Syncfusion
Deliver innovation with ease®

Essential Studio 2013 Volume 4 - v.11.4.0.26

Essential Calculate

Contents

1 Overview

This section covers information on Essential Calculate, its key features, prerequisites to use the control, its compatibility with various OS and browsers, and finally the documentation details complimentary with the product. It comprises the following subsections:

1.1 Introduction to Essential Calculate

Essential Calculate is a 100% Native .NET library that provides calculation functionality by using the Microsoft .NET framework, so that it can be used in any .NET environment, including C#, VB.NET and managed C++. Essential Calculate is a UI independent class library that lets you add formula calculation support to Window Forms, Web Forms and WPF applications. Essential Calculate does not depend on Microsoft Excel and thus allows you to perform calculations independent of Excel.

The range of calculations include simple algebraic expressions such as (1.2^3-1)/8, to formulas using intrinsic functions like 4 * sqrt(exp(8.4)), to formulas relying on variables that are defined through controls on a form such as cos([textBox1] * pi()/180), to spreadsheet-like formulas such as Sum(A2:B14). Essential Calculate lets you parse and compute such expressions, and includes a library of more than 150 functions. This function library is easily extendable. The data used in the calculations can be from any source, ranging from fixed values to values that are entered through the controls, to data tables and Excel spreadsheets.

Essential Calculate allows you to add extensive calculation support to your business objects. It enables to easily setup forms that have calculation dependencies among various controls.

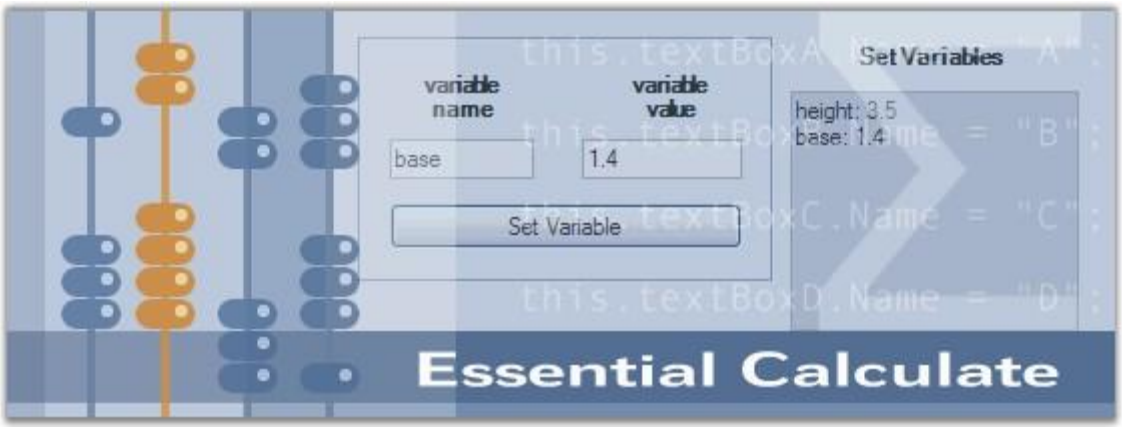


Figure 1: Essential Calculate

Key Features

Important features of Essential Calculate are listed below.

- Extensive calculation support can be added to your own business objects in both Windows Forms and ASP.NET.
- Easily set up forms that have calculation dependencies among various controls.
- Essential Calculate comes with a function library of more than 150 entries and supports cross sheet references.
- It can be used in conjunction with Essential XlsIO, to fully load, manipulate and compute Excel spreadsheets without depending on Excel.
- Essential Calculate does not depend upon Microsoft Excel and thus enables you to perform calculations independent of Excel.

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 Essential Calculate. 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 various platform applications and deploying Essential Calculate into those applications.
- **Concepts and Features**-The features of Essential Calculate are illustrated with use case scenarios, code examples and screen shots under this section.
- **Frequently Asked Questions**-This section illustrates the solutions for various task-based queries about Essential Calculate.

Document Conventions

The following conventions will help you to quickly identify the important sections of information while using the content.

Table 1: Document Conventions

Convention	Icon	Description
Note	 *Note:*	Represents important information.
Example	Example	Represents an example.
Tip		Represents useful hints that will help you in using the controls/features.
Additional Information		Represents additional information

		on the topic.

1.2 Prerequisites and Compatibility

This section covers the requirements mandatory for using Essential Calculate. It also lists operating systems and browsers, compatible with the product.

Prerequisites

The prerequisites details are listed below:

Table 2: Prerequisites

Development Environments	• Visual Studio 2012 (Ultimate, Premium, Professional and Express) • Visual Studio 2010 (Ultimate, Premium, Professional and Express) • Visual Studio 2008 (Team System, Professional, Standard & Express) • Visual Studio 2005 (Professional, Standard & Express)
.NET Framework versions	• .NET 4.5 • .NET 4.0 • .NET 3.5 SP1 • .NET 2.0

Compatibility

The compatibility details are listed below:

Table 3: Compatibility

Operating Systems	• Windows 8 (32 bit and 64 bit) • Windows Server 2012 (32 bit and 64 bit) • Windows 7 (32 bit and 64 bit) • Windows Server 2008 (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 Calculate.

Table 4: Type of Documentation

Type of Documentation	Location
Readme	**Windows Forms**-[drive:]\Program Files\Syncfusion\Essential Studio\x.x.x.x\Infrastructure\Data\Release Notes\readme.htm **ASP.NET**-[drive:]\Program Files\Syncfusion\Essential Studio\x.x.x.x\Infrastructure\Data\asp release notes\readme.htm **WPF**-[drive:]\Program Files\Syncfusion\Essential Studio\x.x.x.x\Infrastructure\Data\WPF release notes\readme.htm
Release Notes	**Windows Forms**-[drive:]\Program Files\Syncfusion\Essential Studio\x.x.x.x\Infrastructure\Data\Release Notes\Release Notes.htm **ASP.NET**-[drive:]\Program Files\Syncfusion\Essential Studio\x.x.x.x\Infrastructure\Data\asp release notes\Release Notes.htm **WPF**-[drive:]\Program Files\Syncfusion\Essential Studio\x.x.x.x\Infrastructure\Data\WPF release notes\Release Notes.htm
User Guide (this document)	**Online** http://help.syncfusion.com/resources (Navigate to the Calculate User Guide.) ***Note: Click Download as PDF to access a PDF version.*** **Installed Documentation** Dashboard -> Documentation -> Installed Documentation.
Class Reference	**Online** http://help.syncfusion.com/resources (Navigate to the Reporting User Guide. Select *Calculate*, and then click the Class Reference link found in the upper right section of the page.)

	Installed Documentation Dashboard -> Documentation -> Installed Documentation.

2 Installation and Deployment

This section covers information on the install location, samples, licensing, patches update and updation of the recent version of Essential Studio. It comprises the following subsections:

2.1 Installation

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

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 and Installation

Where to Find Samples?

This section covers the location of the installed samples and describes the procedure to run the samples through the sample browser and online. It also lists the location of utilities, assemblies, and source code.

Sample Installation Location

The Windows Forms Calculate samples are installed in the following location, locally on the disk:

[Install Location]:\...\Syncfusion\Essential Studio\[Version Number]\Windows\Calculate.Windows\Samples\2.0

The Calculate Web samples are installed in the following location, locally on the disk:

[Install Location]:\...\Syncfusion\Essential Studio\[Version Number]\Web\Calculate.Web\Samples\3.5

Viewing Samples

To view the samples:

1. Click **Start → All Programs → Syncfusion → Essential Studio <version number> → Dashboard**.

Essential Studio UI Edition Samples are listed by default.

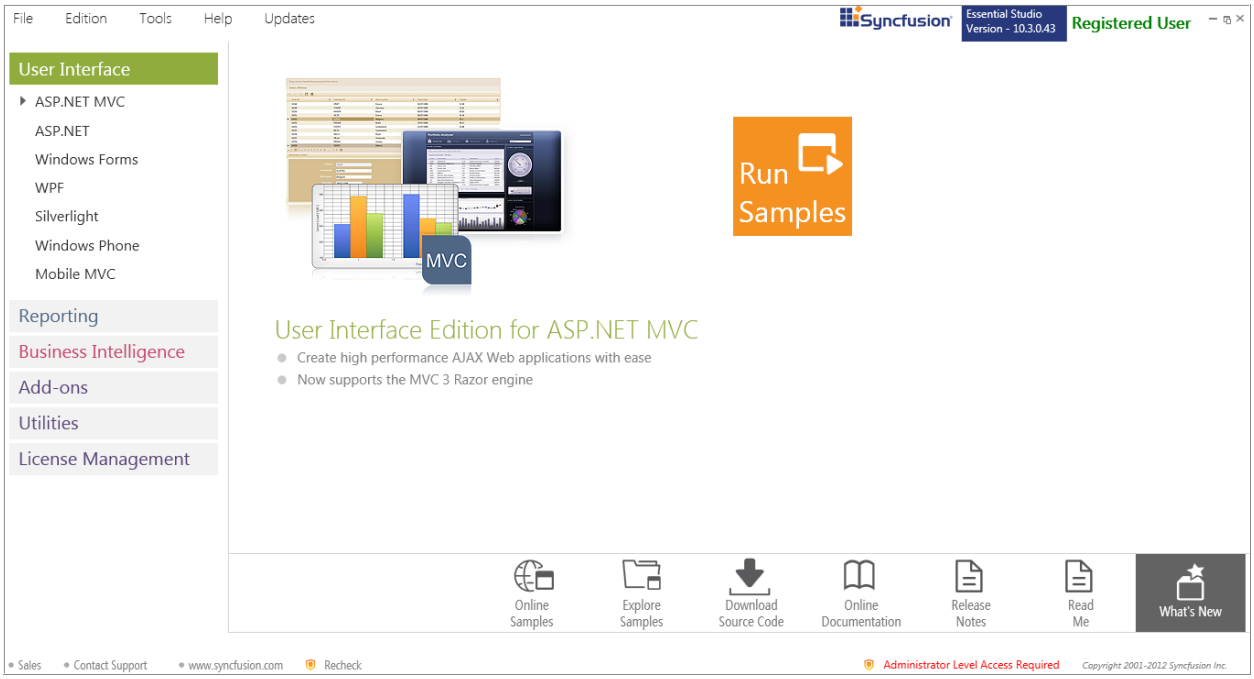


Figure 2: Syncfusion Essential Studio Dashboard

2. Select **Reporting** edition.

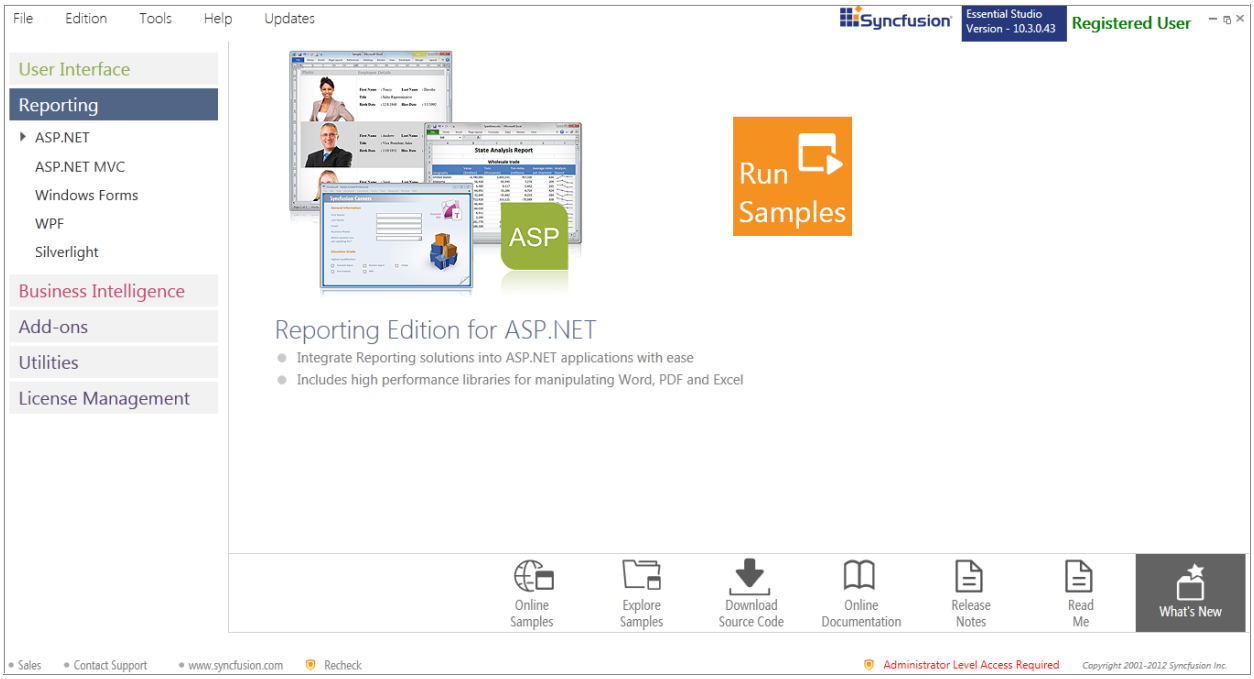


Figure 3: Syncfusion Essential Reporting Samples

3. The steps to view the **Calculate** samples in various platforms are discussed below:

Windows Forms

1. In the **Dashboard** window, click **Run Samples for Windows Forms** under **Reporting Edition** panel.

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.

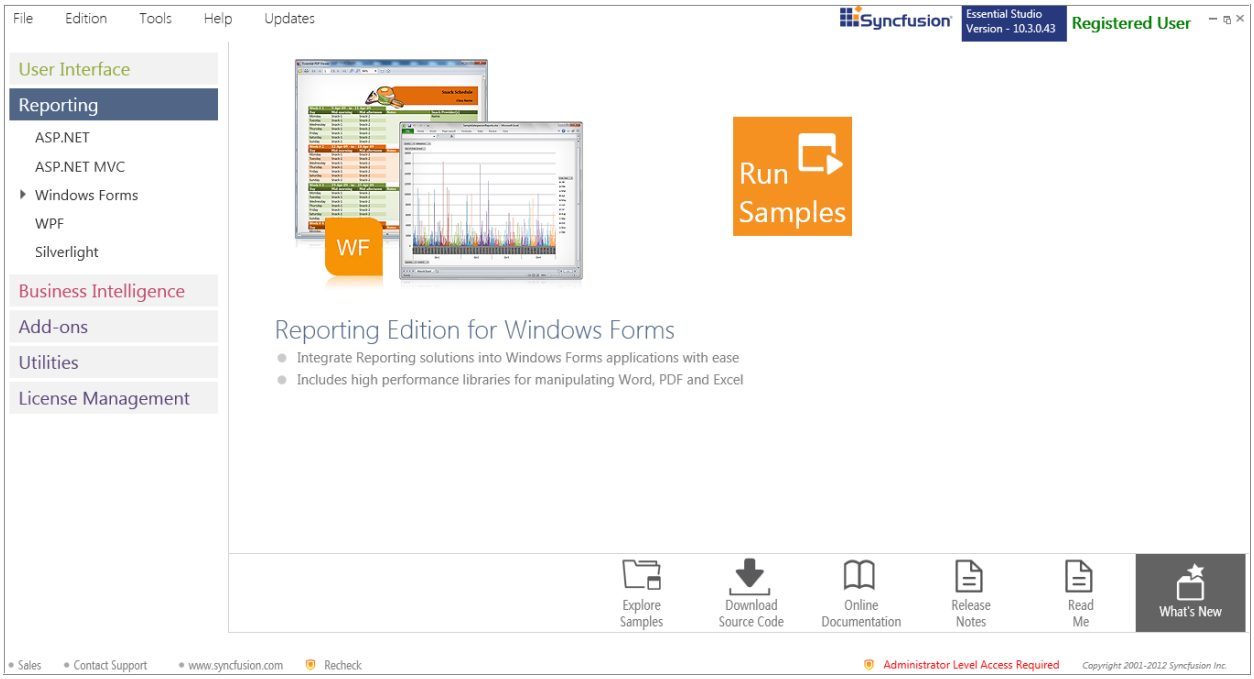


Figure 4: View Options Displayed for Samples

The **Windows Forms** Sample Browser window is displayed.

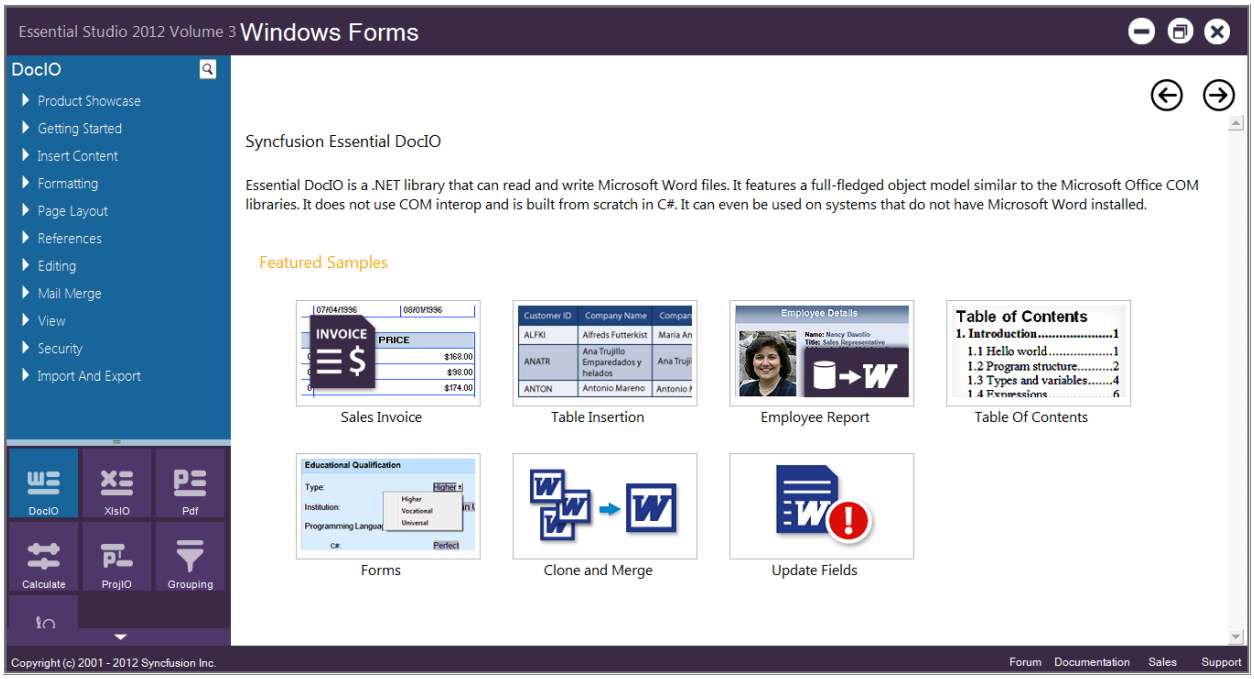


Figure 5: Calculate Samples displayed on the Contents Pane

2. Click **Calculate** in the **Contents** tab on the bottom-left pane. The **Calculate** samples are displayed.

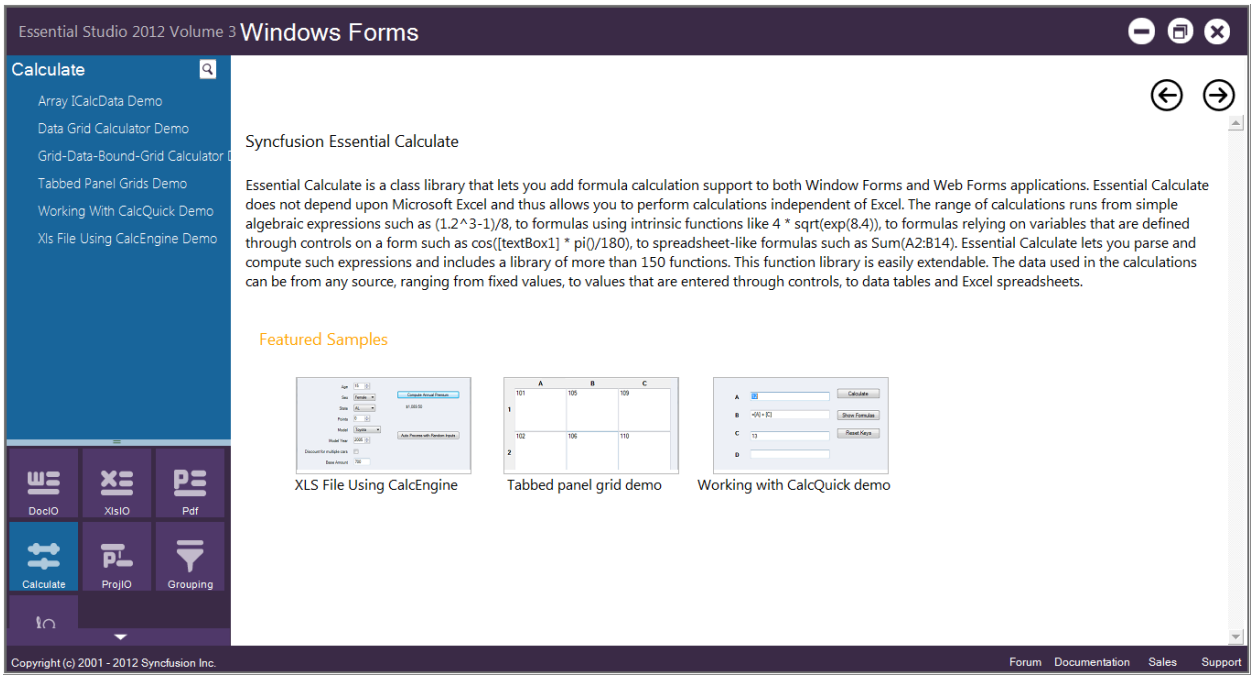


Figure 6: Calculate samples displayed in the Windows Forms Sample Browser

3. 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.

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

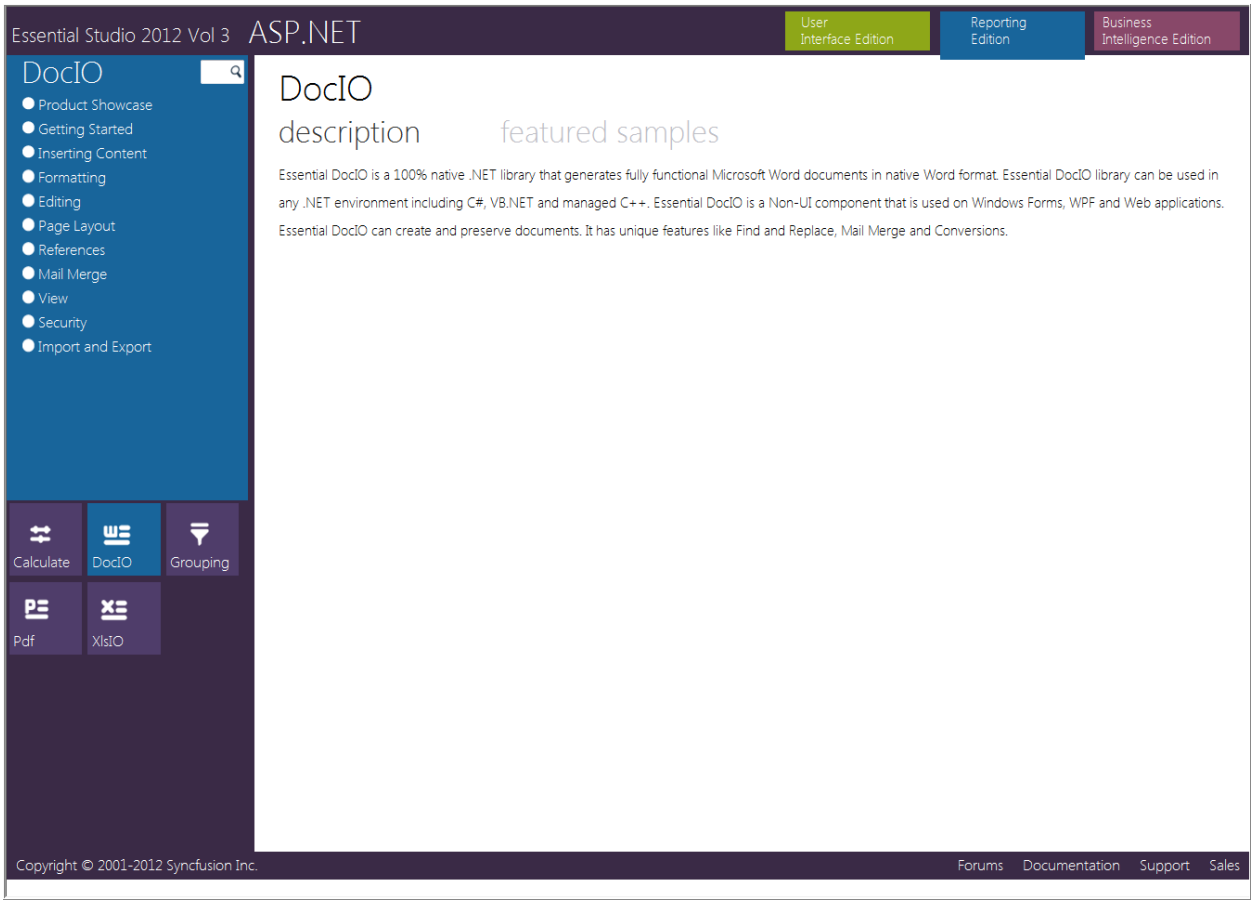


Figure 7: ASP.NET Sample Browser

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

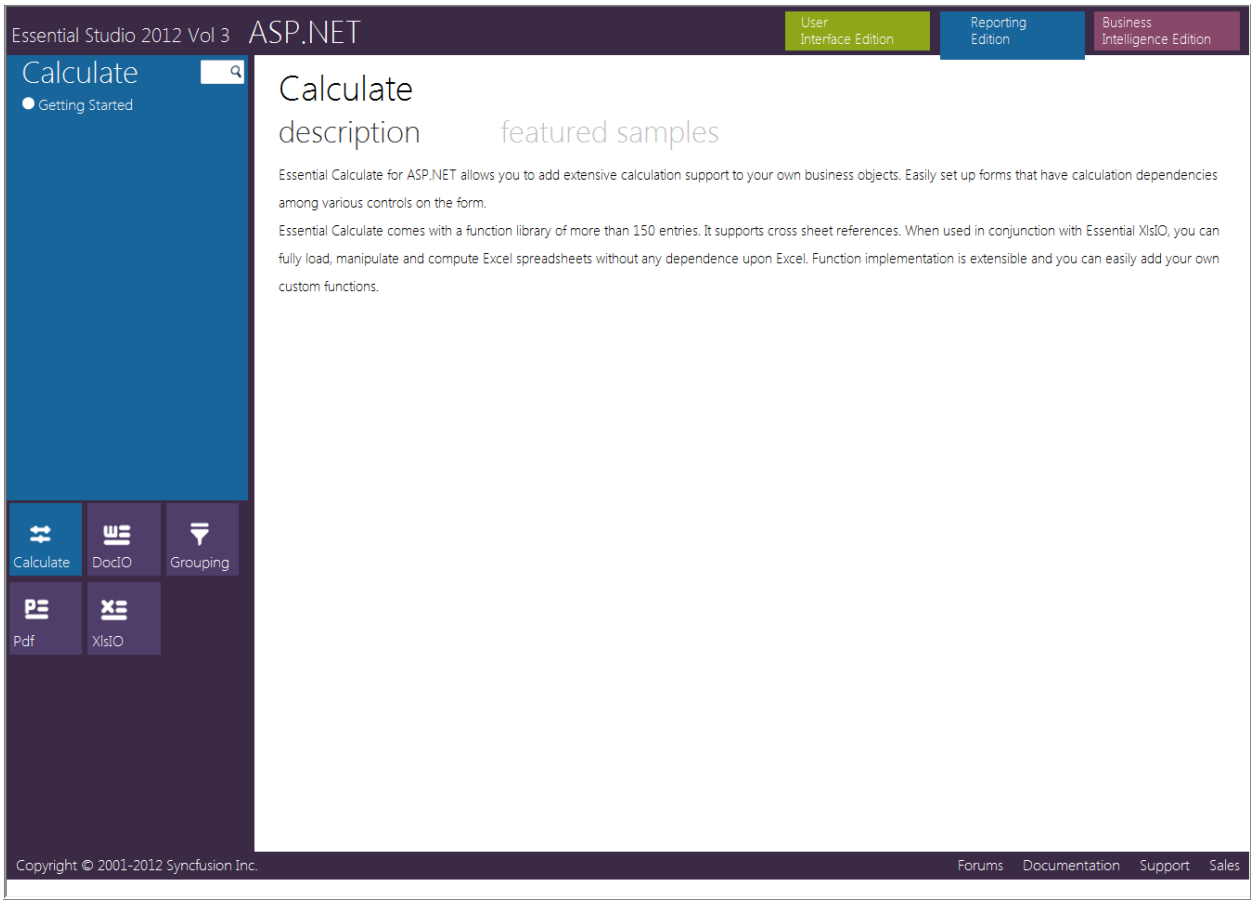


Figure 8: Calculate samples displayed in the ASP.NET Sample Browser

3. Select any sample and browse through the features.

Source Code Location

Windows Forms Source Code

The default location of the Windows Forms Calculate source code is:

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

ASP.NET Source Code

The default location of the ASP.NET Calculate source code is:

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

2.3 Deployment Requirements

This section elaborates on the deployment requirements for using Essential Calculate in various applications. It comprises the following topics:

2.3.1 DLLs

The following dlls need to be referenced in your application for the usage of Essential Calculate.

- Syncfusion.Core.dll
- Syncfusion.Calculate.Base.dll
- Syncfusion.Calculate.Windows.dll
- Syncfusion.Shared.Base.dll
- Syncfusion.Shared.Web.dll

2.3.2 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.

Essential Calculate allows working in medium trust by using following assemblies.

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

3 Getting Started

This section covers information on the following topics.

3.1 Class Diagram

The following illustration shows the Class Diagram for Essential Calculate.

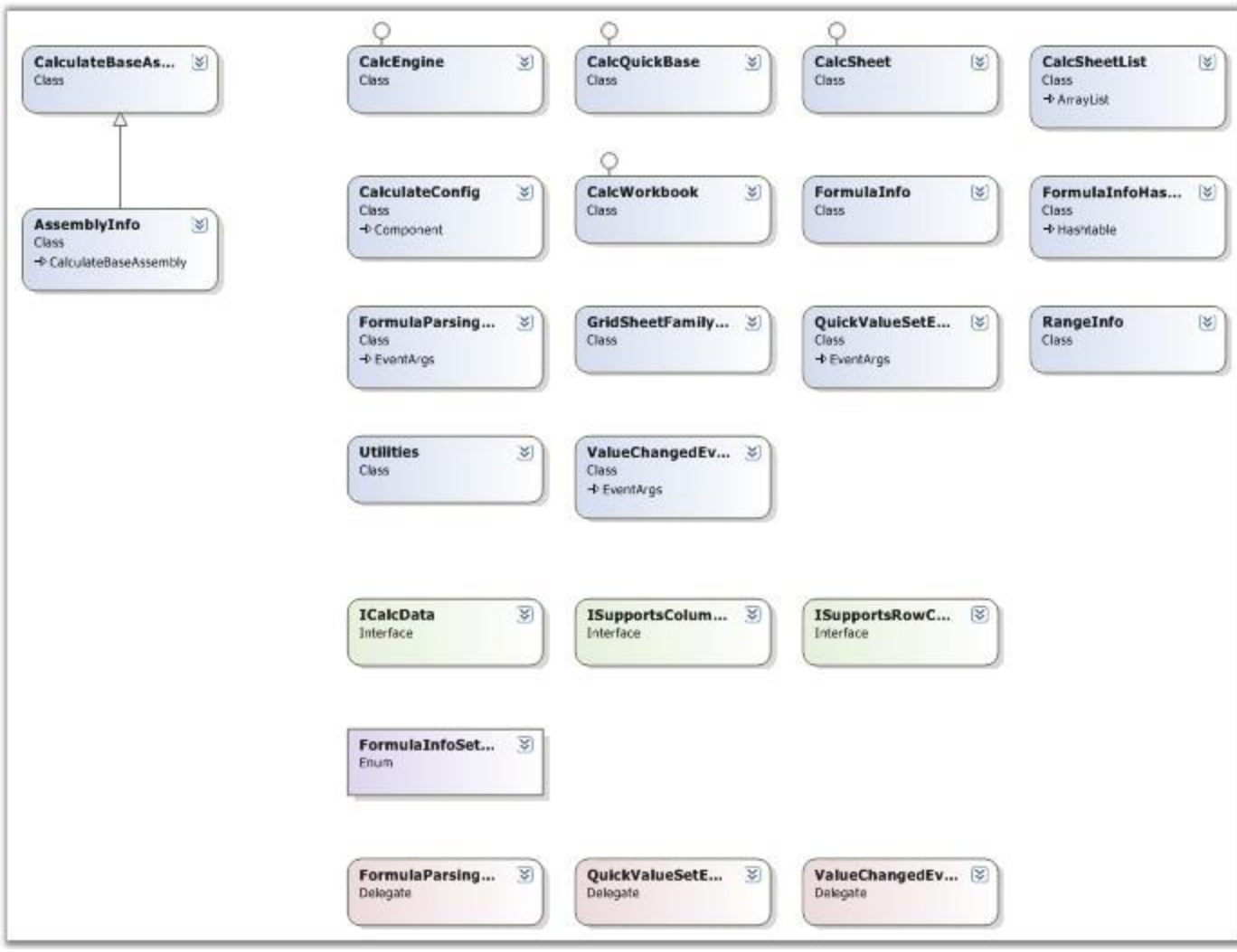


Figure 9: Class Diagram for Essential Calculate

3.2 Creating Platform Application

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

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**.

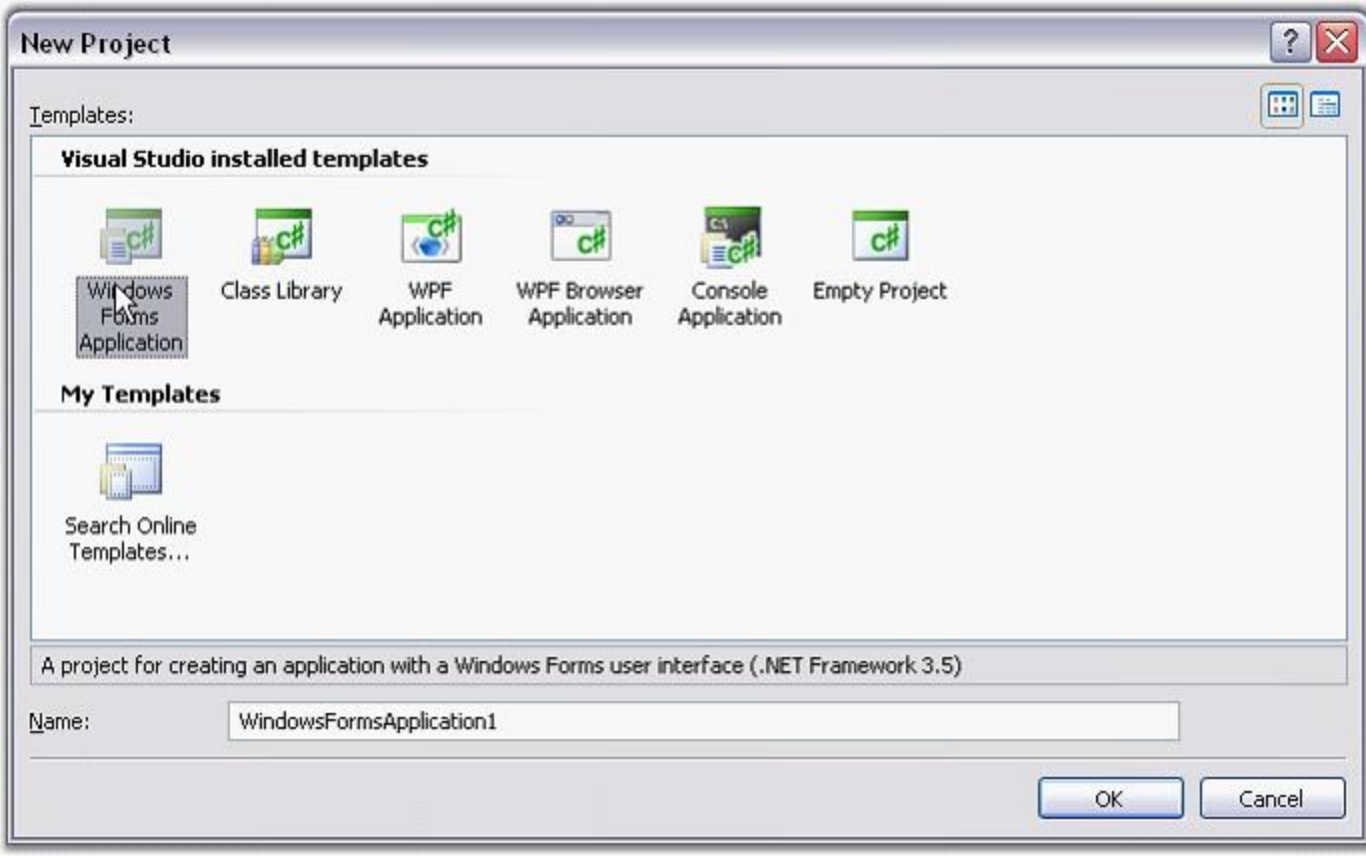


Figure 10: New Project Dialog Box

A windows application is created.

2. 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. Now you need to deploy Essential Calculate into this Windows application. Refer Windows topic for detailed information.

Web Application

1. Open Microsoft Visual Studio. Go to **File** menu and click **New Website**. In the New Website dialog box, select **ASP.NET Web Site** template, name the website and click **OK**.

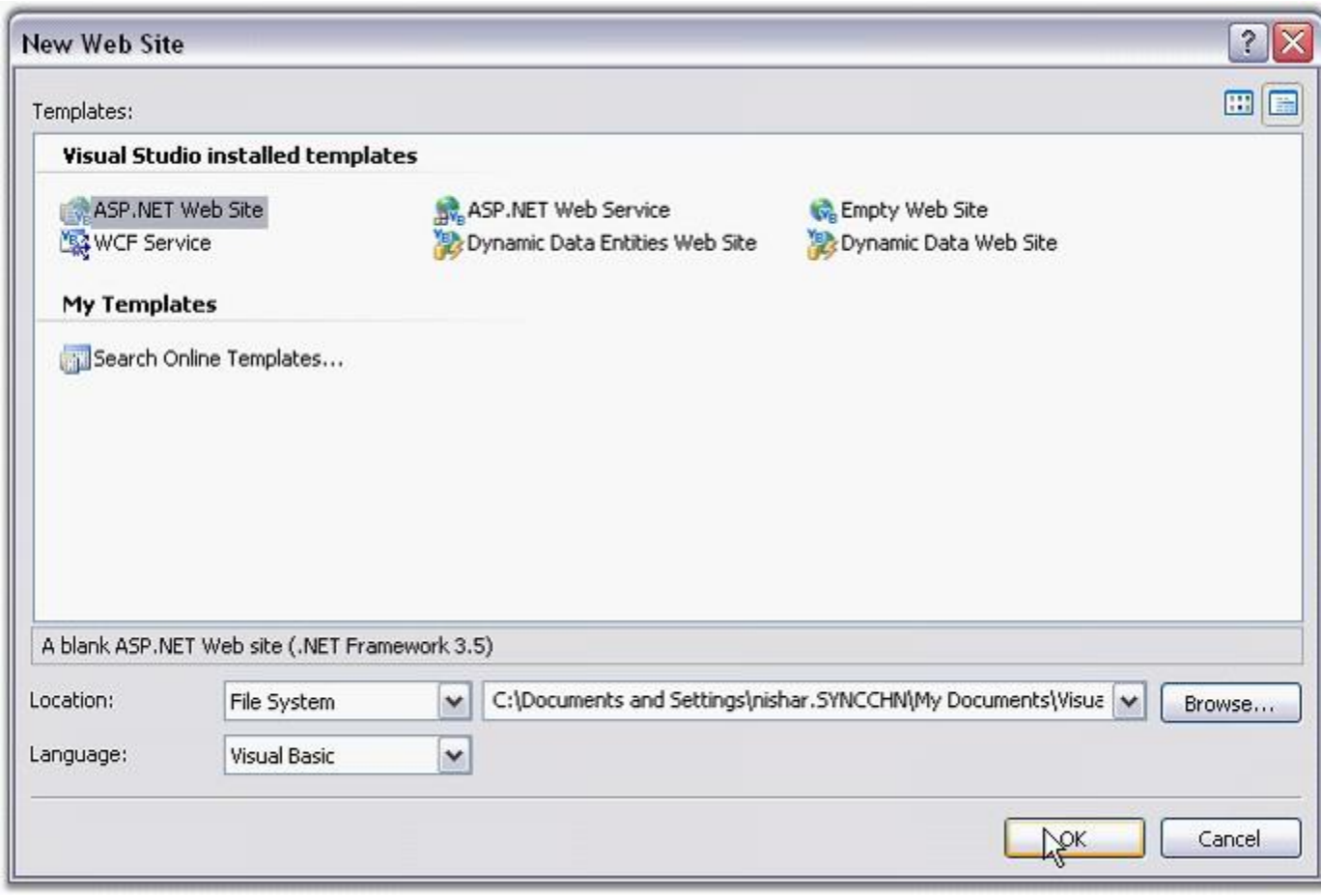


Figure 11: New Web Site Dialog Box

A Web application is created.

2. Now you need to deploy Essential Calculate into this ASP.NET application. Refer ASP.NET topic for detailed information.

WPF Application

1. 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**.

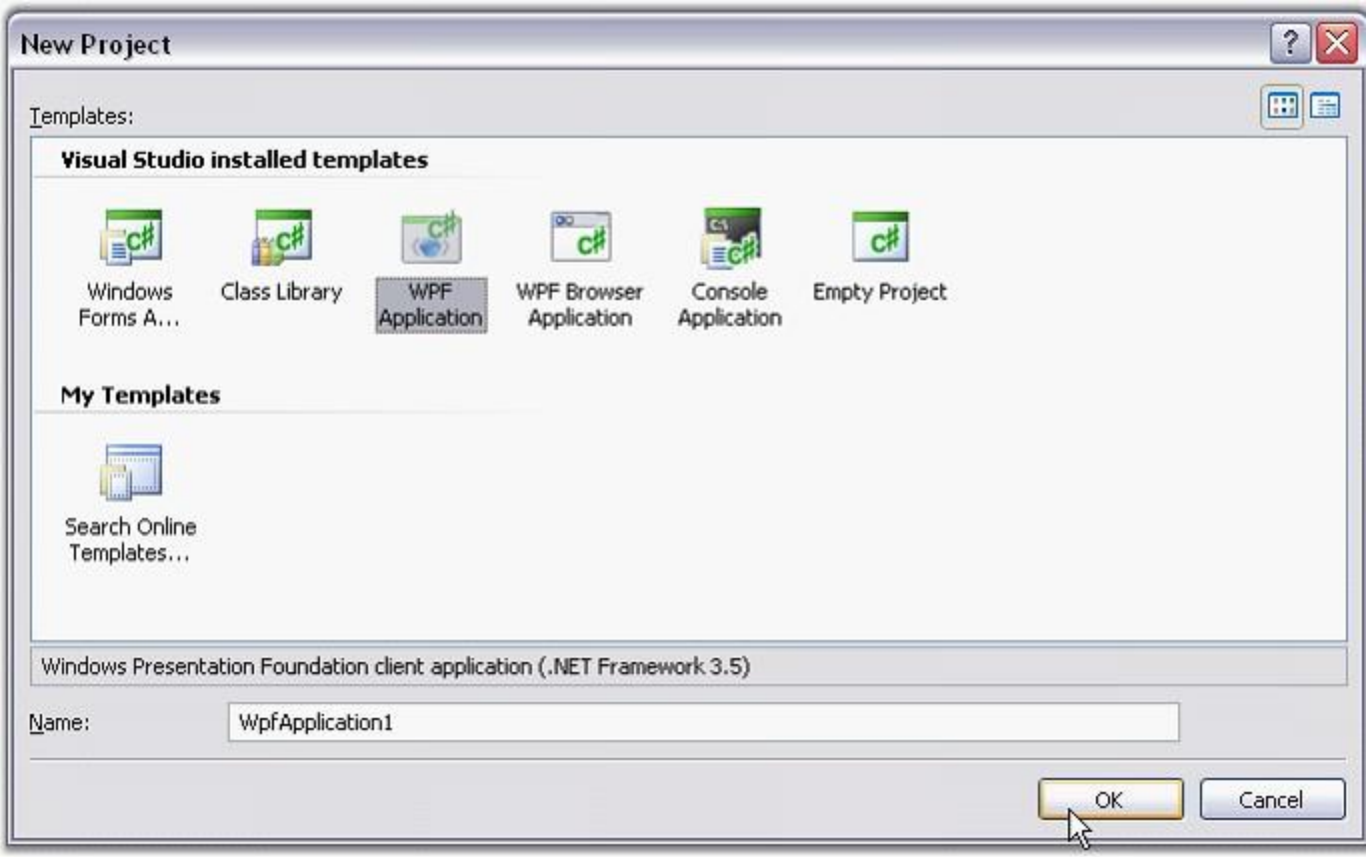


Figure 12: New Project Dialog Box

A new WPF application is created.

2. 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. Now you need to deploy Essential Calculate into this WPF application. Refer WPF topic for detailed information.

For more information refer the following topic.

3.3 Deploying Essential Calculate

We have now created a platform application in the previous topic (Creating Platform Application).

This section will guide you to deploy Essential Calculate in those applications under the following topics:

- **Windows**-Step-by-step procedure to deploy Calculate in a Windows application.
- **ASP.NET**-Step-by-step procedure to deploy Calculate in an ASP.NET application.
- **WPF**-Step-by-step procedure to deploy Calculate in a WPF application.

3.3.1 Windows

Now, you have created a Windows application (refer Creating Platform Application). This section illustrates how to deploy Essential Calculate into this Windows application.

Deploying Essential Calculate in a Windows Application

The following steps will guide you to deploy Essential Calculate:

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. In order to use the Essential Calculate library in your project, add the **CalculateConfig** component found in the toolbox to a project to enable support for Calculate.

Figure 13: Toolbox

This will add references to the following dependent assemblies of your project:

- Syncfusion.Core.dll
- Syncfusion.Compression.Base.dll
- Syncfusion.Calculate.Base.dll

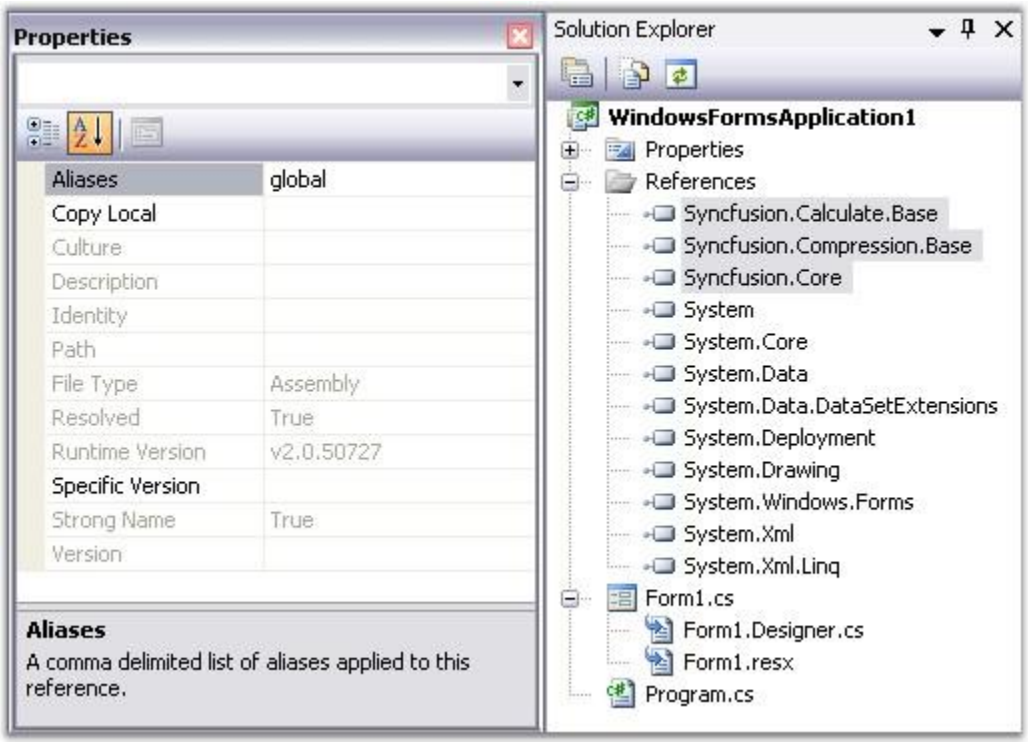


Figure 14: Solution Explorer

4. The libraries are added to the GAC during the product installation. Note that the CalculateConfig is a component provided for convenient usage of Essential Calculate. Hence, the product can also be used just by manually adding reference to the above assemblies.

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

5. Then create a CalculateEngine. The **CalcQuickBase** class is used to create a CalculateEngine.

```
[C#]

// Create a new CalculateQuickBase. This object represents the
CalculateEngine.
CalcQuickBase cq = new CalcQuickBase();
```



```
[VB.NET]

' Create a new CalculateQuickBase. This object represents the
CalculateEngine.
Dim cq As CalcQuickBase
cq = New CalcQuickBase()
```

6. Use the **ParseAndCompute** method to perform calculations by using the CalculateEngine.

```
[C#]

// Perform calculations by using Essential Calculate.
string formula = "if(20>10,\"BIG\",\"Small\")";
string value = cq.ParseAndCompute(formula);
```

```
[VB.NET]

' Perform calculations by using Essential Calculate.
Dim formula As String = "if(20>10,""BIG"",""Small"")"
Dim value As String = cq.ParseAndCompute(formula)
```

7. You can also modify the default behavior of the CalculateEngine by using the **Engine** property.

Example

The default format of appending quotation marks to the concatenated string can be eliminated by using the following code.

```
[C#]

// Strings concatenated by using the ampersand operator will be returned
without quotation marks.
cq.Engine.UseNoAmpersandQuotes = true;
```

```
[VB.NET]

' Strings concatenated by using the ampersand operator will be returned
without quotation marks.
cq.Engine.UseNoAmpersandQuotes = True
```

 ***Note: Engine is a class that is defined as a "property" in Essential Calculate.***

Essential Calculate is deployed in your Windows application.

3.3.2 ASP.NET

Now, you have created an ASP.NET application (refer Creating Platform Application). This section illustrates how to deploy Essential Calculate into this ASP.NET application.

Deploying Essential Calculate in an ASP.NET Application

This section provides information and instructions for deploying ASP.NET applications that use Essential Calculate. 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 Calculate 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.

The following steps will guide you to deploy Essential Calculate 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 Syncfusion assemblies need to be in the bin folder that is beside the aspx files.

 ***Note: They can also be in the GAC, in which case, they should be referenced in Web.config file.***

```
[ASPX]

<configuration>
   <system.web>
     <compilation>
      <assemblies>
        <add assembly="Syncfusion.Calculate.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.

3. **Data Files**-If you have .xml, .mdb, or other data files, ensure that they have sufficient security permission. Authenticated users should have full control over the files and the directories in order to give ASP.NET code, enough permission to open the file during runtime.

Refer to the document in the following path, for step by step process of Syncfusion assemblies deployment in ASP.NET:

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

Note: Application with Essential Calculate needs the following dependent assemblies for deployment.

- - Syncfusion.Core.dll
- - Syncfusion.Compression.Base.dll
- - Syncfusion.Calculate.Base.dll

4. Create a CalculateEngine. The **CalcQuickBase** class is used to create a CalculateEngine. Use the **ParseAndCompute** method to perform calculations by using the CalculateEngine.

You can also modify the default behavior of the CalculateEngine by using the **Engine** property.

```
[C#]

// Create a new CalculateQuickBase. This object represents the CalculateEngine.
CalcQuickBase cq = new CalcQuickBase();

' Perform calculations by using ParseAndCompute method of Essential Calculate.
Dim formula As String = "if(20>10,""BIG"",""Small"")"
Dim value As String = cq.ParseAndCompute(formula)

// Strings concatenated by using the ampersand operator will be returned without
quotation marks.
cq.Engine.UseNoAmpersandQuotes = true;
```

Note: The Engine is a class that is defined as a "property" in Essential Calculate.

Essential Calculate is now deployed in your ASP.NET application.

3.3.3 WPF

Now, you have created a WPF application (refer Creating Platform Application). This section illustrates how to deploy Essential Calculate into this WPF application.

Deploying Essential Calculate in a WPF Application

The following steps will guide you to deploy Essential Calculate:

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.Base.dll
   - Syncfusion.Calculate.Base.dll

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

3. Then create a CalculateEngine. The **CalcQuickBase** class is used to create a CalculateEngine.

[C#]

```
// Create a new CalculateQuickBase. This object represents the
CalculateEngine.
CalcQuickBase cq = new CalcQuickBase();
```

[VB.NET]

```
' Create a new CalculateQuickBase. This object represents the
CalculateEngine.
Dim cq As CalcQuickBase
cq = New CalcQuickBase()
```

4. Use the **ParseAndCompute** method to perform calculations by using the CalculateEngine.

[C#]

```
// Perform calculations by using Essential Calculate.
string formula = "if(20>10,\"BIG\",\"Small\")";
string value = cq.ParseAndCompute(formula);
```

[VB.NET]

```
' Perform calculations by using Essential Calculate.
Dim formula As String = "if(20>10,""BIG"",""Small"")"
Dim value As String = cq.ParseAndCompute(formula)
```

5. You can also modify the default behavior of the CalculateEngine by using the **Engine** property.

Example

The default format of appending quotation marks to the concatenated string can be eliminated by using the following code.

```
[C#]

// Strings concatenated by using the ampersand operator will be returned without quotation marks.
cq.Engine.UseNoAmpersandQuotes = true;
```

```
[VB.NET]

' Strings concatenated by using the ampersand operator will be returned without quotation marks.
cq.Engine.UseNoAmpersandQuotes = True
```

 ***Note: Engine is a class that is defined as a "property" in Essential Calculate.***

Essential Calculate is now deployed in your WPF application.

3.4 Feature Summary

The features of Essential Calculate are listed below.

- Essential Calculate comes with a function library of more than 150 entries and supports cross sheet references.
- It can be used in conjunction with Essential XlsIO to fully load, manipulate and compute Excel spreadsheets without depending on Excel.
- Essential Calculate does not depend on Microsoft Excel and thus enables you to perform calculations independent of Excel.
- You can add extensive calculation support to your own business objects in both Windows Forms and ASP.NET applications.

- Easily set up forms that have calculation dependencies among various controls.
- With Essential Calculate, you can set properties that will indicate that you want formula dependencies to be tracked so that the values are automatically updated when a dependent value changes. Or you can turn off the overhead of tracking dependencies and have formulas calculated from scratch when you need a particular formula value.
- Essential Calculate can be used in manual mode or automatic mode.
- The manual mode works when you explicitly request for a value. At that point, the calculation is done from scratch to obtain the computed value. So, if your formula depends on several other values in the form, and when you request the computed value, the other values will be retrieved and used to compute the requested formula.
- In automatic mode, Essential Calculate maintains a dependency list. Hence, when a value is changed, any formula that depends on it, is recalculated at that particular point. When you request for a formula value, the formula value is not computed from scratch; instead it is retrieved from where Essential Calculate stores computed values.

3.5 Quick Start

This section will show you how easy it is to get started using Essential Calculate. It will give you a basic introduction to the concepts you need to know before getting started with the product and some tips and ideas on how to use Essential Calculate in your projects.

Console Application Using CalcQuickBase

In this section, you will learn how to use the CalcQuickBase object to perform arbitrary calculations from a Console Application. This will show you the minimal steps that are required to use Essential Calculate to add calculation support to an application. This quick application lets you type algebraic expressions using a Console.ReadLine and display the calculated results using a Console.WriteLine.

Windows Application Using CalcQuickBase

In this section, you will create a Windows Application that uses a **CalcQuickBase** object that handles arbitrary variables in its calculations.

Adding Calculation Support to an Arbitrary Array with an ICalcData Interface

This section will demonstrate how to add calculation support to an arbitrary array of doubles by adding an additional row and column to hold summary information.

3.5.1 Simple Console Application Using CalcQuickBase

In this section, you will create a Console Application that requests a string from the user. The application requires the string to be an algebraic expression like: 4.1 + 3.21 or SQRT(2) * 14.2 or (3 + Sqrt(2)) / (2 - Cos(2.1))

Once you enter an expression, the application uses a **CalcQuick** object to perform the requested calculation and displays it to the Console. The process continues until you enter an empty string. The step-by-step procedure to create a simple console application is discussed under the following topic:

3.5.1.1 Console Application CalcQuickBase

The step-by-step procedure to create a simple console application is as follows:

1. From Visual Studio, use File | New | Project to create a new Console Application named CalcQuickBaseTutorial. After creating the project, open the References node in the Solution Explorer and add a reference to Syncfusion.Calculate.Base. At this point, your Solution Explorer window should appear similar to this one.

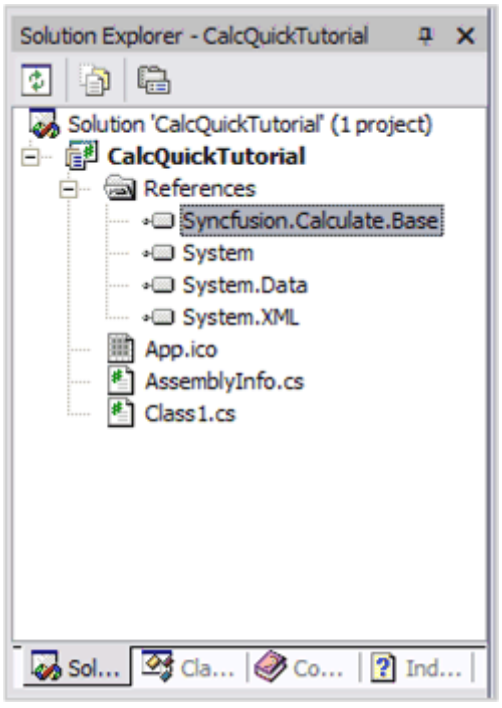


Figure 15: Essential Calculate Reference Added to the Project

2. In the Main method, add the code to create a **CalcQuickBase** object. Also add the code to loop through the process of retrieving a string and using **CalcQuickBase.ParseAndCompute** to perform the calculation that is represented by the string. Given below is the code that handles these tasks.

[C#]

```
using System;
using Syncfusion.Calculate;

namespace CalcQuickBaseTutorial
{
        /// <summary>
        /// Summary description for Class1.
        /// </summary>
        class Class1
        {
                /// <summary>
                /// The main entry point for the application.
                /// </summary>
                [STAThread]
                static void Main(string[] args)
                {
                        CalcQuickBase cq = new CalcQuickBase();

                        string s;
                        while( (s = Console.ReadLine()) != "" )
                        {
                                string val = cq.ParseAndCompute(s);
                                Console.WriteLine("value= " + val);

                                // Blank line
                                Console.WriteLine("");
                        }
                }
        }
}
```

[VB.NET]

```
Imports System
Imports Syncfusion.Calculate

Namespace CalcQuickBaseTutorial
    Class Class1
```

```
        Public Overloads Shared Sub Main()
            Dim cq As New CalcQuickBase
            Dim s As String = Console.ReadLine()
            Do While s <> ""
                Dim val As String = cq.ParseAndCompute(s)
                Console.WriteLine(("value= " + val))
                Console.WriteLine("")

                ' Blank line
                s = Console.ReadLine()
            Loop

        ' Main
        End Sub

    ' Class1
    End Class

' CalcQuickBaseTutorial
End Namespace
```

3. Once the code is entered, run the application by pressing F5. Then enter an expression such as 1+2 and press Enter. Enter additional algebraic combinations of constants and named functions from the Function Library like Sin, Cos, Sum and Pi. Press Enter without entering anything to terminate the program. Below is a typical display of this.

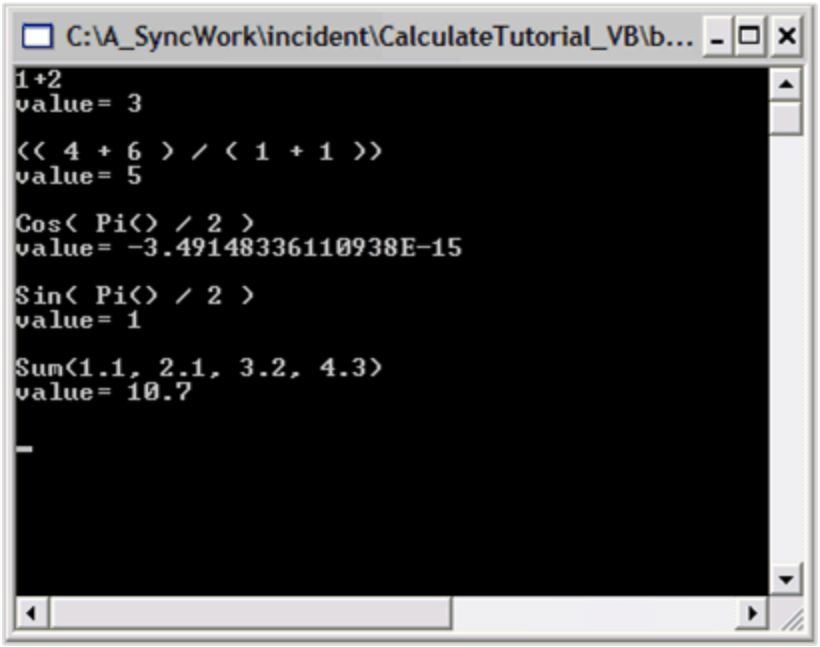


Figure 16: Application Display

3.5.2 Windows Application Using Variables and CalcQuickBase

In this section, you will learn how to create a Windows Application that allows you to register variables with a **CalcQuick** object and then use these variables in algebraic expressions. For example, you might register a variable named Rate to be 0.07 and a variable named Amount to be 1500, and then compute a quantity represented by the formula Rate * Amount or by (1 + Rate) * Amount.

3.5.2.1 Windows Forms CalcQuickBase

1. In Visual Studio, use File | New | Project to create a new Windows Forms Application. Right-click the References in Solution Explorer and add a reference to Syncfusion.Calculate.Base.

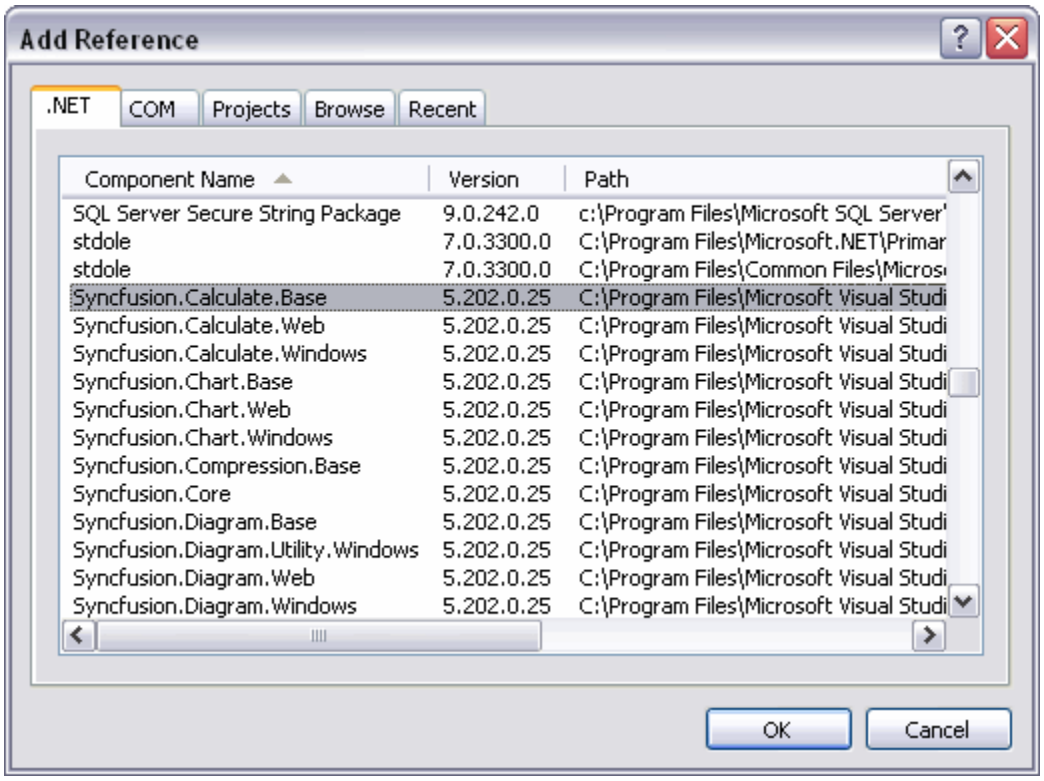


Figure 17: Essential Calculate Reference Being Added to the Project

2. Using the designer, drop three text boxes, three labels, two buttons and one list box onto the form as shown in the picture below.

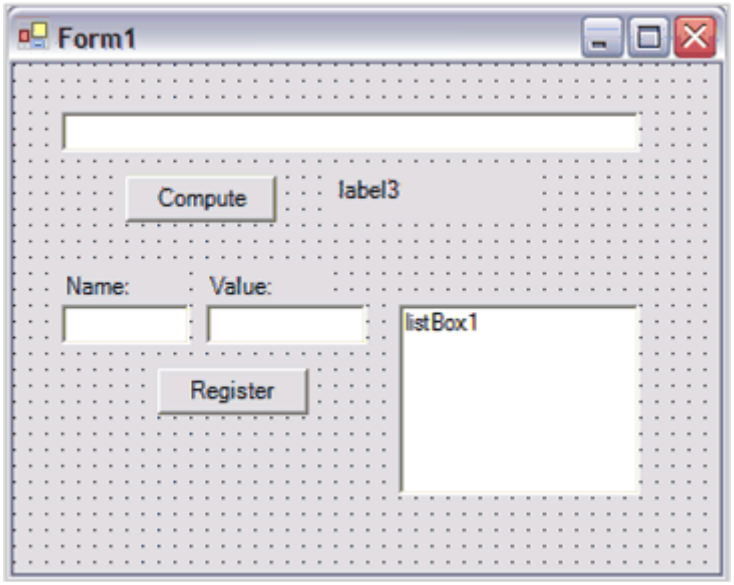


Figure 18: Form Showing Controls

3. Double-click the form in the designer to add a **Form.Load** event handler. Add the code which, is shown below to the project.

```
[C#]

using Syncfusion.Calculate;

//...

private CalcQuickBase cq;
private void Form1_Load(object sender, System.EventArgs e)
{
        cq = new CalcQuickBase();

        this.button1.Click += new EventHandler(button1_Click);
        this.button2.Click += new EventHandler(button2_Click);
}
private void button1_Click(object sender, EventArgs e)
{
        // Compute
        this.label3.Text =
                                this.cq.ParseAndCompute(this.textBox1.Text);
}
private void button2_Click(object sender, EventArgs e)
{
        // Register name.
        string key = this.textBox2.Text;
```



```
        if(key.Length > 0)
        {
                // The value.
                this.cq[key] = this.textBox3.Text;

                // Just display it in the list box.
                this.listBox1.Items.Add(string.Format("[{0}] = {1}",
                        key, this.textBox3.Text));

        }
}
```

[VB]

```
Imports Syncfusion.Calculate

'...
Dim cq As CalcQuickBase
Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs)
    cq = New CalcQuickBase

    AddHandler Me.button1.Click, AddressOf button1_Click
    AddHandler Me.button2.Click, AddressOf button2_Click

' Form1_Load
End Sub

Private Sub button1_Click(ByVal sender As Object, ByVal e As EventArgs)

    ' Compute
    Me.label3.Text = Me.cq.ParseAndCompute(Me.textBox1.Text)

' Button1_Click
End Sub

Private Sub button2_Click(ByVal sender As Object, ByVal e As EventArgs)

    ' Register name.
    Dim key As String = Me.textBox2.Text
    If key.Length > 0 Then

        ' The value.
        Me.cq(key) = Me.textBox3.Text

        ' Just display it in the list box.
        Me.listBox1.Items.Add(String.Format("[{0}] = {1}", key,
```



```
Me.textBox3.Text))
    End If

' Button2_Click
End Sub
```

4. Run the sample by pressing F5. Then in the Name text box, enter Rate and in the Value text box, enter .07. Now press the Register button. Similarly, enter Amount in the Name text box, 15000 in the Value text box followed by pressing the Register button.

5. In the top text box, which is empty, enter the formula: **[Rate] * [Amount]**

6. Press the Compute button. You will then see a screen similar to the one below.

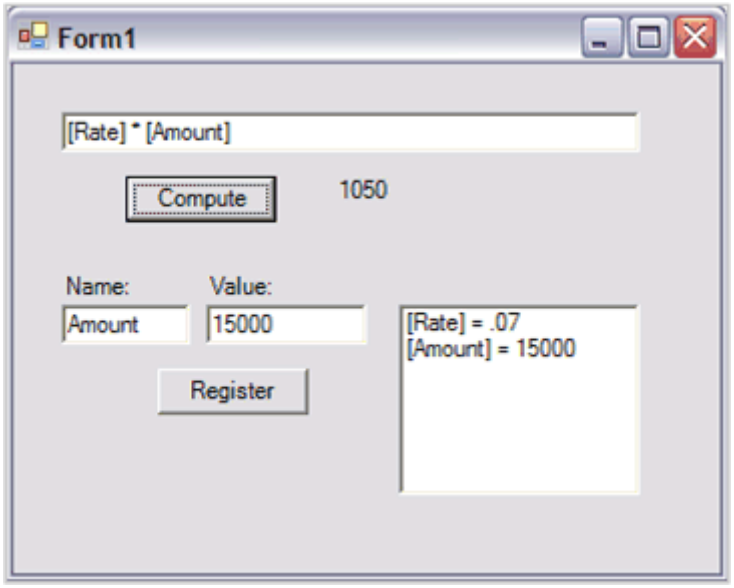


Figure 19: Form Showing Two Variables Registered and a Sample Calculation

The computed product, 1050, is displayed next to the Compute button.

By examining the code, notice that you register a name with a **CalcQuickBase** object by using its name as an indexer on the CalcQuickBase object and assign the desired value to this indexed object. Then to use the name within a formula, you enclose it within square brackets.

3.5.3 Adding Calculation Support to an Array Using ICalcData

For rectangular business objects, implementing an ICalcData interface lets you use values from your object in calculations. This section illustrates the process of adding calculation support to an Array.

3.5.3.1 ICalcData

1. In Visual Studio, use File | New | Project to create a new Windows Forms Application. Right-click the References in Solution Explorer and add a reference to Syncfusion.Calculate.Base.

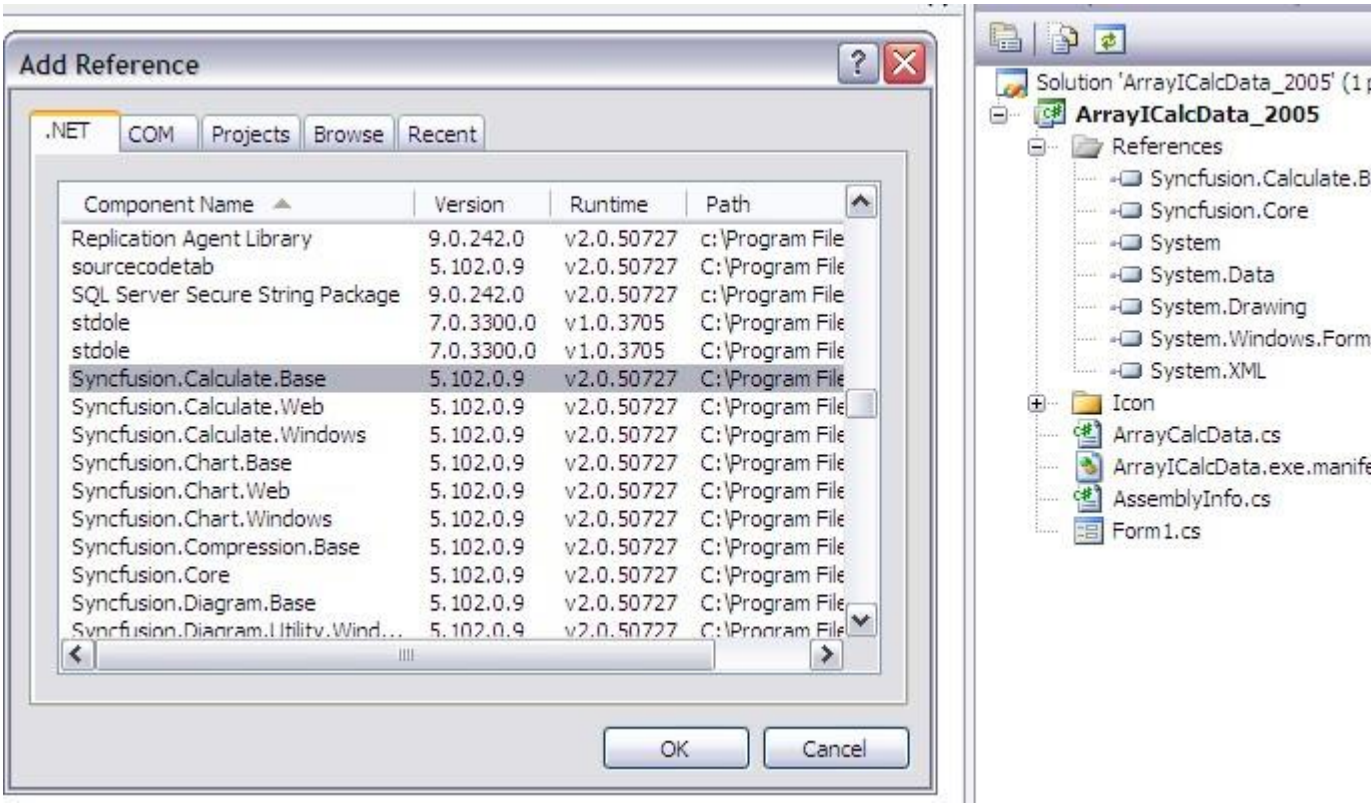


Figure 20: Essential Calculate Reference Being Added to the Project

2. As you drop the controls on the form, accept the default names so that you can copy and paste the code snippets later in this lesson.

3. The form has two buttons: the first is the Generate Data button and the second is the Set button. Drop a text box on the form and set its **MultiLine** property to True so that you can size it to occupy most of the form.

4. Drop the final three text boxes in left to right order under the Generate Data button, adding three labels to identify these text boxes.

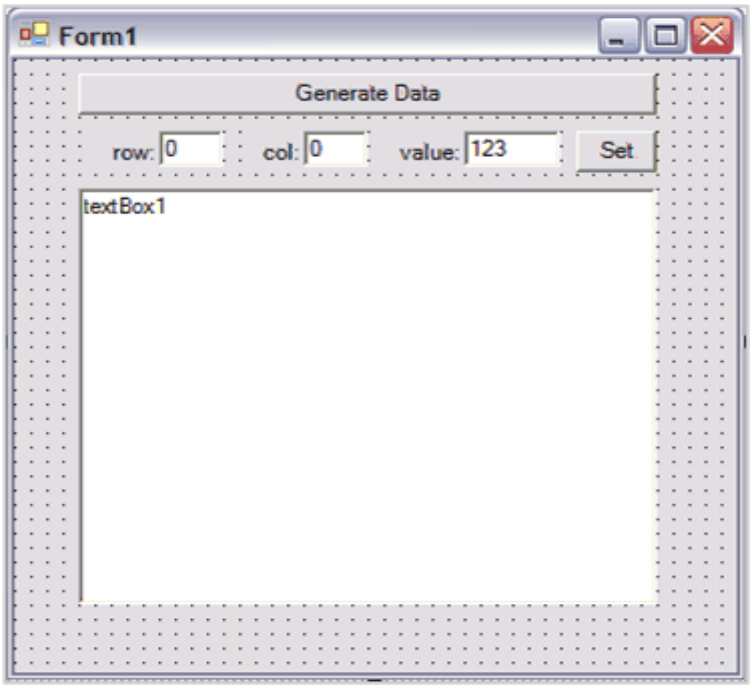


Figure 21: Form with Controls Positioned

You now have your basic form. Before leaving this form in the designer, double click both buttons to add the button handler code stubs to your Form1 code. You can add the code to these stubs later.

In order to add arbitrary calculation support to an object, that object must implement the ICalcData interface. In this sample, you may want to add calculation support to a double array. To do so, you must create a wrapper class that accepts a double object in its constructor and then returns these double values in its indexer. In addition you can extend the wrapper class by adding an additional row that holds the sum of the column values in the double array and by adding an addition column that holds the sum of the row values in the double array.

1. The first step is to add the class. To do so, right-click your project in the Solution Explorer window, select Add, and select Add Class.

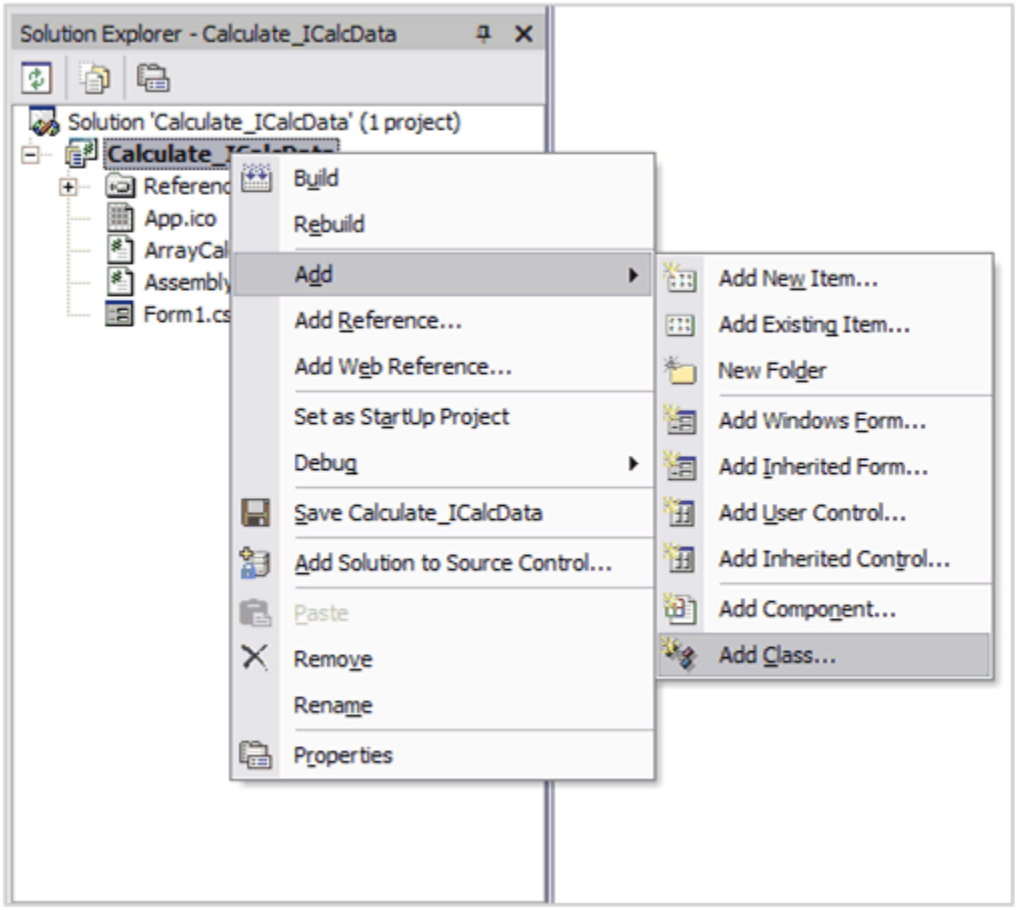


Figure 22: Adding a Class File to Your Project

2. In the dialog that appears, name the class as ArrayCalcData.cs (or ArrayCalcData.vb depending upon the language you are using).

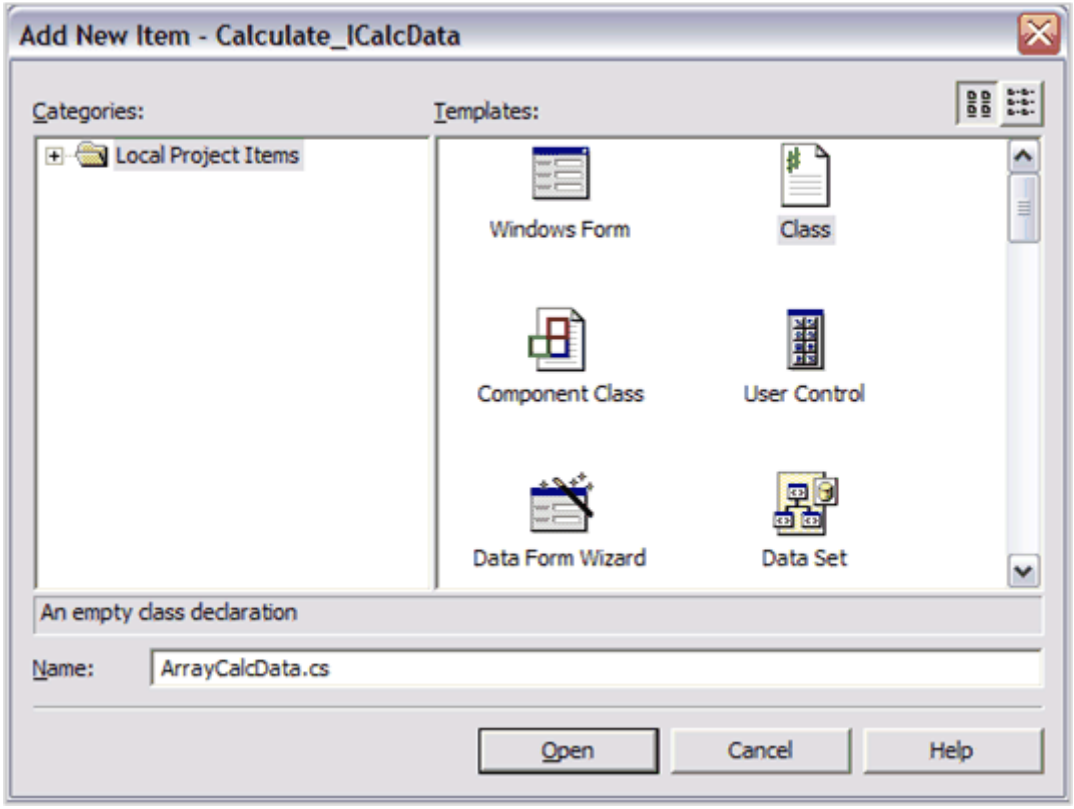


Figure 23: Naming the Class File

3. After adding the class, Visual Studio displays the class implementation file whose contents are shown below. In the next few steps, you must add the code to this class to create the functionality you need.

```
[C#]

using System;

namespace Calculate_ICalcData
{
        /// <summary>
        /// Summary description for ArrayCalcData.
        /// </summary>
        public class ArrayCalcData
        {
                public ArrayCalcData()
                {
                        //
                        // TODO: Add constructor logic here.
                        //
                }
        }
}
```



```
[VB]

Imports System

Namespace Calculate_ICalcData
        ''' <summary>
        ''' Summary description for ArrayCalcData.
        ''' </summary>
        Public Class ArrayCalcData
            Public Sub New()
                '
                ' TODO: Add constructor logic here.
                '
            End Sub
        End Class
End Namespace
```

4. You need to add fields to hold the double array, the extra row and extra column. Additionally, you can add fields to hold the row and column counts for convenience. Here is the code.

```
[C#]

using System;
using Syncfusion.Calculate;

namespace Calculate_ICalcData
{
        public class ArrayCalcData
        {
                /// <summary>
                /// Original double array.
                /// </summary>
                private double[,] dValues;

                /// <summary>
                /// Vector holding the sum of the rows.
                /// </summary>
                /// <remarks>
                /// Serves as the last column.
                /// </remarks>
                private object[] rowSums;

                /// <summary>
                /// Vector holding the sum of the columns.
                /// </summary>
                /// <remarks>
                /// Serves as the last row.
                /// </remarks>
```

```
                private object[] colSums;

                int rowCount;
                int colCount;

                //...
        }
}
```

```
[VB]

Imports System
Imports Syncfusion.Calculate

Public Class ArrayCalcData

    '/ <summary>
    '/ Original double array.
    '/ </summary>
    Private dValues(,) As Double

    '/ <summary>
    '/ Vector holding the sum of the rows.
    '/ </summary>
    '/ <remarks>
    '/ Serves as the last column.
    '/ </remarks>
    Private rowSums() As Object

    '/ <summary>
    '/ Vector holding the sum of the columns.
    '/ </summary>
    '/ <remarks>
    '/ Serves as the last row.
    '/ </remarks>
    Private colSums() As Object

    Dim rowCount As Integer
    Dim colCount As Integer

End Class
```

5. Here you are making a constructor that accepts a double array as an argument. In the constructor code, you must save the reference that is to be the passed in a double array, initialize two **RowCount** and **ColCount** fields and allocate the two additional arrays that are needed to hold the added sums you want.

If you notice, you are actually populating these added arrays with formula strings such as "=Sum(A5:D5)". While using Essential Calculate with an ICalcData interface, Essential Calculate uses Excel-like notation to refer to the cells in a rectangular collection. A1 is the first cell in the first row, B1 is second cell in the first row, and so on. So, "=Sum(A5:D5)" sums columns 1 through 4 in row 5. The method **RangeInfo.GetAlphaLabel** used in the code, converts a numerical value like 1, 2, or 3, into the proper column letter A, B, or C.

[C#]

```
/// <summary>
/// Wraps the double array with an extra row and column that holds calculated sums.
/// </summary>
/// <param name="dValues"></param>
public ArrayCalcData(double[,] dValues)
{
        this.dValues = dValues;
        rowCount = dValues.GetLength(0);
        colCount = dValues.GetLength(1);
        rowSums = new object[rowCount + 1];
        colSums = new object[colCount + 1];

        // Initializes the formulas for the row sums.
        // eg. "=Sum(A5:D5)"  to sum row 5
        for(int row = 0; row <= rowCount; ++row)
        {
                rowSums[row] = string.Format("=Sum(A{1}:{0}{1})",
                        RangeInfo.GetAlphaLabel(colCount), row + 1);
        }

        // Initializes the formulas for the column sums.
        // eg. "=Sum(B1:B5)"  to sum column 2
        for(int col = 0; col <= colCount; ++col)
        {
                colSums[col] = string.Format("=Sum({0}1:{0}{1})",
                        RangeInfo.GetAlphaLabel(col + 1), rowCount);
        }
}
```

[VB]

```
'/ <summary>
'/ Wraps the double array with an extra row and column that holds calculated sums.
'/ </summary>
'/ <param name="dValues"></param>
Public Sub New(ByVal dValues(,) As Double)
        Me.dValues = dValues
        rowCount = dValues.GetLength(0)
        colCount = dValues.GetLength(1)
        rowSums = New Object(rowCount + 1) {}
```

```
        colSums = New Object(colCount + 1) {}

        ' Initializes the formulas for the row sums.
        ' eg. "=Sum(A5:D5)"  to sum row 5
              Dim row As Integer
         Dim row As Integer
        For row = 0 To rowCount - 1
            rowSums(row) = String.Format("=Sum(A{1}:{0}{1})", _
                        RangeInfo.GetAlphaLabel(colCount - 1), row + 1)
        Next

        ' Initializes the formulas for the column sums.
        ' eg. "=Sum(B1:B5)"  to sum column 2
        Dim col As Integer
        For col = 0 To colCount - 1
            colSums(col) = String.Format("=Sum({0}1:{0}{1})", _
                 RangeInfo.GetAlphaLabel((col + 1)), rowCount - 1)
        Next

' New
End Sub
```

6. Now you can add the code stubs for implementing the ICalcData interface. For the implementation, you only need to add the code to the first two methods in the interface. **WireParentObject** will not be used in our code. **ValueChanged** is an event that you will raise in the **SetValueRowCol** implementation.

7. Using Visual Studio 2003 with C#, add a colon after the class name in the class declaration and type ICalcData. Pressing the tab key will add the method stubs. Given below is a picture showing this technique.

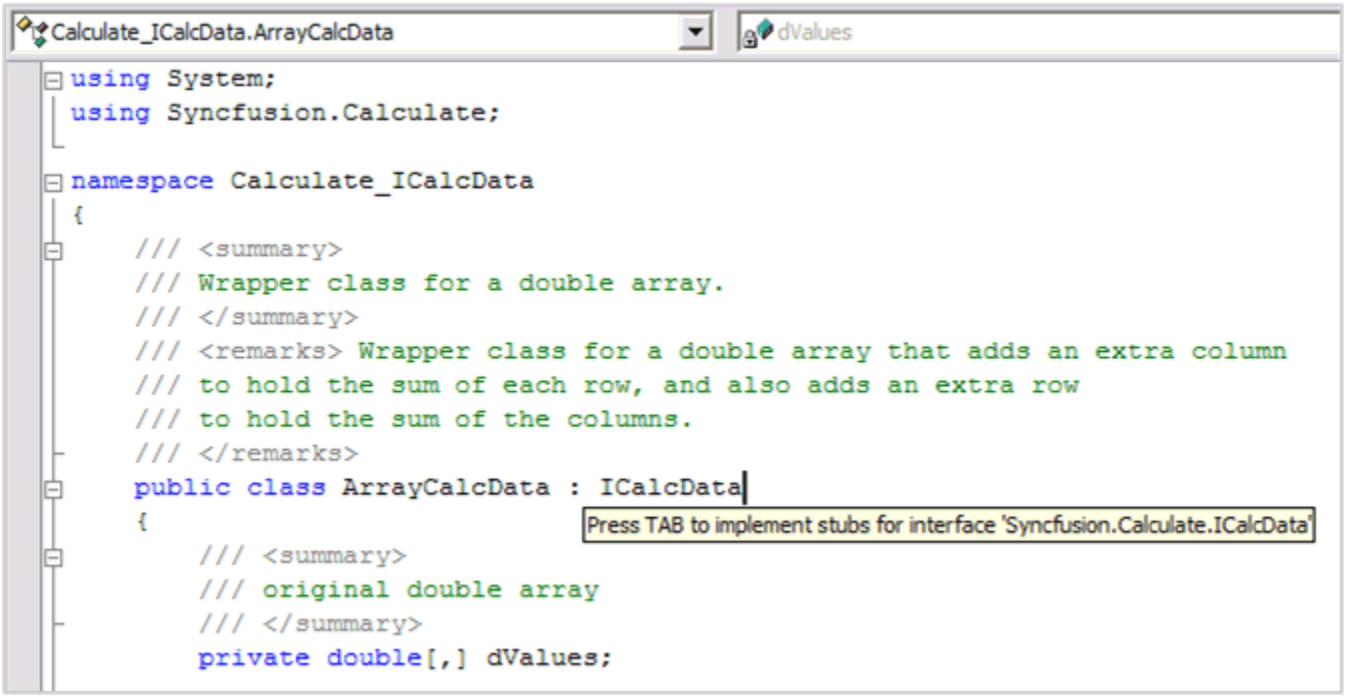


Figure 24: Implementing the ICalcData Interface in C#

If you are using VB.NET, then you can add the ICalcData stubs using the drop-down lists at the top of the edit window in Visual Studio. In the left drop-down, select the ICalcData interface as shown below.

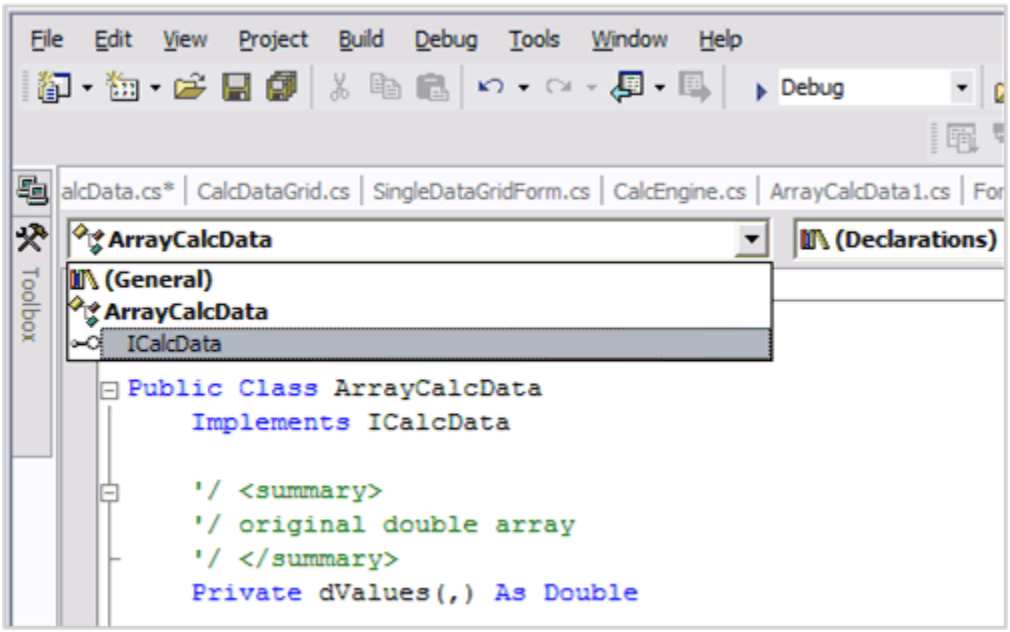


Figure 25: Implementing the ICalcData Interface in VB, Choosing the Interface

Then, in the right drop-down, click each of the four items listed to add the proper code stubs.

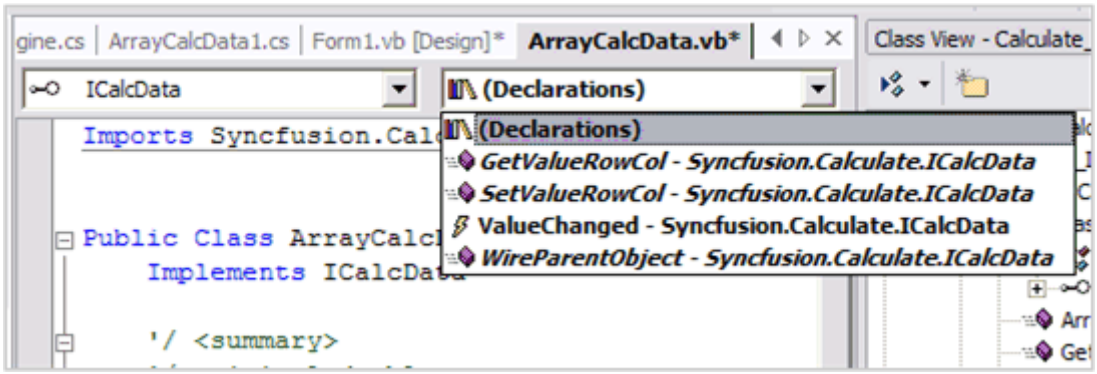


Figure 26: Implementing the ICalcData Interface in VB, Adding the Members

After doing these steps, you will be able to see these methods in the class code. (In the C# code, the region may be collapsed.)

[C#]

```
public object GetValueRowCol(int row, int col)
{
        // TODO:  Add ArrayCalcData1.GetValueRowCol implementation.
        return null;
}

public void SetValueRowCol(object value, int row, int col)
{
        // TODO:  Add ArrayCalcData1.SetValueRowCol implementation.
}

public event Syncfusion.Calculate.ValueChangedEventHandler ValueChanged;

public void WireParentObject()
{
     // TODO:  Add ArrayCalcData1.WireParentObject implementation.
}
```

[VB]

```
Public Function GetValueRowCol(ByVal row As Integer, ByVal col As Integer) _
                As Object Implements
Syncfusion.Calculate.ICalcData.GetValueRowCol
End Function
```

```
Public Sub SetValueRowCol(ByVal value As Object, ByVal row As Integer, ByVal
_
                col As Integer) Implements
Syncfusion.Calculate.ICalcData.SetValueRowCol
End Sub

Public Sub WireParentObject() Implements
Syncfusion.Calculate.ICalcData.WireParentObject
End Sub

Public Event ValueChanged(ByVal sender As Object, ByVal e As _
Syncfusion.Calculate.ValueChangedEventArgs) Implements _
                        Syncfusion.Calculate.ICalcData.ValueChanged
```

8. **GetValueRowCol** should return the value for a given row and column index. The indexes are one-based by convention. Here is the code that you must use. If the last column is requested, the value in the colSums array is returned. If the last row is requested, the value in the rowSums array is returned. Otherwise, the value in the double array is returned.

[C#]

```
/// <summary>
/// Gets the value into either the double array or column vector or row
vector.
/// </summary>
/// <param name="row">One-based row index.</param>
/// <param name="col">One-based column index.</param>
/// <returns>The value.</returns>
/// <remarks> By convention, ICalcData uses one-based indexes.</remarks>
public object GetValueRowCol(int row, int col)
{
        if(row-1 == rowCount)
        {
                return colSums[col-1];
        }
        else if(col-1 == colCount)
        {
                return rowSums[row-1];

        }
        else
                return this.dValues[row-1, col-1];
}
```

[VB]

```
'/ <summary>
'/ Gets the value into either the double array or column vector or row
vector.
'/ </summary>
'/ <param name="row">One-based row index.</param>
'/ <param name="col">One-based column index.</param>
'/ <returns>The value.</returns>
'/ <remarks> By convention, ICalcData uses one-based indexes.</remarks>
Public Function GetValueRowCol(ByVal row As Integer, ByVal col As Integer) _
               As Object Implements
Syncfusion.Calculate.ICalcData.GetValueRowCol
        If row = rowCount Then
            Return colSums((col - 1))
        Else
            If col = colCount Then
                Return rowSums((row - 1))

            Else
                Return Me.dValues(row - 1, col - 1)
            End If
        End If

' GetValueRowCol
End Function
```

9. SetValueRowCol is used to set the value at a specified row and column index. You can also use it to raise the ValueChanged event. The **CalcEngine** listens to this event to trigger calculations as values are entered initially and later modified. It is this event that brings calculation support into the object.

Here is the code you must use. The code is the reverse process of the GetValueRowCol method. At the end of the method, the ValueChanged event is raised.

```
[C#]

/// <summary>
/// Sets the value into either the double array or column vector or row
vector.
/// </summary>
/// <param name="row">One-based row index.</param>
/// <param name="col">One-based column index.</param>
/// <param name="value">The value to be set.</param>
/// <remarks> This setter raises the ICalcData.ValueChanged event which,
/// is listened to by the CalcEngine to manage the calculations.
///
/// By convention, ICalcData uses one-based indexes.
```



```
/// </remarks>
public void SetValueRowCol(object value, int row, int col)
{
        if(row-1 == rowCount)
        {
                colSums[col-1] = value;
        }
        else if(col-1 == colCount)
        {
                rowSums[row-1] = value;
        }
        else
                this.dValues[row-1, col-1] = double.Parse(value.ToString());

        ValueChangedEventArgs e1 = new ValueChangedEventArgs(row, col,
value.ToString());
        ValueChanged(this, e1);
}
```

[VB]

```
'/ <summary>
    '/ Sets the value into either the double array or column vector or row
vector.
    '/ </summary>
    '/ <param name="row">One-based row index.</param>
    '/ <param name="col">One-based column index.</param>
    '/ <param name="value">The value to be set.</param>
    '/ <remarks> This setter raises the ICalcData.ValueChanged event which,
    '/ is listened to by the CalcEngine to manage the calculations.
    '/
    '/ By convention, ICalcData uses one-based indexes.
    '/ </remarks>
    Public Sub SetValueRowCol(ByVal value As Object, ByVal row As Integer, _
                              ByVal col As Integer) Implements                _
                              Syncfusion.Calculate.ICalcData.SetValueRowCol

        If row = rowCount Then
            colSums(col - 1) = value
        Else
            If col = colCount Then
                rowSums(row - 1) = value
            Else
                Me.dValues(row - 1, col - 1) = Double.Parse(value.ToString())
            End If
        End If
```



```
        Dim e1 As New ValueChangedEventArgs(row, col, value.ToString())
        RaiseEvent ValueChanged(Me, e1)

' SetValueRowCol
End Sub
```

10. You have to add an indexer definition to the class for two reasons: it is a straight-forward way to force the user to go though the GetValueRowCol and SetValueRowCol methods, allowing the CalcEngine to monitor these changes. It also makes your ArrayCalcData class look like an array.

Here is the code you must use. It is just a Get and Set implementation that goes through the **ICalcData** interface methods.

[C#]

```
/// <summary>
/// Gets / sets the value of this ArrayCalcData object.
/// </summary>
/// <remarks>
/// Since this class wraps a double array, the indexing is zero-based but,
/// the ICalcData methods requires one-based indexing by convention. So,
/// one is added to the indexes when the ICalcData methods are called.
/// </remarks>
public object this[int row, int col]
{
        get{return GetValueRowCol(row + 1, col + 1);}
        set{SetValueRowCol(value, row + 1, col + 1);}
}
```

[VB]

```
'/ <summary>
'/ Gets / sets the value of this ArrayCalcData object.
'/ </summary>
'/ <remarks>
'/ Since this class wraps a double array, the indexing is zero-based but,
'/ the ICalcData methods requires one-based indexing by convention. So,
'/ one is added to the indexes when the ICalcData methods are called.
'/ </remarks>
Default Public Property Item(ByVal row As Integer, ByVal col As Integer) As
Object
        Get
            Return GetValueRowCol(row + 1, col + 1)
        End Get
        Set(ByVal Value As Object)
```



```
            SetValueRowCol(Value, row + 1, col + 1)
        End Set
End Property
```

11. Your Generate Data button handler will create a random double array and populate it with random double values. It also creates an instance of an ArrayCalcData class to wrap the double array that it creates. It then creates a CalcEngine object that uses this ArrayCalcData object as its datasource. The **ShowObject** method will just display the values from your ArrayCalcData object in the multiline text box that you added to the form.

The engine.UseDependencies property will tell the CalcEngine that you want it to track dependencies so the value will automatically recalculate when dependent values change. The engine.RecalculateRange call allows the CalcEngine to traverse all the data and set up the necessary dependencies to do the calculations.

Here is the code.+--+

```
[C#]

using Syncfusion.Calculate;

//. . .

private Random r = new Random();
private ArrayCalcData data;
int nRows;
int nCols;

/// <summary>
/// Populates the double array.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void button1_Click(object sender, System.EventArgs e)
{
        // Creates some sample data.
        this.nRows = r.Next(10) + 2;
        this.nCols = r.Next(3) + 1;
        double[,] a = new double[nRows,nCols];
        for(int row = 0; row < nRows; ++ row)
                for(int col = 0; col < nCols; ++col)
                        a[row, col] = ((double)r.Next(100)) / 10;

        // Creates an ArrayCalcData object and passes it in this array.
        this.data = new ArrayCalcData(a);
```

```
        // Creates a CalcEngine object using this ArrayCalcData object.
        CalcEngine engine = new CalcEngine(this.data);

        // Turns on dependency tracking so that values set with the Set
button take effect immediately.
        engine.UseDependencies = true;

        // Calls RecalculateRange so any formulas in the data can be
initially computed.
        engine.RecalculateRange(RangeInfo.Cells(1, 1, nRows + 1, nCols + 1),
data);

        ShowObject();
}

/// <summary>
/// Displays the ArrayCalcData values in a text box.
/// </summary>
private void ShowObject()
{
        this.textBox1.Text = "";
        for(int i = 0; i <= this.nRows; ++i)
        {
                for(int j = 0; j <= this.nCols; ++j)
                {
                        this.textBox1.Text += this.data[i, j].ToString() +
"\t";

                }
                this.textBox1.Text += "\r\n";
        }
}
```

```
[VB]

Imports Syncfusion.Calculate

'...

Private r As New Random
Private data As ArrayCalcData
Dim nRows As Integer
Dim nCols As Integer

'/ <summary>
'/ Populates the double array.
```



```
'/ </summary>
'/ <param name="sender"></param>
'/ <param name="e"></param>
Private Sub button1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) _
                                                Handles Button1.Click
        ' Creates some sample data.
        Me.nRows = r.Next(10) + 2
        Me.nCols = r.Next(3) + 1
        Dim a(nRows, nCols) As Double
        Dim row As Integer
        For row = 0 To nRows - 1
            Dim col As Integer
            For col = 0 To nCols - 1
                a(row, col) = CDbl(r.Next(100)) / 10
            Next
        Next

        'Creates an ArrayCalcData object and passes it in this array.
        Me.data = New ArrayCalcData(a)

        ' Creates a CalcEngine object using this ArrayCalcData object.
        Dim engine As New CalcEngine(Me.data)

        ' Turns on dependency tracking so that values set with the Set button
take effect immediately.
        engine.UseDependencies = True

        ' Calls the RecalculateRange so any formulas in the data can be
initially computed.
        engine.RecalculateRange(RangeInfo.Cells(1, 1, nRows + 1, nCols + 1),
data)

        ShowObject()

' Button1_Click
End Sub


'/ <summary>
'/ Displays the ArrayCalcData values in a text box.
'/ </summary>
Private Sub ShowObject()
        Me.TextBox1.Text = ""
        Dim i As Integer
        For i = 0 To Me.nRows
```



```
            Dim j As Integer

            For j = 0 To Me.nCols
                Me.TextBox1.Text += Me.data(i, j).ToString() +
ControlChars.Tab
            Next
            Me.TextBox1.Text += ControlChars.Cr + ControlChars.Lf
        Next

' ShowObject
End Sub
```

Here is a typical display that you might see if you run the application at this point. Recall that the data is random so you may see fewer or more rows and columns. The column on the right is the sum of the columns that precedes it in the same row. The row at the bottom is the sum of the columns above it. You can click the Generate Data button several times to see different data.

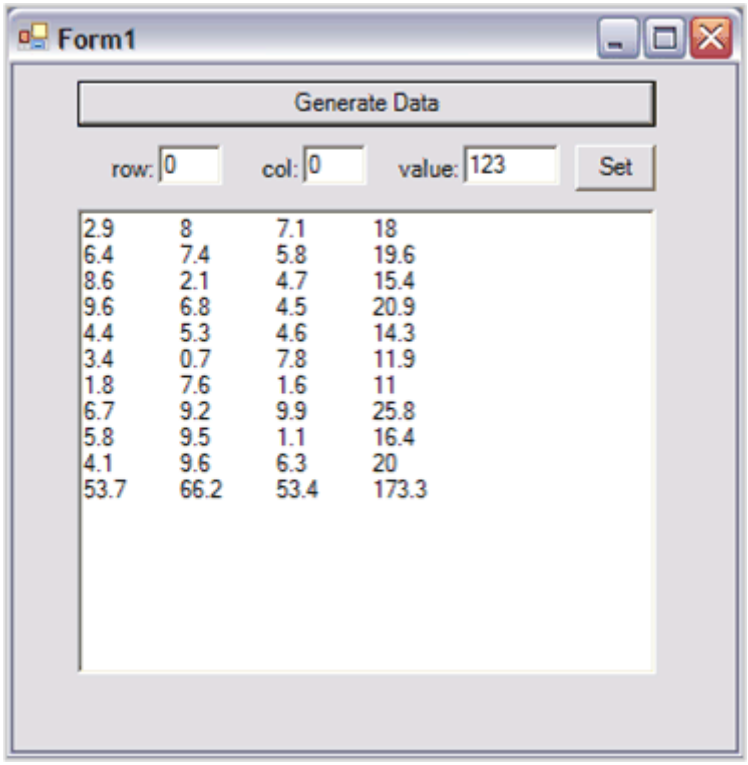


Figure 27: Sample Display Showing Double Array, SumColumn and SumRow

12. The Set button will allow you to set a particular value in the displayed data so you can see the effect of changing this value on the calculations in the last row and last column. Recall that your data store is mimicking an array of doubles, so it indexes from zero even though the ICalcData interface expects one-based indexing. The implementation code takes this into account.

13. Here is the code you must use. It will allow you to specify the row, column and value. Note there is no error checking code shown here, so enter only values that make sense for the displayed data.

[C#]

```
/// <summary>
/// Populates a single entry in the ArrayCalcData object.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void button2_Click(object sender, System.EventArgs e)
{
        if (this.nRows == 0)
        {
                MessageBox.Show("Generate data first.");
                return;
        }
        int row = int.Parse(this.textBox2.Text);
        int col = int.Parse(this.textBox3.Text);
        string val = this.textBox4.Text;
        this.data[row, col] = val;

        ShowObject();
}
```

[VB]

```
'/ <summary>
'/ Populates a single entry in the ArrayCalcData object.
'/ </summary>
'/ <param name="sender"></param>
'/ <param name="e"></param>
Private Sub button2_Click(ByVal sender As Object, ByVal e As
System.EventArgs) _
                                                Handles Button2.Click
        If Me.nRows = 0 Then
            MessageBox.Show("Generate data first.")
            Return
        End If

        Dim row As Integer = Integer.Parse(Me.textBox2.Text)
        Dim col As Integer = Integer.Parse(Me.textBox3.Text)
        Dim val As String = Me.textBox4.Text

        Me.data(row, col) = val
```



```
        ShowObject()

' Button2_Click
End Sub
```

On the previous screen, if you click the Set button, it will place 123 in cell 0,0. Notice the calculations update automatically to reflect the new value.

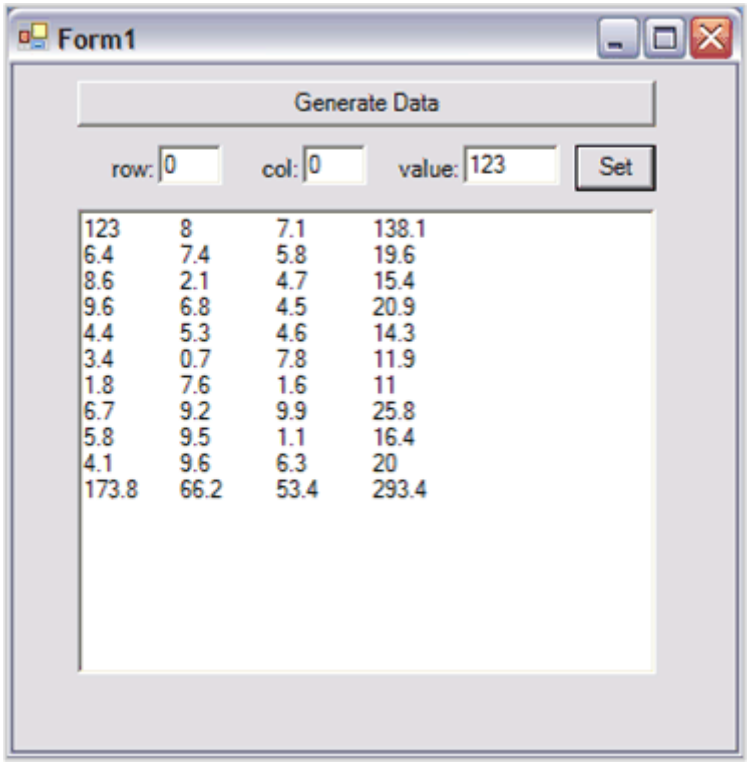


Figure 28: Sample Display After Setting 123 into Array Element (0, 0)

4 Concepts and Features

This section discusses the concepts and features of Essential Calculate. It includes the following topics.

4.1 Adding Calculation Support

Essential Calculate has a **CalcQuickBase** class that enables you to easily add formula parsing calculation support to arbitrary business objects. In addition, you can add calculation support to any data class by having that class implement the **ICalcData** interface.

The following sections will discuss this class and interface:

4.1.1 CalcQuickBase

The simplest way to use Essential Calculate is through an instance of its **CalcQuickBase** class. This class provides options to directly parse and compute a formula, or register variable names that can later be used in more complex formulas involving these variables. After registering the variables, it provides options to perform manual or automatic calculations.

This section discusses the usage of the CalcQuickBase class, under the following topics:

4.1.1.1 Manual Calculations

Manual calculations requires you to explicitly request Essential Calculate to compute the value and return it. You can do this by invoking methods in the CalcQuickBase class. There are several methods available which are discussed under the following topics:

- **ParseAndCompute**-This method will accept a formula string, parse it, and then return the computed value of the parsed formula. You can also directly invoke computation methods for any of the library functions of Essential Calculate through the **CalcEngine** class.
- **Indexer** method by using Variables

4.1.1.1.1 ParseAndCalculate Method

If you have an algebraic expression that just contains constants and function library methods, the most straight forward way of using Essential Calculate is to invoke its ParseAndCalculate method. Using CalcQuickBase makes this very simple. Consider, for example, the below form with a text box and a button on it. When you click the button, the computed value of the formula is displayed in the text box.

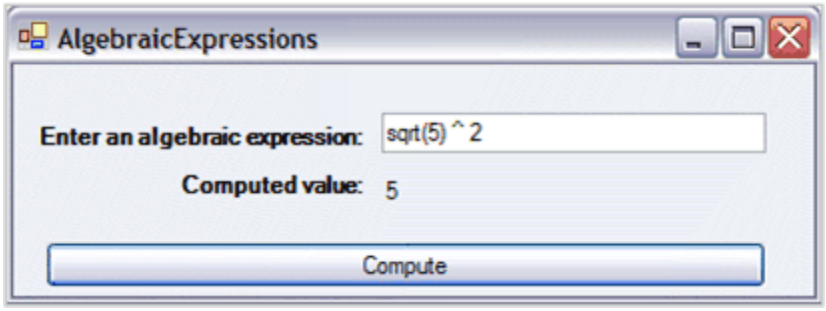


Figure 29: Simple Expression

The code that provides this functionality is very straight-forward. Add a reference to **Syncfusion.Calculate** in your project. Then instantiate a CalcQuickBase object, and invoke its ParseAndCalculate method from the button handler. Now you can type a formula in the text box and click the button to get the computed value. The following code illustrates this.

[C#]

```
using Syncfusion.Calculate;

private CalcQuickBase calculator = null;

private void Form1_Load(object sender, EventArgs e)
{
      this.calculator = new CalcQuickBase();
}

private void button1_Click(object sender, EventArgs e)
{
      string s = calculator.ParseAndCompute(this.textBox1.Text);
      this.label3.Text = s;
}
```

[VB]

```
Imports Syncfusion.Calculate
```



```
Private calculator As CalcQuickBase = Nothing

Private Sub Form1_Load(sender As Object, e As EventArgs)
      Me.calculator = New CalcQuickBase()

     ' Form1_Load
End Sub

Private Sub button1_Click(ByVal sender As Object, ByVal e As EventArgs)
      Dim s As String = calculator.ParseAndCompute(Me.textBox1.Text)
      Me.label3.Text = s

' Button1_Click
End Sub
```

In this discussion, it is assumed that the formulas involved contain only constants and library references. On the other hand, you can use the ParseAndCompute method to explicitly parse and calculate formulas that use variables as well. But, before you do that, you need to know how to register variables and assign values to them.

4.1.1.1.2 Indexer Method using Variables

In this section, you will learn how to use variable names within formulas to represent particular values. A variable name must begin with an alphabetical character and can contain only letters and digits. It is not case-sensitive. To register a string as a variable name and set its value is a single step operation, you must simply index the **CalcQuickBase** object with the name and assign the value to it.

[C#]

```
this.calculator["base"] = 3;
this.calculator["height"] = 2.5;
```

[VB]

```
Me.calculator("base") = 3
Me.calculator("height") = 2.5
```

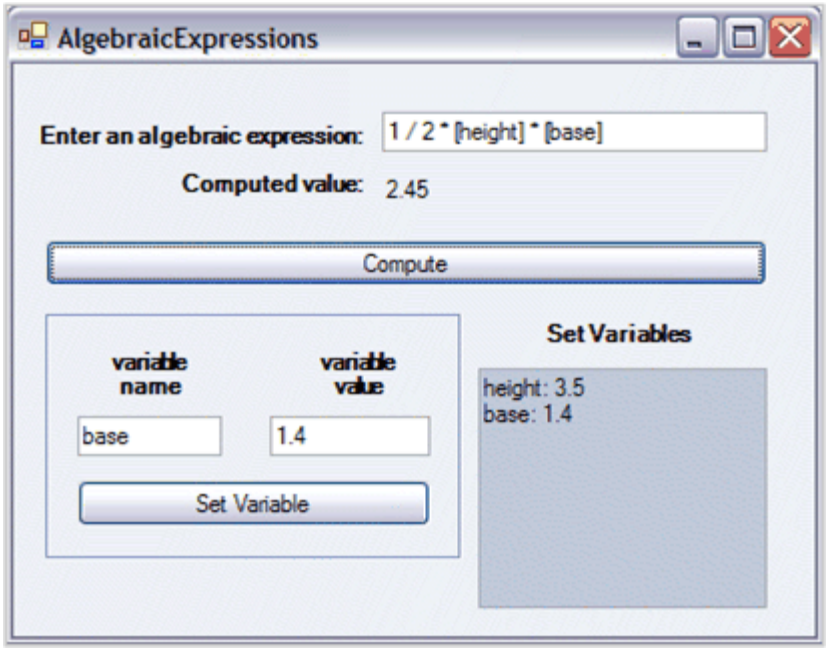



Figure 30: Simple Variables

When a name is used as an indexer on the CalcQuickBase object, the object checks a collection of variables that it maintains.

- If the name is not in the collection, it is added to the collection as a new item with the assigned value.
- If the name is already in the collection, its assigned value is changed to the new value.

Then when a formula is **parsed**, the CalcQuickBase object replaces all occurrences of variable names with their current values. To use a named variable in a formula, you must enclose it within brackets, as shown in the following formula:

[base]*[height]/2

The preceding formula takes the value represented by the base and multiplies it by the height of the value divided by two.

As a convention, if you want a variable to actually hold a string that is a formula and be treated as a formula, so that it is parsed and computed through the indexing code, then begin that string with a special character, the **CalcQuickBase.FormulaCharacter** (CalcEngine.FormulaCharacter). The default value of this character is "=". If you invoke the **ParseAndCompute** directly, any string you pass will be treated as a formula, whether or not it begins with FormulaCharacter.

4.1.1.1.2.1 The FormulaInfo Class

The collection that **CalcQuickBase** maintains in order to hold information on variables, is a collection of FormulaInfo objects. The **FormulaInfo** class has the following public properties.

- **FormulaText**-String holding the formula as originally entered
- **ParsedFormula**-String holding the **parsed** version of the formula1
- **FormulaValue**-String holding the last computed value for the formula

Indexing is an instance of the CalcQuickBase class, which sets the **FormulaText** property and gets the **FormulaValue** property for the FormulaInfo object that is associated with the variable name and used as the indexer. A FormulaInfo object is dynamically created, if you use a variable name that is not in the **CalcQuickBase FormulaInfo** collection. To retrieve FormulaText, you must use the **CalcQuickBase.GetFormula** method and pass it in the variable name.

While using an indexer to get a value from a CalcQuickBase object, the FormulaValue property is returned. So, the question arises, as to exactly "when" this FormulaValue is computed from the FormulaText that has been set into this FormulaInfo object. When a new value for FormulaValue is computed, it is controlled by an internal member of FormulaInfo, the calcID. The CalcQuickBase object maintains a calcID value as well. Whenever the FormulaInfo.FormulaValue is requested, the CalcQuickBase.calcID is compared to the FormulaInfo.calcID, and if they do not match, the FormulaInfo.FormulaValue is recomputed and its FormulaInfo.calcID is set to match the CalcQuickBase.calcID. So, FormulaValue is only computed when the calcID value does not match the CalQuick.calcID value. Also, you can force new computations by calling the **SetDirty** method on your CalcQuickBase instance.

The actual collection of FormulaInfo objects (and some related collections) are protected members of the CalcQuickBase class. In order to access the objects of these collections directly, you must derive the CalcQuickBase class of Essential Calculate. The Calculate class reference has more information on these protected collections.

You can access the underlying Calculate.Engine object through the public read-only **Engine** property of the CalcQuickBase class. You can then use this Engine property to add custom functions to the Function Library that is available for the CalcQuickBase object.

4.1.1.2 Automatic Calculations

Essential Calculate enables you to monitor value changes, and then automatically recomputes formulas that depend upon the changed values. There is an additional overhead associated with automatic calculations that enables you to determine "when" you want to use this feature.

4.1.1.2.1 Using Explicit Events

By default, CalcQuickBase does not try to track any dependencies among the variables you set. Thus, if you have a formula like

"=[TestValue]+2", the computed value of this formula will not change automatically if you modify your TestValue variable. To enable automatic recalculation of dependent variables, you must set the CalcQuickBase.AutoCalc property to True. Once this is done, the CalcQuickBase object (through its embedded CalcEngine object) maintains the required dependency information.

In practice, some additional work needs to be done. When a variable is auto-changed, nothing will actually happen until you try to use it. For example, assume that you have a series of text boxes on a form with some of the text boxes holding numerical values and some text boxes holding formulas that reference these values through variables that you have registered with a CalcQuickBase object.

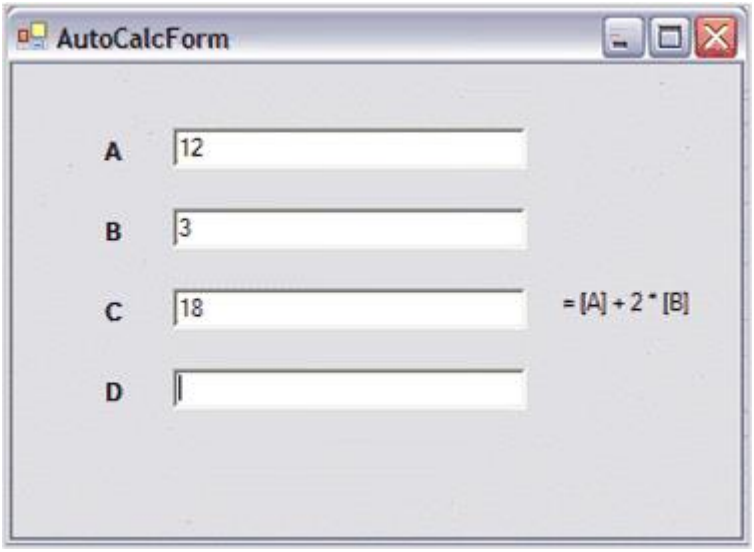


Figure 31: AutoCalc

In the above screenshot, Text Box C is set to a formula that references the values from Text Box A and Text Box B. So, once the value in Text Box A or Text Box B changes, the value in Text Box C should also change.

In order to get this to work, two things must be done. First, when the variable registered as A or B is modified, your code must set the new values in the CalcQuickBase object by using indexers. The reason for you do this is, so that CalcQuickBase has no knowledge of Text Box A and Text Box B. It knows about the variables A and B which you have registered through indexing, but it has no knowledge that these values are actually coming from particular text boxes. After you have set the new values using indexers, the CalcQuickBase object knows that variable C will change and modify its value. But, the CalcQuickBase object that modifies its value for variable C has no effect on the text box that holds C. This is the point where your code needs to play its second role. You need to get the value of variable C and put it into the appropriate text box. The common question is, "How does the code know that it has to retrieve C so that it can update Text Box C appropriately?" This is made possible as the CalcQuickBase object raises an event, **CalcQuickBase.ValueSet**, everytime a value that it is tracking is modified. So, your code can listen for this event and set the proper values.

The following code illustrates the above process.

[C#]

```
private CalcQuickBase calculator = null;

private void Form_Load(object sender, System.EventArgs e)
{

        // 1) Instantiate a CalcQuickBase object.
        calculator = new CalcQuickBase();

        // 2) Populate your controls.
        this.textBoxA.Text = "12";
        this.textBoxB.Text = "3";
        this.textBoxC.Text = "= [A] + 2 * [B]";

        // Must enter formula names before turning on calculations.
        // 3) Assign formula object names.
        calculator["A"] = this.textBoxA.Text;
        calculator["B"] = this.textBoxB.Text;
        calculator["C"] = this.textBoxC.Text;
        calculator["D"] = this.textBoxD.Text;

        // 4) Turn on auto calculations.
        this.calculator.AutoCalc = true;

        // 5) Subscribe to the event to set newly calculated values.
        this.calculator.ValueSet += new
QuickValueSetEventHandler(calculator_ValueSet);

        // 6) Subscribe to some events to trigger the setting of values into
CalcQuickBase.
        this.textBoxA.Leave +=new EventHandler(textBoxA_Leave);
        this.textBoxB.Leave +=new EventHandler(textBoxB_Leave);
```



```
        this.textBoxC.Leave +=new EventHandler(textBoxC_Leave);
        this.textBoxD.Leave +=new EventHandler(textBoxD_Leave);

         // 7) Allow the CalcQuickBase sheet to create dependency lists.
        // Necessary for auto-calculations.
        this.calculator.RefreshAllCalculations();
}

// 8) Raised when a variable value is calculated.
private void calculator_ValueSet(object sender, OuickValueSetEventArgs e)
{
        switch(e.Key)
        {
                case "A":
                        this.textBoxA.Text = this.calculator[e.Key].ToString();
                        break;
                case "B":
                        this.textBoxB.Text = this.calculator[e.Key].ToString();
                        break;
                case "C":
                        this.textBoxC.Text = this.calculator[e.Key].ToString();
                        break;
                case "D":
                        this.textBoxD.Text = this.calculator[e.Key].ToString();
                        break;
                default:
                        break;
        }
}

// 9) Handles the changing of the text in the text box so the CalQuick object
can be updated as the text changes.
private void textBoxA_Leave(object sender, EventArgs e)
{
        if(this.textBoxA.Modified)
        {
                calculator["A"] = this.textBoxA.Text;
                this.textBoxA.Modified = false;
        }
}
// ..... same for textBoxB_Leave, textBoxC_Leave, textBoxD_Leave
```

[VB]

```
Private calculator As CalcQuickBase = Nothing


Private Sub Form_Load(sender As Object, e As System.EventArgs)
        ' 1) Instantiate a CalcQuickBase object.
        calculator = New CalcQuickBase()
```



```
        ' 2) Populate your controls.
        Me.textBoxA.Text = "12"
        Me.textBoxB.Text = "3"
        Me.textBoxC.Text = "= [A] + 2 * [B]"

        ' Must enter formula names before turning on calculations.
        ' 3) Assign formula object names.
        calculator("A") = Me.textBoxA.Text
        calculator("B") = Me.textBoxB.Text
        calculator("C") = Me.textBoxC.Text
        calculator("D") = Me.textBoxD.Text

        ' 4) Turn on auto calculations.
        Me.calculator.AutoCalc = True

        ' 5) Subscribe to the event to set newly calculated values.
        AddHandler Me.calculator.ValueSet, AddressOf calculator_ValueSet

        ' 6) Subscribe to some events to trigger the setting of values into
CalcQuickBase.
        AddHandler Me.textBoxA.Leave, AddressOf textBoxA_Leave
        AddHandler Me.textBoxB.Leave, AddressOf textBoxB_Leave
        AddHandler Me.textBoxC.Leave, AddressOf textBoxC_Leave
        AddHandler Me.textBoxD.Leave, AddressOf textBoxD_Leave

        ' 7) Allow the CalcQuickBase sheet to create dependency lists necessary
for auto calculations.
        Me.calculator.RefreshAllCalculations()

' Form_Load
End Sub

' 8) Raised when a variable value is calculated.
Private Sub calculator_ValueSet(ByVal sender As Object, ByVal e As
OuickValueSetEventArgs)
        Select Case e.Key
            Case "A"
                Me.textBoxA.Text = Me.calculator(e.Key).ToString()
            Case "B"
                Me.textBoxB.Text = Me.calculator(e.Key).ToString()
            Case "C"
                Me.textBoxC.Text = Me.calculator(e.Key).ToString()
            Case "D"
                Me.textBoxD.Text = Me.calculator(e.Key).ToString()
            Case Else
        End Select

' Calculator_ValueSet
End Sub
```



```
' 9) Handles the changing of the text in the text box so the CalcQuickBase
object can be updated as the text changes.
Private Sub textBoxA_Leave(ByVal sender As Object, ByVal e As EventArgs)
        If Me.textBoxA.Modified Then
            calculator("A") = Me.textBoxA.Text
            Me.textBoxA.Modified = False
        End If

' TextBoxA_Leave
End Sub
```

The following is an explanation of the numbered steps given in the preceding Form_Load.

1. Instantiates the CalcQuickBase instance.

2. Sets some initial text into the first three text boxes. The first two values are just numerical entries but, the third value will be treated as a formula by the CalcQuickBase object as it begins with a "=".

3. This step registers the variable names A, B, C and D with the CalcQuickBase object, and sets the initial values of these variables to the contents of the text boxes.

4. Here the **AutoCalc** property is turned on so CalcQuickBase will start tracking any dependencies that it notes among the variables registered with it. Note that this step was done after the initial registering of the variables in step 3. So, any relations among the registered variables are unknown to the CalcQuickBase object. This shortcoming will be addressed in step 7 and the rationale for this order will be discussed there.

5. Subscribe to CalcQuickBase's ValueSet event so that the code can react to any automatic changing of the registered variables' values and place them into the appropriate text box so your display will immediately reflect any change.

6. Subscribe to the text box events so that you can update the CalcQuickBase object when the text in a text box has changed.

7. This step forces the recalculation of all variables registered with the CalcQuickBase object. This has to be done after the AutoCalc property has been set to **True**, so that the dependencies between variables can be monitored. The reason to postpone setting AutoCalc until after the initial registration of the variables, is to avoid problems that might occur because of CalcQuickBase trying to set up dependency chains even before all the variables have been registered. Initializing the variables, turning on AutoCalc, and then calling **RefreshAllCalculations**, avoids this potential problem.

8. This is the event handler that moves a freshly computed variable into the text box that it is related to.

9. These four event handlers signal when the user leaves a modified text box. At that point, the CalcQuickBase object is updated to reflect the new value that has been entered by the user.

4.1.1.2.2 Using RegisterControlArray

Using explicit events to manage the auto-calculation in **CalcQuickBase** used with controls on a form is straight-forward enough but, does require subscribing to multiple events and writing code in the handlers. **CalcQuickBase.RegisterControlArray** will handle all this work for you and will streamline adding calculation support to a form or **UserControl**. There are two assumptions on the controls to which you want to bind the calculations. The first assumption is that the control is either a text box or a combo box. (To support other controls, you will have to derive CalcQuickBase and override RegisterControl). The second assumption is that the variable name that you want to use to represent the control value in formulas is Control.Name.

Here is the code that will do exactly the same work as our previous example by using explicit events to support auto-calculation. Notice that all the event handling has been removed. There are only three steps that are related to adding the calculation support, which includes instantiating the CalcQuickBase object, and calling the **RegisterControlArray** and **RefreshAllCalculations** methods.

```
[C#]

CalcQuickBase calculator = null;

private void Form1_Load(object sender, System.EventArgs e)
{
        // 1) Make sure controls have the names you want to use as variables.
This can be done either
        //   from code as here or from the designer.
        this.textBoxA.Name = "A";
        this.textBoxB.Name = "B";
        this.textBoxC.Name = "C";
        this.textBoxD.Name = "D";

        // 2) Initially populate the controls. Again, this can be done from the
designer.
        this.textBoxA.Text = "12";
        this.textBoxB.Text = "3";
        this.textBoxC.Text = "= [A] + 2 * [B]";

        // 3) Instantiate a CalcQuickBase object.
        calculator = new CalcQuickBase();

        // 4) Register the controls used in calculations. The formula variables
used are the Control.Name strings.
        this.calculator.RegisterControlArray(new Control[]
        {
```

```
                this.textBoxA,
                this.textBoxB,
                this.textBoxC,
                this.textBoxD
        });

         // 5) Allow the CalcQuickBase object to create dependency lists among
the formula variables necessary
        //   for autocalculations and do the initial computations.
        this.calculator.RefreshAllCalculations();
}
```

[VB]

```
' 1) Make sure controls have the names you want to use as variables. This can
be done either
' from code as here or from the designer.
Me.textBoxA.Name = "A"
Me.textBoxB.Name = "B"
Me.textBoxC.Name = "C"
Me.textBoxD.Name = "D"

' 2) Initially populate the controls. Again, this can be done from the
designer.
Me.textBoxA.Text = "12"
Me.textBoxB.Text = "3"
Me.textBoxC.Text = "= [A] + 2 * [B]"

' 3) Instantiate a CalcQuickBase object.
calculator = New CalcQuickBase

' 4) Register the controls used in calculations. The formula variables used are
the Control.Name strings.
Me.calculator.RegisterControlArray(New Control() {Me.textBoxA, Me.textBoxB,
Me.textBoxC, Me.textBoxD})

' 5) Allow the CalcQuickBase object to create dependency lists among the
formula variables necessary for autocalculations and do the initial
computations.
Me.calculator.RefreshAllCalculations()
```

The following is an explanation of the numbered steps in the preceding Form_Load.

1. Ensures that variable names are set as desired.
2. Sets the initial text into the first three text boxes.
3. Instantiates the CalcQuickBase instance.

4. Calls the RegisterControlArray method. In this method, the CalcQuickBase object will handle all the event code that you have manually added in the previous section. It does make the assumption that the controls are either text boxes or combo boxes with the appropriate names.

5. This step forces the recalculation of all variables that are registered with the CalcQuickBase object.

4.1.1.3 Resetting Keys by using Calculate Engine

This method provides support for resetting keys (which happens backend) using Calculate Engine.

The user can reset or clear the keys by using this method.

Samples Installation Location:

CalcQuick WF samples are installed under the following location:

C:\Syncfusion\EssentialStudio\<Version Number>\Windows\Calculate.Windows\Samples \2.0\Working With CalcQuick Demo

Viewing Samples:

1. Follow steps 1 to 2 of viewing Windows samples in section **2.2 Samples and Installation**.

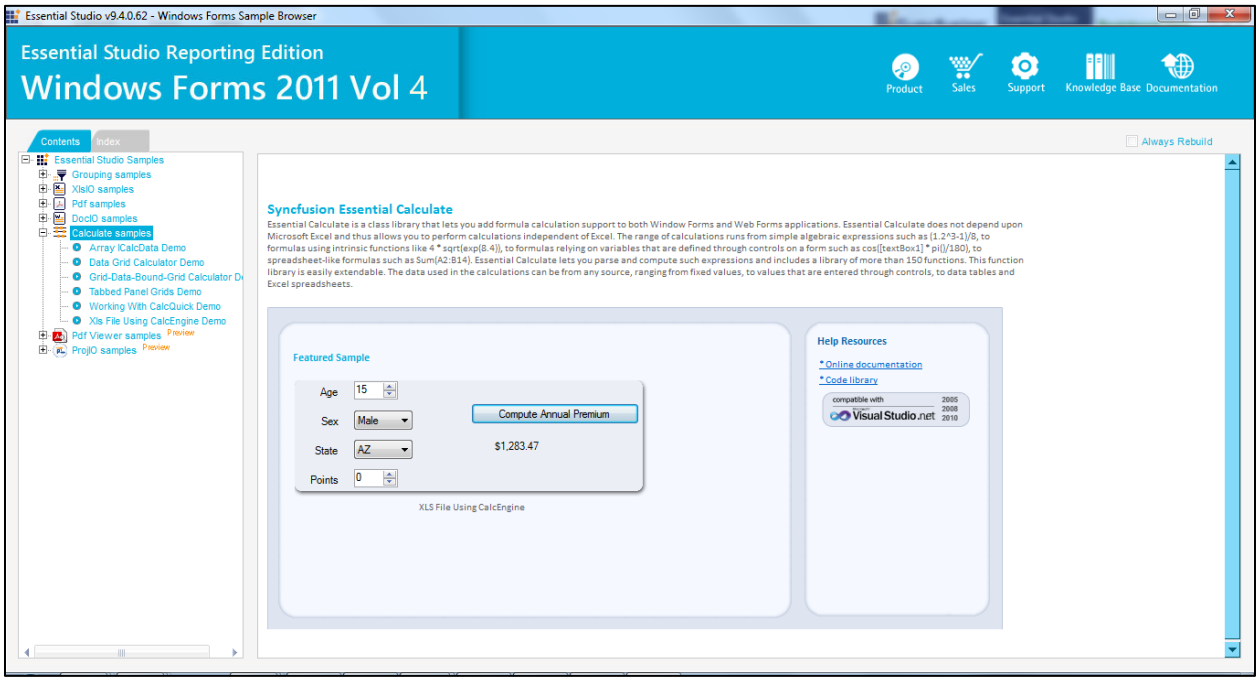


Figure 32: WF Edition Sample Browser

2. Select **Working With CalcQuick Demo** from the samples provided and browse through the features.

Figure 33: Working With CalcQuick Demo

4.1.1.3.1 Methods

Name	Description
ResetKeys()	Clears the keys used by the Calculate engine

4.1.1.4 Summary

CalcQuickBase is the simplest way to add calculation support to your code. You can create an instance of it, and then just start by using it through either its ParseAndCompute method, or by indexing it with a variable name. You can use CalcQuickBase either in manual calculation mode or in an automatic calculation mode. Automatic calculations will require you to either handle certain events or use the RegisterControlArray method for Windows Forms text box and combo box controls.

4.1.2 General Calculation Support - ICalcData

Essential Calculate enables you to add calculation support to arbitrary business objects through its ICalcData interface. In this section, you will learn how to define this interface and use it with a Windows Forms DataGrid.

4.1.2.1 The ICalcData Interface

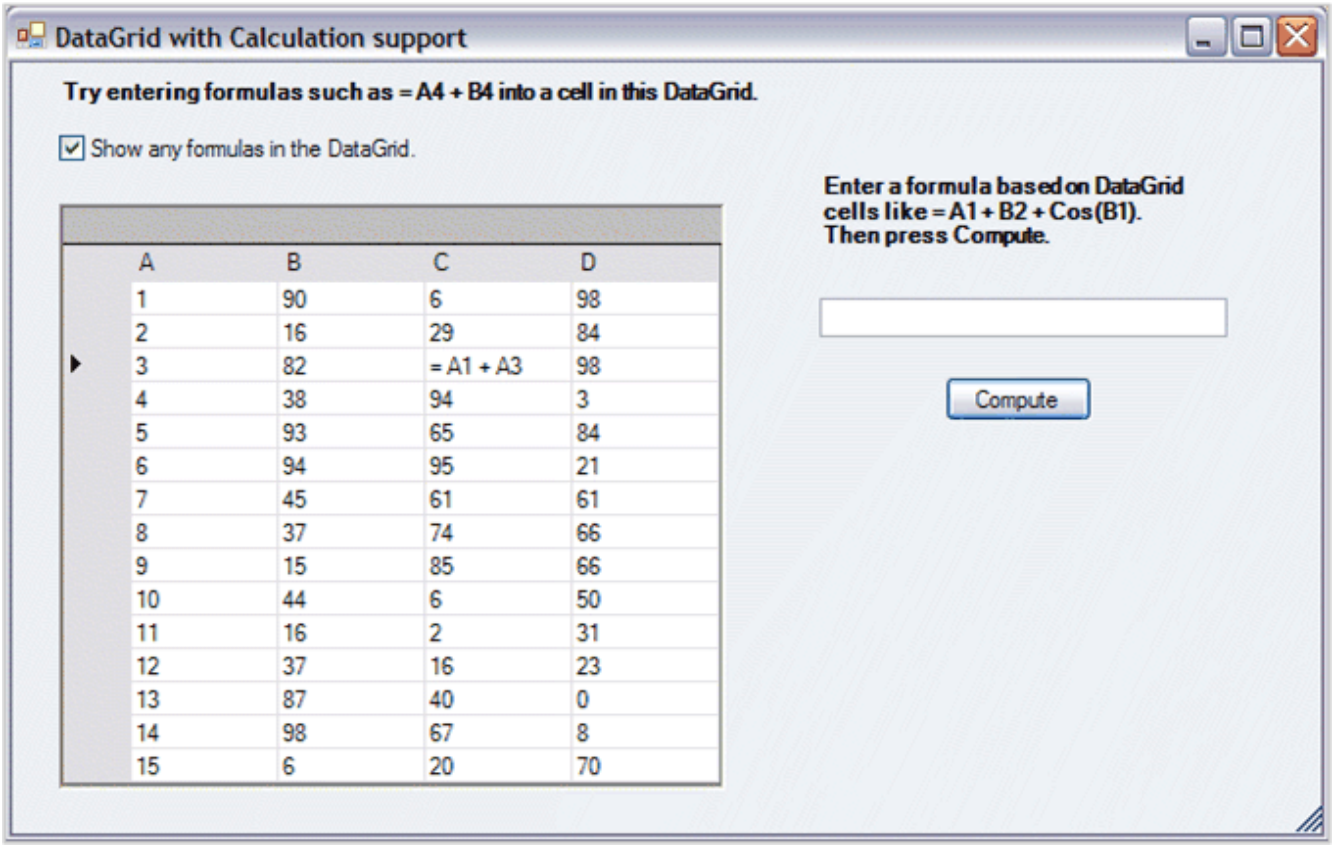
ICalcData has three methods and one event. This interface allows the **CalcEngine** class in Essential Calculate to communicate with arbitrary data sources that implement this interface.

- **GetValueRowCol**-Returns the data value of a specified row and column
- **SetValueRowCol**-Sets the data value of a specified row and column
- **WireParentObject**-A callback to the data object that occurs as the CalcEngine is being created. The purpose is to give the data object a chance to do any initialization steps it may need, such as subscribe to events to handle changes in data notifications.
- **ValueChanged**-An event that is raised whenever data changes. The ICalcData implementer raises this event when the data changes. The CalcEngine listens to this event and accordingly reacts to data changes. It is through this event that formulas are processed and dependencies are tracked by the CalcEngine.

4.1.2.2 Working with System.Windows.Forms.DataGrid

A Windows Forms Data Grid is a rectangular container that holds data in cells. Such a container is a natural medium for using calculation support. To add Essential Calculation support to classes that represent data in a row/column format like a Data Grid, you will have to derive the class and implement the ICalcData interface. This interface contains three methods and one event. Once you add the appropriate interface implementation, your derived object will have formula support.

The following is a discussion of using Essential Calculate with a Data Grid as an ICalcData object is based on the Essential Studio\Windows\Calculation.Windows\Samples\DataGridCalculator sample that ships with the product. The sample has a derived DataGrid class that implements ICalcData. Below is a screen shot of a sample screen. The sample sets the column header text to A, B, and so on, and places 1, 2, and so on, in the first column along with random integers in the other columns. This is done to remind you of the Excel-like cell notation of A1, A2, B2, and so on. This is the notation supported by Essential Calculate formulas using ICalcData objects.



	A	B	C	D
1	90	6	98
2	16	29	84
3	82	= A1 + A3	98
4	38	94	3
5	93	65	84
6	94	95	21
7	45	61	61
8	37	74	66
9	15	85	66
10	44	6	50
11	16	2	31
12	37	16	23
13	87	40	0
14	98	67	8
15	6	20	70

Figure 34: Data Grid

You can copy the code that defines the derived DataGrid object to your projects and have immediate support for calculations in a Data Grid using **Data Table** data sources. Before we begin with the details of this sample and the derived DataGrid class, we will discuss the ICalcData interface describing the purpose of its implementation details.

4.1.2.2.1 Using CalcDataGrid as a Single Spreadsheet

Essential Calculate provides a derived DataGrid object that implements ICalcData to support calculations. But you need to know how to use such an object on a **User Control**. Essentially, you can drop an instance of the CalcDataGrid object onto your form, create an instance of CalcEngine by using your CalcDataGrid as its ICalcData object and then calculate things so that the initial display shows the calculated values. The following code illustrates this.

```
[C#]

private Syncfusion.Calculate.CalcEngine engine;
private DataTable dt;
```



```
private void SingleDataGridForm_Load(object sender, System.EventArgs e)
{

        // Set up your DataTable.
        this.dt = GetTheDataTable();

        // Set the datasource to a DataTable.
        this.dataGrid1.DataSource = this.dt;

        // Set any formulas you want
        // or they might already be in the DataTable.
        this.dataGrid1[2,2] = "= A1 + A3";

        // 1) Reset static members of CalcEngine.
        Syncfusion.Calculate.CalcEngine.ResetSheetFamilyID();

        // 2) Create a CalcEngine object and tie it to the DataGrid that
implements ICalcData.
        engine = new Syncfusion.Calculate.CalcEngine(this.dataGrid1);

        // 3) Set the CalcEngine to track dependencies required for auto
updating.
        engine.UseDependencies = true;

        // 4) Call RecalculateRange so any formulas in the data can be
initially computed.
        engine.RecalculateRange(RangeInfo.Cells(1, 1, dt.Rows.Count,
dt.Columns.Count), this.dataGrid1);
}
```

```
[VB]

Private engine As Syncfusion.Calculate.CalcEngine
Private dt As DataTable

Private Sub SingleDataGridForm_Load(sender As Object, e As System.EventArgs)

        ' Set up your DataTable.
        Me.dt = GetTheDataTable()

       ' Set the datasource to a DataTable.
        Me.dataGrid1.DataSource = Me.dt

        ' Set any formulas you want
        ' or they might already be in the DataTable.
```

```
        Me.dataGrid1(2, 2) = "= A1 + A3"

        ' 1) Reset static members of CalcEngine.
        Syncfusion.Calculate.CalcEngine.ResetSheetFamilyID()

        ' 2) Create a CalcEngine object and tie it to the DataGrid that
implements ICalcData.
        engine = New Syncfusion.Calculate.CalcEngine(Me.dataGrid1)

        ' 3) Set the CalcEngine to track dependencies required for auto
updating.
        engine.UseDependencies = True

        ' 4) Call RecalculateRange so any formulas in the data can be
initially computed.
        engine.RecalculateRange(RangeInfo.Cells(1, 1, dt.Rows.Count,
dt.Columns.Count), Me.dataGrid1)

' SingleDataGridForm_Load
End Sub
```

The following is an explanation of the preceding code.

1. **ResetSheetFamilyID** clears any static members of the CalcEngine class and sets the engine state to operate with a single ICalcData object.

2. Creates an instance of the CalcEngine object.

3. Sets the engine object to track calculation dependencies so that cells can be automatically updated as other cell values change.

4. **RecalculateRange** goes through the existing cell contents and calculates any formulas for the initial display.

4.1.2.2.2 Using Several CalcDataGrids in a Workbook

Essential Calculate supports cross-references among several ICalcData objects. This support allows you to have a workbook of CalcDataGrids by using a Windows Forms Tab control. The following discussion is based on the Calculation\Samples\DataGridCalculator sample that ships with the product.

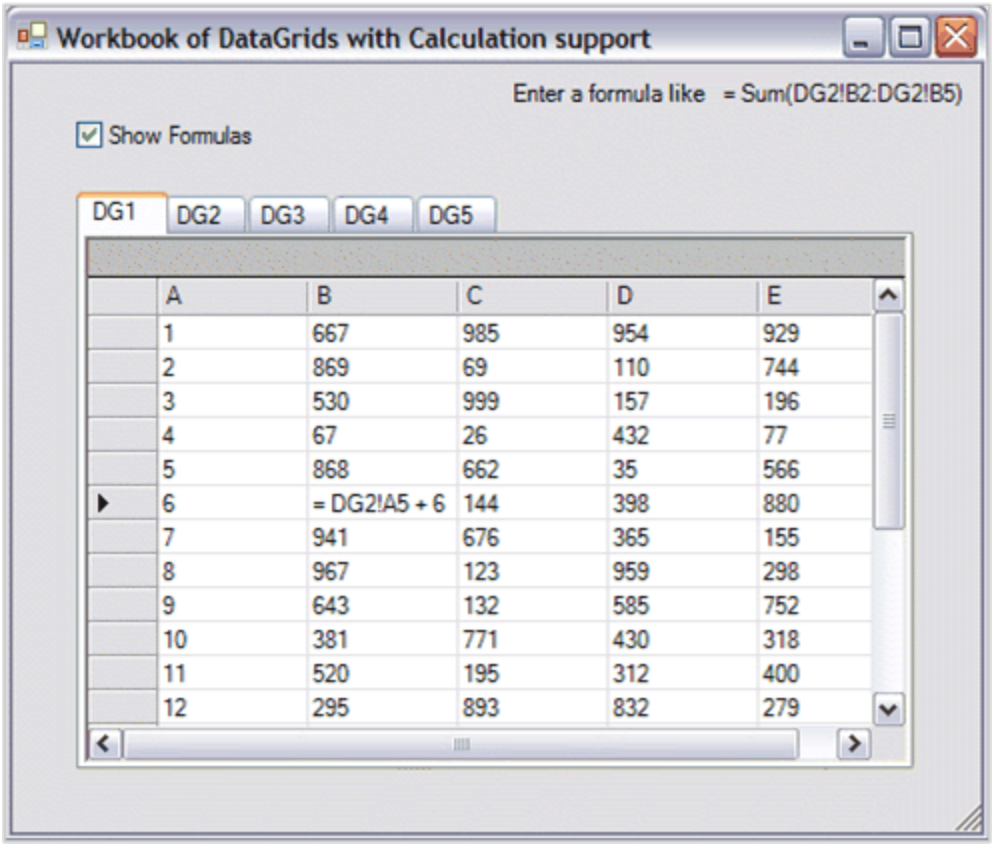


Figure 35: Workbook

To support cross-references among several ICalcData objects, you must register the objects with a single instance of the CalcEngine. Part of the information provided during registration are names associated with each ICalcData object. To reference a particular cell in an ICalcData object, you need to use its name along with the column and row to determine the proper reference. For example, say, in an ICalcData object whose name is Sales; To reference column 5, row 3, you must use Sales!E3. The name is separated from the column and row reference by an exclamation point. In the above screenshot, the formula = DG2!A5 + 6 would add 6 to the value in row 5, column 1 in sheet DG2.

Here is the FormLoad method for the sample screenshot depicted above. Using the designer, a TabControl is dropped on a form and five TabPages are added. The pages are named DG1, DG2, …, DG5. On each TabPage, a CalcDataGrid is set with the DockStyle set to **Fill**. This is all done through the designer.

```
[C#]

Syncfusion.Calculate.CalcEngine engine;

private void DataGridWorkBookForm_Load(object sender, System.EventArgs e)
```



```
{

        // 1) Set the datasource for the DataGrid.
        this.dataGrid1.DataSource = GetARandomTable();
        this.dataGrid2.DataSource = GetARandomTable();
        this.dataGrid3.DataSource = GetARandomTable();
        this.dataGrid4.DataSource = GetARandomTable();
        this.dataGrid5.DataSource = GetARandomTable();

        // 2) Call this to reset static members.
        Syncfusion.Calculate.CalcEngine.ResetSheetFamilyID();

        // 3) Create the engine.
        engine = new Syncfusion.Calculate.CalcEngine(this.dataGrid1);

        // 4) Track dependencies required for auto calculations.
        engine.UseDependencies = true;

        // 5) Register multiple ICalcData objects for cross sheet references.
        int sheetfamilyID = CalcEngine.CreateSheetFamilyID();
        engine.RegisterGridAsSheet("DG1", this.dataGrid1, sheetfamilyID);
        engine.RegisterGridAsSheet("DG2", this.dataGrid2, sheetfamilyID);
        engine.RegisterGridAsSheet("DG3", this.dataGrid3, sheetfamilyID);
        engine.RegisterGridAsSheet("DG4", this.dataGrid4, sheetfamilyID);
        engine.RegisterGridAsSheet("DG5", this.dataGrid5, sheetfamilyID);
}
```

```
[VB]

Dim engine As Syncfusion.Calculate.CalcEngine

Private Sub DataGridWorkBookForm_Load(sender As Object, e As
System.EventArgs)

        ' 1) Set the datasource for the DataGrid.
        Me.dataGrid1.DataSource = GetARandomTable()
        Me.dataGrid2.DataSource = GetARandomTable()
        Me.dataGrid3.DataSource = GetARandomTable()
        Me.dataGrid4.DataSource = GetARandomTable()
        Me.dataGrid5.DataSource = GetARandomTable()

        ' 2) Call this to reset static members in case the other form loaded
first.
        Syncfusion.Calculate.CalcEngine.ResetSheetFamilyID()

        ' 3) Create the engine.
```



```
        engine = New Syncfusion.Calculate.CalcEngine(Me.dataGrid1)

        ' 4) Track dependencies required for auto calculations.
        engine.UseDependencies = True

        ' 5) Register multiple ICalcData objects for cross sheet references.
        Dim sheetfamilyID As Integer = CalcEngine.CreateSheetFamilyID()
        engine.RegisterGridAsSheet("DG1", Me.dataGrid1, sheetfamilyID)
        engine.RegisterGridAsSheet("DG2", Me.dataGrid2, sheetfamilyID)
        engine.RegisterGridAsSheet("DG3", Me.dataGrid3, sheetfamilyID)
        engine.RegisterGridAsSheet("DG4", Me.dataGrid4, sheetfamilyID)
        engine.RegisterGridAsSheet("DG5", Me.dataGrid5, sheetfamilyID)

' DataGridWorkBookForm_Load
End Sub
```

The following is an explanation of the preceding code.

1. Assign the datasources to the DataGrids.

2. **ResetSheetFamilyID** clears any static members of the CalcEngine class and sets the engine state to operate with a single ICalcData object.

3. Creates an instance of the CalcEngine object.

4. Sets the engine object to track calculation dependencies so that cells can be automatically updated as other cell values change.

5. This is the code that registers a name for each ICalcData object so that the CalcEngine can support references across ICalcData objects.

4.1.2.3 Conventions

There are two conventions that are honored by Essential Calculate. While processing strings that are used as data values, any string beginning with "=" is treated as a formula that is to be parsed and evaluated. You can change the "=" to some other character by setting this static (Shared in VB) member, CalcEngine.FormulaCharacter. If you call Parse routines directly from code, the FormulaCharacter is optional.

The second convention involves zero-based and one-based indexing. It should be noted that a lot of data sources use zero-based indexing to access values, but in CalcEngine one-based indexing is used to mimic Excel. This leads to possible indexing conflicts. To keep things consistent and to make sure that it is clear what should be used, Essential Calculate expects any indexes (rows / column integer values) to be one-based. This means that you may have to tweak the indexes that are passed through the ICalcData methods and event arguments to make them consistent with any zero-based data sources that you might be using. One such example is the DataGrid discussed in this section. You can see the index-based adjustments in the following code samples.

```
[C#]

public class CalcDataGrid : DataGrid, Syncfusion.Calculate.ICalcData
{
        public CalcDataGrid() : base()
        {
                // Avoid the complexity of sorting.
                this.AllowSorting = false;
        }

// 1) Used to subscribe to the DataTable.ColumnChanged event. This
ColumnChanged event will raise the required ValueChanged event.

        // Without this ValueChanged event, the CalcEngine would have no
knowledge of the data.
        public void WireParentObject()
        {
                // Assumes grid's datasource is a DataTable.
                DataTable dt = this.DataSource as DataTable;
                dt.ColumnChanged += new
DataColumnChangeEventHandler(dt_ColumnChanged);

                // Avoids the complexity of a new row.
                dt.DefaultView.AllowNew = false;
        }

        // 2) This event handler raises the required ICalcData.ValueChanged
event when the data in the DataTable changes.
        private void dt_ColumnChanged(object sender,
DataColumnChangeEventArgs e)
        {
                CurrencyManager cm = (CurrencyManager)
this.BindingContext[this.DataSource, this.DataMember];
                DataTable dt = this.DataSource as DataTable;
                int pos = cm.Position;
                int field = dt.Columns.IndexOf(e.Column);
                string val = this[pos, field].ToString();
```

```
                // e1.RowIndex, e1.ColIndex needs to be one-based.
                Syncfusion.Calculate.ValueChangedEventArgs e1 = new
ValueChangedEventArgs(pos + 1, field + 1, val);
                ValueChanged(this, e1);
        }

        // 3) Returns the value at the one-based row and col.
        public object GetValueRowCol(int row, int col)
        {
                // Row, col are one-based.
                return this[row-1, col-1];
        }

        // 4) Sets the value at the one-based row and col.
        public void SetValueRowCol(object value, int row, int col)
        {
                // Row, col are one-based.
                DataTable dt = this.DataSource as DataTable;
                dt.Rows[row - 1][col - 1] = value;
        }

        // 5) Required ICalcData event.
        public event ValueChangedEventHandler ValueChanged;
}
```

[VB]

```
Public Class CalcDataGrid
Inherits DataGrid
Implements Syncfusion.Calculate.ICalcData

Public Sub New()
        ' Avoids the complexity of sorting.
        Me.AllowSorting = False
End Sub

'1) Used to subscribe to the DataTable.ColumnChanged event. This event will
raise the required ValueChanged event. Without this           ValueChanged
event, the CalcEngine would have no knowledge of the data.
Public Sub WireParentObject() Implements ICalcData.WireParentObject
        ' Assumes the grid's datasource is a DataTable.
        Dim dt As DataTable = Me.DataSource
        AddHandler dt.ColumnChanged, AddressOf dt_ColumnChanged

        ' Avoids the complexity of a new row.
```



```
        dt.DefaultView.AllowNew = False
End Sub

' 2) This event handler raises the required ICalcData.ValueChanged event when
the data in the DataTable changes.
Private Sub dt_ColumnChanged(ByVal sender As Object, ByVal e As
DataColumnChangeEventArgs)
        Dim cm As CurrencyManager = CType(Me.BindingContext(Me.DataSource,
Me.DataMember), CurrencyManager)
        Dim dt As DataTable = Me.DataSource
        Dim pos As Integer = cm.Position
        Dim field As Integer = dt.Columns.IndexOf(e.Column)
        Dim val As String = Me(pos, field).ToString()

        ' e1.RowIndex, e1.COlIndex needs to be one-based.
        Dim e1 = New ValueChangedEventArgs(pos + 1, field + 1, val)
        ValueChanged(Me, e1)
End Sub

' 3) Returns the value at the one-based row and col.
Public Function GetValueRowCol(ByVal row As Integer, ByVal col As Integer) As
Object Implements ICalcData.GetValueRowCol
      ' Row, col are one-based.
        Return Me(row - 1, col - 1)
End Function

' 4) Sets the value at the one-based row and col.
Public Sub SetValueRowCol(ByVal value As Object, ByVal row As Integer, ByVal
col As Integer) Implements ICalcData.SetValueRowCol
      ' Row, col are one-based.
        Dim dt As DataTable = Me.DataSource '
        dt.Rows(row - 1)(col - 1) = value
End Sub

' 5) Required ICalcData event.
Public Event ValueChanged As ValueChangedEventHandler Implements
ICalcData.ValueChanged
' CalcDataGrid
End Class
```

The following is an explanation of the preceding code.

1. This is the implementation of the **ICalcData.WireParentObject**. One of the requirements of the ICalcData object is to tell the CalcEngine "when" values have changed. This is necessary so that the CalcEngine can maintain the proper state of its data structures. The ICalcData object performs this notification task by raising the **ICalcData.ValueChanged**

event whenever a data value changes. WireParentObject is a callback method from CalcEngine which gives the ICalcData object a chance to do whatever it needs to initially set up this notification process. In this particular case, this means subscribing to the **DataTable.ColumnChanged** event whose event handler will raise the required ValueChanged event. These type of actions cannot be done in the ICalcData constructor as the data source of the DataGrid would not have been set at this point. Using the WireParentObject callback, enables the DataGrid to have things completely set up before the WireParentObject is triggered when the CalcEngine is created.

2. This is the DataTable.ColumnChanged event handler. Its purpose is to raise the ICalcData.ValueChanged event. It uses the **CurrencyManager** to retrieve the row position of the changed row, and looks up the changed column in the Columns collection. One point to note is that both these values are zero-based indexes. Anytime you interact with an ICalcData object, the indexes have to be one-based. So, when the **ValueChangedEventArgs** are created, the indexes are incremented by one.

3. This **GetValueRowCol** implementation returns the value in the DataGrid for a certain row and column index, which are passed in by the caller. These values must be one-based in the call list. So, before they are used to retrieve the value from the DataGrid, they are decremented to be proper zero-based indexes on the DataGrid.

4. This **SetValueRowCol** implementation sets the value in the DataGrid for a certain row and column index, which are passed in by the caller. Again the row and column indexes are adjusted for the base difference.

5. This is the declaration of the ValueChanged event that was raised in the ColumnChanged event discussed in item two.

4.2 Web Control Performance

Syncfusion Essential studio makes use of the class named ScriptResourceAttribute which is used to define a resource in an assembly to be used from a client script file.

Then the resource files which are all used in the Syncfusion controls are gzipped and served over the network. The following screen shot shows this.

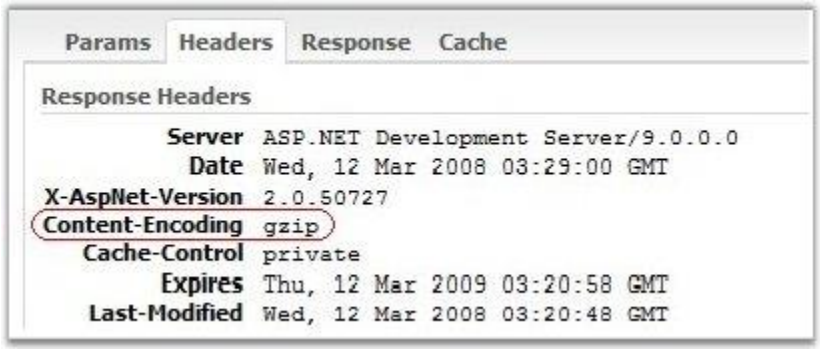


Figure 36: Web Control Performance

In order to achieve this, you need to set the following attributes in the project's web.config file.

```
<configuration system.web.extensions/>
. . .
      <scripting>
            <ScriptResourceHandler enableCompression="true"
enableCaching="true" />
      </scripting>
. . .
</system.web.extensions>
```

As the resource files get gzipped

- It saves the precious network band-width.
- It reduces the load-time. As a result, the web form, which consists of the Syncfusion controls, will get loaded faster on the client browser.
- It also reduces the network traffic.

4.3 Working with an Excel Spreadsheet

You can use the Microsoft Excel to design spreadsheets that can be used on systems where MS Excel is not installed. This can be done by using a combination of Essential XlsIO and Essential Calculate, where the former can be used to read and write the spreadsheet and later to actually do the computation as values in the spreadsheet are modified.

Example

To illustrate this process, consider a sample project, Essential Studio\x.x.x\Windows\Calculation.Windows\Samples\2.0\XlsFileUsingExcelRW.

Note: This requires you to have Essential XlsIO installed in addition to Essential Calculate. MS Excel is not required.

The spreadsheet you are using is a car insurance calculator. It uses Names to manage variable values and has the following four sheets.

- **Inputs**-Contains the input values for the car insurance calculations like the state, age, and so on.
- **LookUps**-Contains data that determine insurance rates. For example, each state has a weight assigned to it; each age has a weight assigned to it, and so on.
- **Calculate**-Does the actual calculations. Based on the input values from the input sheet, formulas in this sheet, look up appropriate weights from the LookUps sheet, and compute the car insurance cost depending upon these weights.
- **Outputs**-Contains the computed results obtained from the Calculate sheet.

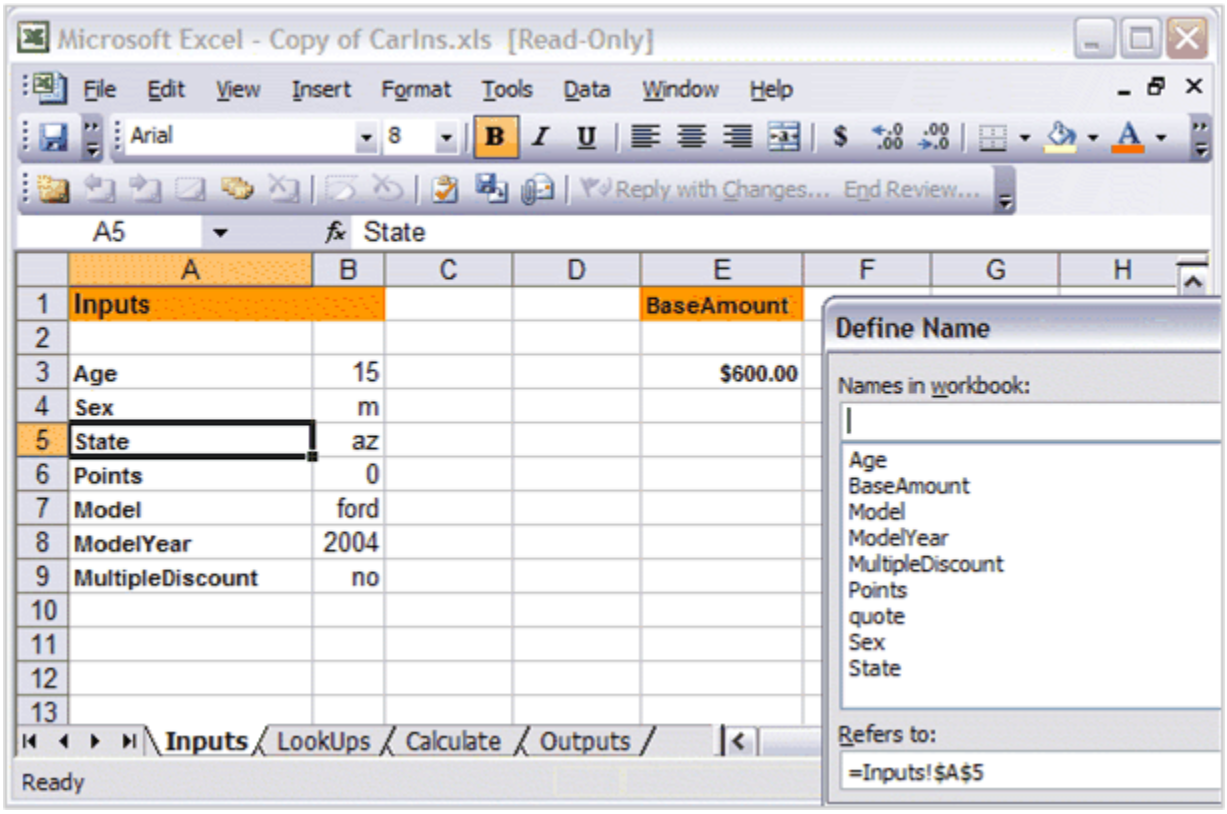
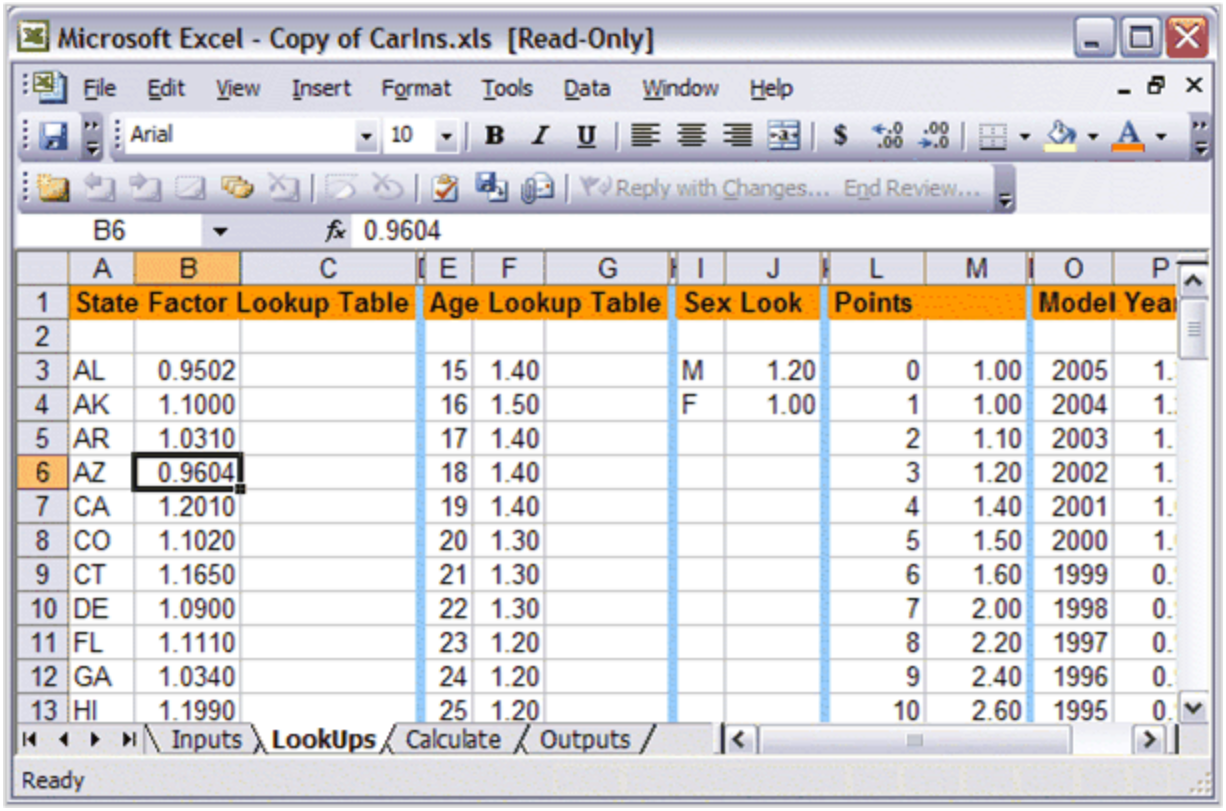


Figure 37: Worksheet that Receives Inputs



	A	B	C	E	F	G	I	J	L	M	O	P
1	State Factor Lookup Table			Age Lookup Table			Sex Look		Points		Model	Yea
2												
3	AL	0.9502		15	1.40		M	1.20	0	1.00	2005	1.
4	AK	1.1000		16	1.50		F	1.00	1	1.00	2004	1.
5	AR	1.0310		17	1.40				2	1.10	2003	1.
6	AZ	0.9604		18	1.40				3	1.20	2002	1.
7	CA	1.2010		19	1.40				4	1.40	2001	1.
8	CO	1.1020		20	1.30				5	1.50	2000	1.
9	CT	1.1650		21	1.30				6	1.60	1999	0.
10	DE	1.0900		22	1.30				7	2.00	1998	0.
11	FL	1.1110		23	1.20				8	2.20	1997	0.
12	GA	1.0340		24	1.20				9	2.40	1996	0.
13	HI	1.1990		25	1.20				10	2.60	1995	0.

Figure 38: Worksheet that Holds LookUp Tables

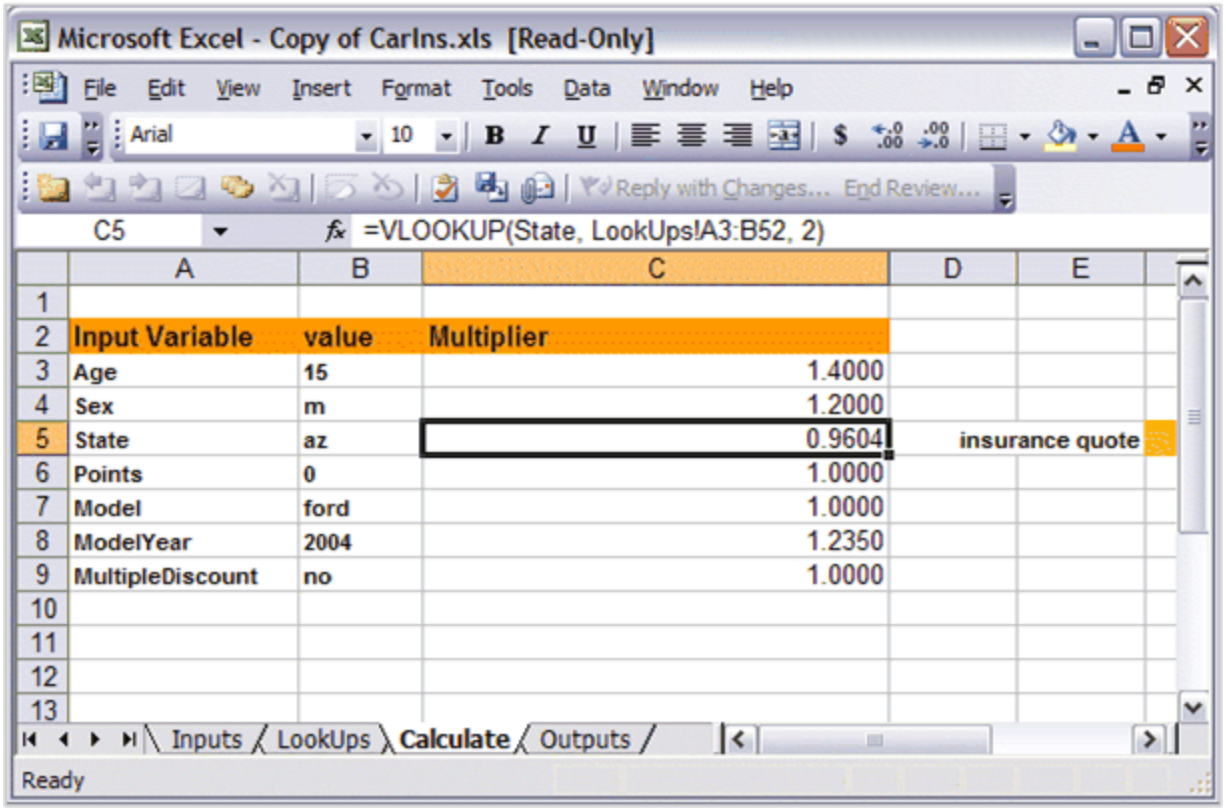


Figure 39: Worksheet that Performs Calculations

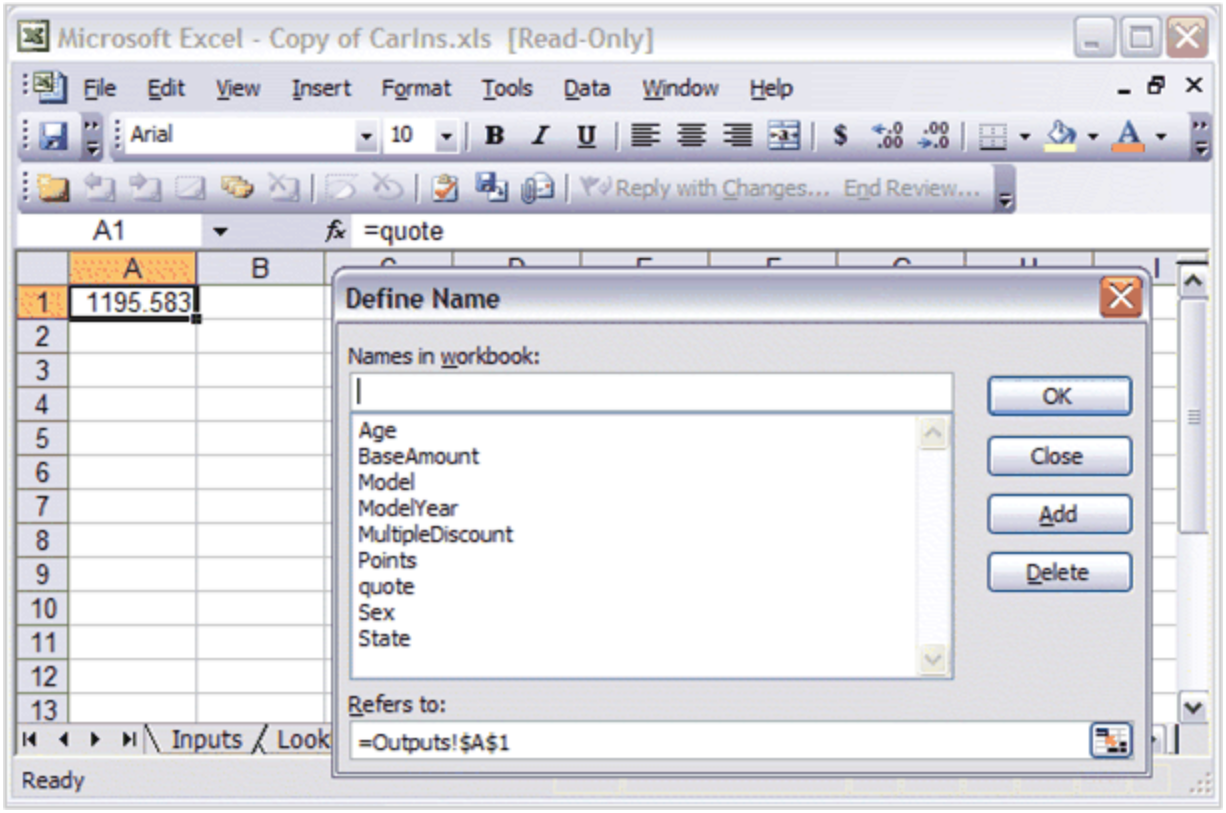


Figure 40: Dialog Box Showing Named Variables

This layout represents a general calculation design process which you can use for batch processing of information. The idea is that you change the inputs (all on a single sheet) and then return the outputs (all from a single sheet). There may be a web service or a server application that will allow clients to upload inputs and then download outputs. Or it could just be a batch processing calculation engine. Using this technique, you can use Excel to design complex calculations and then have a simple application that runs on systems without Excel, to input new values and retrieve computed results.

For example, consider the below form which accepts input values from the user. Once the values are set, the user clicks a button on the form that puts these values into the inputs sheet and then retrieves the insurance costs from the Outputs sheet and displays it on the form.

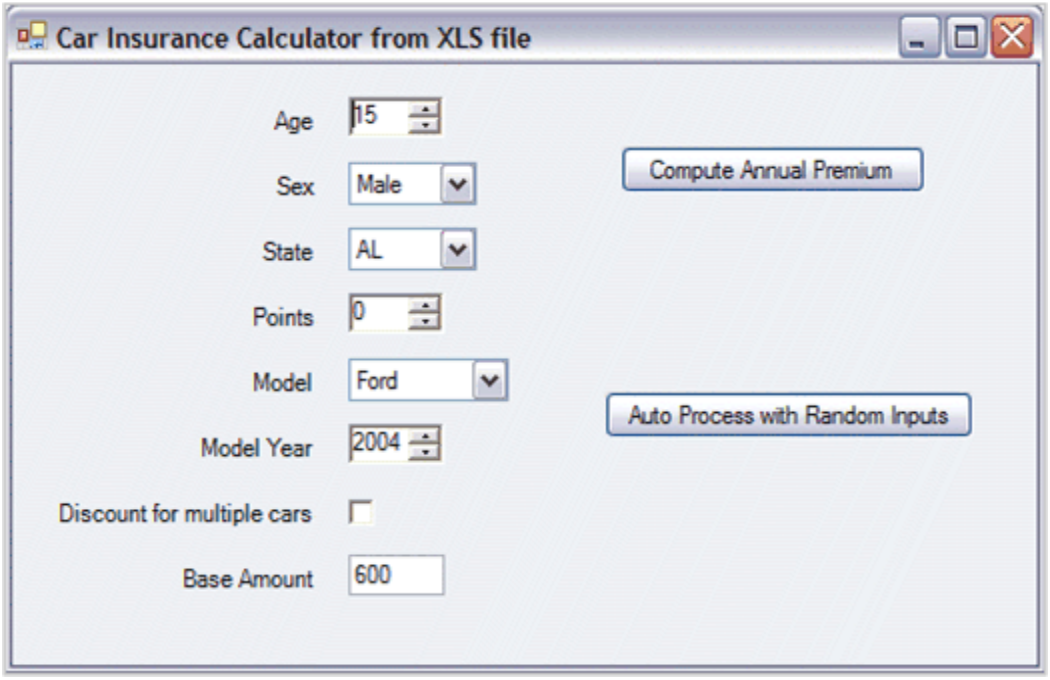


Figure 41: Form interface for our Excel Workbook

Before learning about the actual code used in this sample to access XLS files, you need to know about a couple of classes in Essential Calculate as well as the role that Essential XlsIO will play.

4.3.1 CalcSheet and CalcWorkbook Classes

In the Adding Calculation Support section, you would have learnt how to support referencing multiple ICalcData objects in a workbook fashion. The technique used there relies on registering each ICalcData object directly with a single instance of the CalcEngine. Different ICalcData objects are managed by tying together in a Tab Control as the Tab Pages. To support a general workbook structure where there are no support objects like Tab Pages and Tab Controls to provide the links, the Essential Calculate library includes two classes: **CalcSheet** and **CalcWorkbook**.

- The **CalcSheet** class is an ICalcData derived object that plays the role of a single worksheet.
- It does have the optional facility to hold row/column type data objects that can be set through indexing an instance of the class.
- This class will allocate storage to hold such data.
- The CalcWorkbook class is a collection of CalcSheets.
- You can use these classes to manage the support for working with Excel spreadsheets.

For more detailed information on these classes, check out the class reference.

4.3.2 Using Essential XlsIO

Essential XlsIO will give you an Excel-like Automation-type support without having MS Excel installed on the host system. This means that you can use this library to read and write an XLS file and hold its contents in memory.

Limitation-You cannot perform actual computations on the contents of the XLS file. Essential Calculate adds this ability.

A sample which illustrates the usage of Essential XlsIO with Essential Calculate is available in the following sample installation location:

<Install Location>\Windows\Calculate.Windows\Samples\2.0\Xls File Using CalcEngine Demo\cs

In this sample, the following two classes are used:

- **ExcelRWCalcSheet** which is derived from **CalcSheet** and implements Syncfusion.XlsIO.IWorksheet
- **ExcelRWWorkbook** which is derived from **CalcWorkbook** and implements Syncfusion.XlsIO.IWorkbook.

These classes uses XlsIO library through the supported interfaces to populate a CalcWorkbook object from an XLS file. In addition, the derived classes use overrides to get and set the data through the XlsIO objects that holds the XLS data, instead of relying on the internal data storage that is available in CalcSheet. This gives us the ability to change values in the CalcWorkbook object and view the newly computed results. So, when you want to use an XLS file in your business objects and modify the values or get new calculated results, you can add these two classes to your project and utilize the support immediately in the same manner as this sample.

4.3.3 Car Insurance Sample Details

The following sample code snippet creates an XlsIO workbook from the Excel spreadsheet that was shown earlier. It creates a form that enables you to change input values for the spreadsheet, and then displays the corresponding cost of insurance for these input values.

Given below is the code from the Form.Load event handler.

```
[C#]

private ExcelRWCalcWorkbook calcWB;
```



```
private void Form1_Load(object sender, System.EventArgs e)
{
        // 1) Instantiates the workbook object from the spreadsheet file.
        calcWB = ExcelRWCalcWorkbook.CreateFromXLS(@"CarIns.xls");

        // 2) Do all calculations so that dependencies are known.
        this.calcWB.Engine.LockDependencies = false;
        this.calcWB.CalculateAll();

       // this.calcWB.Engine.CalculatingSuspended = true;
       this.calcWB.Engine.LockDependencies = true;

        // 3) Set some initial values on the form.
        this.comboBoxSex.SelectedIndex = 0;
        this.comboBoxState.SelectedIndex = 0;
        this.comboBoxModel.SelectedIndex = 0;
}
```

```
[VB]

Private calcWB As ExcelRWCalcWorkbook

Private Sub Form1_Load(sender As Object, e As System.EventArgs)
        ' 1) Instantiate the workbook object from the spreadsheet file.
        calcWB = ExcelRWCalcWorkbook.CreateFromXLS("CarIns.xls")

        ' 2) Do all calculations so that dependencies are known.
        Me.calcWB.Engine.LockDependencies = False
        Me.calcWB.CalculateAll()

       ' Me.calcWB.Engine.CalculatingSuspended = True
        Me.calcWB.Engine.LockDependencies = True

        ' 3) Set some initial values on the form.
        Me.comboBoxSex.SelectedIndex = 0
        Me.comboBoxState.SelectedIndex = 0
        Me.comboBoxModel.SelectedIndex = 0
End Sub
```

The following is an explanation of the preceding code.

1. Uses a static member, ExcelRWWorkbook.CreateFromXLS, to read and instantiate an ExcelRWWorkbook object from the given XLS file. (The CreateFromXLS method relies on Essential XlsIO to actually do this work.)

2. A call to **CalculateAll** is made so that all dependencies can be properly set within the underlying **CalcEngine**. By default, dependent management is locked in these classes. So, you will have to toggle **LockDependencies** to allow the engine to track them. It works this way for this sample, as you are not changing any relations among the values like adding or editing actual formulas, so the dependency relations among the values do not change. Thus, these dependencies only need to be done once and not continually updated as values change. The sample requests the calculated values to be refreshed from the beginning and does not rely on auto-calculations. There is another property setting that is commented out, i.e., setting **CalculatingSuspended** to True tells the engine to skip any calculations that might be triggered by changing values. This will postpone calculations until the property is reset to False. At that point, you will have to do a **RecalculateAll** call or use an explicit **PullUpdatedValue** call to ensure that the computed values are current. Suspending calculations makes sense if you are updating many entries and do not need intermediate values calculated.

3. Sets initial values for the combo boxes on the form. This next set of code shows what will happen when you click the button. At this point, the values need to be moved from the controls on the form into the ExcelRWCalcWorkbook object and the newly computed result is retrieved.

```
[C#]

private void button1_Click(object sender, System.EventArgs e)
{

        // 1) Moves input values from the form into the calcsheet.
        SetSheetInputs();

       // 2) Calculations are not suspended, so just getting the value
triggers the calculation. So these two lines are not needed.....
       // this.calcWB.Engine.UpdateCalcID();
       //
this.calcWB.Engine.PullUpdatedValue(this.calcWB.GetSheetID("Outputs"), 1, 1);

        // 3) Get the value from cell 1,1 on the output sheet.
        double d;

if(double.TryParse(calcWB["Outputs"][1,1].ToString(),NumberStyles.Any,null,ou
t d))
        {
                // Cell 1,1 on the outputs sheet has the result.
                this.labelPrice.Text = string.Format("{0:C2}", d);
        }
        else
                this.labelPrice.Text = "---";
}
```



```
private int ageRow = 3;
private int sexRow = 4;
private int stateRow = 5;
private int pointsRow = 6;
private int modelRow = 7;
private int modelYearRow = 8;
private int multipleDiscountRow = 9;

// 4) Set the input values into the CalcSheets.
private void SetSheetInputs()
{
        CalcSheet inputSheet = this.calcWB["Inputs"];
        inputSheet[ageRow,2] = this.numericUpDownAge.Value.ToString();
        inputSheet[sexRow,2] = this.comboBoxSex.Text[0].ToString();
        inputSheet[stateRow,2] = this.comboBoxState.Text;
        inputSheet[pointsRow,2] = this.numericUpDownPoints.Value.ToString();
        inputSheet[modelRow,2] = this.comboBoxModel.Text;
        inputSheet[modelYearRow,2] = numericUpDownModelYear.Value.ToString();
        inputSheet[multipleDiscountRow,2] = checkBoxMCars.Checked ? "Yes" :
"No";
        inputSheet[3, 5] = this.textBoxBaseAmount.Text;
}
```

```
[VB]

Private Sub button1_Click(sender As Object, e As System.EventArgs)

        ' 1) Moves input values from the form into the CalcSheet.
        SetSheetInputs()

        ' 2) Calculations not suspended, so just getting the value triggers
the computation. So these two lines are not needed.....
        ' Me.calcWB.Engine.UpdateCalcID()
        '
Me.calcWB.Engine.PullUpdatedValue(this.calcWB.GetSheetID("Outputs"), 1, 1)

        ' 3) Get the value from cell 1,1 on the output sheet.
        Dim d As Double
        If Double.TryParse(calcWB("Outputs")(1, 1).ToString(),
NumberStyles.Any, Nothing, d) Then

          ' Cell 1,1 on the outputs sheet has the result.
            Me.labelPrice.Text = String.Format("{0:C2}", d)
        Else
            Me.labelPrice.Text = "---"
        End If
```



```
' Button1_Click
End Sub

Private ageRow As Integer = 3
Private sexRow As Integer = 4
Private stateRow As Integer = 5
Private pointsRow As Integer = 6
Private modelRow As Integer = 7
Private modelYearRow As Integer = 8
Private multipleDiscountRow As Integer = 9


' 4) Set the input values into the CalcSheets.
Private Sub SetSheetInputs()
        Dim inputSheet As CalcSheet = Me.calcWB("Inputs")
        inputSheet(ageRow, 2) = Me.numericUpDownAge.Value.ToString()
        inputSheet(sexRow, 2) = Me.comboBoxSex.Text(0).ToString()
        inputSheet(stateRow, 2) = Me.comboBoxState.Text
        inputSheet(pointsRow, 2) = Me.numericUpDownPoints.Value.ToString()
        inputSheet(modelRow, 2) = Me.comboBoxModel.Text
        inputSheet(modelYearRow, 2) =
Me.numericUpDownModelYear.Value.ToString()
        If Me.checkBoxMultipleCars.Checked Then
            inputSheet(multipleDiscountRow, 2) = "Yes"
        Else
            inputSheet(multipleDiscountRow, 2) = "No"
        End If
        inputSheet(3, 5) = Me.textBoxBaseAmount.Text

' SetSheetInputs
End Sub
```

The following is an explanation of the preceding code.

1. Calls a method to take the values from the controls on the form and move them into the Inputs sheet of the workbook.

2. This is a commented code. You will need it if **CalculatingSuspended** property is set to **True**. This code guarantees that the value in cell A1 of the Outputs sheet is current. But, since calculations are being done as the values are set in step 1, this code is not needed.

3. The computed value is in cell A1 of the Output sheet. To retrieve this value, you need to index the workbook with the sheet name to return the sheet, and then you need to index this sheet with the row and column to return the value, calcWB["Outputs"][1,1].

4. This is the code that actually moves the values from the controls on the form to the Inputs sheet. Again, you need to index the workbook with the sheet name and then use the appropriate row and column indexes to set the values into the Inputs sheet.

The last set of code you look at will show you how to handle a batch processing requirement, looping through setting Inputs and retrieving the Output value. In this code, you will have to set CalculatingSuspended to True; so, you are setting multiple values in a manner that does not trigger any intermediate calculations. Doing so increases performance about 10% in this sample, with only eight inputs. If you had hundreds of inputs, the increased performance will be more significant.

[C#]

```
private void button2_Click(object sender, System.EventArgs e)
{
        // Runs 1000 iterations.
        int num = 1000;

        this.Cursor = Cursors.WaitCursor;
        DateTime start = DateTime.Now;
        CalcSheet inputSheet = this.calcWB["Inputs"];
        Random r = new Random();

        this.calcWB.Engine.CalculatingSuspended = true;

        for(int i = 0; i < num; ++ i)
        {

                // 1) Sets random values into the Inputs sheet.
                inputSheet[ageRow,2] = (r.Next(74) + 15).ToString();
                inputSheet[sexRow,2] = r.Next(2) == 1 ? "M" : "F";
                inputSheet[stateRow,2] =
this.comboBoxState.Items[r.Next(50)];
                inputSheet[pointsRow,2] = r.Next(15).ToString();
                inputSheet[modelRow,2] = r.Next(11).ToString();
                inputSheet[modelYearRow,2] = (33 + r.Next(1972)).ToString();
                inputSheet[multipleDiscountRow,2] = r.Next(2) == 1 ? "Yes" :
"No";
                inputSheet[3, 5] = this.textBoxBaseAmount.Text;

                // 2) Calculations are suspended so need to pull the computed
value to make sure it has been calculated with the latest changes.
                this.calcWB.Engine.UpdateCalcID();

calcWB.Engine.PullUpdatedValue(this.calcWB.GetSheetID("Outputs"), 1, 1);

                // 3) Gets the value from cell 1,1 on the output sheet.
```



```
                double d;
                if(double.TryParse(calcWB["Outputs"][1,1].ToString(),
NumberStyles.Any, null, out d))
                {
                        this.labelPrice.Text = string.Format("{0:C2}", d);
                }
                else
                        this.labelPrice.Text = "---";

                // Allows the label to update.
                this.labelPrice.Refresh();
        }
        this.calcWB.Engine.CalculatingSuspended = false;
        this.Cursor = Cursors.Default;
        this.labelPrice.Text = string.Format("{0} updates in {1} seconds",
num,  (TimeSpan)(DateTime.Now - start) );
}
```

[VB]

```
Private Sub button2_Click(sender As Object, e As System.EventArgs)

        ' Runs 1000 iterations
        Dim num As Integer = 1000

        Me.Cursor = Cursors.WaitCursor
        Dim start As DateTime = DateTime.Now
        Dim inputSheet As CalcSheet = Me.calcWB("Inputs")
        Dim r As New Random()

        Me.calcWB.Engine.CalculatingSuspended = True

        Dim i As Integer = 0
        While i < num

            ' 1) Sets random values into the Inputs sheet.
            inputSheet(ageRow, 2) = (r.Next(74) + 15).ToString()
            If r.Next(2) = 1 Then
                inputSheet(sexRow, 2) = "M"
            Else
                inputSheet(sexRow, 2) = "F"
            End If
            inputSheet(stateRow, 2) = Me.comboBoxState.Items(r.Next(50))
            inputSheet(pointsRow, 2) = r.Next(15).ToString()
            inputSheet(modelRow, 2) = r.Next(11).ToString()
            inputSheet(modelYearRow, 2) = (33 + r.Next(1972)).ToString()
```



```
            If Me.checkBoxMultipleCars.Checked Then
                inputSheet(multipleDiscountRow, 2) = "Yes"
            Else
                inputSheet(multipleDiscountRow, 2) = "No"
            End If
            inputSheet(3, 5) = Me.textBoxBaseAmount.Text

           ' 2) Calculations are suspended so need to pull the computed value
to make sure it has been calculated with the latest changes.
            Me.calcWB.Engine.UpdateCalcID()
            calcWB.Engine.PullUpdatedValue(Me.calcWB.GetSheetID("Outputs"),
1, 1)

            ' 3) Gets the value from cell 1,1 on the output sheet.
            Dim d As Double
            If Double.TryParse(calcWB("Outputs")(1, 1).ToString(),
NumberStyles.Any, Nothing, d) Then
                Me.labelPrice.Text = String.Format("{0:C2}", d)
            Else
                Me.labelPrice.Text = "---"
            End If

            ' Allows the label to update.
            Me.labelPrice.Refresh()
        End While
        Me.calcWB.Engine.CalculatingSuspended = False
        Me.Cursor = Cursors.Default
        Me.labelPrice.Text = String.Format("{0} updates in {1} seconds", num,
CType(DateTime.Now - start, TimeSpan))

' Button2_Click
End Sub
```

The following is an explanation of the preceding code.

1. Setting random values into the Inputs sheet using the proper row and column indexers.

2. Calling **UpdateCalcID** and PullUpdatedValue guarantees that the value in the Outputs sheet at cell (1,1) reflects the current values in the workbook.

3. Here, you retrieve the value on the Output sheet at cell (1,1).

4.4 Supported Algebra

The explicit look of a valid formula in Essential Calculate may vary depending upon the context of the formula. For example, if you are using a formula in a **CalcSheet** class based on an Excel spreadsheet, then something like = A1 + A3 will be valid since CalcSheet recognizes the "A1" and "A3" as valid cell references. But, if you are using a **CalcQuickBase** object to manage formulas for controls on a Windows Form, then the same = A1 + A3 will not be valid since CalcQuickBase only recognizes registered names inside square brackets as valid arguments. Hence, all Essential Calculation formulas support the same algebra with the exception of what comprises the definition of valid arguments.

This section comprises the following topics:

4.4.1 Operators

The following is a list of the operators which are supported by Essential Calculate.

Unary Arithmetic Operator

- Unary Minus Sign

Binary Arithmetic Operators

+ Addition

- Subtraction

* Multiplication

/ Division

^ Exponentiation

Binary Literal Operator

& Concatenation

Binary Logical Operators

< Less Than

> Greater Than

= Equal To

<= Less Than Or Equal

>= Greater Than Or Equal

<> Not Equal

All operations are subject to the following hierarchy of operations. The level 1 operations are done first, followed by level 2, and so on. Within the same level, the operations are performed from left to right in the order in which they are encountered during the parsing of the formula.

1. - (Unary Minus)
2. * /
3. + -
4. < > = <= >= <>
5. & (Concatenation)

If you want to change the default operators precedence, then use parentheses to explicitly indicate the operation order.

Examples

	Formulas	Computed Value
1.	= 6 / 2 + 1	4
2.	= 6 / (2 + 1)	2
3.	= 2 + 4 / 2	4
4.	= (2 + 4) / 2	3

Logical operations return specific values: True or False. If you need specific numerical values associated with any logical expression, then use the logical expression as the first argument in the Formula Library IF-function, with the second argument being the numerical value of True and the third argument being the numerical value of False. If you use a well-formed logical expression in a larger calculation, True evaluates to numerical 1 and False evaluates to numerical 0 for use in the calculations.

4.4.2 Square Brackets in CalcQuickBase Formulas

If you are using a **CalcQuickBase** object to add calculation support to your business object, then you must use strings as indexers on the CalcQuickBase instance to get and set values. These strings are referred to as the value's Name. If you need to use a Name in a formula, then you should enclose the string within brackets, [ ]. In step three of the code below, four names A, B, C, and D are registered. Notice that the formula entered in step two uses the values from A and B by enclosing these names in brackets.

```
[C#]

// 1) Instantiates a CalcQuickBase object.
```



```
calculator = New CalcQuickBase();

// 2) Populate your controls.
this.textBoxA.Text = "12";
this.textBoxB.Text = "3";
this.textBoxC.Text = "= [A] + 2 * [B]";

// Must enter formula names before turning on calculations.
// 3) Assigns formula object names.
calculator("A") = this.textBoxA.Text;
calculator("B") = this.textBoxB.Text;
calculator("C") = this.textBoxC.Text;
calculator("D") = this.textBoxD.Text;
```

```
[VB]

' 1) Instantiates a CalcQuickBase object.
calculator = New CalcQuickBase()

' 2) Populate your controls.
Me.textBoxA.Text = "12"
Me.textBoxB.Text = "3"
Me.textBoxC.Text = "= [A] + 2 * [B]"

' Must enter formula names before turning on calculations.
' 3) Assigns formula object names.
calculator("A") = Me.textBoxA.Text
calculator("B") = Me.textBoxB.Text
calculator("C") = Me.textBoxC.Text
calculator("D") = Me.textBoxD.Text
```

4.4.3 Equal Sign, the Formula Character

To indicate that a particular string should be treated as a formula, you must start the string with a special character, **CalcEngine.FormulaCharacter**. This property is static (Shared in VB), so you can change the formula character within your code. It's default value is the equal sign, (=).

In general, in order for Essential Calculate to recognize a string as containing a formula; the string is required to start with the CalcEngine.FormulaCharacter. There is one exception though, if you explicitly call a CalcEngine **Parse** method like **CalcEngine.ParseFormula** or **CalcEngine.ParseAndComputeFormula**, including the formula character as the first character in the passed string, it is optional.

4.4.4 Using Function Library Formulas

Function Library formulas may be used as a stand alone formula, and they can be incorporated into more complex formulas.

Formula	Comment
= Sin(3.14159)	Returns the sine of 3.14159 radians
= 2 * Sin(3.14159) + Sqrt(2)	Returns 2 * sine of 3.14159 radians plus the square root of 2.
= 2 * Pi()	Returns 2 * pi.

Some library functions may not have arguments but, you must still include the parentheses to indicate that you are using a library function. For example, = 2 * Pi(), shows the proper use of the library function Pi.

4.5 Function Library

The **Function Library** contains many functions from statistics, finance and mathematics, along with other general purpose functions. There are more than 150 entries in the library, and it is easy to add your own calculations.

The functions are discussed in the below topics.

4.5.1 Add Function

CalcQuickBase relies on a Calculate.Engine object through an **ICalcData** interface to provide its calculation support. To add functions to the Formula Library available to your CalcQuickBase object, you need to add them to the **CalcQuickBase's** underlying Engine object. You can access this engine object through the public "read-only" property, CalcQuickBase.Engine. Once you have a reference to the CalcQuickBase's Engine object, you can add library functions by following the steps given below.

Adding a custom function to the Formula Library is a two step process.

The first step is to write a method that actually does the calculation work for your custom function. The second step is to register this method with the CalcEngine. So, if your CalcEngine object is a member of a form, you can add your additional function methods to the form and then register these methods with the CalcEngine object after the object has been created, in Form_Load for example.

The above steps have been explained in detail in the following topics:

4.5.1.1 Step 1-Writing the Method

The method must have the signature specified by the delegate, **Syncfusion.Calculate.CalcEngine.LibraryFunction**. It accepts a string argument and returns a string value. So here is a minimal implementation. The sample found in \Windows\Calculate.Windows\Samples\2.0\ DataGridCalculator has code that adds a custom function.

[C#]

```
public string ComputeMymin(string args)
{
   // Computes someString using the values from args.
   return someString;
}
```

[VB]

```
Public Function ComputeMymin(args As String) As String

' Computes someString using the values from args.
Return someString

' ComputeMymin
End Function
```

This is the only requirement on the method. You are free to use any kind of conventions with respect to passing arguments and within your implementation code. So, args may be a single entry like A1 or 153 or, it may be something more complex like A1:C15. It is up to your implementation code to parse args, use the information passed in, to compute the value and then return this value as a string. If you want your arguments to be standard items like cell references, numbers, other formulas, etc., **CalcEngine** exposes some parsing tools that you can use to minimize the amount of code you may need to write. But, you are not limited to what CalcEngine exposes, you are free to design and implement any argument conventions you want, as long as your method has the required signature.

Here add a method that accepts an argument list and then returns the minimum value in this list. The list may be individual cell references and cell ranges, or numbers. This sample code uses CalcEngine methods to handle the parsing and retrieving of values from the argument list.

 ***Note: List separators can vary depending upon culture. While it is reasonable to use a comma as a separator in en-US, this is not the case with fr-FR where the comma is used as a decimal separator. For this reason, CalcEngine.ParseArgumentSeparator is a static member that holds the list separator that is recognized by the parsing of algorithms in the CalcEngine.***

In the following code, **CalcEngine.ParseArgumentSeparator** is used to split the args into a list. Additionally, the code makes use of two utility methods in the CalcEngine object, **GetCellsFromArgs** and **GetValueFromArg.** GetCellsFromArgs accepts a range like A3:C6 and returns a string array of the individual cell references in the range. GetValueFromArg will accept a cell reference, formula, or number and return the numerical value that the cell holds.

```
[C#]

// This sample computes the minimum of an arbitrary range.
// Example:    = Mymin(A1:C3)
// Example:    = mymin(a1,c2,a4,b2,100)
public string ComputeMymin(string args)
{
       // Assumes that this.engine is the CalcEngine object.

        double min = double.MaxValue;
        double d;
        string s1;

        foreach(string r in args.Split(new
char[]{CalcEngine.ParseArgumentSeparator}))
        {
                // Cell range
                if(r.IndexOf(':') > -1)
                {
                        foreach(string s in engine.GetCellsFromArgs(r))
                        {
                              // s is a cell line a21 or c3...
                                try
                                {
                                        s1 = engine.GetValueFromArg(s);
                                        if(s1 != ""
                                                && double.TryParse(s1,
System.Globalization.NumberStyles.Number,
                                                        null, out d))
                                        {
                                                min = Math.Min(min, d);
```



```
                                        }
                                }
                                catch(Exception ex)
                                {
                                        return ex.Message;
                                }
                        }
                }
                else
                {
                        try
                        {
                                s1 = engine.GetValueFromArg(r);
                                if(s1 != ""
                                        && double.TryParse(s1,
System.Globalization.NumberStyles.Number,
                                                                null, out d))
                                {
                                        min = Math.Min(min, d);
                                }
                        }
                        catch(Exception ex)
                        {
                                return ex.Message;
                        }
                }
        }
        if(min != double.MaxValue)
                return min.ToString();
        return "";
}
```

[VB]

```
' This sample computes the minimum of an arbitrary range.
' Example:    = Mymin(A1:C3)
' Example:    = mymin(a1,c2,a4,b2,100)
Public Function ComputeMymin(ByVal args As String) As String

        ' Assumes that this.engine is the CalcEngine object.
        Dim min As Double = Double.MaxValue
        Dim d As Double
        Dim s1 As String

        Dim r As String
        For Each r In args.Split(New Char()
```



```
{CalcEngine.ParseArgumentSeparator})

            ' Cell range
            If r.IndexOf(":"c) > -1 Then
                Dim s As String
                For Each s In engine.GetCellsFromArgs(r)

                    ' s is a cell line a21 or c3...
                    Try
                        s1 = engine.GetValueFromArg(s)
                        If s1 <> "" And Double.TryParse(s1,
System.Globalization.NumberStyles.Number, Nothing, d) Then
                            min = Math.Min(min, d)
                        End If
                    Catch ex As Exception
                        Return ex.Message
                    End Try
                Next s
            Else
                Try
                    s1 = engine.GetValueFromArg(r)
                    If s1 <> "" And Double.TryParse(s1,
System.Globalization.NumberStyles.Number, Nothing, d) Then
                        min = Math.Min(min, d)
                    End If
                Catch ex As Exception
                    Return ex.Message
                End Try
            End If
        Next r
        If min <> Double.MaxValue Then
            Return min.ToString()
        End If
        Return ""

        ' ComputeMymin
End Function
```

4.5.1.2 Step 2-Registering the Method with the CalcEngine

The second step for adding your own formula is to register your method with the **CalcEngine** object. This is done with the **AddFunction** method. This method accepts the string that is used when you reference the function in a spreadsheet formula, and the second argument is a delegate for which you pass your method. The only requirement here is that the function name should start with an alpha character and should only contain alpha-numeric characters. Additionally, the string cannot be the name of any existing library function.

```
[C#]

// Add formula name Mymin to the Library.
this.engine.AddFunction("Mymin", new LibraryFunction(ComputeMymin));
```

```
[VB]

' Add formula name Mymin to the Library.
Me.engine.AddFunction("Mymin", AddressOf ComputeMymin)
```

By convention, within the Essential Calculate library, the C# implementation method for each of the library functions that are shipped with the word "Compute" is named and followed by the name of the library function. The above code confirms to this convention, with the function name being 'Mymin' and the method name being 'ComputeMymin'. Our library functions are public members of the **CalcEngine** class, so that you can access them directly if it serves your purpose. Additionally, if you own the source code version, you can see implementation details that may be of use to you if you try to implement many custom library methods on your own.

 ***Note: Once this is done, you can use your custom method in the same manner as the default library functions.***

4.5.2 Remove and Replace Function

This section discusses the Remove and Replace Function available for the CalcEngine.

Remove Function

Removing unused functions from the Function Library, reduces the memory usage and speeds up parsing as well. Also, if you are only using a selected few Library functions, you may want to remove the unused ones. This can be done using the methods given below.

- To remove all functions, you can clear the hash table that holds them by using the **engine.LibraryFunctions.Clear** method.

```
[C#]
```



```
// Remove all functions from the Library.
engine.LibraryFunctions.Clear();
```

```
[VB]

// Remove all functions from the Library.
engine.LibraryFunctions.Clear()
```

After clearing all functions, you can add few functions that will be used often. To know how to add functions, see Add Function.

- To remove a single function from the Function Library, use the **CalcEngine.RemoveFunction** method, passing a "function name" as the string that references this function, from a formula.

```
[C#]

// Remove formula name MyMin from the Library.
engine.RemoveFunction("MyMin");
```

```
[VB]

' Remove formula name MyMin from the Library.
engine.RemoveFunction("MyMin")
```

Replace Function

To replace a function with another implementation, you must remove the original name, and add the same name again with a different delegate method.

4.5.3 Functions

Here is a list of the Function Library entries included with Essential Calculate which includes more than 150 entries. For detailed information on each function, see Function Reference Section.

4.5.3.1 Logical

This section lists the logical functions included with Essential calculate in the below table.

Table 5: logical functions

Function Name	Description
And	Returns TRUE if all conditions are TRUE. It returns FALSE if any of the conditions are FALSE.
If	Returns one value if a specified condition evaluates to TRUE, or another value if it evaluates to FALSE.
IF-THEN-ELSE	Statement can only be used in VBA code. Assume there are two conditions that are evaluated. Once a condition is found to be true, the IF-THEN-ELSE statement will execute the corresponding code and not evaluate the conditions any further.
True	Function returns a logical value of TRUE.
Case(VBA only)	It has the functionality of an IF-THEN-ELSE statement.
Nested Ifs(Upto 7)	It is possible to nest multiple IF functions within one Excel formula. You can nest up to 7 IF functions to create a complex IF THEN ELSE statement.
Not	This function returns the reversed logical value.
False	Returns a logical value of FALSE.
NestedIfs (More than)	Use this method to nest up to or more than 7 IF conditions.
Or	Returns TRUE if any of the conditions are TRUE. Otherwise, it returns FALSE.

4.5.3.2 Text

This section lists the Text functions included with Essential calculate in the below table.

Table 6: Text functions

Function Name	Description
Concatenate	Joins arguments into a single string.
Left	Returns the first character or characters in a text string, based on the number of characters you specify.
Right	Returns the last character or characters in a text string, based on the

	number of characters you specify.
Text	Returns a formatted string.
Value	Returns a number from a string.
Len	Returns the length of a string .
MID	Returns a text segment of a character string.

4.5.4 Date and Time

This section lists the Date and Time functions included with Essential calculate in the below table.

Table 7: Date and Time functions

Function Name	Description
Date	Returns the sequential serial number that represents a particular date.
Datevalue	Returns the serial number of the date value.
Day	Returns the day of the month.
Days360	Returns the number of days between two dates using 360-day year.
Hour	Returns the hour from 0 to 23.
Minute	Returns the minute from 0 to 59.
Month	Returns the month from 1 to 12.
Now	Returns the current date and time.
Second	Returns the second from 0 to 59.
Time	Returns serial number time.
Timevalue	Returns the serial number time from a string.
Today	Returns the current date.

Weekday	Returns the weekday from 1 to 7.
Year	Returns the year from 1900 to 9999.

4.5.4.1 LookUps and Information

This section lists the LookUp and Information functions included with Essential calculate in the below table.

Table 8: LookUp and Information functions

Function Name	Description
HLookUp	Finds a value in one row and returns the corresponding value in another row.
VLookUp	Finds a value in one column and returns the corresponding value in another column.
IsError	Returns True if cell holds an error.
IsNumber	Returns True if cell holds a number.

4.5.5 Financial

This section lists the Financial functions included with Essential calculate in the below table.

Table 9: Financial functions

Function Name	Description
Db	Returns the depreciation using fixed declining balance calculation.
Ddb	Returns the depreciation using double declining balance calculation.
Fv	Returns future value of an investment.
Ipmt	Returns interest payment.
Irr	Returns internal rate of return.

Ispmt	Returns interest payment.
Mirr	Returns internal rate of return.
Nper	Returns number of payment periods.
Npv	Returns net present value.
Pmt	Returns loan payment.
Ppmt	Returns principal payment.
Pv	Returns present value.
Rate	Returns interest rate per period.
Sln	Returns straight-line depreciation.
Syd	Returns sum-of-years depreciation.
Vdb	Returns the depreciation using double declining balance calculation.

4.5.6 Math and Trig functions

This section lists the Math and Trig functions included with Essential calculate in the below table.

Table 10: Math and Trig functions

Function Name	Description
Abs	Returns the absolute value of a value.
Acos	Returns the inverse cosine.
Acosh	Returns the inverse hyperbolic cosine.
Asin	Returns the inverse sine.
Asinh	Returns the inverse hyperbolic sine.
Atan	Returns the inverse tangent.
Atan2	Returns the inverse tangent.

Atanh	Returns the inverse hyperbolic tangent.
Avg	Returns the arithmetic mean of the listed arguments.
Ceiling	Returns the rounded up value.
Combin	Returns the number of combinations.
Cos	Returns the cosine.
Cosh	Returns the hyperbolic cosine.
Degrees	Returns the degrees represented by the given radians.
Even	Returns the number rounded up to the nearest event integer.
Exp	Returns the value of e raised to the given argument.
Fact	Returns the factorial of the given value.
Floor	Returns the value rounded down to the nearest value.
Int	Returns the value rounded down to nearest integer.
Ln	Returns the natural logarithm of the value.
Log	Returns the logarithm to a given base.
Log10	Returns the logarithm base 10.
Mod	Returns the remainder after a division.
Odd	Returns the value to the nearest odd integer.
Pi	Returns the pi.
Pow	Returns the value of one number raised to another number.
Product	Returns the product of the listed values.
Radians	Returns the radians from given degrees.
Rand	Returns a random value.
Round	Returns the number to a specified number of digits.
Rounddown	Returns the number rounded down.

Roundup	Returns the number rounded up.
Sign	Returns the sign of the number.
Sine	Returns the sine.
Sinh	Returns the hyperbolic sine.
Sqrt	Returns the square root of a number.
Sum	Returns the sum of the listed values.
Sumif	Returns the conditional sum of the listed values.
SumProduct	Returns the sum of the products.
Sumsq	Returns the sum of the squares.
Sumx2my2	Returns the differences of the squares.
Sumx2py2	Returns the sum of the squares.
Sumxmy2	Returns the squares of the differences.
Tan	Returns the tangent.
Tanh	Returns the hyperbolic tangent.
Trunc	Returns the number without a fractional part.

4.5.6.1 Multinomial

The MULTINOMIAL function returns the ratio of the factorial of a sum of values to the product of factorials.

Syntax

The syntax of the MULTINOMIAL function is

=MULTINOMIAL(number1, (number2), …)

#NUM! - Occurs if any of the supplied arguments are less than 0.

#VALUE! - Occurs if any of the supplied arguments are non-numeric.

Example:

FORMULA	RESULT
=MULTINOMIAL(2, 3, 4)	1260

4.5.6.2 ISEVEN

The ISEVEN function returns TRUE if given number is an even number, and returns FALSE if the given number is odd.

Syntax:

The syntax of the ISEVEN function is

=ISEVEN (value)

The given value must be a numeric value. If it is non-integer value, the value is rounded down.

 ***Note: If the given value is nonnumeric, the ISEVEN function returns the '#VALUE!' error value.***

Example:

FORMULA	RESULT
=ISEVEN(-1)	FALSE
=ISEVEN(2.5)	TRUE
=ISEVEN(5)	FALSE

4.5.6.3 ISODD

The ISODD function returns TRUE if the given number is an odd number, and returns FALSE if the given number is even.

Syntax:

The syntax of the ISODD function is

=ISODD (value)

The given value must be a numeric value. If it is a non-integer value, the value is rounded down.

 ***Note: If the given value is nonnumeric, ISODD function returns the '#VALUE!' error value.***

Example:

FORMULA	RESULT
=ISODD(-1)	TRUE
=ISODD(2.5)	FALSE
=ISODD(5)	TRUE

4.5.6.4 N

The N function converts the given value into a numeric value.

Syntax:

The syntax of the N function is

=N (value)

A value is required.

Numeric values are converted as numeric values. A date value is converted as a serial number. The Logic operator TRUE returns a value of 1. The other values are returned as 0.

Example:

FORMULA	RESULT
=N(7)	7
=N("EVEN")	0
=N(1/1/2008)	39448
=N(TRUE)	1

4.5.6.5 NA

The NA function returns the #N/A error. This error message is produced when a formula is unable to find a value that it needs. This error message denotes 'value not available.'

Syntax:

The syntax of NA function is

=NA()

The NA function syntax has no arguments.

 ***Note: The function doesn't have any arguments.***

Example:

FORMULA	RESULT
=NA()	#NA

4.5.6.6 ERROR.TYPE

The Error.Type function returns an integer for the given error value that denotes the type of the given error.

Syntax:

The syntax of the ERROR.TYPE function is

= ERROR.TYPE(value)

The given value is required.

Here is the return value of function:

Given Value	Return value of function
#NULL!	1
#DIV/0!	2
#VALUE!	3
#REF!	4
#NAME?	5
#NUM!	6
#N/A	7
#GETTING_DATA	8

Anything else	#N/A

Example:

FORMULA	RESULT
= ERROR.TYPE(#NULL!)	1
= ERROR.TYPE(even)	#NA

4.5.6.7 SUBTOTAL

The SUBTOTAL function returns a subtotal in a list. Once the subtotal list is created, you can modify it by editing the SUBTOTAL function.

Syntax:

The syntax of the SUBTOTAL function is

=SUBTOTAL (function_Number, ref1, (ref2),...)

A function_Number is required. This specifies which function to use in calculating subtotals within a list. Here is the list of functions supported by Syncfusion:

FUNCTION NUMBER	FUNCTION NAME
1	AVERAGE
2	COUNT
3	COUNTA
4	MAX
5	MIN
6	PRODUCT
7	STDEV
8	STDEVP
9	SUM
10	VAR

Ref1 The first named range which is used for the subtotal. This value is required.

Ref2 This value is optional.

Note: If the subtotal function has any nested subtotal functions, then the nested subtotal is ignored for double counting.

4.5.6.8 MROUND

The MROUND function rounds a given number up or down to the nearest multiple of a given number.

Syntax

The syntax of the MROUND function is

= MROUND (number, multiple)

Number – The value to be rounded. This value is required.

Multiple – This value is required.

The number must be is greater than or equal to half the value of multiple.

#NUM! - Occurs if the number and multiple have different signs.

#VALUE! - Occurs if any of the given arguments are non-numeric.

Example:

FORMULA	RESULT
=MROUND(10,2.6)	8
=MROUND(-10,-2.6)	-8
=MROUND(10,-2)	#NUM!

4.5.6.9 RANDBETWEEN

The RANDBETWEEN function returns a random number that is between the given ranges. This function returns a new random number each time in recalculation.

Syntax

The syntax of the RANDBETWEEN Function is

=RANDBETWEEN (start_num, end_num)

start_num – Required. This is the smallest integer.

end_num – Required. This is the largest integer.

#NUM! - Occurs if the end_num value is larger than start_num value.

#VALUE! - Occurs if any of the given arguments are non-numeric.

Example:

FORMULA	RESULT
= RANDBETWEEN (10,20)	12. The value is generated randomly between the given values.

4.5.6.10 SQRTPI

The SQRTPI function returns the square root of a given number multiplied by π. Here π is the constant math value.

Syntax

The syntax of the SQRTPI function is

=SQRTPI (number)

number – Required.

#NUM! - If the number is less than zero (0).

#VALUE! - Occurs if any of the given arguments are non-numeric.

Example:

FORMULA	RESULT
= SQRTPI (2)	2.506628
= SQRTPI (-2)	#NUM!

4.5.6.11 QUOTIENT

The QUOTIENT function returns the integer portion of a division between two given numbers. The returned value will be integer value.

Syntax

The syntax of the QUOTIENT function is

=QUOTIENT (numerator, denominator)

Numerator – Required.

Denominator – Required.

Remarks:

#VALUE! - Occurs if any of the given arguments are non-numeric.

Example:

FORMULA	RESULT
= QUOTIENT (10,3)	3
= QUOTIENT (-20,6)	-3

4.5.6.12 FACTDOUBLE

The FACTDOUBLE function returns the double factorial of a given value. The given value must be an integer value.

Syntax

The syntax of the FACTDOUBLE function is

= FACTDOUBLE (number).

number – Required.

#NUM! - If the number is less than zero (0).

#VALUE! - Occurs if any of the given arguments are non-numeric

Example:

FORMULA	RESULT
= FACTDOUBLE (6)	48
= FACTDOUBLE (-2)	#NUM!

4.5.6.13 GCD

The GCD function returns the greatest common divisor of two or more given values. The values must be a numeric value.

Syntax

The syntax of the GCD function is

= GCD (number1, number2, ...)

Number1 – Required.

If any value is not an integer, then it will be rounded down.

#NUM! - If the number is less than zero (0).

#VALUE! - Occurs if any of the given arguments are non-numeric.

Example:

FORMULA	RESULT
= GCD (5,3,2)	1
= GCD (-2)	#NUM!

4.5.6.14 LCM

The LCM function returns the least common multiple of two or more given values. The values must be numeric values.

Syntax

The syntax of the LCM function is

= LCM (number1, number2, ...)

number1 – Required.

If any value is not an integer, then it will be rounded down.

#NUM! - If the number is less than zero (0).

#VALUE! - Occurs if any of the given arguments are non-numeric.

Example:

FORMULA	RESULT
= LCM (5,2)	10
= LCM (-2)	#NUM!

4.5.6.15 ROMAN

The ROMAN function converts an Arabic number to a Roman numeral. This function returns a text string depicting the Roman numeral form of the given number.

Syntax

The syntax of the ROMAN function is

=ROMAN(number, (form))

number – Required.

Form – Optional, this value will specify the style of the Roman numeral.

If number is not an integer, then it will be rounded down.

Form	Type
0 or omitted	Classic.
1	More concise. See example below.
2	More concise. See example below.
3	More concise. See example below.
4	Simplified.

#VALUE! - Occurs if any of the given values is non-numeric, or for values less than 0 and greater than 3999.

Example:

FORMULA	RESULT
= ROMAN (499,0)	CDXCIX
= ROMAN (499,1)	ID
=ROMAN(-100)	#VALUE!

4.5.6.16 IFERROR

The IFERROR function tests if an initial given value (or expression) returns an error, and if so, this function returns a second given argument. Otherwise, the function returns the initial tested value.

Syntax

The syntax of the IFERROR function is

= IFERROR (value, value_error)

value – Required. This is a value to check the error.

value_error – Required. This value will be returned if the value has an error.

If the value_error is an empty cell, then the function takes the error value as empty string.

Example:

FORMULA	RESULT
= IFERROR (200/55, “ERROR in DIVISION”)	3
= IFERROR (200/0, “ERROR in DIVISION”)	ERROR in DIVISION

4.5.6.17 T

The T function tests whether the given value is text or not. If the given value is text, then it returns the given text. Otherwise, the function returns an empty text string.

Syntax

The syntax of the T function is

=T(value)

value – Required. This is a value to be checked.

If the value is not a number or logical value, then the function returns an empty string.

Example:

FORMULA	RESULT
=T(“SYNCFUSION”)	SYNCFUSION
=T(TRUE)	
=T(10)	

4.5.6.18 XOR

The XOR function returns the exclusive OR for the given arguments.

Syntax

XOR (logical_value1, logical_value2,…)

Logical_value1: Required. This can be either TRUE or FALSE, and can be logical values, arrays, or references.

Notes

If the given arguments do not have the logical values, XOR returns the #VALUE! error value.

Example

FORMULA	RESULT
= XOR(5>0, 7<9)	FALSE
=XOR(3>12, 4>6)	TRUE

4.5.6.19 IFNA

The IFNA function returns the value specified if the formula returns the #N/A error value; otherwise, it returns the result of the given formula.

Syntax

=IFNA (Formula_value, value_if_na)

Formula_value: Required. The argument that is checked for the #N/A error value.

value_if_na: Required. The value returned if the formula evaluates to the #N/A error value.

Example

FORMULA	RESULT
=IFNA(“#N/A”,”Incorrect”)	Incorrect
=IFNA(1602,”incorrect”)	1602

4.5.6.20 CLEAN

The CLEAN function is used to remove the non-printable characters from the given text, represented by numbers 0 to 31 of the 7-bit ASCII code.

Syntax

=Clean(Text)

Text: Required. String or text from which to remove nonprintable characters.

Example

FORMULA	RESULT
=Clean(Syncfusion)	Syncfusion
= Clean("Text*")	Text

4.5.6.21 ISREF

The ISREF function returns the logical value TRUE if the given value is a reference value; otherwise, the function returns FALSE.

Syntax

=ISREF(given_value)

given_value: Required. The value that is to be tested. The value argument can be a blank (empty cell), error, logical value, text, number, or reference value, or a name referring to any of these.

Example

FORMULA	RESULT
=ISREF("Region1")	FALSE
=ISREF(=ISLOGICAL(TRUE))	TRUE

4.5.6.22 AVERAGEIF

The AVERAGEIF function finds the average of values in a given array that satisfy a given criteria, and returns the average value of the corresponding values in a second given array.

Syntax

=AVERAGEIF(range, criteria, average_range)

range: Array of values to be tested against the given criteria.

criteria: The condition to be tested in each of the values of the given range.

average_range: Numeric values to be evaluated against the criteria and averaged.

Notes

- If range is blank or a text value, AVERAGEIF returns the #DIV/0! error value.
- If a cell in criteria is empty, AVERAGEIF treats it as a 0 value.

- If no cells in the range meet the criteria, AVERAGEIF returns the #DIV/0! error value.

Example

Input Table

	A	B
1	Earning	Tax
2	100000	3000
3	200000	6000
4	300000	7500
5	400000	9000

FORMULA	RESULT
=AVERAGEIF(B2:B5,"<7000")	4500
=AVERAGEIF(A2:A5,">250000")	350000

4.5.6.23 AVERAGEIFS

The AVERAGEIFS function finds the average of values in a given array that satisfy a set of given criteria.

Syntax

= AVERAGEIFS(average_range, criteria_range1, criteria1, [criteria_range2, criteria2], ...)

average_range: Specific set of values to be averaged if the criteria range meets the provided criteria.

criteria_range1: Array of values to be tested against the given criteria.

criteria1: The condition to be tested on each of the values of the given range.

Notes

- If average_range is blank or a text value, AVERAGEIFS returns the #DIV/0! error value.
- If a cell in a criteria range is empty, AVERAGEIFS treats it as a 0 value.
- If cells in average_range cannot be translated into numbers, AVERAGEIFS returns the #DIV/0! error value.

Example

Input Table

	A	B	C
1	Earning	Tax	other
2	100000	3000	100
3	200000	6000	200
4	300000	7500	300
5	400000	9000	500

FORMULA	RESULT
AVERAGEIFS(C2:C5, B2:B5, ">7000", B2:B5, "<10000")	400

4.5.6.24 NETWORKDAYS

The NETWORKDAYS function is used to calculate the number of whole work days between two given dates. This includes all weekdays from Monday to Friday, but excludes a supplied list of holidays.

Syntax

= NETWORKDAYS(start_date, end_date, [holidays])

start_date: The start of the period to find the working days

end_date: The end of the period to find the working days.

[holidays]: An optional argument, which specifies an array of dates that are not to be counted as working days.

Notes

- If any argument is not a valid date, NETWORKDAYS returns the #VALUE! error value.

Example

FORMULA	RESULT
=NETWORKDAYS(DATE(2012,10,1),DATE(2013,3,1))	110

4.5.6.25 SUMIFS

The SUMIFS function sums the values in a given array that satisfy a set of given criteria.

Syntax:

=SUMIFS(sum_range, criteria_range1, criteria1, [criteria_range2, criteria2], …)

criteria_range1: Array of values to be tested against the given criteria.

criteria1: The condition to be tested on each of the values of given range.

sum_range: The range of values to be summed if the associated criteria range meets the specified criteria.

Notes

- Cells in the sum_range argument that contain TRUE evaluate to 1; cells in sum_range that contain FALSE evaluate to 0 (zero).

Example

Input Table

	A	B	C
1	Earning	Tax	other
2	100000	3000	100
3	200000	6000	200
4	300000	7500	300
5	400000	9000	500

FORMULA	RESULT
SUMIFS(C2:C5, B2:B5, ">7000", B2:B5, "<10000")	800

4.5.6.26 ADDRESS

The ADDRESS function returns the address of a cell in a worksheet given specified row and column numbers.

Syntax

ADDRESS(row_num, column_num, [abs_num], [a1], [sheet_text])

row_num: A numeric value that specifies the row number.

column_num: A numeric value that specifies the column number

abs_num: Optional. A numeric value that specifies the type of reference to return.

A1: A logical value that specifies the A1 or R1C1 reference style.

Example

FORMULA	RESULT
=ADDRESS(2,3,2,FALSE)	R2C[3].
=ADDRESS(2,3,1,FALSE,"[Book1]Sync1")	[Book1] Sync1!R2C3

4.5.6.27 LOOKUP

The LOOKUP function returns a value either from a one-row or one-column range, or from an array. The LOOKUP function has two syntax forms: vector and array.

Vector Form: The vector form of LOOKUP looks in a one-row or one-column range for a value, and then returns a value from the same position in a second one-row or one-column range.

Syntax

=LOOKUP(lookup_value, lookup_vector, result_vector)

Array form: The array form of LOOKUP looks in the first row or column of an array for the specified value, and then returns a value from the same position in the last row or column of the array.

Syntax

=LOOKUP(lookup_value, array)

Notes

- If the LOOKUP function can't find the lookup_value, the function matches the largest value in lookup_vector that is less than or equal to lookup_value.
- If lookup_value is smaller than the smallest value in lookup_vector, LOOKUP returns the #N/A error value.

Example

Input Table

	A	B	C
1	Earning	Tax	other
2	100000	3000	100
3	200000	6000	200
4	300000	7500	300
5	400000	9000	500

FORMULA	RESULT
=LOOKUP(6000,B2:B5,C2:C5)	200
=LOOKUP("C", {"a","b","c","d";1,2,3,4})	3

4.5.6.28 SEARCH

The SEARCH function returns the location of a substring in a string. This function is NOT case-sensitive.

Syntax

SEARCH(substring, string, [start_position])

substring: Required. The text to be found.

string: Required. The text in which to search for the value of the substring.

start_num: Optional. The starting position for searching in the string.

Notes

- If the value of find_text is not found, the #VALUE! error value is returned.
- If the start_num argument is omitted, it is assumed to be 1.
- If start_num is not greater than 0, or is greater than the length of the string argument, the #VALUE! error value is returned.

Example

FORMULA	RESULT
=SEARCH("base","database")	5

4.5.7 Statistics

This section lists the Statistic functions included with Essential calculate in the below table.

Table 11: Statistic functions

Function Name	Description
Avedev	Returns average deviation.
Average	Returns the arithmetic mean.
Averagea	Returns the arithmetic mean.
Binomdist	Returns the cumulative beta probability density function.
Chidist	Returns the chi-squared distribution.
Chiinv	Returns the inverse of Chidist.
Chitest	Returns the chi-squared distribution test for independence.
Confidence	Returns the confidence interval.
Correl	Returns the correlation between two sets of data.
Count	Returns the count of the data listed in the arguments.
Counta	Returns the count of the data listed in the arguments.
Countblank	Returns the count of the empty cells listed in the arguments.
Countif	Returns the count of values that meet a specified criteria.
Covar	Returns covariance.
CritBinom	Returns the critical value for the cumulative binomial distribution.
Devsq	Returns the sum of the squares of the mean deviation.
Expondist	Returns the exponential distribution.
Fdist	Returns the F probability distribution.
Finv	Returns the inverse of Fdist.

Fisher	Returns the Fisher transformation.
Fisherinv	Returns the inverse of Fisher.
Forecast	Returns the value forecasted by the linear trend.
Gammadist	Returns the gamma distribution.
Gammainv	Returns the inverse of Gammadist.
Gammaln	Returns the natural logarithm of gamma function.
Geomean	Returns the geometric mean.
Harmean	Returns the harmonic mean.
Hypgeomdist	Returns the hypergeometric distribution.
Intercept	Returns the Y intercept of the least squares fit line.
Kurt	Returns the kurtosis of the set of arguments.
Large	Returns the kth largest value.
Loginv	Returns the inverse of the Lognormdist.
Lognormdist	Returns the cumulative lognormal distribution function.
Max	Returns the largest value among the arguments.
Maxa	Returns the largest value among the arguments.
Median	Returns the median value among the arguments.
Min	Returns the smallest value among the arguments.
Mina	Returns the smallest value among the arguments.
Mode	Returns the most frequently occurring value.
Negbinomdist	Returns the negative binomial distribution.
Normdist	Returns the normal cumulative distribution.
Norminv	Returns the inverse of Normdist.
Pearson	Returns the Pearson product.

Percentile	Returns the kth percentile of the given values.
Percentrank	Returns the position of a value in the range of values.
Permut	Returns the number of permutations.
Poisson	Returns the Poisson distribution.
Prob	Returns a probability.
Quartile	Returns which quarter a value belongs to within an ordered set of data.
Rank	Returns the position of a value in an ordered list.
Rsq	Returns the square of the Pearson product moment correlation coefficients.
Skew	Returns the skewness of a distribution.
Slope	Returns the slope of the linear regression line.
Small	Returns the kth smallest value.
Standardize	Returns the normalized value.
Stdev	Returns the sample standard deviation.
Stdeva	Returns the sample standard deviation.
Stdevp	Returns the population standard deviation.
Stdevpa	Returns the population standard deviation.
Steyx	Returns the standard error.
Trimmean	Returns the mean after removing out-liers.
Var	Returns the sample variance.
Vara	Returns the sample variance.
Varp	Returns the population variance.
Varpa	Returns the population variance.
Weibull	Returns the Weibull distribution.

Ztest	Returns the P-value of a z-test.

4.6 Inside CalcEngine

Following are the topics discussed in this section.

4.6.1 Tracking the Information

To track information used during calculations, **CalcEngine** manages several hash tables. Here is a table of the public hash tables in CalcEngine and a description of their keys and values:

Table 12: Hash Table

Hash Table	Key	Value	Description
FormulaInfoTable	Cell reference	FormulaInfo object	Tracks formula/value information for this cell.
DependentCells	Cell reference	Hashtable object	Tracks cells that depend on this cell.
DependentFormulaCells	Formula cell reference	Hashtable object	Tracks cells that the formula cell depends upon.
NamedRanges	Name string	Value string	Associates the named range with its value.
LibraryFunctions	Function name	LibraryFunction delegate	Associates the function name with its method.

Within CalcEngine, all data is assumed to be part of a rectangular array reference through cell coordinates like A1, C18, and so on. Even **CalcQuickBase** does not require or use such cell-type notation internally on the user side. When it communicates with CalcEngine, it converts its [name]-type notation into cell references that CalcEngine can understand. It is these cell references that are used as keys for the first three listed hash tables.

4.6.2 Parsing

This section discusses the Parse function available for the CalcEngine.

CalcEngine.Parse method does the following:

- Accepts a string formula, for example = A2 + 5.
- Checks whether it is a valid formula that **CalcEngine** can understand
- Returns a string that represents a parsed version of the formula that can be more readily computed.

The parsed formula is a Reverse Polish Notation expression using tokens to compactly represent the entered formula. The parsing recognizes and replaces **NamedRanges** with their corresponding value. The parsing also recognizes library functions and tokenises them as well.

4.6.3 Calculating

This section discusses the Calculate function available for the CalcEngine.

CalcEngine.Calculate is the method that actually performs calculations. It does the following:

- It accepts a parsed formula
- Uses a stack oriented calculation technique to convert the parsed formula into the value that it represents.

 ***Note: The value returned is a string holding the computed quantity.***

4.6.4 How Things Work

1. What happens when you enter the formula = A1 + 5 into a cell in a CalcSheet object?

2. Here lets assume that CalcSheet is using its own internal data storage to hold values, so that it makes it simple to understand what is going on within CalcEngine. If the data is being held in some other object (like a DataGrid with a DataTable datasource) things will look the same from within the CalcEngine.

3. Here is a sketch of the major steps taken within the library code when you enter a formula into a cell assuming CalcEngine.UseDependencies is True. The actual processing is more involved; these steps should give you an outline of what happens:

4. The string is tested to see whether it begins with an equal sign. If not, CalcSheet stores the entered string in its internal memory so that it will be available if needed. A check is made to see if this cell is a key in the DependentCells collection. If it is, then all cells depending upon this cell are recomputed. This recomputing is a recursive process as changing a cell, that depends upon the changed cell which triggers the recomputing needs of the newly changed cell and so on.

5. If the entered string does begin with an equal sign, the CalcEngine sees this as an entered formula. At this point, the CalcEngine checks to see if the cell is a key in the FormulaInfoTable.

6. If the cell is a key in the FormulaInfoTable, the corresponding FormulaInfo object is retrieved and updated. This amounts to the following:

   - Parsing the string.
   - Computing the string.
   - Saving the original formula, the parsed formula and the computed value in the FormulaInfo object.

7. If the cell is not a key in the FormulaInfoTable, a new FormulaInfo object is created. This new FormulaInfo object is populated from the entered string. This amounts to the following:

   - Parsing the string.
   - Computing the string.
   - Saving the original formula, the parsed formula and the computed value in the FormulaInfo object.

There are several other scenarios that must be handled in the CalcEngine. They include things like the newly entered string changes from an existing formula cell to a non-formula cell. In this situation, the CalcEngine uses the DependFormulaCells collection to remove dependencies that are no longer needed.

All this dependent tracking is done conditionally depending upon CalcEngine.UseDependencies. Additionally, you can turn off formula calculations using **CalcEngine.CalculatingSuspended**.

4.6.5 Error Messages

The error messages that are displayed by Essential Calculate can be found in this string array in the **CalcEngine**. After a CalcEngine object has been created, you can change the text of these messages by changing the array values.

```
[C#]
```



```
public string[] FormulaErrorStrings = new string[]
{
        "binary operators cannot start an expression",          //0
        "cannot parse",                                         //1
        "bad library",                                          //2
        "invalid char in front of",                             //3
        "number contains 2 decimal points",                     //4
        "expression cannot end with an operator",               //5
        "invalid characters following an operator",             //6
        "invalid character in number",                          //7
        "mismatched parentheses",                               //8
        "unknown formula name",                                 //9
        "requires a single argument",                           //10
        "requires 3 arguments",                                 //11
        "invalid Math argument",                                //12
        "requires 2 arguments",                                 //13
        "#NAME?",                                               //14
        "too complex",                                          //15
        "circular reference: ",                                 //16
        "missing formula",                                      //17
        "improper formula",                                     //18
        "invalid expression",                                   //19
        "cell empty",                                           //20
        "bad formula",                                          //21
        "empty expression",                                     //22
        "",                                                     //23
        "mismatched string quotes",                             //24
        "wrong number of arguments",                            //25
        "invalid arguments",                                    //26
        "iterations do not converge",                           //27
        "Control named '{0}' is already registered"             //28
};
```

4.7 Function Reference Section

This section discusses the library functions that are shipped in the Essential Calculate library. The arguments required by each of these functions are listed in bold text. Optional arguments are listed in a normal text.

4.7.1 ABS

Returns the absolute value of a number. The absolute value of a non-negative number is the number itself. The absolute value of a negative number is -1 times the number.

Syntax

ABS(number)

where:

number is the real number for which you want the absolute value.

4.7.2 ACOS

Returns the inverse cosine of a number. Inverse cosine is also referred to as arccosine. The arccosine is the angle whose cosine is the given number. The returned angle is given in radians in the range of 0 to pi.

Syntax

ACOS(number)

where:

number is the cosine of the angle that you want and must be between -1 and 1.

4.7.3 ACOSH

Returns the inverse hyperbolic cosine of a number. The number must be greater than or equal to 1. The inverse hyperbolic cosine is the value whose hyperbolic cosine is the given number.

Syntax

ACOSH(number)

where:

number is any real number that is greater than or equal to 1.

4.7.4 AND

Returns **True** if all the arguments have a logical value of True and returns False if at least one argument is False.

Syntax

AND(logical1, logical2, ...)

where:

logical1, logical2, ... are multiple conditions you want to test for True or False.

Remarks

- The arguments must evaluate to logical values (True or False).
- If an argument does not evaluate to True or False, those values are ignored.
- There must be at least one value in the argument list.

4.7.5 ASIN

Returns the inverse sine of a number. Inverse sine is also referred to as arcsine. The arcsine is the angle whose sine is the given number. The returned angle is given in radians in the range from -pi/2 to +pi/2.

Syntax

ASIN(number)

where:

number is the sine of the angle that you want and must be between -1 and 1.

4.7.6 ASINH

Returns the inverse hyperbolic sine of a number. The inverse hyperbolic sine is the value whose hyperbolic sine is the given number, so ASINH(SINH(number)) equals number.

Syntax

ASINH(number)

where:

number is any real number.

4.7.7 ATAN

Returns the inverse tangent of a number. Inverse tangent is also known as arctangent. The arctangent is the angle whose tangent is a number. The returned angle is given in radians in the range from -pi/2 to +pi/2.

Syntax

ATAN(number)

where:

number is the tangent of the angle that you want.

4.7.8 ATAN2

Returns the inverse tangent of the specified x and y co-ordinates. The arctangent is the angle from the x-axis to a line containing the origin (0, 0) and the point (x_num, y_num). The angle is given in radians between -pi and pi, excluding -pi.

Syntax

ATAN2(x_num,y_num)

where:

x_num is the X coordinate of the point.

y_num is the Y coordinate of the point.

Remarks

- A positive result represents a counterclockwise angle from the x-axis; a negative result represents a clockwise angle.
- ATAN2(a,b) equals ATAN(b/a), except that a can equal 0 in ATAN2.

4.7.9 ATANH

Returns the inverse hyperbolic tangent of a number. Number must be strictly between -1 and 1. The inverse hyperbolic tangent is the value whose hyperbolic tangent is number, so ATANH(TANH(number)) equals the given number.

Syntax

ATANH(number)

where:

number is any real number that is between 1 and -1.

4.7.10 AVEDEV

Returns the average of the absolute mean deviations of data points. AVEDEV is a measure of the variability in a data set.

Syntax

AVEDEV(number1, number2, ...)

where:

number1, number2, ... are arguments for which you want the average of the absolute deviations. You can also use a single array or a reference to an array instead of arguments separated by commas.

Remarks

- The arguments must either be numbers or names, arrays or references that contain numbers.
- If an array or reference argument contains text, logical values or empty cells, those values are ignored; however, cells with the a zero value are included.
- The equation for average deviation is:

$$\frac{1}{n}\sum|x-\bar{x}|$$

where:

x-bar is the arithmetic mean of the data.

4.7.11 AVERAGE

Returns the average (arithmetic mean) of the arguments.

Syntax

AVERAGE(number1, number2, ...)

where:

number1, number2, ... are numeric arguments for which you want the average.

Remarks

- The arguments must either be numbers or names, arrays or references that contain numbers.
- If an array or reference argument contains text, logical values or empty cells, those values are ignored; however, cells with a zero value are included.

4.7.12 AVERAGEA

Calculates the average (arithmetic mean) of the values in the list of arguments. In addition to numbers and text logical values such as True and False are also included in the calculation.

Syntax

AVERAGEA(value1, value2,...)

where:

value1, value2, ... are cells, ranges of cells, or values for which you want the average.

Remarks

- The arguments must be numbers, names, arrays, or references.
- Array or reference arguments that contain text evaluate as 0 (zero). If the calculation should not include text values in the average, then use the AVERAGE function.
- Arguments that contain True evaluate as 1; arguments that contain False evaluate as 0 (zero).

4.7.13 AVG

Returns the average (arithmetic mean) of the arguments.

Syntax

AVG(number1, number2,...)

where:

number1, number2, ... are numeric arguments for which you want the average.

Remarks

- This method is the same as AVERAGE and is included for compatibility purposes.

4.7.14 BINOMDIST

Returns the individual term binomial distribution probability.

Syntax

BINOMDIST(number_s, trials, probability_s, cumulative)

where:

number_s is the number of successes in trials.

trials is the number of independent trials.

probability_s is the probability of success on each trial.

Cumulative is a logical value that determines the form of the function. If cumulative is True, then BINOMDIST returns the cumulative distribution which, is the probability that there are at most number_s successes; if False, it returns the probability that there are exactly number_s successes.

Remarks

- Number_s and trials are truncated to integers.
- Number_s should be >= 0 and <= trials.
- Probability_s should be >=0 and <= 1.
- The binomial probability mass function is:

$$b(x,n,p) = \binom{n}{x} p^{n}(1-p)^{n-x}$$

where:

$\binom{n}{x}$ is COMBIN(n,x).

- The cumulative binomial distribution is:

$$B(x,n,p) = \sum_{y=0}^{x} b(y,n,p)$$

4.7.15 CEILING

Returns number rounded up, away from zero, to the nearest multiple of significance. For example, if you want to avoid using pennies in your prices and your product is priced at $4.82, use the formula =CEILING(4.82,0.05) to round prices up to the nearest nickel.

Syntax

CEILING(number, significance)

where:

number is the value you want to round off.

significance is the multiple to which you want to round.

Remarks

- Both values must be numeric.
- Regardless of the sign of a number, a value is rounded up when adjusted away from zero. If the number is an exact multiple of significance, no rounding occurs.

4.7.16 Char

The **Char** function returns the character whose number code is defined in the parameter.

Syntax:

Char(number)

where,

- **number** is the numeric value to retrieve the character.

4.7.17 CHIDIST

Returns the one-tailed probability of the chi-squared (□□) distribution. The □□ distribution is associated with a □□ test.

Syntax

CHIDIST(x, degrees_freedom)

where:

x is the value at which you want to evaluate the distribution.

degrees_freedom is the number of degrees of freedom.

Remarks

- Both arguments should be numeric.
- degrees_freedom >= 1 and < 10^10.
- CHIDIST is calculated as follows:

CHIDIST = P(X > x)

where:

X is a χ^2 random variable.

4.7.18 CHIINV

Returns the inverse of the one-tailed probability of the chi-squared (χ^2) distribution. If probability = CHIDIST(x,...), then CHIINV(probability,...) = x. Use this function to compare observed results with expected ones in order to decide whether your original hypothesis is valid.

Syntax

CHIINV(probability, degrees_freedom)

where:

probability is a probability associated with the chi-squared distribution.

degrees_freedom is the number of degrees of freedom.

Remarks

- Probability must be >= 0 and <= 1.
- degrees_freedom >=1 and = 10^10.

Given a value for probability, CHIINV seeks the value x such that CHIDIST(x, degrees_freedom) = probability. Thus, precision of CHIINV depends on precision of CHIDIST. CHIINV uses an iterative search technique.

4.7.19 CHITTEST

Returns the test for independence. CHITEST returns the value from the chi-squared (c2) distribution for the statistic and the appropriate degrees of freedom.

Syntax

CHITEST(actual_range, expected_range)

where:

actual_range is the range of data that contains observations to test against expected values.

expected_range is the range of data that contains the ratio of the product of row totals and column totals to the grand total.

Remarks

- The χ^2 test first calculates a χ^2 statistic using the formula:

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{\left(A_{ij} - E_{ij}\right)^2}{E_{ij}}$$

where:

Aij = actual frequency in the i-th row, j-th column

Eij = expected frequency in the i-th row, j-th column

r = number of rows

c = number of columns

A low value of χ^2 is an indicator of independence. The use of CHITEST is most appropriate when Eij's are not too small. Some statisticians suggest that each Eij should be greater than or equal to 5.

4.7.20 Choose

The **Choose** function returns the value from a range of values on a specific index.

Syntax:

Choose(index, valuearray)

where,

- **index** is to specify the index from where you want to retrive the value.
- **valuearray** is the array of value from where you want to take the value.

4.7.21 Column

The **Column** function returns the column index of the provided column in range.

Syntax:

Column(range)

where,

- **range** is to provide the column in range.

4.7.22 COMBIN

Returns the number of combinations for a given number of items. Use COMBIN to determine the total possible number of groups for a given number of items.

Syntax

COMBIN(number, number_chosen)

where:

number is the number of items.

number_chosen is the number of items in each combination.

Remarks

- Numeric arguments are truncated to integers.
- A combination is any set or subset of items, regardless of their internal order. Combinations are distinct from permutations where the internal order is significant.
- The number of combinations is as follows, where number = n and number_chosen = k:

$$\binom{n}{k} = \frac{P_{k,n}}{k!} = \frac{n!}{k!\,(n-k)!}$$

where:

$$P_{k,n} = \frac{n!}{(n-k)!}$$

4.7.23 CONCATENATE

Joins several text strings into one text string.

Syntax

CONCATENATE (text1, text2,...)

where:

text1, text2, ... are text items to be joined into a single text item. The text items can be text strings, numbers, or single-cell references.

Remarks

- The **"&"** operator can be used instead of CONCATENATE to join text items.

4.7.24 CONFIDENCE

Returns a value that you can use to construct a confidence interval about a population mean. The confidence interval is a range of values. In your sample, mean x is at the center of this range and the range is x ± CONFIDENCE. For example, if x is the sample mean of delivery times for products ordered through the mail, x ± CONFIDENCE is a range of population means.

Syntax

CONFIDENCE(alpha, standard_dev,size)

where:

alpha is the significance level used to compute the confidence level. The confidence level equals 100*(1 - alpha)%, or in other words, an alpha of 0.05 indicates a 95 percent confidence level.

standard_dev is the population standard deviation for the data range and is assumed to be known.

size is the sample size.

Remarks

- All arguments must be non-numeric.
- Alpha must be > 0 and < 1.
- Standard_dev must be > 0.
- Size must be >= 1.

4.7.25 CORREL

Returns the correlation coefficient of the array1 and array2 cell ranges.

Syntax

CORREL(array1, array2)

where:

array1 is a cell range of values.

array2 is the second cell range of values.

Remarks

- **array1** and **array2** must have the same number of data points.
- The equation for the correlation coefficient is:

$$Correl(X,Y) = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sqrt{\sum(x - \bar{x})^2 \sum(y - \bar{y})^2}}$$

where:

x-bar and **y-bar** are the sample means AVERAGE(array1) and AVERAGE(array2).

4.7.26 COS

Returns the cosine of the given angle.

Syntax

COS(number)

where:
number is the angle in radians for which you want the cosine.

4.7.27 COSH

Returns the hyperbolic cosine of a number.

Syntax

COSH(number)

where:
number is any real number for which you want to find the hyperbolic cosine.

Remarks

- The formula for the hyperbolic cosine is:

$$\cosh(z) = \frac{e^{z} + e^{-z}}{2}$$

4.7.28 COUNT

Counts the number of items in a list that contains numbers.

Syntax

COUNT(value1, value2,...)

where:

value1, value2, ... are arguments that can contain or refer to a variety of different types of data, but only numbers are counted.

Remarks

- Arguments that are numbers, dates or text representations of numbers are counted; arguments that are error values or text that cannot be translated into numbers are ignored.
- If an argument is an array or reference, only numbers in that array or reference are counted. Empty cells, logical values, text or error values in the array or reference are ignored.

4.7.29 COUNTA

Counts the number of cells that are not empty.

Syntax

COUNTA(value1, value2,...)

where:

value1, value2, ... are arguments representing the values you want to count. In this case, a value is any type of information, excluding empty cells.

4.7.30 COUNTBLANK

Counts empty cells in a specified range of cells.

Syntax

COUNTBLANK(range)

where:

range is the range from which you want to count the blank cells.

Remarks

- Cells with formulas that return "" (empty text) are also counted. Cells with zero values are not counted.

4.7.31 COUNTIF

Counts the number of cells within a range that meet the given criteria.

Syntax

COUNTIF(range, criteria)

where:

range is the range of cells from which you want to count cells.

criteria is the criteria in the form of a number, expression or text that defines which cells will be counted. For example, the criteria can be expressed as ">32".

Remarks

- If and Sumif are other library functions that can be used to conditionally compute values.

4.7.32 COVAR

Returns covariance, the average of the products of deviations for each data point pair.

Syntax

COVAR(array1, array2)

where:

array1 is the first cell range of numbers.

array2 is the second cell range of numbers.

Remarks

- The arguments must either be numbers or be names, arrays or references that contain numbers.
- **array1** and **array2** must have the same number of data points.
- The covariance is:

$$Cov(X,Y) = \frac{\sum(x - \bar{x})(y - \bar{y})}{n}$$

where:

X is array1

Y is array2

x-bar and **y-bar** are the sample means AVERAGE(array1) and AVERAGE(array2).

n is the sample size.

4.7.33 CRITBINOM

Returns the smallest value for which, the cumulative binomial distribution is greater than or equal to a criterion value.

Syntax

CRITBINOM(trials, probability_s, alpha)

where:

trials is the number of Bernoulli trials.

probability_s is the probability of a success on each trial.

alpha is the criterion value.

Remarks

- Trials must be >= 0.
- Probability_s must be >=0 and <= 1.
- Alpha must be >= 0 and <= 1.

4.7.34 DATE

Returns the sequential serial number that represents a particular date.

Syntax

DATE(year, month, day)

where:

year can be one to four digits. Year is interpreted based on 1900.

- If a year is between 0 (zero) and 1899 (inclusive), the value is added to 1900 to calculate the year. For example, DATE(102,11,12) returns November 12, 2002 (1900+102).
- If a year is between 1900 and 9999 (inclusive), the value is used as is, for example, DATE(2002,11,12) returns November 12, 2002.

month is a number representing the month of the year.

day is a number representing the day of the month.

Remarks

- Dates are stored as sequential serial numbers so that they can be used in calculations. By default, January 1, 1900 is serial number 1 and November 12, 2002 is serial number 37572 because it is 37572 days after January 1, 1900.

4.7.35 DATEVALUE

Returns the serial number of the date represented by the date_text.

Syntax

DATEVALUE(date_text)

where:

date_text is the text that represents a date as a formatted string. For example, "11/12/2002" or "12-Nov-2002" are text strings within quotation marks that represent dates. If the year portion of the date_text is omitted, DATEVALUE uses the current year from your computer's built-in clock. The time information in the date_text is ignored.

Remarks

- Dates are stored as sequential serial numbers so that they can be used in calculations. By default, January 1, 1900 is serial number 1, and November 12, 2002 is serial number 37572 because it is 37572 days after January 1, 1900.
- Most functions automatically convert date values to serial numbers.

4.7.36 DAY

Returns the day of a date, represented by a serial number. The day is given as an integer ranging from 1 to 31.

Syntax

DAY(serial_number)

where:

serial_number is the date of the day you are trying to find. Dates should be entered by using the DATE function or as results of other formulas or functions. For example, use DATE(2002,4,23) for the 23rd day of April, 2002.

4.7.37 DAYS360

Returns the number of days between two dates based on a 360-day year (twelve 30-day months) which, is used in some accounting calculations.

Syntax

DAYS360(start_date, end_date, method)

where:

start_date and end_date are the two dates between which, you want to know the number of days. If start_date occurs after end_date, DAYS360 returns a negative number. Dates should be entered by using the DATE function or as results of other formulas or functions.

method is a logical value that specifies whether to use the U.S. or European method in the calculation. If method is:

- **False or omitted** – The calculation uses the U.S. (NASD) method. If the starting date is the 31st of a month, it becomes equal to the 30th of the same month. If the ending date is the 31st of a month and the starting date is earlier than the 30th of a month, the ending date becomes equal to the 1st of the next month; otherwise the ending date becomes equal to the 30th of the same month.
- **True** – The calculation uses the European method. Starting dates and ending dates that occur on the 31st of a month become equal to the 30th of the same month.

4.7.38 DB

Returns the depreciation of an asset for a specified period using the fixed-declining balance method.

Syntax

DB(cost, salvage, life, period, month)

where:

cost is the initial cost of the asset.

salvage is the value at the end of the depreciation (sometimes called the salvage value of the asset).

life is the number of periods over which, the asset is being depreciated (sometimes called the useful life of the asset).

period is the period for which, you want to calculate the depreciation. Period must use the same units as life.

month is the number of months in the first year. If month is omitted, it is assumed to be 12.

Remarks

- The fixed-declining balance method computes the depreciation at a fixed rate. DB uses the following formulas to calculate the depreciation for a period:

(cost - total depreciation from prior periods) * rate

where:

rate = 1 - ((salvage / cost) ^ (1 / life)), rounded to three decimal places

- Depreciation for the first and last periods is a special case. For the first period, DB uses this formula:

cost * rate * month / 12

- For the last period, DB uses this formula:

((cost - total depreciation from prior periods) * rate * (12 - month)) / 12

4.7.39 DDB

Returns the depreciation of an asset for a specified period using the double-declining balance method or some other method you specify.

Syntax

DDB(cost, salvage, life, period, factor)

where:

cost is the initial cost of the asset.

salvage is the value at the end of the depreciation (sometimes called the salvage value of the asset).

life is the number of periods over which, the asset is being depreciated (sometimes called the useful life of the asset).

period is the period for which, you want to calculate the depreciation. Period must use the same units as life.

factor is the rate at which, the balance declines. If factor is omitted, it is assumed to be 2 (the double-declining balance method).

 ***Note: All five arguments must be positive numbers.***

Remarks

- The double-declining balance method computes the depreciation at an accelerated rate. Depreciation is highest in the first period and decreases in successive periods. DDB uses the following formula to calculate depreciation for a period:

((cost-salvage) - total depreciation from prior periods) * (factor/life)

4.7.40 DEGREES

Converts radians into degrees.

Syntax

DEGREES(angle)

where:

angle is the angle in radians that you want to convert.

4.7.41 DEVSQ

Returns the sum of squares of deviations of data points from their sample mean.

Syntax

DEVSQ(number1, number2,...)

where:

number1, number2, ... are arguments for which, you want to calculate the sum of squared deviations. You can also use a single array or a reference to an array instead of arguments separated by commas.

Remarks

- The arguments must be numbers or names, arrays or references that contain numbers.
- The equation for the sum of squared deviations is:

$$DEVSQ = \sum (x - \bar{x})^2$$

4.7.42 Dollar

The **Dollar** function converts a number to text, using a currency format.

The format used is $#,##0.00_);($#,##0.00).

Syntax:

Dollar (number, decimal_places)

where,

- number is the number you want to convert to text.
- decimal_places is the number of digits in decimal places you want to display. The value will be rounded accordingly.

4.7.43 EVEN

Returns the number rounded up to the nearest even integer.

Syntax

EVEN(number)

where:

number is the value that is to be rounded.

Remarks

- Regardless of the sign of the number a value is rounded up when adjusted away from zero. If the number is an even integer no rounding occurs.

4.7.44 Exact

The **Exact** function compares two values ignoring the styles and returns the boolean value as true or false.

Syntax:

Exact(value1, value2)

where,

- **value1** is the first value you want to compare .
- **value2** is the second value you want to compare.

4.7.45 EXP

Returns e raised to the power of the given number.

Syntax

EXP(number)

where:

number is the exponent applied to the base e.

4.7.46 EXPONDIST

Returns the exponential distribution.

Syntax

EXPONDIST(x, lambda, cumulative)

where:

x is the value of the function.

lambda is the parameter value.

cumulative is a logical value that indicates which, form of the exponential function is to be provided. If cumulative is True, EXPONDIST returns the cumulative distribution function; if False, it returns the probability density function.

Remarks

- The equation for the probability density function is:

$$f(x;\lambda) = \lambda e^{-\lambda x}$$

- The equation for the cumulative distribution function is:

$$F(x;\lambda)=1-e^{-\lambda x}$$

4.7.47 FACT

Returns the factorial of a number. The factorial of a number is the product of all positive integers <= the given number.

Syntax

FACT(number)

where:

number is the non-negative number for which you want the factorial of. If the number is not an integer, it is truncated.

4.7.48 False

The **False** function returns the logical value for the false.

Syntax:

False(stringvalue)

where:

- **stringvalue** is to provide an empty string.

4.7.49 FDIST

Returns the F probability distribution.

Syntax

FDIST(x, degrees_freedom1, degrees_freedom2)

where:

x is the value at which to evaluate the function.

degrees_freedom1 is the numerator degrees of freedom.

degrees_freedom2 is the denominator degrees of freedom.

Remarks

- All arguments must be numeric.
- X must be >= 0.
- Both degrees_freedom1 and degrees_freedom2 must be >= 1 and < 10^10.
- FDIST is calculated as follows:

FDIST=P(F>x)

where:

F is a random variable that has an F distribution with degrees_freedom1 and degrees_freedom2 degrees of freedom.

4.7.50 Find

The **Find** finction finds a portion of a string from a particular text and returns the location of the string.

Syntax:

Find(lookfor, lookin, start)

where,

- **lookfor** is the text you want to search.
- **lookin** is the the text in which you want to search.
- **start** specifies the starting position of the text from where you want to start searching in the text. This is optional.

4.7.51 Finv

The **Finv** function returns the inverse of the F probability distribution. If p = FDIST(x,...), then FINV(p,...) = x.

Using F distribution, you can compare the degree of variability for two data sets.

Syntax:

FINV(probability,deg_freedom1,deg_freedom2)

The FINV function syntax has the following three arguments (Argument is a value that provides information to an action, an event, a method, a property, a function, or a procedure):

- **Probability** is a probability associated with the F cumulative distribution.
- **Deg_freedom1** is the numerator degrees of freedom.
- **Deg_freedom2** is the denominator degrees of freedom.

4.7.52 FISHER

Returns the Fisher transformation at x. This transformation produces a function that is normally distributed rather than skewed.

Syntax

FISHER(x)

where:

x is a numeric value for which you want the transformation.

Remarks

- X must be > -1 and < 1.
- The equation for the Fisher transformation is:

$$z' = \frac{1}{2} ln\left(\frac{1+x}{1-x}\right)$$

4.7.53 FISHERINV

Returns the inverse of the Fisher transformation. If y = FISHER(x), then FISHERINV(y) = x.

Syntax

FISHERINV(y)

where:

y is the value for which you want to perform the inverse of the transformation.

Remarks

- The equation for the inverse of the Fisher transformation is:

$$x = \frac{e^{2y} - 1}{e^{2y} + 1}$$

4.7.54 Fixed

The **Fixed** function rounds off to a specified number of decimal places and returns the value in text format.

Syntax:
Fixed (number, decimal_places, no_commas)

where,

- number is the number you want to round.
- decimal_places is the number of decimal places you want to display in the result.
- no_commas is a logical value. This displays commas when it is set to FALSE, do not display commas when it is set to TRUE.

4.7.55 FLOOR

Rounds off the given number down, toward zero, to the nearest multiple of significance.

Syntax

FLOOR(number, significance)

where:

number is the numeric value that you want to round off.

significance is the multiple to which, you want to round the number off.

Remarks

- Number and significance must have the same sign.
- Regardless of the sign of the number, a value is rounded down when adjusted away from zero. If a number is an exact multiple of significance, no rounding occurs.

4.7.56 FORECAST

Calculates a future value by using existing values using a linear regression. The predicted value is a y-value for a given x-value.

Syntax

FORECAST(x, known_ys, known_xs)

where:

x is the data point for which you want to predict a value.

known_ys is the dependent array or range of data.

known_xs is the independent array or range of data.

Remarks

- The equation for FORECAST is:

a+bx

where:

$$a = \bar{y} - b\,\bar{x}$$

$$b = \frac{\sum (x - \bar{x})(y - \bar{y})}{\sum (x - \bar{x})^2}$$

x-bar and **y-bar** are the sample means AVERAGE(known_xs) and AVERAGE(known_ys).

4.7.57 FV

The **FV** function returns the future value of an investment, based on an interest rate and a constant payment schedule.

Syntax:

FV(interest_rate, number_payments, payment, PV, Type)

where,

- interest_rate is the interest rate for the investment.
- number_payments is the number of payments for the annuity.
- payment is the payment made on each period.
- PV is the present value of the payments. This is optional. The FV function assumes PV value as 0, when this parameter is omitted.
- Type indicates the payments due. Type accepts the following values:
    - 0 - Payments at the end of the period (default).
    - 1 - Payments at the beginning of the period.

This is optional. The FV function assumes Type value as 0, when this parameter is omitted.

4.7.58 GAMMADIST

Returns the gamma distribution.

Syntax

GAMMADIST(x, alpha,beta, cumulative)

where:

x is the value at which, you want to evaluate the distribution.

alpha is a parameter to the distribution.

beta is a parameter to the distribution. If beta = 1, GAMMADIST returns the standard gamma distribution.

cumulative is a logical value that determines the form of the function. If cumulative is True, GAMMADIST returns the cumulative distribution function; if False, it returns the probability density function.

Remarks

- X must be >= 0.
- Alpha and beta must be > 0.
- The equation for the gamma probability density function is:

$$f(x;\alpha,\beta) = \frac{1}{\beta^{\alpha}\Gamma(\alpha)} x^{\alpha-1} e^{\frac{-x}{\beta}}$$

- The standard gamma probability density function is:

$$f(x;\alpha) = \frac{x^{\alpha-1} e^{-x}}{\Gamma(\alpha)}$$

- When alpha = 1, GAMMADIST returns the exponential distribution with:

$$\lambda = \frac{1}{\beta}$$

4.7.59 Gammainv

The **Gammainv** function returns the inverse function for the GAMMADIST function.

Syntax:

Gammainv(p, alpha, beta)

where,

- **p** is the probability associated with the gamma distribution.
- **alpha** is a parameter of the distribution.
- **beta** is a parameter of the distribution.

4.7.60 GAMMALN

4.7.60.1 GAMMAINV

Returns the inverse of the gamma cumulative distribution. If p = GAMMADIST(x,...), then GAMMAINV(p,...) = x.

Syntax

GAMMAINV(probability, alpha, beta)

where:

probability is the probability associated with the gamma distribution.

alpha is a parameter to the distribution.

beta is a parameter to the distribution.

Remarks

- Probability must be >= 0 and <= 1.
- Alpha and beta must be positive.

Given a value for probability, GAMMAINV seeks value x such that GAMMADIST(x, alpha, beta, True) = probability. Thus, precision of GAMMAINV depends on the precision of GAMMADIST. GAMMAINV uses an iterative search technique.

4.7.61 GEOMEAN

Returns the geometric mean of an array or range of positive data.

Syntax

GEOMEAN(number1, number2,...)

where:

number1, number2, ... are arguments for which you want to calculate the mean.

Remarks

- The arguments must be either numbers or names, arrays or references that contain numbers.
- All values must be positive.
- The equation for the geometric mean is:

$$GM = \left(\prod_{1}^{n} y_i \right)^{\frac{1}{n}}$$

4.7.62 GROWTH

This feature enables you to calculate predicted exponential growth using existing data. This calculates and returns an array of values used for the regression analysis. Growth enables you to perform a regression analysis.

Table 13: Method Table

Method	Description	Parameters	Type	Return Type	Reference links
Growth()	Calculates the Growth for an array of cells.	Known y's, Known x's, new_x's	Method	String	N/A

The following is the formula to calculate Growth for an array of cells in a column:

[Syntax]

=GROWTH(known_y's, [known_x's], [new_x's],

Known_y's: A set of y-values you already know in a relationship, where y = b*m^x.

Known_x's: An optional set of x-values that you may already know in the relationship, where y = b*m^x.

New_x's: New x-values for which you want GROWTH to return corresponding y-values.

[Code]

=Growth(B2:B7,A2:A7,C6:C7)

4.7.63 HARMEAN

Returns the harmonic mean of a data set. The harmonic mean is the reciprocal of the arithmetic mean of reciprocals.

Syntax

HARMEAN(number1, number2,...)

number1, number2, ... are arguments for which you want to calculate the mean.

Remarks

- The arguments must be either numbers or names, arrays or references that contain numbers.
- All data values must be positive.
- The equation for the harmonic mean is:

$$H = \frac{n}{\sum \frac{1}{y_i}}$$

4.7.64 HLOOKUP

Searches for a value in the top row of the array of values and then returns a value in the same column from a row you specify in the array. Use HLOOKUP when your comparison values are located in a row across the top of a table of data and you want to look down a specified number of rows. Use VLOOKUP when your comparison values are located in a column to the left of the data you want to find.

Syntax

HLOOKUP(lookup_value, table_array, row_index_num, range_lookup)

where:

lookup_value is the value to be found in the first row of the table. Lookup_value can be a value, a reference or a text string.

table_array is a table of information in which, data is looked up. Use a reference to a range or a range name.

row_index_num is the row number in table_array from which, the matching value will be returned. A row_index_num of 1 returns the first row value in table_array, a row_index_num of 2 returns the second row value in table_array and so on.

range_lookup is a logical value that specifies whether you want HLOOKUP to find an exact match or an approximate match. If True or omitted, an approximate match is returned. In other words, if an exact match is not found, the next largest value that is less than the lookup_value is returned. (This requires your lookup values to be sorted.) If False, HLOOKUP will find an exact match.

4.7.65 HOUR

Returns the hour of a time value. The hour is given as an integer, ranging from 0 (12:00 A.M.) to 23 (11:00 P.M.).

Syntax

HOUR(serial_number)

where:

serial_number is the time that contains the hour you want to find. Times may be entered as text strings within quotation marks (for example, "6:00 PM"), as decimal numbers (for example, 0.75, which represents 6:00 PM), or as results of other formulas or functions (for example, TIMEVALUE("6:00 PM")).

4.7.66 Hypgeomdist

The **Hypgeomdist** function returns the hypergeometric distribution.

Syntax:

Hypgeomdist(sample, numberofsample, population, numberofpopulation)

where,

- **sample** is the number of successes in the sample.
- **numberofsample** is the size of the sample.
- **population** is the number of successes in the population.
- **numberofpopulation** is the population size.

4.7.67 HYPEGEOMDIST

Returns the hypergeometric distribution. HYPGEOMDIST returns the probability of a given number of sample successes, given the sample size, population successes and population size.

Syntax

HYPGEOMDIST(sample_s, number_sample, population_s, number_population)

where:

sample_s is the number of successes in the sample.

number_sample is the size of the sample.

population_s is the number of successes in the population.

number_population is the population size.

Remarks

- All arguments are truncated to integers.
- sample_s must be >= 0 less than both number_sample and population_s.
- number_sample must be >= 0 and < number_population.
- population_s must be >= 0 and < number_population.
- number_population must b >= 0.
- The equation for the hypergeometric distribution is:

$$P(X = x) = h(x;n,M,N) = \frac{\binom{M}{x}\binom{N - M}{n - x}}{\binom{N}{n}}$$

where:

x = sample_s

n = number_sample

M = population_s

N = number_population

4.7.68 IF

Returns one value if a condition you specify evaluates to True and another value if it evaluates to False.

Use IF to conduct conditional tests on values and formulas.

Syntax

IF(logical_test, value_if_true, value_if_false)

where:

logical_test is any value or expression that can be evaluated to True or False.

value_if_true is the value that is returned if a logical_test is True.

value_if_false is the value that is returned if a logical_test is False.

Remarks

- Countif and Sumif are additional methods that provide conditional calculations.

4.7.69 Index

The **Index** function returns the exact value from the provided row index and column index from a specific range.

Syntax:

Index(range,row,col)

where,

- **range** is a string to mention the specific range.
- **row** is the integer that indicates the specific row index.
- **col** is the integer that indicates the specific column index.

4.7.70 Indirect

The **Indirect** function returns the reference as a string instead of providing the content or range within it.

Syntax:

Indirect(content)

where,

- **content** is the string that provides the textual representation of the cell.

4.7.71 INT

Rounds a number down to the nearest integer.

Syntax

INT(number)

where:

number is the real number that you want to round down to an integer.

4.7.72 INTERCEPT

Calculates the point at which, the least squares fit line will intersect the y-axis.

Syntax

INTERCEPT(known_y's, known_x's)

where:

known_y's is the dependent set of observations or data.

known_x's is the independent set of observations or data.

Remarks

- The equation for the intercept of the regression line, a, is:

$$a = \bar{y} - b\,\bar{x}$$

where:

$$b = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sum(x - \bar{x})^2}$$

x-bar and **y-bar** are the sample means AVERAGE(known_x's) and AVERAGE(known_y's).

4.7.73 IPMT

Returns the interest payment for a given period for an investment based on periodic, constant payments and a constant interest rate.

Syntax

IPMT(rate, per, nper, pv, fv, type)

where:

rate is the interest rate per period.

per is the period for which, you want to find the interest and must be in the range 1 to nper.

nper is the total number of payment periods in an annuity.

pv is the present value or the lump-sum amount that a series of future payments is worth right now.

fv is the future value or a cash balance that you want to attain after the last payment is made. If fv is omitted, it is assumed to be 0 (the future value of a loan, for example, is 0).

type is the number 0 or 1 and indicates when payments are due. If type is omitted, it is assumed to be 0. If type = 0, payments are made at the end of the period. If type is 1, payments are made at the beginning of the period.

Remarks

- Make sure that you are consistent about the units you use for specifying rate and nper. If you make monthly payments on a four-year loan at 12 percent annual interest, use 12%/12 for rate and 4*12 for nper. If you make annual payments on the same loan, use 12% for rate and 4 for nper.

4.7.74 IRR

Returns the internal rate of return for a series of cash flows represented by the numbers in values. The cash flows must occur at regular intervals such as monthly or annually.

Syntax

IRR(values, guess)

where:

values is an array or a reference to cells that contain numbers for which, you want to calculate the internal rate of return. Values must contain at least one positive value and one negative value to calculate the internal rate of return. IRR uses the order of values to interpret the order of cash flows. Be sure to enter your payment and income values in the sequence you want.

guess is a number that you guess is close to the result of IRR. An iterative technique is used for calculating IRR. In most cases, you do not need to provide a guess for the IRR calculation. If a guess is omitted, it is assumed to be 0.1 (10 percent).

4.7.75 IsBlank

The **IsBlank** function checks for blank or null values.

Syntax:
IsBlank(value)

where,
value is the value that you want to test. If the value is blank, this function will return TRUE. If the value is not blank, the function will return FALSE.

4.7.76 IsErr

The IsErr function checks whether a value is an error.

Syntax:

IsErr(value)

where,

value is the value that you want to test. If the value is an error value (except #N/A), this function will return TRUE/FALSE to indicate whether a value is an error.

4.7.77 ISERROR

Returns True if the value is a string that starts with a #.

Syntax

ISERROR(value)

where:

value is the value that is to be tested.

4.7.78 IsLogical

The **IsLogiacl** function checks whether a value is a logical value and returns a TRUE or FALSE.

Syntax:
IsLogical(value)

where,
This value is the value that you want to test. If the value is a TRUE or FALSE value, this function will return TRUE. Otherwise, it will return FALSE.

4.7.79 IsNA

The **IsNA** function returns a boolean value after determining that the provided value is a #NA error value.

Syntax:
IsNA(value)

where,

- **value** the function will test.

4.7.80 IsNonText

The **IsNonText** function returns the boolean value after determining that the provided value is not a string.

Syntax:

IsNonText(text)

where,

- **text** is the value you want to test whether it is a string or not.

4.7.81 ISNUMBER

Returns True if the value parses as a numeric value.

Syntax
ISNUMBER(value)

where:

value is the value that is to be tested.

4.7.82 ISPMT

Calculates the interest paid during a specific period of an investment.

Syntax

ISPMT(rate, per, nper, pv)

where:

rate is the interest rate for the investment.

per is the period for which, you want to find the interest and must be between 1 and nper.

nper is the total number of payment periods for the investment.

pv is the present value of the investment. For a loan, pv is the loan amount.

Remarks

- Make sure that you are consistent about the units you use for specifying rate and nper. If you make monthly payments on a four-year loan at an annual interest rate of 12 percent, use 12%/12 for rate and 4*12 for nper. If you make annual payments on the same loan, use 12% for rate and 4 for nper.

4.7.83 IsText

The **IsText** function returns a boolean value after determining that the provided value is a string.

Syntax:

IsText(text)

where,

- **text** is the value you want to test to check if it is a string or not.

4.7.84 KURT

Returns the kurtosis of a data set. Kurtosis characterizes the relative peakedness or flatness of a distribution compared with the normal distribution. Positive kurtosis indicates a relatively peaked distribution. Negative kurtosis indicates a relatively flat distribution.

Syntax

KURT(number1, number2, ...)

where:

number1, number2, ... are arguments for which you want to calculate kurtosis. You can also use a single array or a reference to an array instead of arguments separated by commas.

Remarks

- The arguments must be either numbers or names, arrays or references that contain numbers.
- If an array or reference argument contains text, logical values or empty cells, those values are ignored; however, cells with the value zero are included.

- If there are fewer than four data points or if the standard deviation of the sample equals zero, KURT returns the #DIV/0! error value.
- Kurtosis is defined as:

$$\left\{\frac{n(n+1)}{(n+1)(n+2)(n+3)}\sum\left(\frac{x_i-\bar{x}}{s}\right)^4\right\}-\frac{3(n+1)^2}{(n+2)(n+3)}$$

where:

s is the sample standard deviation.

4.7.85 LARGE

Returns the k-th largest value in a data set.

Syntax

LARGE(array, k)

where:

array is the array or range of data for which, you want to determine the k-th largest value.

k is the position (from the largest) in the array or cell range of data to return.

Remarks

- If n is the number of data points in a range, then LARGE(array,1) returns the largest value, and LARGE(array,n) returns the smallest value.

4.7.86 LEFT

LEFT returns the first character or characters in a text string, based on the number of characters you specify.

Syntax

LEFT(text, num_chars)

where:

text is the text string that contains the characters which, you want to extract.

num_chars specifies the number of characters which, you want LEFT to extract.

Remarks

- Num_chars must be greater than or equal to zero.
- If num_chars is greater than the length of text, LEFT returns all the text.
- If num_chars is omitted, it is assumed to be 1.

4.7.87 LN

Returns the natural logarithm of a number. Natural logarithms are based on the constant e (2.718281828459...).

Syntax

LN(number)

where:

number is the positive real number for which, you want the natural logarithm.

Remarks

- LN is the inverse of the EXP function.

4.7.88 LEN

LEN returns the length of a text string, including spaces.

Syntax

Len(text)

where:

text is the text string whose length is to be determined.

4.7.89 LOG

Returns the logarithm of a number to the base that you specify.

Syntax

LOG(number, base)

where:

number is the positive real number for which, you want the logarithm.

base is the base of the logarithm. If base is omitted, it is assumed to be 10.

4.7.90 LOG10

Returns the base-10 logarithm of a number.

Syntax

LOG10(number)

where:

number is the positive real number for which, you want the base-10 logarithm.

4.7.91 LOGEST

This feature enables you to calculate predicted exponential growth using existing data. This calculates and returns an array of values used for the regression analysis. Logest calculates and returns an array of values that is used in regression analysis.

Table 14: Method Table

Method	Description	Parameters	Type	Return Type	Reference links
Logest()	Calculates Logest for an array of cells.	known_y's, known_x's, const, stats	**Method**	String	N/A

The following is the formula to calculate Logest for an array of cells in a column:

[Syntax]

=LOGEST(known_y's, [known_x's], [const], [stats])

Known_y's : A set of y-values you already know in a relationship, where y = b*m^x.

Known_x's : An optional set of x-values that you may already know in a relationship, where y = b*m^x.

Const : A logical value specifying whether to force the constant b to equal 1.

Stats : A logical value specifying whether to return additional regression statistics.

[Code]

= Logest(B2:B7,A2:A7,TRUE,FALSE)

4.7.92 LOGINV

Returns the inverse of the lognormal cumulative distribution function of x, where ln(x) is normally distributed with parameters mean and standard_dev. If p = LOGNORMDIST(x,...), then LOGINV(p,...) = x.

Syntax

LOGINV(probability, mean, standard_dev)

where:

probability is the probability associated with the lognormal distribution.

mean is the mean of ln(x).

standard_dev is the standard deviation of ln(x).

Remarks

- Probability must be >= 0 and < 1.
- Standard_dev must be positive.
- The inverse of the lognormal distribution function is:

$$LOGINV(p,\mu,\sigma) = e^{\{\mu + \sigma x [NORMSINV(p)]\}}$$

4.7.93 LOGNORMDIST

Returns the cumulative lognormal distribution of x, where ln(x) is normally distributed with parameters mean and standard_dev.

Syntax

LOGNORMDIST(x, mean, standard_dev)

where:

x is the value at which, the function can be evaluated.

mean is the mean of ln(x).

standard_dev is the standard deviation of ln(x).

Remarks

- Both x and standard_dev must be positive.

- The equation for the lognormal cumulative distribution function is:

$$LOGNORMDIST(x,\mu,\sigma) = NORMSDIST\left(\frac{ln(x)-\mu}{\sigma}\right)$$

4.7.94 Lower

The **Lower** function converts all characters in the specified text string to lowercase. Characters in the string that are not letters, are not changed.

Syntax:

Lower(text)

where,

- text is the string you want to convert to lowercase.

4.7.95 Match

The **Match** function searches for a specified value in an array and returns the relative position of that item.

Syntax:

Match(value, array, match_type)

where,

- this value is the value you want to search in the array.
- array is a range of cells that contains the value you want to search.
- match_type is the type of match you want to perform.

match_type accepts the following values:

- 1 - The Match function will find the largest value that is less than or equal to the specified value. Ensure that the array is sorted in ascending order.

- 0 - The Match function will find the first value that is equal to the specified value. The array can be sorted in any order.

- - 1 - The Match function will find the smallest value that is greater than or equal to the specified value. Ensure that the array is sorted in descending order.

Note:

- The Match function does not distinguish between uppercase and lowercase when searching.
- If the Match function does not find a match, it returns #N/A error.
- match_type is optional. The Match Function assumes match_type as 1 when the parametter is omitted.
- If the match_type parameter is 0 and a text value, then you can use wildcards in the value parameter.

Where,

* - matches any sequence of characters

? - matches any single character

4.7.96 MAX

Returns the largest value in a set of values.

Syntax

MAX(number1, number2, ...)

where:

number1, number2, ... are numbers for which you want to find the maximum value.

4.7.97 MAXA

Returns the largest value in a list of arguments. Text and logical values such as True and False are compared as well as numbers.

Syntax

MAXA(value1, value2, ...)

where:

value1, value2, ... are values for which you want to find the largest value.

Remarks

- You can specify arguments that are numbers, empty cells, logical values or text representations of numbers. Arguments that are error values cause errors. If the calculation does not include text or logical values, use the MAX worksheet function instead.
- If an argument is an array or reference, only values in that array or reference are used. Empty cells and text values in the array or reference are ignored.
- Arguments that contain True evaluate as 1; arguments that contain text or False evaluate as 0 (zero).
- If the arguments contain no values, MAXA returns 0 (zero).

4.7.98 MEDIAN

Returns the median of the given numbers. The median is the number in the middle of a set of numbers; that is, half the numbers have values that are greater than the median and half have values that are less.

Syntax

MEDIAN(number1, number2, ...)

where:

number1, number2, ... are numbers for which you want the median.

Remarks

- If there is an even number of numbers in the set, then MEDIAN calculates the average of the two numbers in the middle.

4.7.99 MID

MID returns a text segment of a character string. The parameters specify the starting position and the number of characters.

Syntax

MID(text, start_position, num_chars)

where:

text is the text containing the characters to extract.

start is the position of the first character in the text to extract.

number specifies the number of characters in the part of the text.

4.7.100 MIN

Returns the smallest number in a set of values.

Syntax

MIN(number1, number2, ...)

where:

number1, number2, ... are numbers for which you want to find the minimum value.

Remarks

- If an argument is an array or reference, only numbers in that array or reference are used. Empty cells, logical values or text in the array or reference are ignored. If logical values and text should not be ignored, use MINA.

4.7.101 MINA

Returns the smallest value in the list of arguments. Text and logical values such as True and False are compared as well as numbers.

Syntax

MINA(value1, value2, ...)

where:

value1, value2, ... are values for which, you want to find the smallest value.

Remarks

- Arguments that contain True evaluate as 1; arguments that contain text or False evaluate as 0 (zero).

4.7.102 MINUTE

Returns the minutes of a time value. The minute is given as an integer, ranging from 0 to 59.

Syntax

MINUTE(serial_number)

where:

serial_number is the time that contains the minute you want to find. Times may be entered as text strings within quotation marks (for example, "6:00 PM"), as decimal numbers (for example, 0.75, which represents 6:00 PM), or as results of other formulas or functions (for example, TIMEVALUE("6:00 PM")).

Remarks

- Time values are a portion of a date value and are represented by a decimal number (for example, 12:00 PM is represented as 0.5).

4.7.103 MIRR

Returns the modified internal rate of return for a series of periodic cash flows.

Syntax

MIRR(values, finance_rate, reinvest_rate)

where:

values is an array or a reference to cells that contain numbers. These numbers represent a series of payments (negative values) and income (positive values) occurring at regular periods. Values must contain at least one positive value and one negative value to calculate the modified internal rate of return.

finance_rate is the interest rate you pay on the money used in the cash flows.

reinvest_rate is the interest rate you receive on the cash flows as you reinvest them.

Remarks

- MIRR uses the order of values to interpret the order of cash flows. Be sure to enter your payment and income values in the sequence you want and with the correct signs (positive values for cash received, negative values for cash paid).
- If n is the number of cash flows in values, frate is the finance_rate, and rrate is the reinvest_rate, then the formula for MIRR is:

$$\left(\frac{-NPV(rrate, values[positive]) * (1+rrate)^{n}}{NPV(frate, values[negative]) * (1+frate)}\right)^{\frac{1}{n-1}} - 1$$

4.7.104 MOD

Returns the remainder after the number is divided by a divisor. The result has the same sign as the divisor.

Syntax

MOD(number, divisor)

where:

number is the number for which, you want to find the remainder.

divisor is the value by which, you want to divide the number.

Remarks

- The MOD function can be expressed in terms of the INT function:

MOD(n, d) = n – d * INT(n/d)

4.7.105 MODE

Returns the most frequently occurring or repetitive, value in an array or range of data.

Syntax

MODE(number1, number2, ...)

where:

number1, number2, ... are arguments for which you want to calculate the mode.

Remarks

- In a set of values, the mode is the most frequently occurring value.

4.7.106 MONTH

Returns the month of a date represented by a serial number. The month is given as an integer, ranging from 1 (January) to 12 (December).

Syntax

MONTH(serial_number)

where:

serial_number is the date of the month you are trying to find. Dates should be entered by using the DATE function or as results of other formulas or functions. For example, use DATE(2002,11,12) for the 12th day of Nov, 2002.

Remarks

- Dates are stored as sequential serial numbers so that they can be used in calculations. By default, January 1, 1900 is serial number 1 and January 1, 2008 is serial number 39448 because it is 39,448 days after January 1, 1900.

4.7.107 NEGBINOMDIST

Returns the negative binomial distribution. NEGBINOMDIST returns the probability that there will be number_f failures before the number_s-th success, when the constant probability of a success is probability_s.

Syntax

NEGBINOMDIST(number_f, number_s, probability_s)

where:

number_f is the number of failures.

number_s is the threshold number of successes.

probability_s is the probability of a success.

Remarks

- number_s must be >= 1.
- probability_s must be >= 0 and <= 1.
- number_f must be >= 0.
- The equation for the negative binomial distribution is:

$$nb(x;r;p) = \binom{x + r - 1}{r - 1} p^r (1-p)^x$$

where:

x is number_f

r is number_s

p is probability_s

4.7.108 NORMDIST

Returns the normal distribution for the specified mean and standard deviation.

Syntax

NORMDIST(x, mean, standard_dev, cumulative)

where:

x is the value for which, you want the distribution.

mean is the arithmetic mean of the distribution.

standard_dev is the standard deviation of the distribution.

cumulative is a logical value that determines the form of the function. If cumulative is True, NORMDIST returns the cumulative distribution function; if False, it returns the probability mass function.

Remarks

- Standard_dev must be > 0.
- The equation for the normal density function (cumulative = False) is:

$$f(x,\mu,\sigma) = \frac{e^{-\left(\frac{(x-\mu)^2}{2\sigma^2}\right)}}{\sqrt{2\pi}\,\sigma}$$

- When cumulative = True, the formula is the integral from negative infinity to x of the given formula.

4.7.109 NORMINV

Returns the inverse of the normal cumulative distribution for the specified mean and standard deviation.

Syntax

NORMINV(probability, mean, standard_dev)

where:

probability is a probability corresponding to the normal distribution.

mean is the arithmetic mean of the distribution.

standard_dev is the standard deviation of the distribution.

Remarks

- Probability must be >= 0 and <= 1.
- standard_dev must be > 0.

Given a value for probability, NORMINV seeks value x such that NORMDIST(x, mean, standard_dev, True) = probability. NORMINV uses an iterative search technique.

4.7.110 NormsDist

The **NormsDist** function returns the probability that the observed value of a standard normal random variable will be less than or equal to the parameter.

Syntax:

NormsDist(value)

where,

- **value** is a numeric value that checks with the random variable.

4.7.111 NormsInv

The **NormsInv** function returns the standard normal random variable that has *Mean* 0 and *Standard Deviation* 1

Syntax:

NormsDist(value)

where,

- **value** is probability of the standard deviation.

4.7.112 NOT

Reverses the value of its argument.

Syntax

NOT(logical)

where:

logical is a value or expression that can be evaluated to True or False.

4.7.113 NOW

Returns the serial number of the current date and time.

Syntax

NOW( )

Remarks

- Dates are stored as sequential serial numbers so that they can be used in calculations. By default, January 1, 1900 is serial number 1 and January 1, 2008 is serial number 39448 because it is 39,448 days after January 1, 1900
- Numbers to the right of the decimal point in the serial number represent the time; numbers to the left represent the date. For example, the serial number .5 represents the time 12:00 noon.

4.7.114 NPER

Returns the number of periods for an investment based on periodic, constant payments and a constant interest rate.

Syntax

NPER(rate, pmt, pv, fv, type)

where:

rate is the interest rate per period.

pmt is the payment made each period; it cannot change over the life of the annuity.

pv is the present value or the lump-sum amount that a series of future payments is worth right now.

fv is the future value or a cash balance that you want to attain after the last payment is made. If fv is omitted, it is assumed to be 0 (the future value of a loan, for example, is 0).

type is the number 0 or 1 and indicates when payments are due. If type equals:

- 0 - payments are due at the end of the period
- 1 - payments are due at the beginning of the period

4.7.115 NPV

Calculates the net present value of an investment by using a discount rate and a series of future payments (negative values) and income (positive values).

Syntax

NPV(rate, value1, value2, ...)

where:

rate is the rate of discount over the length of one period.

value1, value2, ... are arguments representing the payments and income. Value1, value2, ... must be equally spaced in time and occur at the end of each period. NPV uses the order of value1, value2, ... to interpret the order of cash flows. Be sure to enter your payment and income values in the correct sequence.

Remarks

- The NPV investment begins one period before the date of the value1 cash flow and ends with the last cash flow in the list. The NPV calculation is based on future cash flows. If

your first cash flow occurs at the beginning of the first period, the first value must be added to the NPV result, not included in the value arguments.

- If n is the number of cash flows in the list of values, the formula for NPV is:

$$NPV = \sum_{i=1}^{n} \frac{values_i}{(1+rate)^i}$$

4.7.116 ODD

Returns the number rounded up to the nearest odd integer.

Syntax

ODD(number)

where:

number is the value to be rounded off.

Remarks

- Regardless of the sign of a number, a value is rounded up when adjusted away from zero. If the number is an odd integer, no rounding occurs.

4.7.117 Offset

The **Offset** function returns a reference to a range that is offset a number of rows and columns from any given range or cell.

Syntax:

Offset(range, rows, columns, height, width)

where,

- range is the starting range from which you want to apply the offset.
- rows is the number of rows you want to apply as the offset to the range. This can be either a positive or negative number.

- columns is the number of columns you want to apply as the offset to the range. This can be either a positive or a negative number.
- height is the number of rows that you want the returned range to be.
- width is the number of columns that you want the returned range to be.

4.7.118 OR

Returns True if any argument is True; returns False if all arguments are **False**.

Syntax

OR(logical1, logical2, ...)

where:

logical1, logical2, ... are conditions you want to test that can be either True or False.

Remarks

- The arguments must evaluate to logical values such as True or False or in arrays or references that contain logical values.

4.7.119 PEARSON

Returns the Pearson product moment correlation coefficient, r, a dimensionless index that ranges from -1.0 to 1.0 inclusive and reflects the extent of a linear relationship between two data sets.

Syntax

PEARSON(array1, array2)

where:

array1 is a set of independent values.

array2 is a set of dependent values.

Remarks

- The arguments must be either numbers or names, array constants or references that contain numbers.
- The formula for the Pearson product moment correlation coefficient, r, is:

$$r = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sqrt{\sum(x - \bar{x})^2 \sum(y - \bar{y})^2}}$$

where:

x-bar and **y-bar** are the sample means AVERAGE(array1) and AVERAGE(array2).

4.7.120 PERCENTILE

Returns the k-th percentile of values in a range.

Syntax

PERCENTILE(array, k)

where:

array is the array or range of data that defines relative standing.

k is the percentile value in the range 0..1, inclusive.

Remarks

- k must be >=10 and <= 1.
- If k is not a multiple of 1/(n - 1), PERCENTILE interpolates to determine the value at the k-th percentile.

4.7.121 PERCENTRANK

Returns the rank of a value in a data set as a percentage of the data set.

Syntax

PERCENTRANK(array, x, significance)

where:

array is the range of data with numeric values that defines relative standing.

x is the value for which, you want to know the rank.

significance is an optional value that identifies the number of significant digits for the returned percentage value. If omitted, PERCENTRANK uses three digits (0.xxx).

Remarks

- Significance must be >= 1.
- If x does not match one of the values in the array, PERCENTRANK interpolates to return the correct percentage rank.

4.7.122 Permut

The **Permut** function returns the number of permutations of n items taken at k time.

Syntax:

Permut(n, k)

where,

- **n** is the number of items.
- **k** is the time taken.

4.7.123 PI

Returns the number 3.14159265358979, the mathematical constant pi, accurate to 15 digits.

Syntax

PI( )

4.7.124 PMT

Calculates the payment for a loan based on constant payments and a constant interest rate.

Syntax

PMT(rate, nper, pv, fv, type)

where:

rate is the interest rate for the loan.

nper is the total number of payments for the loan.

pv is the present value or the total amount that a series of future payments is worth now; also known as the principal.

fv is the future value or a cash balance you want to attain after the last payment is made. If fv is omitted, it is assumed to be 0 (zero), that is, the future value of a loan is 0.

type is the number 0 (zero) or 1 and indicates when payments are due. If type equals:

- 0 - payments are due at the end of the period
- 1 - payments are due at the beginning of the period

Remarks

- The payment returned by PMT includes principal and interest but no taxes, reserve payments or fees sometimes associated with loans.
- Make sure that you are consistent about the units you use for specifying rate and nper. If you make monthly payments on a four-year loan at an annual interest rate of 12 percent, use 12%/12 for rate and 4*12 for nper. If you make annual payments on the same loan, use 12 percent for rate and 4 for nper.

4.7.125 POISSON

Returns the Poisson distribution.

Syntax

POISSON(x, mean, cumulative)

where:

x is the number of events.

mean is the expected numeric value.

cumulative is a logical value that determines the form of the probability distribution returned. If cumulative is True, POISSON returns the cumulative Poisson probability that the number of random events occurring will be between zero and x inclusive; if False, it returns the Poisson probability mass function that the number of events occurring will be exactly x.

Remarks

- X must be >= 0.
- Mean must be > 0.
- POISSON is calculated as follows:

For cumulative = False:

$$POISSON = \frac{e^{-\lambda}\lambda^{x}}{x!}$$

For cumulative = True:

$$CUMPOISSON = \sum_{k=0}^{x} \frac{e^{-\lambda}\lambda^{k}}{k!}$$

4.7.126 Pow

The **Pow** function returns the result of a number raised to a power.

Syntax:

POW(number, power)

where,

- **number** is the base number. It can be any real number.
- **power** is the exponent to which, the base number is raised.

4.7.127 POWER

Returns the result of a number raised to a power.

Syntax

POWER(number, power)

where:

number is the base number. It can be any real number.

power is the exponent to which, the base number is raised.

4.7.128 PPMT

Returns the payment on the principal for a given period, for an investment based on periodic, constant payments and a constant interest rate.

Syntax

PPMT(rate, per, nper, pv, fv, type)

where:

rate is the interest rate per period.

per specifies the period and must be in the range of 1 to nper.

nper is the total number of payment periods in an annuity.

pv is the present value— the total amount that a series of future payments is worth now.

fv is the future value or a cash balance that you may want to attain after the last payment is made. If fv is omitted, it is assumed to be 0 (zero), that is, the future value of a loan is 0.

type is the number 0 or 1 and indicates when payments are due. If type equals:

- 0 - Payments are due at the end of the period.
- 1 - Payments are due at the beginning of the period.

Remarks

- Make sure that you are consistent about the units you use for specifying rate and nper. If you make monthly payments on a four-year loan at 12 percent annual interest, use 12%/12 for rate and 4*12 for nper. If you make annual payments on the same loan, use 12% for rate and 4 for nper.

4.7.129 PROB

Returns the probability whose values are in a range that is between two limits. If upper_limit is not supplied, returns the probability that values in x_range are equal to lower_limit.

Syntax

PROB(x_range, prob_range, lower_limit, upper_limit)

where:

x_range is the range of numeric values of x with which, there are associated probabilities.

prob_range is a set of probabilities associated with values in x_range.

lower_limit is the lower bound on the value for which, you want a probability.

upper_limit is the optional upper bound on the value for which, you want a probability.

Remarks

- Any value in prob_range must be > 0 and < 1.
- If upper_limit is omitted, PROB returns the probability of being equal to lower_limit.

4.7.130 PRODUCT

Multiplies all the numbers given as arguments and returns the product.

Syntax

PRODUCT(number1, number2, ...)

where:

number1, number2, ... are numbers that you want to multiply.

4.7.131 PV

Returns the present value of an investment. The present value is the total amount that a series of future payments is worth now.

Syntax

PV(rate, nper, pmt, fv, type)

where:

rate is the interest rate per period. For example, if you obtain an automobile loan at a 10% annual interest rate and make monthly payments, your interest rate per month is 10%/12 or 0.83%. You would enter 10%/12 or 0.83% or 0.0083, into the formula as the rate.

nper is the total number of payment periods in an annuity. For example, if you get a four-year car loan and make monthly payments, your loan has 4*12 (or 48) periods. You would enter 48 into the formula for nper.

pmt is the payment made for each period and cannot change over the life of the annuity. Typically, pmt includes principal and interest but, no other fees or taxes. For example, the monthly payments on a $10,000, four-year car loan at 12 percent are $263.33. You will have to enter -263.33 into the formula as the pmt. If pmt is omitted, you must include the fv argument.

fv is the future value or a cash balance that you want to attain after the last payment is made. If fv is omitted, it is assumed to be 0 (the future value of a loan, for example, is 0). For example, if you want to save $50,000 to pay for a special project in 18 years, then $50,000 is the future value. You could then make a conservative guess at an interest rate and determine how much you must save each month. If fv is omitted, you must include the pmt argument.

type is the number 0 or 1 and indicates when payments are due. If type equals:

- 0 - Payments are due at the end of the period.
- 1 - Payments are due at the beginning of the period.

Remarks

- Make sure that you are consistent about the units you use for specifying rate and nper. If you make monthly payments on a four-year loan at 12 percent annual interest, use 12%/12 for rate and 4*12 for nper. If you make annual payments on the same loan, use 12% for rate and 4 for nper.
- In annuity functions, the cash you pay out such as a deposit to savings is represented by a negative number; the cash you receive such as a dividend check is represented by a positive number.
- One financial argument is solved for in terms of the others. If rate is not 0, then,

$$pv*(1+rate)^{nper} + pmt(1+rate*type) * \left(\frac{(1+rate)^{nper}-1}{rate}\right) + fv = 0$$

If rate is 0:

(pmt * nper) + pv + fv = 0

4.7.132 QUARTILE

Returns the quartile of a data set.

Syntax

QUARTILE(array, quart)

where:

array is the array or cell range of numeric values for which, you want the quartile value.

quart indicates which, value to return.

If quartile equals:	Value returned:
0	Minimum value
1	First quartile (25th percentile)
2	Median value (50th percentile)
3	Third quartile (75th percentile)
4	Maximum value

4.7.133 RADIANS

Converts degrees to radians.

Syntax

RADIANS(angle)

where:

angle is an angle in degrees that you want to convert.

4.7.134 RAND

Returns an evenly distributed random number greater than or equal to 0 and less than 1.

Syntax

RAND( )

4.7.135 RANK

Returns the rank of a number in a list of numbers. The rank of a number is its size relative to other values in a list. (If you were to sort the list, the rank of the number would be its position.)

Syntax

RANK(number, ref, order)

where:

number is the number whose rank you want to find.

ref is an array of or a reference to a list of numbers.

order is a number specifying how to rank numbers.

- If the order is 0 (zero) or omitted, the number is ranked as if ref were a list sorted in descending order.
- If the order is any nonzero value, the number is ranked as if ref were a list sorted in ascending order.

Remark

- RANK gives duplicate numbers of the same rank. However, the presence of duplicate numbers will affect the ranks of subsequent numbers.

4.7.136 RATE

Returns the interest rate per period of an annuity. RATE is calculated by iteration and may not converge to a unique solution.

Syntax

RATE(nper, pmt, pv, fv, type, guess)

where:

nper is the total number of payment periods in an annuity.

pmt is the payment made for each period and cannot change over the life of the annuity. Typically, pmt includes the principal and interest but, no other fees or taxes. If pmt is omitted, you must include the fv argument.

pv is the present value— the total amount that a series of future payments is worth now.

fv is the future value or a cash balance that you want to attain after the last payment is made. If fv is omitted, it is assumed to be 0 (the future value of a loan, for example, is 0).

type is the number 0 or 1 and indicates when payments are due. If type equals:

- 0 - Payments are due at the end of the period.
- 1 - Payments are due at the beginning of the period.

guess is your guess for what the rate will be. If you omit guess, it is assumed to be 10 percent. If RATE does not converge, try different values for guess. RATE usually converges if guess is between 0 and 1.

4.7.137 RIGHT

RIGHT returns the last character or characters in a text string, based on the number of characters you specify.

Syntax

RIGHT(text, num_chars)

where:

text is the text string containing the characters you want to extract.

num_chars specifies the number of characters you want RIGHT to extract.

Remarks

- Num_chars must be greater than or equal to zero.
- If num_chars is greater than the length of text, RIGHT returns all the text.
- If num_chars is omitted, it is assumed to be 1.

4.7.138 ROUND

Rounds a number to a specified number of digits.

Syntax

ROUND(number, num_digits)

where:

number is the number you want to round off.

num_digits specifies the number of digits you want to round off.

Remarks

- If num_digits > 0, then number is rounded off to the specified number of decimal places.
- If num_digits = 0, then number is rounded off to the nearest integer.
- If num_digits < 0, then number is rounded off to the left of the decimal point.

4.7.139 ROUNDDOWN

Rounds a number down towards zero.

Syntax

ROUNDDOWN(number, num_digits)

where:

number is any real number that you want rounded down.

Num_digits is the number of digits to which you want to round a number.

Remark

- ROUNDDOWN behaves like ROUND, except that it always rounds a number down.

4.7.140 ROUNDUP

Rounds a number up away from 0 (zero).

Syntax

ROUNDUP(number, num_digits)

where:

number is any real number that you want rounded up.

num_digits is the number of digits to which you want to round a number.

Remarks

- ROUNDUP behaves like ROUND, except that it always rounds a number up.

4.7.141 RSQ

Returns the square of the Pearson product moment correlation coefficient through data points in known_y's and known_x's.

Syntax

RSQ(known_y's, known_x's)

where:

known_y's is an array or range of data points.

known_x's is an array or range of data points.

Remarks

The equation for the Pearson product moment correlation coefficient, r, is:

$$r = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sqrt{\sum(x - \bar{x})^2 \sum(y - \bar{y})^2}}$$

where:

x-bar and y-bar are the sample means AVERAGE(known_x's) and AVERAGE(known_y's).

RSQ returns r2, which is the square of this correlation coefficient.

4.7.142 SECOND

Returns the seconds of a time value. The second is given as an integer in the range 0 (zero) to 59.

Syntax

SECOND(serial_number)

where:

serial_number is the time that contains the seconds you want to find.

Remarks

- Time values are a portion of a date value and are represented by a decimal number (for example, 12:00 PM is represented as 0.5 because it is half of a day).

4.7.143 SIGN

Determines the sign of a number. Returns 1 if the number is positive, zero (0) if the number is 0 and -1 if the number is negative.

Syntax

SIGN(number)

where:

number is any real number.

4.7.144 SIN

Returns the sine of the given angle.

Syntax

SIN(number)

where:

number is the angle in radians for which you want the sine.

SINH

Returns the hyperbolic sine of a number.

Syntax

SINH(number)

where:

number is any real number.

Remarks

- The formula for the hyperbolic sine is,

$$\sinh(z) = \frac{e^{z} - e^{-z}}{2}$$

4.7.145 SinH

The **SinH** function computes the hyperbolic sine of the argument.

Syntax:

SinH(value)

where,

- **value** is to get the sine of the specific value.

4.7.146 SKEW

Returns the skewness of a distribution. Skewness characterizes the degree of asymmetry of a distribution around its mean.

Syntax

SKEW(number1, number2, ...)

where:

number1, number2 ... are arguments for which you want to calculate the skewness. You can also use a single array or a reference to an array instead of arguments separated by commas.

Remarks

The equation for skewness is defined as:

$$\frac{n}{(n-1)(n-2)} \sum \left(\frac{x_i - \bar{x}}{s} \right)^3$$

4.7.147 SLN

Returns the straight-line depreciation of an asset for one period.

Syntax

SLN(cost, salvage, life)

where:

cost is the initial cost of the asset.

salvage is the value at the end of the depreciation (sometimes called the salvage value of the asset).

life is the number of periods over-which the asset is depreciated (the useful life of the asset).

4.7.148 SLOPE

Returns the slope of the linear regression line through data points in known_y's and known_x's. The slope is the rate of change along the regression line.

Syntax

SLOPE(known_y's, known_x's)

where:

known_y's is an array or cell range of numeric dependent data points.

known_x's is the set of independent data points.

Remarks

- The equation for the slope of the regression line is:

$$b = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sum(x - \bar{x})^2}$$

where:

x-bar and **y-bar** are the sample means AVERAGE(known_x's) and AVERAGE(known_y's).

4.7.149 SMALL

Returns the k-th smallest value in a data set.

Syntax

SMALL(array, k)

where:

array is an array or range of numerical data for which, you want to determine the k-th smallest value.

k is the position (from the smallest) in the array or range of data to return.

4.7.150 SQRT

Returns a positive square root.

Syntax

SQRT(number)

where:

number is the number for which you want the square root.

Remarks

- Number must be >= 0.

4.7.151 STANDARDIZE

Returns a normalized value from a distribution characterized by mean and standard_dev.

Syntax

STANDARDIZE(x, mean, standard_dev)

where:

x is the value that you want to normalize.

mean is the arithmetic mean of the distribution.

standard_dev is the standard deviation of the distribution.

Remarks

- standard_dev must be > 0.
- The equation for the normalized value is:

$$Z = \frac{X - \mu}{\sigma}$$

4.7.152 STDEV

Estimates the standard deviation based on a sample. The standard deviation is a measure of how widely values are dispersed from the average value (the mean).

Syntax

STDEV(number1, number2, ...)

where:

number1, number2, ... are number arguments corresponding to a sample of a population.

Remarks

- STDEV assumes that its arguments are a sample of the population. If your data represents the entire population, then compute the standard deviation using STDEVP.
- STDEV uses the following formula:

$$\sqrt{\frac{\sum(x - \bar{x})^2}{(n-1)}}$$

where:

x-bar is the sample mean AVERAGE(number1,number2,…).

n is the sample size.

4.7.153 STDEVA

Estimates standard deviation based on a sample. The standard deviation is a measure of how widely values are dispersed from the average value (the mean). Text and logical values such as True and False are also included in the calculation.

Syntax

STDEVA(value1, value2 , ...)

where:

value1, value2, ... are values corresponding to a sample of a population. You can also use a single array or a reference to an array instead of arguments separated by commas.

Remarks

- Arguments that contain True evaluate as 1; arguments that contain text or False evaluate as 0 (zero).
- STDEVA uses the following formula:

$$\sqrt{\frac{\sum(x - \bar{x})^2}{(n-1)}}$$

where:

x-bar is the sample mean AVERAGE(value1,value2,…).

n is the sample size.

4.7.154 STDEVP

Calculates standard deviation based on the entire population given as arguments.

Syntax

STDEVP(number1, number2, ...)

where:

number1, number2, ... are 1 to 30 number arguments corresponding to a population. You can also use a single array or a reference to an array instead of arguments separated by commas.

Remarks

- STDEVP assumes that its arguments are the entire population. If your data represents a sample of the population, then compute the standard deviation using STDEV.
- STDEVP uses the following formula:

$$\sqrt{\frac{\sum(x - \bar{x})^2}{n}}$$

where:

x is the sample mean AVERAGE(number1,number2,…).

n is the sample size.

4.7.155 STDEVPA

Calculates the standard deviation based on the entire population given as arguments, including text and logical values.

Syntax

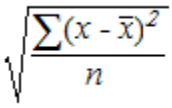
STDEVPA(value1, value2, ...)

where:

value1, value2, ... are values corresponding to a population. You can also use a single array or a reference to an array instead of arguments separated by commas.

Remarks

- Arguments that contain True evaluate as 1; arguments that contain text or False evaluate as 0 (zero).
- STDEVPA uses the following formula:



where:

x-bar is the sample mean AVERAGE(value1,value2,…).

n is the sample size.

4.7.156 STEYX

Returns the standard error of the predicted y-value for each x in the regression.

Syntax

STEYX(known_y's, known_x's)

where:

known_y's is an array or range of dependent data points.

known_x's is an array or range of independent data points.

Remarks

- The equation for the standard error of the predicted y is:

$$\sqrt{\frac{1}{(n-2)}\left[\sum(y-\bar{y})^2-\frac{\left[\sum(x-\bar{x})(y-\bar{y})\right]^2}{\sum(x-\bar{x})^2}\right]}$$

where:

x-bar and **y-bar** are the sample means AVERAGE(known_x's) and AVERAGE(known_y's).

n is the sample size.

4.7.157 SUBSTITUTE

Substitutes new_text for old_text in a text string. Use SUBSTITUTE when you want to replace specific text in a text string; use REPLACE when you want to replace any text that occurs in a specific location in a text string.

Syntax

SUBSTITUTE(text, old_text, new_text, instance_num)

where:

Text is the text or the reference to a cell containing text for which you want to substitute characters.

Old_text is the text you want to replace.

New_text is the text you want to replace old_text with.

Instance_num specifies which occurrence of old_text you want to replace with new_text. If you specify **instance_num**, only that instance of old_text is replaced. Otherwise, every occurrence of old_text in text is changed to new_text.

For example:

The example may be easier to understand if you copy it to a blank worksheet.

	A	
1	Data	
2	Sales Data	
3	Quarter 1, 2008	
4	Quarter 1, 2011	
	Formula	Description (Result)
	=SUBSTITUTE(A2, "Sales", "Cost")	Substitutes Cost for Sales (Cost Data)
	=SUBSTITUTE(A3, "1", "2", 1)	Substitutes first instance of "1" with "2" (Quarter 2, 2008)
	=SUBSTITUTE(A4, "1", "2", 3)	Substitutes third instance of "1" with "2" (Quarter 1, 2012)

4.7.158 Sum

The **Sum** function adds all numbers in a range of cells and returns the result.

Syntax:

Sum(number1, number2, ... number_n)

where,

number1 is the first number, number2 is the second and number_n is the nth number to be added together

4.7.159 SumIf

SumIf function adds the specified range of cells by a given criteria.

Syntax:

SumIf(range, criteria, sum_range)

where,

- range is the range of cells you want to apply the criteria against.

- criteria is used to determine the cells that will be added.
- sum_range are the cells to sum.

4.7.160 SUMPRODUCT

Multiplies corresponding components in the given arrays and returns the sum of those products.

Syntax

SUMPRODUCT(array1, array2, array3, ...)

where:

array1, array2, array3, ... are 2 to 30 arrays whose components you will want to multiply and then add.

Remarks

- The array arguments must have the same dimensions.
- SUMPRODUCT treats array entries that are not numeric as if they were zeros.

4.7.161 SUMSQ

Returns the sum of the squares of the arguments.

Syntax

SUMSQ(number1, number2, ...)

where:

number1, number2, ... are arguments for which you want the sum of the squares. You can also use a single array or a reference to an array instead of arguments separated by commas.

4.7.162 SumXmY2

The **SumXmY2** function calculates the sum of the squares of the differences between the corresponding items in the arrays and returns the sum as results.

Syntax:

SumXmY2(array1, array2)

where,

array1 and array are two ranges or arrays.

4.7.163 SUMX2MY2

Returns the sum of the difference of squares of corresponding values in two arrays.

Syntax

SUMX2MY2(array_x, array_y)

where:

array_x is the first array or range of values.

array_y is the second array or range of values.

Remarks

- If an array or reference argument contains text, logical values or empty cells, those values are ignored; however, cells with the value zero are included.
- The equation for the sum of the difference of squares is:

$$SUMX2MY2 = \sum (x^2 - y^2)$$

4.7.164 SUMX2PY2

Returns the sum of the sum of squares of corresponding values in two arrays. The sum of the sum of squares is a common term in many statistical calculations.

Syntax

SUMX2PY2(array_x, array_y)

where:

array_x is the first array or range of values.

array_y is the second array or range of values.

Remarks

- If an array or reference argument contains text, logical values or empty cells, those values are ignored; however, cells with the value zero are included.
- The equation for the sum of the sum of squares is:

$$SUMX2PY2 = \sum (x^2+y^2)$$

4.7.165 SYD

Returns the sum-of-years' digits depreciation of an asset for a specified period.

Syntax

SYD(cost, salvage, life, per)

where:

cost is the initial cost of the asset.

salvage is the value at the end of the depreciation (sometimes called the salvage value of the asset).

life is the number of periods over which, the asset is depreciated (sometimes called the useful life of the asset).

per is the period and must use the same units as life.

Remarks

- SYD is calculated as follows:

$$\frac{(cost - salvage) * (life - per + 1) * 2}{(life)(life + 1)}$$

4.7.166 TAN

Returns the tangent of a number.

Syntax

TAN(number)

where:

number is the tangent of the angle that you want.

4.7.167 TANH

Returns the hyperbolic tangent of a number.

Syntax

TANH(number)

where:

number is any real number.

Remarks

- The formula for the hyperbolic tangent is:

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$$

4.7.168 TEXT

Converts a value to text in a specific number format.

Syntax

TEXT(value, format_text)

where:

value is a numeric value, a formula that evaluates to a numeric value or a reference to a cell containing a numeric value.

format_text is a number format in text form in the Category box on the Number tab in the Format Cells dialog box.

4.7.169 TIME

Returns the decimal number for a particular time.

The decimal number returned by TIME is a value ranging from 0 (zero) to 0.99999999, representing the times from 0:00:00 (12:00:00 A.M.) to 23:59:59 (11:59:59 P.M.).

Syntax

TIME(hour, minute, second)

where:

hour is a number from 0 (zero) to 23 representing the hour.

minute is a number from 0 to 59 representing the minute.

second is a number from 0 to 59 representing the second.

4.7.170 TIMEVALUE

Returns the decimal number of the time represented by a text string. The decimal number is a value ranging from 0 (zero) to 0.99999999, representing the times from 0:00:00 (12:00:00 A.M.) to 23:59:59 (11:59:59 P.M.).

Syntax

TIMEVALUE(time_text)

where:

time_text is a text string that represents a time as a formatted string; for example, "6:45 PM" and "18:45" text strings within quotation marks that represent time.

Remarks

Date information in time_text is ignored.

4.7.171 TODAY

Returns the serial number of the current date. The serial number is the number of days since Jan 1, 1900.

Syntax

TODAY( )

Remarks

- Dates are stored as sequential serial numbers so that they can be used in calculations. By default, January 1, 1900 is serial number 1 and January 1, 2008 is serial number 39448 because it is 39,448 days after January 1, 1900.

4.7.172 Trim

The Trim function returns a text value with the leading and trailing spaces removed.

Syntax:
Trim(text)

where,

- text is the text value for which you want to remove the leading and the trailing spaces.

4.7.173 TRIMMEAN

Returns the mean of the interior of a data set. TRIMMEAN calculates the mean taken by excluding a percentage of data points from the top and bottom tails of a data set.

Syntax

TRIMMEAN(array, percent)

where:

array is the array or range of values to trim and average.

percent is the fractional number of data points to exclude from the calculation. For example, if percent = 0.2, 4 points are trimmed from a data set of 20 points (20 x 0.2): 2 from the top and 2 from the bottom of the set.

Remarks

- Percent must be >= 0 and <= 1.
- TRIMMEAN rounds off the number of excluded data points down to the nearest multiple of 2. If percent = 0.1, 10 percent of 30 data points equals 3 points. For symmetry, TRIMMEAN excludes a single value from the top and bottom of the data set.

4.7.174 True

The **True** function returns the logical value for True.

Syntax:

True(stringvalue)

where,

- **stringvalue** is to provide an empty string.

4.7.175 TRUNC

Truncates a number to an integer by removing the fractional part of the number.

Syntax

TRUNC(number, num_digits)

where:

number is the number you want to truncate.

num_digits is a number specifying the precision of the truncation. The default value for num_digits is 0 (zero).

Remarks

- TRUNC and INT are similar in that both return integers. TRUNC removes the fractional part of the number. INT rounds numbers down to the nearest integer based on the value of the fractional part of the number. INT and TRUNC are different only when using negative numbers: TRUNC(-4.3) returns -4 but, INT(-4.3) returns -5 because -5 is the lower number.

4.7.176 Upper

The **Upper** function converts all characters in a text string to uppercase.

Syntax:

Upper(text)

where,

- text is the string you want to convert to uppercase.

4.7.177 Value

The **Value** function computes the date or a string that contains the number, and converts it into number format.

Syntax:

Value(range)

where,

- **range** is the string that contains the date or a number.

4.7.178 Var

The Var function returns the variance of a population based on sample of numbers.

Syntax:

Var(number1, number2, ... number_n)

where,

number1, number2, ... number_n are the sample numbers. 30 numbers can be entered.

4.7.179 VarA

The **VarA** function returns the variance of a population based on a sample of numbers, text, and logical values (ie: TRUE or FALSE).

Syntax:

VarA(value1, value2, ... value_n)

where,

value1, value2, ... value_n are the sample values. They can be numbers, text, and logical values. Values that are TRUE are evaluated as 1. Values that are FALSE or text values are evaluated as 0. 30 values can be entered.

4.7.180 VarP

The VarP function returns population variance of the listed values.

Syntax:

VarP(listofvalues)

where,

- listofvalues provides the range or values that contain the population.

4.7.181 VARPA

Calculates variance based on the entire population. In addition to numbers and text, logical values such as True and False are also included in the calculation.

Syntax

VARPA(value1, value2, ...)

value1, value2, ... are arguments corresponding to a population.

Remarks

- VARPA assumes that its arguments are the entire population. If your data represents a sample of the population, you must compute the variance using VARA.
- Arguments that contain True evaluate as 1; arguments that contain text or False evaluate as 0 (zero). If the calculation does not include text or logical values, use the VARP worksheet function instead.
- The equation for VARPA is:

$$\frac{\sum(x - \bar{x})^2}{n}$$

where:

x is the sample mean AVERAGE(value1, value2, …).

n is the sample size.

4.7.182 VDB

Returns the depreciation of an asset for any period you specify, including partial periods, using the double-declining balance method or some other method you specify. VDB stands for variable declining balance.

Syntax

VDB(cost, salvage, life, start_period, end_period, factor, no_switch)

where:

cost is the initial cost of the asset.

salvage is the value at the end of the depreciation (sometimes called the salvage value of the asset).

life is the number of periods over which, the asset is depreciated (sometimes called the useful life of the asset).

start_period is the starting period for which, you want to calculate the depreciation. start_period must use the same units as life.

end_period is the ending period for which, you want to calculate the depreciation. end_period must use the same units as life.

factor is the rate at which, the balance declines. If factor is omitted, it is assumed to be 2 (the double-declining balance method).

no_switch is a logical value specifying whether to switch to straight-line depreciation when depreciation is greater than the declining balance calculation.

- If no_switch is True, straight-line depreciation is not used even when the depreciation is greater than the declining balance calculation.
- If no_switch is False or omitted, straight-line depreciation is used when depreciation is greater than the declining balance calculation.

All arguments except no_switch must be positive numbers.

4.7.183 VLOOKUP

Searches for a value in the left most column of a table and then returns a value in the same row from a column you specify in the table. Use VLOOKUP instead of HLOOKUP when your comparison values are located in a column to the left of the data you want to find.

The V in VLOOKUP stands for "Vertical."

Syntax

VLOOKUP(lookup_value, table_array, col_index_num, range_lookup)

where:

lookup_value is the value to be found in the first column of the array. Lookup_value can be a value, a reference or a text string.

table_array is the table of information in which, data is looked up. Use a reference to a range or a range name.

- If range_lookup is True, the values in the first column of the table_array must be placed in ascending order: ..., -2, -1, 0, 1, 2, ..., A-Z, False, True; otherwise VLOOKUP may not give the correct value. If range_lookup is False, table_array does not need to be sorted.
- The values in the first column of the table_array can be text, numbers or logical values.
- Uppercase and lowercase text are equivalent.

col_index_num is the column number in the table_array from which, the matching value must be returned. A col_index_num of 1 returns the value in the first column of the table_array; a col_index_num of 2 returns the value in the second column of the table_array, and so on.

range_lookup is a logical value that specifies whether you want VLOOKUP to find an exact match or an approximate match. If True or omitted, an approximate match is returned. In other words, if an exact match is not found, the next largest value that is less than the lookup_value is returned.

Remarks

- If VLOOKUP can't find a lookup_value and the range_lookup is True, it uses the largest value that is less than or equal to the lookup_value.

4.7.184 WEEKDAY

Returns the day of the week corresponding to a date. The day is given as an integer, ranging from 1 (Sunday) to 7 (Saturday) by default.

Syntax
WEEKDAY(serial_number,return_type)

where:

serial_number is a sequential number that represents the date of the day you are trying to find. Dates should be entered by using the DATE function or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May 2008.

return_type is a number that determines the type of return value.

If Return_type is:	Number returned:
1 or omitted	Numbers 1 (Sunday) through 7 (Saturday).
2	Numbers 1 (Monday) through 7 (Sunday).
3	Numbers 0 (Monday) through 6 (Sunday).

Remarks

- Dates are stored as sequential serial numbers so that they can be used in calculations. By default, January 1, 1900 is serial number 1 and January 1, 2008 is serial number 39448 because it is 39,448 days after January 1, 1900

4.7.185 Weibull

The **Weibull** function returns the Weibull distribution. This distribution is used in reliability analysis, such as calculating a device's mean time to failure.

Syntax:

WEIBULL(x,alpha,beta,cumulative)

where,

- X is the value at which the function is evaluated.
- Alpha is a parameter to the distribution.
- Beta is a parameter to the distribution.
- Cumulative determines the form of the function.

Remarks

- If x, alpha, or beta is nonnumeric, WEIBULL returns the #VALUE! error value.
- If x < 0, WEIBULL returns the #NUM! error value.
- If alpha ≤ 0 or if beta ≤ 0, WEIBULL returns the #NUM! error value.

4.7.186 Xirr

The **Xirr** function computes the internal rate of return for a schedule of possibly non-periodic cash flows.

Syntax:

Xirr(cashflow, datelist, value)

where,

- **cashflow** is the range of cash flow.
- **datelist** is the list of corresponding date serial number values.
- **value** is an initial guess at the return value.

4.7.187 YEAR

Returns the year corresponding to a date. The year is returned as an integer in the range 1900-9999.

Syntax

YEAR(serial_number)

where:

serial_number is the date of the year you want to find. Dates should be entered by using the DATE function or as results of other formulas or functions. For example, use DATE(2002,11,12) for the 12th day of November 2002.

Remarks

- Dates are stored as sequential serial numbers so that they can be used in calculations. By default, January 1, 1900 is serial number 1 and January 1, 2008 is serial number 39448 because it is 39,448 days after January 1, 1900.

4.7.188 ZTEST

Returns the one-tailed probability-value of a z-test.

Syntax

ZTEST(array, u0, sigma)

where:

array is the array or range of data against which, to test u0

u0 is the value to test.

sigma is the population (known) standard deviation. If omitted, the sample standard deviation is used.

Remarks

- ZTEST is calculated as follows when sigma is not omitted:

$$ZTEST(array, \mu_0) = 1 - NORMSDIST\left(\frac{(\bar{x} - \mu_0)\sqrt{n}}{sigma}\right)$$

or when sigma is omitted:

$$ZTEST(array, \mu_0) = 1 - NORMSDIST\left(\frac{(\bar{x} - \mu_0)\sqrt{n}}{s}\right)$$

where:

x is the sample mean AVERAGE(array); s is the sample standard deviation STDEV(array).

n is the number of observations in the sample COUNT(array).

5 Frequently Asked Questions

This section will help you find answers to common questions regarding Essential Calculate. It includes the below sections:

5.1 CalcQuick

CalcQuick is the simplest way to incorporate calculation support into your code. You can create an instance of it then you can assign variables just by using an indexer on your instance. You can parse and compute formulas based on these variables by calling a single method on your CalcQuick object.

5.1.1 How To Add, Remove, And Modify the Implementation Of Functions In the Function Library In CalcQuickBase?

Refer the Function Library topic to know about this.

5.1.2 How To Calculate a Formula?

To calculate a formula, use the **ParseAndCompute** method from the **CalcQuickBase** class.

```
[C#]

// Declares a CalcQuickBase object.
private CalcQuickBase calculate;

//...

// Creates an instance.
this.calculate = new CalcQuickBase();

//... Set up any variables you need in your calculation.

// Calls the ParseAndCompute method to compute formulas.
string result = this.calculate.ParseAndCompute("4 * 5 + SQRT(3)");
```

```
string result1 = this.calculate.ParseAndCompute("[Rate] * [Amount]");
```

```
[VB]

' Declares a CalcQuickBase object.
private calculate As CalcQuickBase

'...

' Creates an instance.
Me.calculate = New CalcQuickBase()

'... Sets up any variables you need in your calculation.

' Calls the ParseAndCompute method to compute formulas.
Dim result As String = Me.calculate.ParseAndCompute("4 * 5 + SQRT(3)")
Dim result1 As String = Me.calculate.ParseAndCompute("[Rate] * [Amount]")
```

5.1.3 How To Enter Vectors Of Numbers Into CalcQuickBase?

Some formulas, like Intercept, require you to enter the parameters as vectors of numbers. Other formulas, like Sum, accept number vectors as parameter arguments. To use such formulas through a CalcQuickBase object, you must enter the numbers by enclosing them in braces. The following code illustrates this.

```
[C#]

// Sets the number vectors as parameters.
CalcQuickBase["known_Y"] = "{2,3,9,1,8}";
CalcQuickBase["known_X"] = "{6,5,11,7,5}";

// Computes the Intercept returned by these values.
this.textBox1.Text =
CalcQuickBase.ParseAndCompute("Intercept([known_Y],[known_X])");
```

```
[VB]

' Sets the number vectors as parameters.
CalcQuickBase("known_Y") = "{2,3,9,1,8}"
CalcQuickBase("known_X") = "{6,5,11,7,5}"

' Computes the Intercept returned by these values.
Me.textBox1.Text =
```



```
CalcQuickBase.ParseAndCompute("Intercept([known_Y],[known_X])")
```

5.1.4 How To Use Logical Expressions In Other Calculated Expressions?

Logical expressions return a True or False value. If you use a logical expression as part of a calculation, then,

- A True is replaced with 1.
- A False is replaced with 0 as the whole expression is evaluated.

This allows you to easily write and compute formulas that involve logical conditions.

Consider the following expression:

([Cost] < 100) * 1 + ([Cost] >= 100) * ([Cost] < 200) * 3 + ([Cost] >= 200) * ([Cost] < 300) * 5 + ([Cost > 300) * 7

Depending upon the value of cost, this expression returns 1, 3, 5 or 7. This is an example of using a linear combination of logical expressions that times other values.

 ***Note: The logical conditions are mutually exclusive, but, when taken as a whole, cover all possible values of cost. It has the effect of assigning a unique value depending upon the input value.***

5.2 CalcEngine

CalcEngine is the class that encapsulates all the calculation support in Essential Calculate. It has methods that parse and compute expressions and also contains many functions defining the calculations found in the Function Library that ships with Essential Calculate.

5.2.1 How To Force Calculations To Be Processed After They Have Been Suspended?

The following code illustrates how to force calculations to be processed after Engine.CalculatingSuspended has been flipped back to true.

[C#]

```
// Creates some data object that implements ICalcData.
this.data = new ArrayCalcData(a);

// Creates a CalcEngine object using this ICalcData object.
CalcEngine engine = new CalcEngine(this.data);

//Turn off calculations.
engine.CalculatingSuspended = true;

// Makes multiple updates to this.data.
// Turn on calculations.
engine.CalculatingSuspended = false;

// Calls RecalculateRange so any formulas in the data can be computed.
engine.RecalculateRange(RangeInfo.Cells(1, 1, nRows + 1, nCols + 1), data);
```

[VB]

```
' Creates some data object that implements ICalcData.
Me.data = New ArrayCalcData(a)

' Creates a CalcEngine object using this ICalcData object.
Dim engine As New CalcEngine(Me.data)

'...

' Turn off calculations.
engine.CalculatingSuspended = True

' Makes multiple updates to this.data.
' Turn on calculations.
engine.CalculatingSuspended = False

' Calls RecalculateRange so any formulas in the data can be computed.
engine.RecalculateRange(RangeInfo.Cells(1, 1, nRows + 1, nCols + 1), Data)
```

5.2.2 How To Read an XLS File Into Essential Calculate?

To get Essential Calculate to work with an XLS file requires Essential XlsIO. Essential ExcelRW is a library that exposes Excel-Like Automation APIs without any dependence upon Excel. It has the ability to Read / Write XLS files.

For details, see Working with an Excel SpreadSheet.

5.2.3 How To Suspend Calculations While a Series Of Values Are Updated?

You can use the property CalcEngine.CalculatingSuspended to control the calculations that will be performed as values change in your ICalcData object.

[C#]

```
// Creates some data object that implements ICalcData.
this.data = new ArrayCalcData(a);

// Creates a CalcEngine object using this ICalcData object.
CalcEngine engine = new CalcEngine(this.data);
//...
// Turn off calculations.
engine.CalculatingSuspended = true;

// Makes multiple updates to this.data somehow...
// Turn on calculations.
 engine.CalculatingSuspended = false;
```

[VB]

```
' Creates some data object that implements ICalcData.
Me.data = New ArrayCalcData(a)

' Creates a CalcEngine object using this ICalcData object.
Dim engine As New CalcEngine(Me.data)

'...
' Turn off calculations.
```



```
engine.CalculatingSuspended = True

' Makes multiple updates to this.data somehow...
' Turn on calculations.
engine.CalculatingSuspended = False
```

5.3 Common

This section deals with the tasks and solutions that apply to both CalcQuick and CalcEngine.

5.3.1 How to Use a Comma as a Decimal Separator in Formula?

Local settings may require the use of a different decimal separator than the period character that Essential Calculate uses by default. For example, many Local settings use a comma as the decimal separator.

To manage this problem, Essential Calculate exposes two static (Shared in VB.NET) members, which, you can set to specify the character that is recognized as the decimal separator and the character that is recognized as the list separator. The default values of these members are a period and a comma, respectively. You can set these static members at any point in your code before you use any Essential Calculate objects.

```
[C#]

public string Form_Load(object sender, EventArgs e)
{

    // Comma
    CalcEngine.ParseDecimalSeparator = ',';

    // Semicolon
    CalcEngine.ParseArgumentSeparator = ';';

    //.... More code
}
```

```
[VB]
```



```
Public Sub Form_Load(sender as object, e As EventArgs)

     // Comma
     CalcEngine.ParseDecimalSeparator = ","

     // Semicolon
     CalcEngine.ParseArgumentSeparator = ";"

     '..... More code
End Function
```

5.3.2 How to generate error messages or exceptions while performing String-related calculations?

Normally the CalcEngine will not display an invalid error message or exception while performing mathematical operations with string or text. To generate an invalid error message or exception, the **TreatStringAsZero** property must be set to *false*.

For example, if a string is multiplied with a number (for example, ”text” * 10), the calculated result will be zero. But, if the TreatStringAsZero property is set to *false,* the “**#VALUE!**” exception will be generated.

```
[C#]

this.engine.TreatStringsAsZero = false;
```

```
[VB]

Me.engine.TreatStringsAsZero = False
```

Index

