Bold = True

sheet.Range("B2").Text = "Jan"
sheet.Range("C2").Text = "Feb"
sheet.Range("D2").Text = "Mar"

sheet.Range("A3").Text = "Austin"
sheet.Range("A4").Text = "Dallas"
sheet.Range("A5").Text = "Houston"
sheet.Range("A6").Text = "San Antonio"

sheet.Range("B3").Number = 53.75
sheet.Range("B4").Number = 52.85
sheet.Range("B5").Number = 59.77
sheet.Range("B6").Number = 96.15

sheet.Range("C3").Number = 79.79
sheet.Range("C4").Number = 59.22
sheet.Range("C5").Number = 10.09
sheet.Range("C6").Number = 73.02

```

```

sheet.Range("D3").Number = 26.72
sheet.Range("D4").Number = 33.71
sheet.Range("D5").Number = 45.81
sheet.Range("D6").Number = 12.17

sheet.Range("F1").Number = 26.72
sheet.Range("F2").Number = 33.71

sheet.Range("F3").Number = 45.81
sheet.Range("F4").Number = 12.17

' Discontiguous range.
Dim rangesOne As Syncfusion.XlsIO.IRanges = sheet.CreateRangesCollection()
rangesOne.Add(sheet.Range("B3:B6"))
rangesOne.Add(sheet.Range("F1:F2"))

Dim rangesTwo As Syncfusion.XlsIO.IRanges = sheet.CreateRangesCollection()
rangesTwo.Add(sheet.Range("D3:D6"))
rangesTwo.Add(sheet.Range("F3:F4"))

' Adding a New(Embedded chart)to the Worksheet.
Dim shape As Syncfusion.XlsIO.IChartShape = sheet.Charts.Add()
shape.PrimaryCategoryAxis.Title = "City"
shape.PrimaryValueAxis.Title = "Sales (in Dollars)"
shape.ChartTitle = "Texas Books Unit Sales"

' Setting the Series Names in a Legend.
Dim serieOne As Syncfusion.XlsIO.IChartSerie = shape.Series.Add()
serieOne.Name = "Jan"
serieOne.Values = rangesOne

Dim serietwo As Syncfusion.XlsIO.IChartSerie = shape.Series.Add()
serietwo.Name = "March"
serietwo.Values = rangesTwo

' Setting the (Rows & Columns)Property for the Embedded chart.
shape.BottomRow = 40
shape.TopRow = 10
shape.LeftColumn = 3
shape.RightColumn = 15

```

5.2.2 How to define discontinuous ranges?

You can set a discontinuous range by adding different ranges to the Range collection. The following code example illustrates this.

[C#]

```
// Create Range collection.
IRanges rangesOne = sheet.CreateRangesCollection();

// Add different ranges to the Range collection.
rangesOne.Add(sheet.Range["D2:D3"]);
rangesOne.Add(sheet.Range["D10:D11"]);
```

[VB .NET]

```
' Create Range collection.
Dim rangesOne As Syncfusion.XlsIO.IRanges = sheet.CreateRangesCollection()

' Add different ranges to the Range collection.
rangesOne.Add(sheet.Range("D2:D3"));
rangesOne.Add(sheet.Range("D10:D11"));
```

5.2.3 How to format text within a cell?

The text within a cell can be formatted by using the RichText functionality of XlsIO. The following code example illustrates this.

[C#]

```
// Insert Rich Text.
IRange range = sheet.Range["A1"];
range.Text = "RichText";
IRichTextString rtf = range.RichText;

// Formatting first 4 characters.
IFont redFont = workbook.CreateFont();
redFont.Bold = true;
redFont.Italic = true;
redFont.RGBColor = Color.Red;
rtfSetFont(0, 3, redFont);

// Formatting last 4 characters.
IFont blueFont = workbook.CreateFont();
```

```
blueFont.Bold = true;
blueFont.Italic = true;
blueFont.RGBColor = Color.Blue;
rtf.SetFont(4, 7, blueFont);
```

[VB.NET]

```
' Insert Rich Text.
Dim range As Syncfusion.XlsIO.IRange = sheet.Range("A1")
range.Text = "RichText"
Dim rtf As Syncfusion.XlsIO.IRichTextString = range.RichText

' Formatting first 4 characters.
Dim redFont As Syncfusion.XlsIO.IFont = workbook.CreateFont()
redFont.Bold = True
redFont.Italic = True
redFont.RGBColor = Color.Red
rtfSetFont(0, 3, redFont)

' Formatting last 4 characters.
Dim blueFont As Syncfusion.XlsIO.IFont = workbook.CreateFont()
blueFont.Bold = True
blueFont.Italic = True
blueFont.RGBColor = Color.Blue
rtfSetFont(4, 7, blueFont)
```

5.2.4 How to suppress the summary rows and columns using XlsIO?

You can suppress the summary rows and columns by using the IsSummaryRowBelow and IsSummaryColumnRight properties. The following code example illustrates this.

[C#]

```
// Suppress the summary rows at the bottom.
sheet.PageSetup.IsSummaryRowBelow = false;
// Suppress the summary columns to the right.
sheet.PageSetup.IsSummaryColumnRight = false;
```

[VB.NET]

```
' Suppress the summary rows at the bottom.
sheet.PageSetup.IsSummaryRowBelow = False
' Suppress the summary columns to the right.
sheet.PageSetup.IsSummaryColumnRight = False
```

5.2.5 How to zip files using the Syncfusion.Compression.Zip namespace?

You can use the AddFile method of ZipArchive object to compress files by using XlsIO. Following code example illustrates how to use this method.

[C#]

```
Syncfusion.Compression.Zip.ZipArchive zipArchive = new
Syncfusion.Compression.Zip.ZipArchive();
zipArchive.DefaultCompressionLevel =
Syncfusion.Compression.CompressionLevel.Best;

// Add the file you want to zip.
zipArchive.AddFile("../Form1.cs");

// Zip file name and location.
zipArchive.Save("SyncfusionCompressFileSample.zip");
zipArchive.Close();
```

[VB .NET]

```
Dim zipArchive As New Syncfusion.Compression.Zip.ZipArchive()
zipArchive.DefaultCompressionLevel =
Syncfusion.Compression.CompressionLevel.Best

' Add the file you want to zip.
zipArchive.AddFile("../Form1.cs")

' Zip file name and location.
zipArchive.Save("SyncfusionCompressFileSample.zip")
zipArchive.Close()
```

For compressing directories, you can make use of the **AddDirectory** method. The AddDirectory method adds an empty directory file to a ZipArchive. If you want to add all the files inside the directory, then you should manually add these files by using the **AddItem** method.

For example, you can use the following code to add the file from the local drive.

[C#]

```
string fileName = @"C:\Form1.cs";
Syncfusion.Compression.Zip.ZipArchive zipArchive = new
Syncfusion.Compression.Zip.ZipArchive();
zipArchive.DefaultCompressionLevel =
Syncfusion.Compression.CompressionLevel.Best;
Stream stream = new FileStream(fileName, FileMode.Open, FileAccess.Read);
FileAttributes attributes = File.GetAttributes(fileName);
Syncfusion.Compression.Zip.ZipArchiveItem item = new
Syncfusion.Compression.Zip.ZipArchiveItem("Form1.cs", stream, true,
attributes);
zipArchive.AddItem(item);
zipArchive.Save(@"c:\\SyncfusionCompressFileSample.zip");
zipArchive.Close();
```

[VB .NET]

```
Dim fileName As String = "C:\Form1.cs"
Dim zipArchive As New Syncfusion.Compression.Zip.ZipArchive()
zipArchive.DefaultCompressionLevel =
Syncfusion.Compression.CompressionLevel.Best
Dim stream As IO.Stream = New IO.FileStream(fileName, FileMode.Open,
FileAccess.Read)
Dim attributes As IO.FileAttributes = File.GetAttributes(fileName)
Dim item As New Syncfusion.Compression.Zip.ZipArchiveItem("Form1.cs",
stream, True, attributes)
zipArchive.AddItem(item)
zipArchive.Save("c:\\SyncfusionCompressFileSample.zip")
zipArchive.Close()
```

5.2.6 How to zip all the files in subfolders using the Syncfusion.Compression.Zip namespace?

The following code example illustrates how to zip all files in subfolders in XlsIO by using the Syncfusion.Compression.Zip namespace.

[C#]

```

void SubFoldersFiles(string path)
{
    DirectoryInfo dInfo = new DirectoryInfo(path);
    foreach (DirectoryInfo d in dInfo.GetDirectories())
    {
        SubFoldersFiles(d.FullName);
        arr.Add(d);
    }
}

// Zip and save the file.
private void button1_Click(object sender, EventArgs e)
{
    SubFoldersFiles(this.textBox1.Text);

    if (Directory.Exists(this.textBox1.Text))
    {
        AddRootFiles();

        AddSubFoldersFiles();

        // Saving zipped file.
        SaveFileDialog saveDialog = new SaveFileDialog();
        saveDialog.Filter = "Zip Files | *.zip";
        if (saveDialog.ShowDialog() == DialogResult.OK)
            zipArchive.Save(saveDialog.FileName);

        zipArchive.Close();
        label1.Text = "Files Zipped successfully!";
        button1.Enabled = false;
    }
}

private void AddRootFiles()
{
    foreach (string rootfiles in Directory.GetFiles(this.textBox1.Text))
    {
        stream = new FileStream(rootfiles, FileMode.Open, FileAccess.Read);
        att = File.GetAttributes(rootfiles);
        item = new ZipArchiveItem(rootfiles, stream, true, att);
        zipArchive.AddItem(item);
    }
}

```

```

private void AddSubFoldersFiles()
{
    foreach (DirectoryInfo dInfo in arr)
    {
        FileInfo[] fInfo = dInfo.GetFiles();
        foreach (FileInfo file in fInfo)
        {
            string str = file.FullName;
            stream = new FileStream(str, FileMode.Open, FileAccess.Read);
            FileAttributes att = File.GetAttributes(str);
            zipArchive.AddItem(new
Syncfusion.Compression.Zip.ZipArchiveItem(str, stream, true, att));
        }
    }
}

// Browse folder.
private void button2_Click(object sender, EventArgs e)
{
    // Select the folder to be zipped.
    FolderBrowserDialog fldrDialog = new FolderBrowserDialog();
    fldrDialog.Description = "Select the folder to zip. Note: All its
subfolders and its files will zip too.";
    if (fldrDialog.ShowDialog() == DialogResult.OK)
    {
        this.textBox1.Text = fldrDialog.SelectedPath;
        this.button1.Enabled = true;
    }
}

// Close
private void button3_Click(object sender, EventArgs e)
{
    Close();
}

```

[VB.NET]

```

Private Sub SubFoldersFiles(ByVal path As String)
    Dim dInfo As New IO.DirectoryInfo(path)
    For Each d As IO.DirectoryInfo In dInfo.GetDirectories()
        SubFoldersFiles(d.FullName)
        arr.Add(d)
    Next
End Sub

```

```

' Zip and save the file.
Private Sub button1_Click(ByVal sender As Object, ByVal e As EventArgs)
    SubFoldersFiles(Me.textBox1.Text)

    If Directory.Exists(Me.textBox1.Text) Then
        AddRootFiles()

        AddSubFoldersFiles()

        ' Saving zipped file.
        Dim saveDialog As New SaveFileDialog()
        saveDialog.Filter = "Zip Files | *.zip"
        If saveDialog.ShowDialog() = Windows.Forms.DialogResult.OK Then
            zipArchive.Save(saveDialog.FileName)
        End If

        zipArchive.Close()
        label1.Text = "Files Zipped successfully!"
        button1.Enabled = False
    End If
End Sub

Private Sub AddRootFiles()

    For Each rootfiles As String In Directory.GetFiles(Me.textBox1.Text)
        stream = New FileStream(rootfiles, FileMode.Open,
        FileAccess.Read)
        att = File.GetAttributes(rootfiles)
        item = New ZipArchiveItem(rootfiles, stream, True, att)
        zipArchive.AddItem(item)
    Next
End Sub

Private Sub AddSubFoldersFiles()
    For Each dInfo As IO.DirectoryInfo In arr
        Dim fInfo As FileInfo() = dInfo.GetFiles()
        For Each file As FileInfo In fInfo
            Dim str As String = file.FullName
            stream = New FileStream(str, FileMode.Open, FileAccess.Read)
            Dim att As FileAttributes = file.GetAttributes(str)
            zipArchive.AddItem(New
Syncfusion.Compression.Zip.ZipArchiveItem(str, stream, True, att))
        Next
    Next
End Sub

```

```

' Browse folder.
Private Sub button2_Click(ByVal sender As Object, ByVal e As EventArgs)
    ' Select the folder to be zipped.
    Dim fldrDialog As New FolderBrowserDialog()
    fldrDialog.Description = "Select the folder to zip. Note: All its
subfolders and its files will zip too."
    If fldrDialog.ShowDialog() = Windows.Forms.DialogResult.OK Then
        Me.textBox1.Text = fldrDialog.SelectedPath
        Me.button1.Enabled = True
    End If
End Sub

' Close
Private Sub button3_Click(ByVal sender As Object, ByVal e As EventArgs)
    Close()
End Sub

```

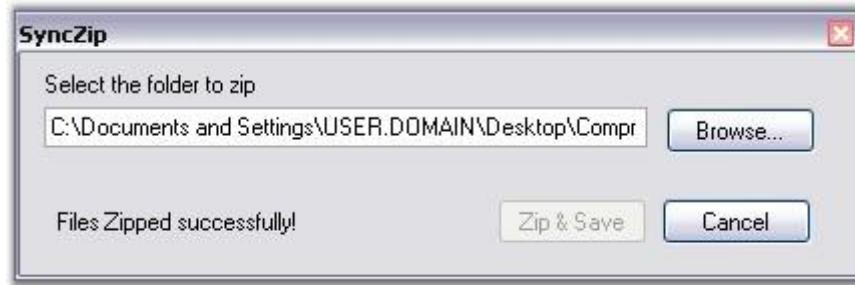


Figure 168: SyncZip dialog box to zip the files

5.2.7 Does Essential XlsIO provide support for Client profile?

Yes, Essential XlsIO provides support for Client profile. In order to use Essential XlsIO in an application (which targeted to Client profile), the user should include the following assemblies

- Syncfusion.Core.dll
- Syncfusion.Compression.Base.dll
- Syncfusion.XlsIO.ClientProfile.dll

Index

3

3-D Chart Wall Settings 229

A

Adding Calculation Engine to an Application 338

Add-Ins 420

Advanced 439

Advantages of Using Excel Reports 12

Alignment Settings 114

Array Formula 333

ASP.NET 38

ASP.NET MVC 54

Attributes 373

B

Background 314

Binding Business Objects to Template Markers
373

Binding Data Objects to Template Markers 372

Border Settings 128

Breaks 311

C

Calculation 335

Calculation Engine 338

Calculation Options 335

Cell Styles 137

Cell Visibility 189

Cells 176

Changes 395

Changing Cell Size 181

Chart Worksheet 218

Charts 210

Check Box 289

Class Diagram 27

Clipboard 206

Combo Box 291

Comments 391

Common 423

Concepts and Features 110

Conditional Formatting 142

Creating a Platform Application 28

CSV Format 96

D

Data 350

Data Sorting for a Given Range of Cells in the
Worksheet 357

Data Validation 352

Defined Names 342

Delete 178

Deploying Essential XlsIO 33

Deployment Requirements 25

Document Properties 412

Documentation 10

Does Essential XlsIO provide support for Client
profile? 450

E

Edit Charts 221

Edit Range 402

Editing 161

Embedded Chart 210

Encryption and Decryption 414

Encryption and Decryption for Excel 2010 419

Excel 2003 258

Excel 2007 259

Excel Engine 65

Excel97to2003 83

External Formula 334

F

Feature Summary 60, 372

Fill Settings 132	How to open an Excel 2007 Macro Enabled Template? 433
Filter 350	How to open an Excel file from Stream? 426
Filtering Chart Series and Categories 235	How to open an existing Xlsx workbook and save it as Xlsx? 427
Find and Replace 167	How to protect certain cells in a spreadsheet? 427
Font Settings 110	How to save a file to stream? 428
Form Controls 288	How to set a line break inside a cell? 429
Formats 181	How to set options to print Titles? 429
Formatting 110	How to set or format a Header/Footer? 429
Formula Auditing 347	How to suppress the summary rows and columns using XlsIO? 444
Formulas 321	How to unfreeze the rows and columns in XlsIO? 430
FreezePane 402	How to use Named Ranges with XlsIO? 431
Frequently Asked Questions 423	How to zip all the files in subfolders using the Syncfusion.Compression.Zip namespace? 446
Function Library 321	How to zip files using the Syncfusion.Compression.Zip namespace? 445
G	I
Getting Started 27	Illustrations 207
H	Import/Export 363
Header/Footer 251	Improving Performance 100
Home 110	Insert 176, 207
How to add chart labels to scatter points? 438	Installation 14
How to change the grid line color of the Excel sheet? 423	Installation and Deployment 14
How to copy a range from one workbook to another? 424	Introduction to Essential XlsIO 7
How to copy and paste the values of the cells that contain only formulas? 423	L
How to create a Chart with a discontinuous range? 439	Links 248
How to create and open Excel Template files by using XlsIO? 432	M
How to Create Template Markers Using XlsIO? 434	Macros 410
How to define discontinuous ranges? 442	Margins 306
How to format text within a cell? 443	N
How to ignore the green error marker in worksheets? 425	Number Formatting 119
How to merge several Excel files to a single file? 426	

O	Sorting Data by Cell Values 359
OLE Objects 295	Sparklines 243
Option Button 293	Split Pane 404
Orientation 308	Spreadsheet 65
Outlines 386	SpreadsheetML 93
Overview 7	Styles 136
P	Supported Elements 102
Page Layout 305	T
Page Setup 306	Tables 256
Paper Size 310	Template Markers 378
Pivot Tables 258	Text Box 288
PivotCharts 282	U
Prepare 412	Use Case Scenario 372
Prerequisites and Compatibility 9	Using Templates 97
Print settings 318	V
Problems of Using MS Excel to Generate Reports 12	View 402
Properties, Methods and Events tables 357	W
Protection 204	Web Application Deployment 59
R	What is the maximum range of Rows and Columns? 431
Range Manipulation 162	Window 402
Reducing size of Excel 2007 & Excel 2010 files 92	Windows 33
Resizing and Positioning of Chart Elements 224	Windows, ASP.NET, WPF, ASP.NET MVC 102
Review 391	Workbook 69
Rich-Text Formatting for Chart Elements 226	Workbook Protection 395
S	Worksheet 74
Samples Location 14	Worksheet Protection 399
Saving a Workbook 82	WPF 42
Scale To Fit 316	X
Sheet Options 317	XLSX 84
Sheet Organization 196	Z
Show/Hide Worksheet Elements 407	Zooming 406
Silverlight 46, 106	
Sorting by Cell Color 362	
Sorting by Font Color 360	

