

Getting Started with STATISTICA Programming

2300 East 14th Street
Tulsa, OK 74104

Phone: (918) 749-1119
Fax: (918) 749-2217

E-mail: DeveloperDocumentation@statsoft.com
Web: www.statsoft.com

Table of Contents

Overview	3
Applications for <i>STATISTICA</i> Visual Basic programs.	4
Additional SVB Help.	5
Start by Recording a Macro.....	7
Customizing Output in a Recorded Macro	9
Programming and the Document Objects	14
<i>STATISTICA</i> Spreadsheet.....	14
Importing Excel File.....	14
Importing Text File	17
Data Management Operations	20
<i>STATISTICA</i> Graphics.....	23
Customizing 2D Graphs	24
Customizing 3D Graphs	27
<i>STATISTICA</i> Workbook.....	31
<i>STATISTICA</i> Report	40
<i>STATISTICA</i> Macro	52
Customizing Data Miner and ByGroup Nodes.....	55
Programming <i>STATISTICA</i> with Other Languages	61
Microsoft® Visual C++	61
Microsoft® Visual Basic.NET	64
Microsoft® C#	66

Overview

The industry standard *STATISTICA* Visual Basic (SVB) language (integrated into *STATISTICA*) provides another user interface to the functionality of *STATISTICA*, and it offers incomparably more than just a “supplementary application programming language” that can be used to write custom extensions.

SVB takes full advantage of the object model architecture of *STATISTICA* and is used to access programmatically every aspect and virtually every detail of the functionality of *STATISTICA*. Even the most complex analyses and graphs can be recorded into Visual Basic macros and later be run repeatedly or edited and used as building blocks of other applications. SVB adds an arsenal of more than 14,000 new functions to the standard comprehensive syntax of Visual Basic, thus comprising one of the largest and richest development environments available.

Along with analyses and graphs, *STATISTICA*'s object model also exposes all of its document types. That is, you can easily customize a graph, manipulate data in a spreadsheet, format a report, and organize a workbook via automation.

In the following chapters, we will discuss all the important aspects of programming *STATISTICA*, its document types, and its analyses. We will also cover some aspects of generating output and how to customize it

Applications for *STATISTICA* Visual Basic programs.

STATISTICA Visual Basic programs can be used for a wide variety of applications, from simple macros recorded to automate a specific (repeatedly used) sequence of tasks, to elaborate custom analytic systems combining the power of optimized procedures of *STATISTICA* with custom developed extensions featuring their own user interface. When properly licensed, scripts for analyses developed this way can be integrated into larger computing environments or executed from within proprietary corporate software systems or Internet or intranet portals.

SVB programs can also be attached to virtually all important “events” in a *STATISTICA* analysis such as opening or closing files, clicking on cells in spreadsheets, etc.; in this manner, the basic user interface of *STATISTICA* can be highly customized for specific applications (e.g., for data entry operations, etc.).

Several scripting languages are included in *STATISTICA*. You can select from:

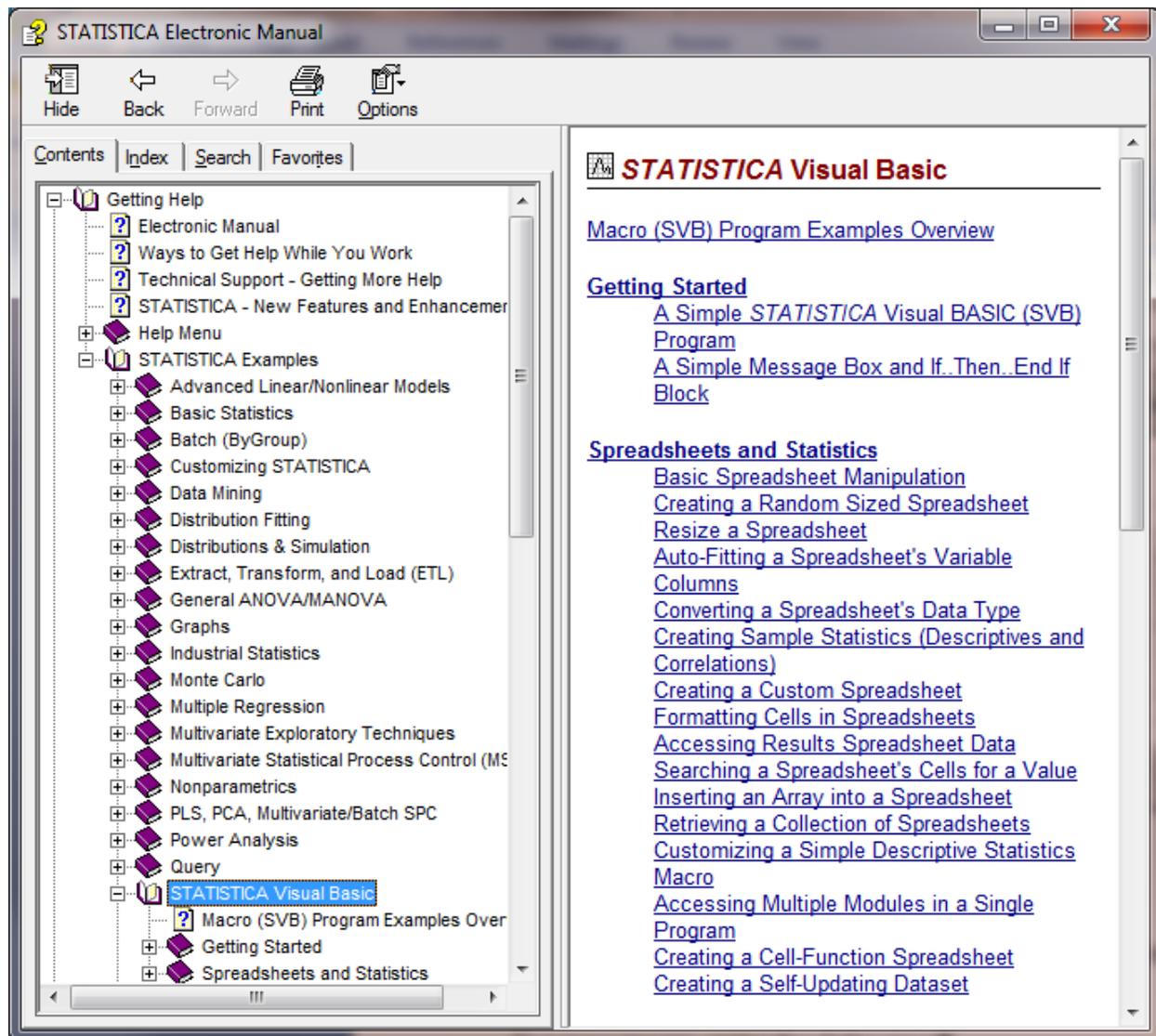
- SVB
- Enhanced SVB
- *STATISTICA* Visual Basic.NET
- R (http://www.statsoft.com/Portals/0/Support/Download/White-Papers/R_Integration_Overview.pdf)

Enhanced *STATISTICA* Visual Basic is a superset of *STATISTICA* Visual Basic, and includes additional features. *STATISTICA* Visual Basic.NET features direct, native access to .NET Assemblies, i.e., not through COM Interop as would be required from standard SVB. Using the R language requires that you have R installed on either the same computer running *STATISTICA* or a computer accessible from the *STATISTICA* Enterprise Server.

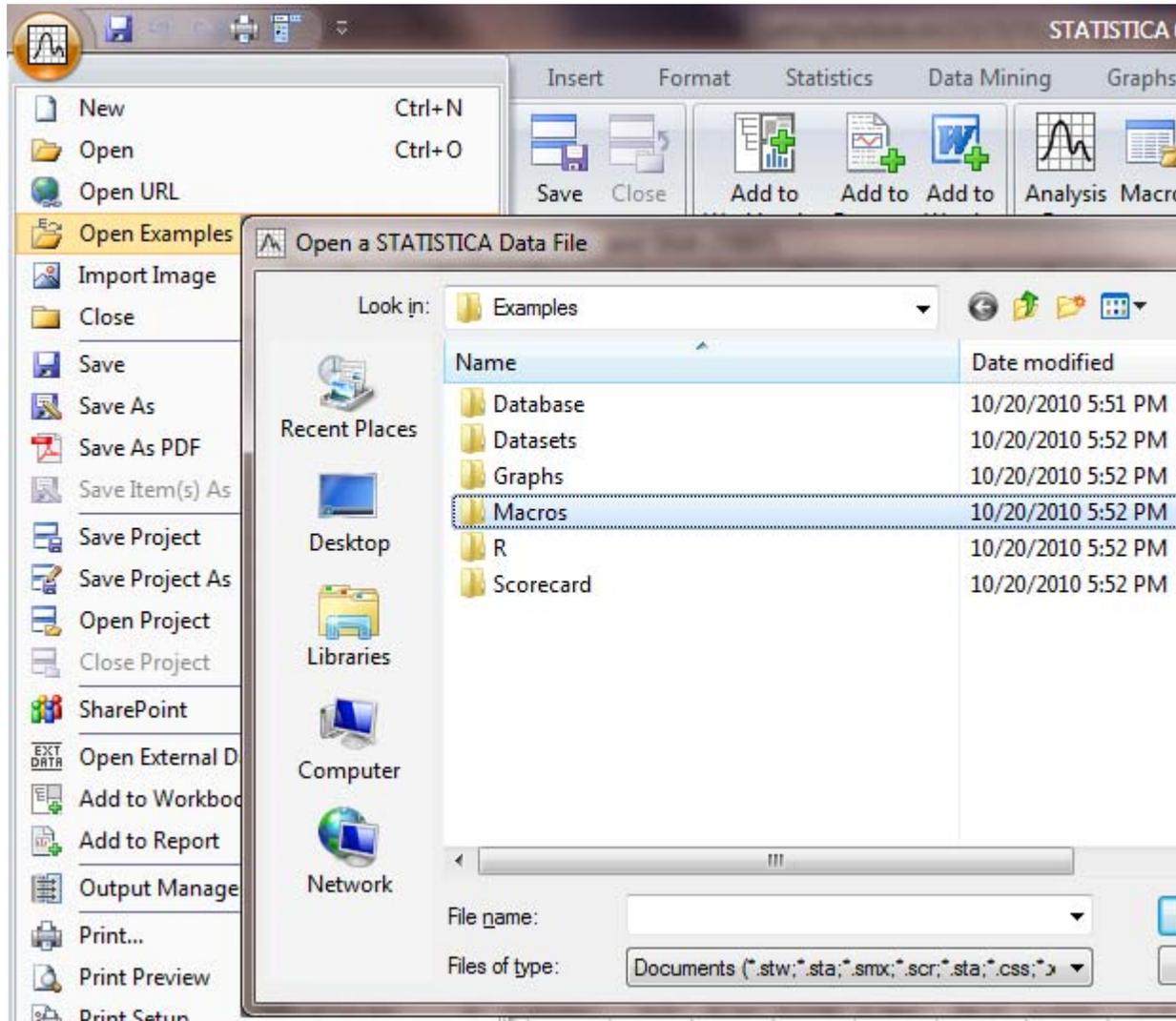
Additional SVB Help.

Descriptions and parameters of the *STATISTICA* Objects can be found in *stv60m.chm*. This help file is installed with *STATISTICA*.

See the *STATISTICA Electronic Manual* for more information and examples. This manual is found by starting *STATISTICA* and typing F1.

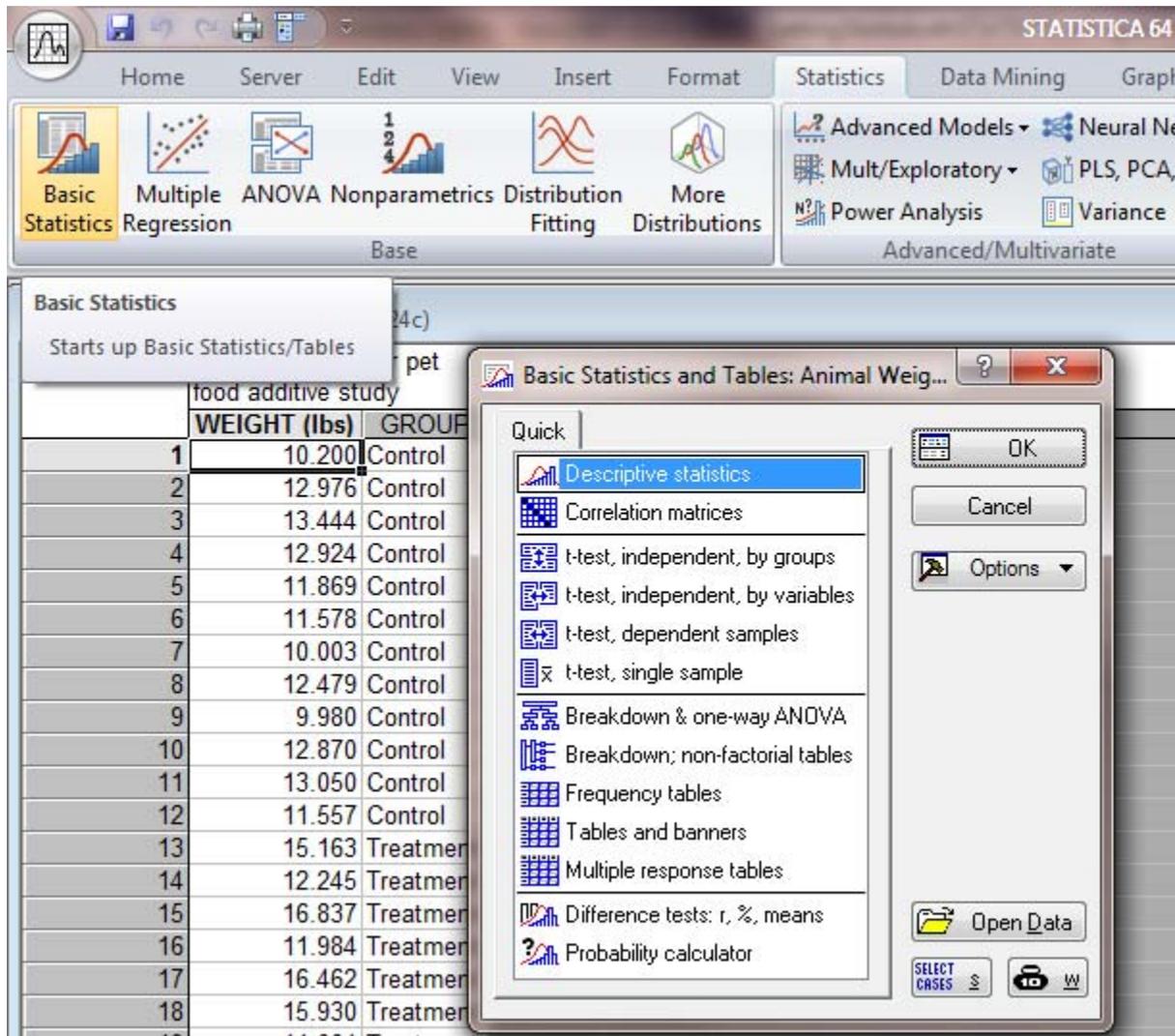


Example macros are installed with *STATISTICA*. Select the *File* menu, *Open Examples* menu and explore the *Macros* folder.

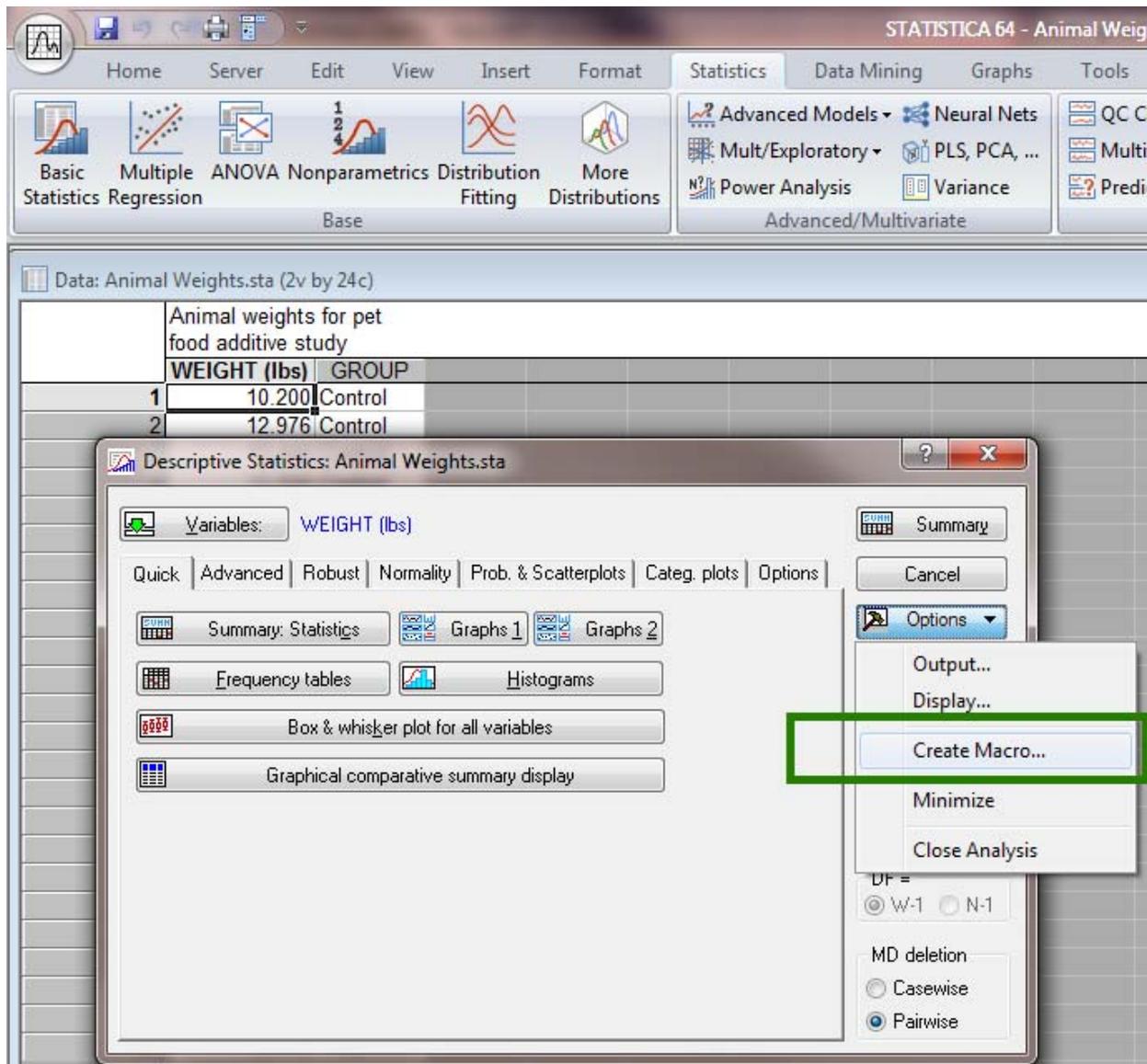


Start by Recording a Macro

In *STATISTICA*, all analyses and their subsequent settings are recorded when you step through them interactively. For example, when you run *Basic Statistics*, *Descriptive Statistics*, all the options and output are recorded while you navigate through the analysis.



Once you are finished with your analysis, click the *Create Macro* from the *Options* button on the analysis dialog. From here, an SVB macro will be generated that contains all of the analysis's settings and output.



You can run this macro on any applicable dataset at anytime without having to go through the analysis again. In addition, you can customize these recorded macros. For example, you can change the analysis properties to include different tests or edit the output.

In the following sections, we shall discuss how to customize the output (i.e., spreadsheets and graphs) within a recorded macro.

Customizing Output in a Recorded Macro

A recorded analysis macro will route all of its output to whatever your current output mode is by calling the `RouteOutput` function. For example, if your output mode is set to workbook, then a macro's output will go into a new workbook. To override this behavior, look for `RouteOutput` in your macro and change it to directly access the generated documents. Consider the following code generated from a *Descriptive Statistics* analysis:

```
Option Base 1
Sub Main

Dim newanalysis As Analysis
Set newanalysis = Analysis (scBasicStatistics,
ActiveDataSet)

With newanalysis.Dialog
    .Statistics = scBasDescriptives
End With

newanalysis.Run

With newanalysis.Dialog
    .Variables = "3 6 9"
    .PairwiseDeletionOfMD = True
    .DisplayLongVariableNames = False
    .ExtendedPrecisionCalculations = False
    .PlotMedianQuartileRange = False
    .PlotMeanSEAndSD = False
    .PlotMeanSD196TimesSD = True
    .PlotMeanSE196TimesSE = False
    .UserDefinedPercentiles = False
    .ValidN = True
    .Mean = True
    .Median = False
    .Mode = False
    .GeometricMean = False
    .HarmonicMean = False
    .ConfLimitsForMeans = False
    .Sum = False
    .StandardDeviation = True
```

```

.Variance = False
.StandardErrorOfMean = False
.MinimumMaximum = True
.LowerUpperQuartiles = True
.Range = False
.QuartileRange = False
.Skewness = False
.Kurtosis = False
.StandardErrorOfSkewness = False
.StandardErrorOfKurtosis = False
.UseNumberOfIntervals = True
.NumberOfIntervals = 10
.NormalExpectedFrequencies = False
.KSAndLillieforsTestForNormality = True
.ShapiroWilkWTest = False
.ConfidenceIntervalForMeansPlot = 95
.CompressedStemAndLeaf = False
End With

newanalysis.RouteOutput(newanalysis.Dialog.Summary).Visible
= True

newanalysis.RouteOutput(newanalysis.Dialog.Histograms).Visi
ble = True

End Sub

```

This will generate a results table and three graphs that will be placed inside of a workbook (assuming your output mode is set to *Workbook*). However, say that we want to customize the output from this macro. To do this, we must remove the calls to `RouteOutput` and instead directly gather the documents from that analysis. Follow these steps to accomplish this:

Step 1:

Find the calls to `RouteOutput` in your macro:

```

newanalysis.RouteOutput(newanalysis.Dialog.Summary).Visible
= True
newanalysis.RouteOutput(newanalysis.Dialog.Histograms).Visi
ble = True

```

Step 2:

Instead of routing our output to the default location, we will directly retrieve them from the analysis. To have a place to put this output, we need to create some document objects. In this case, we know that we will have one spreadsheet and three graphs, so enter this above the calls to `RouteOutput`:

```
Dim SummarySpr As Spreadsheet
Dim Histograms(3) As Graph

newanalysis.RouteOutput(newanalysis.Dialog.Summary).Visible
= True
newanalysis.RouteOutput(newanalysis.Dialog.Histograms).Visi
ble = True
```

Step 3:

Remove the calls to `RouteOutput` so that our code looks like this:

```
Dim SummarySpr As Spreadsheet
Dim Histograms(3) As Graph
```

Step 4:

Now we will set our document objects to the documents in the analysis. The analysis contains its results as a collection. To retrieve a particular result, we set a document object to that item in the collection. For example, to retrieve the summary table we will set our spreadsheet document to the first item in the `Summary` collection by adding these lines:

```
Set SummarySpr = newanalysis.Dialog.Summary(1)
SummarySpr.Visible = True
```

Likewise, to retrieve the three histograms we will set our graph objects to the graphs in the analysis's `Histograms` collection by adding the following lines:

```
Set Histograms(1) = newanalysis.Dialog.Histograms(1)
Histograms(1).Visible = True
Set Histograms(2) = newanalysis.Dialog.Histograms(2)
Histograms(2).Visible = True
Set Histograms(3) = newanalysis.Dialog.Histograms(3)
Histograms(3).Visible = True
```

From here, we can customize the spreadsheet and graphs. For example, add a footnote

to the first graph by adding the following line:

```
Histograms(1).Titles.Add(scgFootnote, "Acme, inc.")
```

Our final macro will look like this:

```
Option Base 1
Sub Main

Dim newanalysis As Analysis
Set newanalysis = Analysis (scBasicStatistics,
ActiveDataSet)

With newanalysis.Dialog
    .Statistics = scBasDescriptives
End With

newanalysis.Run

With newanalysis.Dialog
    .Variables = "3 6 9"
    .PairwiseDeletionOfMD = True
    .DisplayLongVariableNames = False
    .ExtendedPrecisionCalculations = False
    .PlotMedianQuartileRange = False
    .PlotMeanSEAndSD = False
    .PlotMeanSD196TimesSD = True
    .PlotMeanSE196TimesSE = False
    .UserDefinedPercentiles = False
    .ValidN = True
    .Mean = True
    .Median = False
    .Mode = False
    .GeometricMean = False
    .HarmonicMean = False
    .ConfLimitsForMeans = False
    .Sum = False
    .StandardDeviation = True
    .Variance = False
    .StandardErrorOfMean = False
    .MinimumMaximum = True
    .LowerUpperQuartiles = True
    .Range = False
```

```
.QuartileRange = False
.Skewness = False
.Kurtosis = False
.StandardErrorOfSkewness = False
.StandardErrorOfKurtosis = False
.UseNumberOfIntervals = True
.NumberOfIntervals = 10
.NormalExpectedFrequencies = False
.KSAndLillieforsTestForNormality = True
.SapiroWilkWTest = False
.ConfidenceIntervalForMeansPlot = 95
.CompressedStemAndLeaf = False
End With
```

```
'NEW CODE THAT WE ADDED IN THIS EXAMPLE
```

```
Dim SummarySpr As Spreadsheet
Dim Histograms(3) As Graph
```

```
Set SummarySpr = newanalysis.Dialog.Summary(1)
SummarySpr.Visible = True
Set Histograms(1) = newanalysis.Dialog.Histograms(1)
Histograms(1).Visible = True
Set Histograms(2) = newanalysis.Dialog.Histograms(2)
Histograms(2).Visible = True
Set Histograms(3) = newanalysis.Dialog.Histograms(3)
Histograms(3).Visible = True
Histograms(1).Titles.Add(scgFootnote, "Acme, inc.")
```

```
End Sub
```

NOTE

When accessing specific items of an analysis's output collection, it is more efficient to set a `staDocuments` object to the collection and reference that, rather than calling the analysis collection directly. For example:

```
newanalysis.Dialog.Summary(1).SaveAs("C:\Data\GenderSummary.sta")
newanalysis.Dialog.Summary(2).SaveAs("C:\Data\AdvertSummary.sta")
newanalysis.Dialog.Summary(3).Visible = True
```

This is more efficient when written as:

```
Dim s As StaDocuments
Set s=newanalysis.Dialog.Summary
s(1).SaveAs("C:\Data\GenderSummary.sta")
s(2).SaveAs("C:\Data\AdvertSummary.sta")
s(3).Visible = True
```

Programming and the Document Objects

As mentioned earlier, all *STATISTICA* documents are available through automation. In addition, *STATISTICA*'s object model is designed to mimic the application's interface.

***STATISTICA* Spreadsheet**

STATISTICA's spreadsheet is a flexible, tabular document that offers a plethora of data management functionality. For example, data can be sorted, merged, and filtered through the interface and automation. Let us discuss some of these common operations, along with how to automate the importation of data from non-*STATISTICA* files.

Usually the first step to using *STATISTICA* is to import data into the system, so we will begin there. Note that most file importing operations can be recorded by creating a master macro, but we will review some of these operations here in more detail.

The most common file types that are imported into *STATISTICA* are text and Microsoft® Excel documents. The first example will be to import a Microsoft® Excel file:

Importing Excel File

Step 1:

Locate a Microsoft® Excel file. For this example, let us assume that this file is located at "C:\My Documents\Cat Clinic.xls". This file may look something like this:

	A	B	C	D	E	F	G
1		Patient ID	Age (1998)	Weight (1998)	Body Length (1998)	Age (1999)	Weight
2	Tori	601	1	5.8	6.8	2	
3	Cheshire	602	2	8.6	15.2	3	
4	Fluffy	603	3	6.1	13.9	4	
5	Cotton	604	1	7.2	8.2	2	
6	Ms. Isabelle	605	3	12.9	15.7	4	
7	Salem	606	3	9.1	15.8	4	
8	Muffin	607					1
9	Hobbes	608	6	8.1	16.3	7	

Step 2:

In *STATISTICA*, select *File - New* to display the *Create New Document* dialog, select the *Macro* tab, and create a macro.

In a new macro, enter the following:

```
Option Base 1
Option Explicit
Sub Main
    Application.ImportXLSAsSpreadsheet( _
        "C:\My Documents\Cat Clinic.xls", _
        1, True, True, True, 1, -1, 1, _
        -1).Visible = True
End Sub
```

Let us examine the arguments to `ImportXLSAsSpreadsheet`.

The first argument that we pass is the file path to the Excel file, which is "C:\My Documents\Cat Clinic.xls".

The second argument is the sheet number that we want to import. In this case, we will select the first sheet.

The third and fourth parameters determine whether to import the first column as the case names and the first row as the variable names. In this case, the Excel file contains the names of our patients (case names). It also contains descriptive column headers (variable names). To import these case and variable names, we will want to pass `True` to the third and fourth parameters.

The fifth parameter determines whether to include formatting. Because we have marked cells in our Excel file—along with the case names highlighted to reflect the patients' genders—we will pass `True` as our argument for this parameter.

The sixth and seventh parameters specify which row to begin importing from and how many rows to import, respectively. In this case, we will pass 1 to the sixth parameter to import from the first row, and pass -1 to the seventh parameter to import all the rows. Note that if we only wanted the first 10 rows, then we would pass 10 as our argument for the seventh parameter.

The eighth and ninth parameters specify which columns to begin importing from and how many columns to import, respectively. In this case, we will pass 1 to the eighth parameter to import from the first column, and pass -1 to the ninth parameter to import all the columns. Note that if we only wanted the first 10 columns, then we would pass 10 as our argument for the ninth parameter.

Step 3:

Finally, we add `.Visible = True` to the end of our call to `Application.ImportXLSAsSpreadsheet`—this will make the imported spreadsheet visible.

Step 4:

Run the macro and our result spreadsheet will look like this:

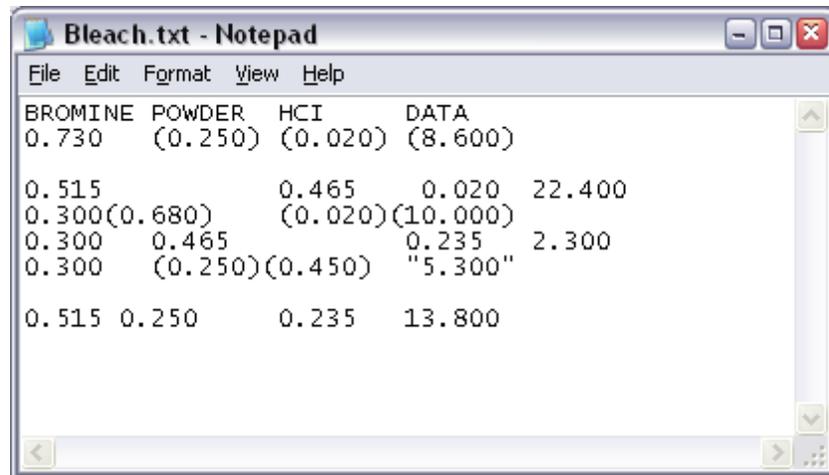
	1	2	3	4	5	6
	Patient ID	Age (1998)	Weight (1998)	Body Length (1998)	Age (1999)	Weight (1999)
Tori	601	1	5.8	6.8	2	8.5
Cheshire	602	2	8.6	15.2	3	9.57
Fluffy	603	3	6.1	13.9	4	6.2
Cotton	604	1	7.2	8.2	2	8.7
Ms. Isabelle	605	3	12.9	15.7	4	13.2
Salem	606	3	9.1	15.8	4	9.7
Muffin	607				1	10.1
Hobbles	608	6	8.1	16.3	7	8.3

Importing Text File

For another example, let us look at some methods for importing a text file.

Step 1:

Locate a text file that contains multiple separators, quoted values, and column names. For this example, create a file that looks something like the following and save it as "C:\My Documents\Bleach.txt":



Step 2:

Create a new macro and enter the following:

```
Option Base 1
Option Explicit
Sub Main
    Application.ImportTextAutoEx2( _
        "C:\My Documents\Bleach.txt", _
        Array(Asc(vbTab), Asc(" "), Asc("("), Asc(")")), _
        1, scTextImportQualifierDoubleQuote, _
        True, False, True, True, _
        True, False, False, False, 0.1000).Visible = True
End Sub
```

Let us examine the arguments to ImportTextAutoEx2.

The first argument that we pass is the file path to the text file, which is "C:\My Documents\Bleach.txt".

The second argument will be an array of integers representing the ASCII values of the column separators. Looking at our text file, it appears that we will need to consider tabs, spaces, and parentheses as column separators. Next, we will pass these values in as an array (by using the Array function). Note that we must convert all of these characters into the ASCII equivalent, so we will pass each of these values to the Asc function. Our argument for the second parameter should now look like this:

```
Array(Asc(vbTab), Asc(" "), Asc("("), Asc(")"))
```

The third parameter represents which row to begin importing from. In this case, we want to import all the rows, so we will pass 1 as our argument.

The fourth parameter represents how to treat quoted values. In this case, we do have one value that is inside of double quotes, so we will pass `scTextImportQualifierDoubleQuote` as our argument.

The fifth argument will specify whether to treat multiple, consecutive separators as one separator. In this case, we do have some values that are separated by more than one separator, so we will pass `True` as our argument.

The sixth and seventh arguments will specify whether to import the first column as the case names and the first row as the variable names, respectively. In this case, we do not want the first column treated as case names, so we will pass `False` as our argument. Next, we do want to import the first row as the variable names, so we will pass `True` as our argument for the seventh parameter.

The eighth argument specifies whether to trim any extra spaces off of the values. We will pass `True` for our argument here.

The ninth argument specifies whether to ignore blank lines. In this example, our text file does have empty lines in it, so we will specify `True`; otherwise, we will have empty cases in our dataset.

The tenth argument specifies whether we should not create text labels. Although our text file does not have labels in it, it is recommended to set this parameter to `False`. If we were to pass in `True`, then any text labels in the file will be imported as missing data.

The eleventh argument specifies whether we should treat a period as missing data. This is useful for importing files saved from SAS. We will set this option to `False`.

The twelfth argument specifies whether we should import textual columns as text variables (rather than double variables with text labels). `False` is recommended for this parameter, unless the columns contain lengthy strings.

The thirteenth argument specifies the threshold value for determining if a column is textual. Because we passed `False` to the last parameter, this parameter will be ignored and we will just pass 0.1000.

Step 3:

Finally, we add `.Visible = True` to the end of our call to `ImportTextAutoEx2`—this will make the imported spreadsheet visible.

Step 4:

Run the macro and our result spreadsheet will look like this:

	1 BROMINE	2 POWDER	3 HCI	4 DATA
1	0.73	0.25	0.02	8.6
2	0.515	0.465	0.02	22.4
3	0.3	0.68	0.02	10
4	0.3	0.465	0.235	2.3
5	0.3	0.25	0.45	5.3
6	0.515	0.25	0.235	13.8

Data Management Operations

STATISTICA spreadsheets offer numerous data operations to organize, clean, and merge your data. Note that most data operations can also be recorded from a master macro (refer to the analysis programming section for further reading). We will look at some of these operations in detail here to see how they work in automation.

Let us look at how to open a spreadsheet, sort it, and then create a subset:

Step 1:

In *STATISTICA*, select *File - New* to display the *Create New Document* dialog, select the *Macro* tab, and create a macro.

Enter the following:

```
Option Base 1
Option Explicit
Sub Main

    Dim Spr As Spreadsheet
    Set Spr = Spreadsheets.Open _
        (Application.Path & _
         "\Examples\Datasets\Cat Clinic Patient Info.sta")
    Spr.Visible = True

End Sub
```

This will open the example dataset *Cat Clinic Patient Info*. Many example datasets are installed with *STATISTICA*.

Step 2:

Next, add the following line to sort the spreadsheet by its case names and then by the first variable:

```
Spr.SortDataEx("0 1", _
    Array(scSortAscending,scSortAscending), _
    Array(scSortByText,scSortByNumeric),False,False)
```

Let us look at the parameters of `SortDataEx` to understand how this function works.

The first parameter indicates which variables to sort and in which order to sort them. In this example, we are sorting by variable 0 (the case names) first, and then by variable 1.

The second parameter indicates how to sort the variables. This parameter accepts an array of sort order keys, so we list all the keys inside of a call to the `Array` function.

The third parameter indicates the collation method to sort the variables. This parameter accepts an array of collation keys, so we list all the keys inside of a call to the `Array` function.

The fourth parameter indicates whether to create a new spreadsheet from the sort operation, rather than sort the original spreadsheet in place. We will set this to `False` so that the original spreadsheet will be sorted.

The fifth parameter indicates whether to include the formatting from the original file. This only applies if you are creating a new spreadsheet, so we will set this to `False`.

Now, create a subset of this dataset where we only have the female cats included.

Step 3:

Add the following:

```
Dim S1 As Spreadsheet
Set S1 = Spr.Subset("1-4", "GENDER='FEMALE' ", "", _
    "BREED='Jack Russell Terrier' ", "")
S1.Visible = True
```

Let us now look at the parameters of the `Subset` function:

The first parameter indicates which variables to include from the original spreadsheet into the subset. In this example, we will include all four variables.

The second parameter indicates the inclusion selection condition. In this case, we will include all cases where the variable *GENDER* is *FEMALE*.

The third parameter indicates the inclusion list (i.e., a list of the cases to include). Because we already included an inclusion statement, we need to leave this parameter blank.

The fourth parameter indicates your exclusion selection condition. In this case, we note that there is a canine (a Jack Russell Terrier) in our dataset and want to remove it from the subset. To accomplish this, we specify to exclude all cases where *BREED* is *Jack Russell Terrier*.

The fifth parameter indicates the exclusion list (i.e., a list of the cases to include). Because we do not want to ignore any specific cases, we just leave this parameter blank.

Here is what our final macro will look like:

```
Option Base 1
Option Explicit
Sub Main

    Dim Spr As Spreadsheet
    Set Spr = Spreadsheets.Open _
        (Application.Path & _
        "\Examples\Datasets\Cat Clinic Patient Info.sta")
    Spr.Visible = True
    Spr.SortDataEx("0 1", _
        Array(scSortAscending,scSortAscending), _
        Array(scSortByText,scSortByNumeric),False,False)
    Dim S1 As Spreadsheet
    Set S1 = Spr.Subset("1-4","GENDER='FEMALE'", "", _
        "BREED='Jack Russell Terrier'", "")
    S1.Visible = True

End Sub
```

STATISTICA Graphics

Now that we have learned how to record an analysis macro and retrieve its output documents, we will now move on to customizing that output. The most common type of output to customize is the graph, which can be edited in a number of ways:

- Custom titles can be added.
- Axes can have custom labels added.
- Point markers can be added and configured in numerous ways.
- Fit lines can be added.

These are just some of the options available to us for customizing graphs. Let us first get started with customizing 2D graphs, and afterwards we shall move onto 3D graphs.

The object model of the graph document type is a hierarchical design where the primary interface is the layout object. This object is returned from the `Content` function of the `Graph` class, and contains interfaces to customize the graph in many ways.

The layout object contains an `Axes` property that returns the collection of axes from the graph. This collection enables you to edit any given axis, such as adding labels or changing its scaling.

Next, the layout object also contains a `Plots` property that returns the plots within the graph. This collection enables you to customize all the properties of any given plot. For example, you can add and edit fit lines (through the `Fits` collection of the plot), change the display of point labels, alter the color of a histogram's bars, etc.

How to create and use layout objects for 2D and 3D graphs shall be discussed in further detail in the following sections.

NOTE

Always create a layout object and set it to a graph's content when customizing a graph. This will provide more explicit data typing and hence better IntelliSense support. For example:

```
Dim graphLayout as Layout2D
Set graphLayout = ActiveGraph.Content
graphLayout.Attribute.Background.Color = RGB(1000, 120, 20)
```

Customizing 2D Graphs

To customize graphs programmatically, the important aspect to understand is that you must use the "layout" of the graph to access most of its functionality. To accomplish this, you must create a `Layout2D` object, set it to the content of the graph, and then use that object to customize the graph. For example, to customize a 2D graph (e.g., a histogram) we would do the following:

Step 1:

Set a 2D graph as the active graph and enter the following into a new macro:

```
Option Base 1
Option Explicit
Sub Main
    Dim graphLayout As Layout2D
    Set graphLayout = ActiveGraph.Content
End Sub
```

Step 2:

Use the `graphLayout` variable—which is pointing to the 2D graph's content—to access the first plot. Most graphs will usually have just one plot, so for brevity we will make this assumption. When you access the plot, you will be able to find most of the features that you would normally find on the *Graph Options* dialog interactively.

For example, let us say that we need to turn on point labels that display the Y axis value. To accomplish this, simply add the following lines:

```
graphLayout.Plots(1).DisplayPointLabels = True
graphLayout.Plots(1).DisplayPointLabelsX = True
```

Next, we want to set the format of these point labels to US Dollars, so we will add this line:

```
graphLayout.Plots(1).LabelsFormat = "$#,##0;($#,##0)"
```

As another example, say that we want to remove the legend. Again, we will find this in the plot of the graph's layout. Add the following code to achieve this:

```
graphLayout.Plots(1).LegendType = scgPlotLegendOff
```

Along with the plot of a graph, the axes are also an important aspect of the graph that is commonly customized. To access this area, use the `AXES` property of the `graphLayout` object.

For example, to change the scaling of the Y axis, we need to set the scaling to manual and then set our new range. To do this, enter the following:

```
graphLayout.Axes(scgLeftY).StepMode = scgManualStep
graphLayout.Axes(scgLeftY).SetManualRange(1,10)
```

Note that unlike plots (that were simply indexed by a numeric value), axes must be indexed more explicitly. That is to say, we must pass in an enumeration value to specify which axis we want to access. The following is a table of these constants:

Enumeration Value	Axis
scgHistogramLeftY	Histogram's left Y axis
scgHistogramRightY	Histogram's right Y axis
scgLeftY	Left Y axis
scgrightY	Right Y axis
scgTop	Top axis (parallel to the X axis)
scgV	V axis (3D graphs only)
scgX	X axis
scgY	Y axis
scgZ	Z axis (3D graphs only)

Some more examples of using the `Axes` property could be turning on minor tick marks on the X axis:

```
graphLayout.Axes(scgX).DisplayMinorTickMark = True
```

...or changing the X axis's font:

```
graphLayout.Axes(scgX).Font.Face.FaceName = "MS UI Gothic"
```

Our final code for the 2D graph customization example will look like this:

```

Option Base 1
Option Explicit
Sub Main
    Dim graphLayout As Layout2D
    Set graphLayout = ActiveGraph.Content
    graphLayout.Plots(1).DisplayPointLabels = True
    graphLayout.Axes(scgLeftY).StepMode = scgManualStep
    graphLayout.Axes(scgLeftY).SetManualRange(1,10)
    graphLayout.Axes(scgX).DisplayMinorTickMark = True
    graphLayout.Axes(scgX).Font.Face.FaceName = _
        "MS UI Gothic"
End Sub

```

Customizing 3D Graphs

Customizing 3D graphs is very similar to customizing 2D graphs. The only difference is that rather than using a `Layout2D` object, you will need to use a different layout object meant for the specific graph. For example, a 3D histogram will require a `Layout3DHistograms` object, while a Ternary plot will require a `LayoutTernary` object. The following table details these relations:

Graph Type	Layout Type Required
All 2D graphs	Layout2D
3D Histogram	Layout3DHistograms
3D Ternary Plot	LayoutTernary
3D Line Plot	Layout3DLines
3D Scatterplot	Layout3DScatterplot
3D Matrix Plot	LayoutMatrix
Icon Plot	LayoutIcons

NOTE

Different graph types have different layout types for their respective content. Consult the above table to see which layout objects should be used for a particular graph type.

Customizing 3D graphs is comparable to 2D graphs in that it relies on accessing the plot and axes of the graph's layout. For example, to customize a 3D scatterplot, we would do the following:

Step 1:

Create a 3D scatterplot and enter the following into a new macro:

```
Option Base 1
Option Explicit
Sub Main
    Dim graphLayout As Layout3DScatterplot
    Set graphLayout = ActiveGraph.Content
End Sub
```

Step 2:

Remembering from 2D graphs that all the plot features are available in the `Plots` property of the layout object, we will now add a fit line to the graph by adding this line:

```
graphLayout.Plots(1).Fits.Add
```

Now a linear fit (the default for a 3D scatterplot) will be added to our graph. Note that if we wanted a different type of fit (e.g., Quadratic), then you could do something like this:

```
graphLayout.Plots(1).Fits.Add.FitType = scgFit3DQuadratic
```

Step 3:

It is important to note that most of the customization features available to 2D graphs are also available to 3D graphs. For example, all the customizations that we made in the previous 2D graph example could be used in our 3D graph macro. Take this code from the 2D graph example and paste it into our current 3D macro:

```

graphLayout.Plots(1).DisplayPointLabels = True
graphLayout.Axes(scgLeftY).StepMode = scgManualStep
graphLayout.Axes(scgLeftY).SetManualRange(1,10)
graphLayout.Axes(scgX).DisplayMinorTickMark = True
graphLayout.Axes(scgX).Font.Face.FaceName = "MS UI Gothic"

```

After running the macro, note that all the changes made to the 2D graph (in the previous example) were now made to the 3D graph.

Step 4:

Along with most of the features of 2D graphs, the `Plots` property of a 3D graph's layout also has some of its own specific functionality. For a 3D scatterplot, it is possible to add customize spike lines to the points. Add the following lines to your macro:

```

graphLayout.Plots(1).DisplaySpike3D = True
graphLayout.Plots(1).Spike3D.Type.Value = scgSolid
graphLayout.Plots(1).Spike3D.ForegroundColor = RGB(0,255,0)

```

This will add spike lines between the points and the X axis, set them to solid lines, and finally set their color to neon green.

Step 5:

Editing the axes of a 3D graph is basically the same as a 2D graph. For example, to set the step size of the X axis, add these lines:

```

graphLayout.Axes(scgX).StepMode = scgManualStep
graphLayout.Axes(scgX).StepSize = 2

```

Another example could be to add a custom label to the X axis. Add these lines to change the value "2" to appear as "two pounds":

```

Dim custScaleValue As CustomScaleValue
Set custScaleValue =
graphLayout.Axes(scgX).CustomScaleValues.Add( )
custScaleValue.Value = 2
custScaleValue.Label = "two pounds"

```

In the above example, we first create a `CustomScaleValue` object to point to the custom label when we create it. Next, we call the `Add` function from the `CustomScaleValues` property of the X axis, which in turn returns a `CustomScaleValue` object. Finally, we set our custom label to replace the value "2" with "two pounds" by using the `CustomScaleValue` object that was returned from the `Add` function.

Our final 3D scatterplot customization macro will look like this:

```
Option Base 1
Option Explicit
Sub Main
    Dim graphLayout As Layout3DScatterplot
    Set graphLayout = ActiveGraph.Content
    graphLayout.Plots(1).Fits.Add.FitType = _
        scgFit3DQuadratic

    graphLayout.Plots(1).DisplayPointLabels = True
    graphLayout.Axes(scgLeftY).StepMode = scgManualStep
    graphLayout.Axes(scgLeftY).SetManualRange(1,10)
    graphLayout.Axes(scgX).DisplayMinorTickMark = True
    graphLayout.Axes(scgX).Font.Face.FaceName = _
        "MS UI Gothic"

    graphLayout.Plots(1).DisplaySpike3D = True
    graphLayout.Plots(1).Spike3D.Type.Value = scgSolid
    graphLayout.Plots(1).Spike3D.ForegroundColor = _
        RGB(0,255,0)

    graphLayout.Axes(scgX).StepMode = scgManualStep
    graphLayout.Axes(scgX).StepSize = 2

    Dim custScaleValue As CustomScaleValue
    Set custScaleValue = _
        graphLayout.Axes(scgX).CustomScaleValues.Add()
    custScaleValue.Value = 2
    custScaleValue.Label = "two pounds"
End Sub
```

***STATISTICA* Workbook**

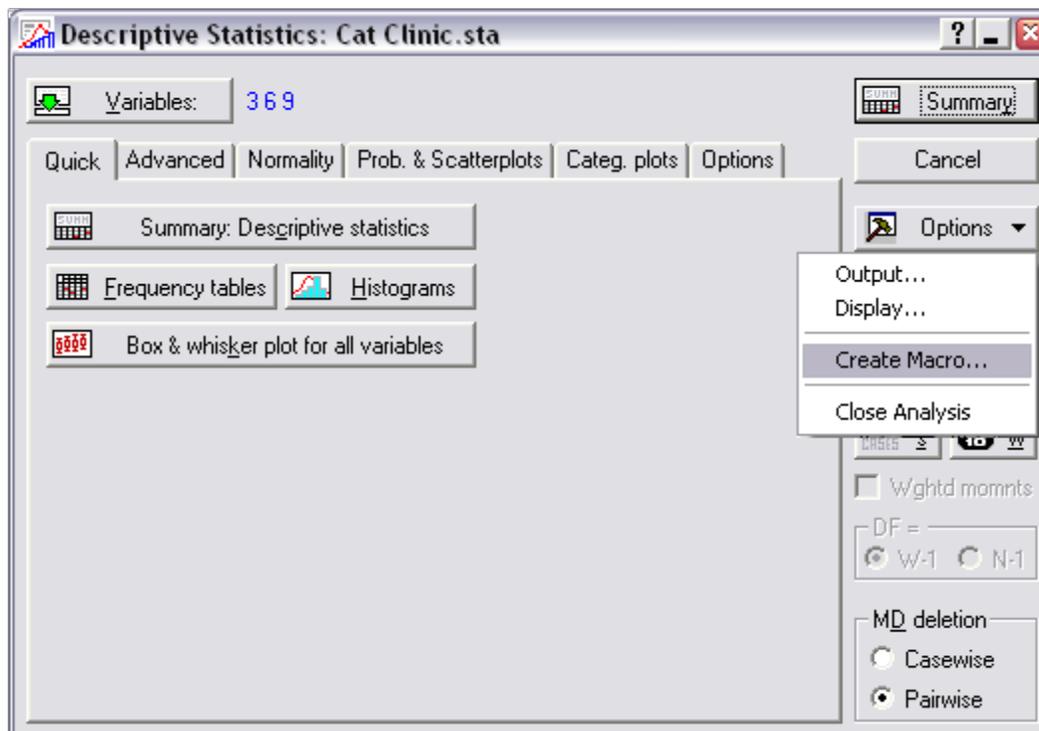
The default mode of output in *STATISTICA* is the workbook. The workbook is an OLE container document, meaning that it can store any ActiveX document. This not only includes most *STATISTICA* documents, but also many others, including Microsoft® Office documents.

Programmatically speaking, workbooks store their documents and folders as `WorkbookItem` objects. Every workbook has a root folder (the `Root` property will return this), and from there you can customize any subitem by setting a document variable to the desired node. You can also edit a node in a workbook as a `WorkbookItem` (e.g., renaming the item).

As an example, let us iterate through every document node in a workbook and append a time stamp to their respective names.

Step 1:

Perform a descriptive statistics on three variables and also generate some histograms. When finished, select the *Create Macro...* option from the *Options* button on the analysis dialog:



This is what our macro should look like:

```
Option Base 1
Sub Main
```

```
Dim newanalysis As Analysis
Set newanalysis = Analysis (scBasicStatistics, ActiveDataSet)
```

```
With newanalysis.Dialog
    .Statistics = scBasDescriptives
End With
```

```
newanalysis.Run
```

```
With newanalysis.Dialog
    .Variables = "3 6 9"
    .PairwiseDeletionOfMD = True
    .DisplayLongVariableNames = False
    .ExtendedPrecisionCalculations = False
    .PlotMedianQuartileRange = False
    .PlotMeanSEAndSD = False
```

```
.PlotMeanSD196TimesSD = True
.PlotMeanSE196TimesSE = False
.UserDefinedPercentiles = False
.ValidN = True
.Mean = True
.Median = False
.Mode = False
.GeometricMean = False
.HarmonicMean = False
.ConfLimitsForMeans = False
.Sum = False
.StandardDeviation = True
.Variance = False
.StandardErrorOfMean = False
.MinimumMaximum = True
.LowerUpperQuartiles = False
.Range = False
.QuartileRange = False
.Skewness = False
.Kurtosis = False
.StandardErrorOfSkewness = False
.StandardErrorOfKurtosis = False
.UseNumberOfIntervals = True
.NumberOfIntervals = 10
.NormalExpectedFrequencies = False
.KSAndLillieforsTestForNormality = True
.ShapiroWilkWTest = False
.ConfidenceIntervalForMeansPlot = 95
.CompressedStemAndLeaf = False
```

End With

```
newanalysis.RouteOutput(newanalysis.Dialog.Summary).Visible =
True
```

```
newanalysis.RouteOutput(newanalysis.Dialog.Histograms).Visible
= True
```

```
newanalysis.Dialog.ResultsVariables = "6 | 9"
newanalysis.RouteOutput(newanalysis.Dialog.Scatterplot2D).Visib
le = True
```

End Sub

Step 2:

Because we are going to iterate through every node in the workbook, we will need to set up a loop to do this. First create an iterator variable, which will be a `WorkbookItem`, and then set it to the root folder of the active workbook:

```
Dim CurrentItem As WorkbookItem
Set CurrentItem = ActiveWorkbook.Root
```

Next, set up the loop to iterate through every node in the workbook:

```
While (TypeName(CurrentItem) <> "Nothing")
    Set CurrentItem = ActiveWorkbook.NextItem(CurrentItem)
    If TypeName(CurrentItem) = "Nothing" Then Exit While
Wend
```

Basically, this is a while loop that will keep going until our `WorkbookItem` variable is set to nothing. When the return value of `NextItem` is nothing, then that indicates that we have reached the end of the workbook.

Step 3:

Inside of the loop, we will first see if the current node (workbook item) is a document, and if so add a time stamp to the end of its name. To do this, first we verify that the node is not a folder. If it is not a folder, then add the current date and time to its name. Add these lines to the bottom of the while loop:

```
If CurrentItem.Type <> scWorkbookItemTypeFolder Then
    CurrentItem.Name = CurrentItem.Name & " " & Now
End If
```

Our final macro should now look like this:

```
Option Base 1
Sub Main

Dim newanalysis As Analysis
Set newanalysis = Analysis (scBasicStatistics, ActiveDataSet)

With newanalysis.Dialog
    .Statistics = scBasDescriptives
```

End With

newanalysis.Run

```
With newanalysis.Dialog
  .Variables = "3 6 9"
  .PairwiseDeletionOfMD = True
  .DisplayLongVariableNames = False
  .ExtendedPrecisionCalculations = False
  .PlotMedianQuartileRange = False
  .PlotMeanSEAndSD = False
  .PlotMeanSD196TimesSD = True
  .PlotMeanSE196TimesSE = False
  .UserDefinedPercentiles = False
  .ValidN = True
  .Mean = True
  .Median = False
  .Mode = False
  .GeometricMean = False
  .HarmonicMean = False
  .ConfLimitsForMeans = False
  .Sum = False
  .StandardDeviation = True
  .Variance = False
  .StandardErrorOfMean = False
  .MinimumMaximum = True
  .LowerUpperQuartiles = False
  .Range = False
  .QuartileRange = False
  .Skewness = False
  .Kurtosis = False
  .StandardErrorOfSkewness = False
  .StandardErrorOfKurtosis = False
  .UseNumberOfIntervals = True
  .NumberOfIntervals = 10
  .NormalExpectedFrequencies = False
  .KSAndLillieforsTestForNormality = True
  .ShapiroWilkWTest = False
  .ConfidenceIntervalForMeansPlot = 95
  .CompressedStemAndLeaf = False
```

End With

```
newanalysis.RouteOutput(newanalysis.Dialog.Summary).Visible =
```

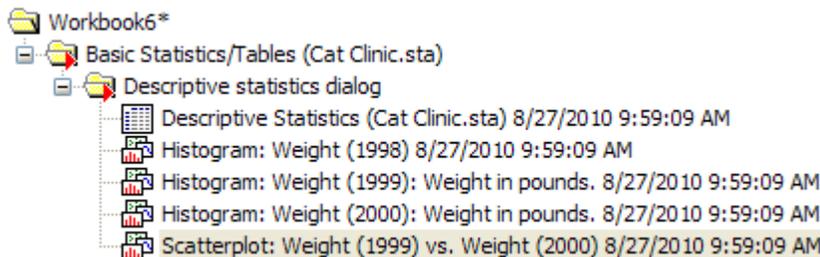
```
True
```

```
newanalysis.RouteOutput(newanalysis.Dialog.Histograms).Visible  
= True
```

```
newanalysis.Dialog.ResultsVariables = "6 | 9"  
newanalysis.RouteOutput(newanalysis.Dialog.Scatterplot2D).Visib  
le = True
```

```
Dim CurrentItem As WorkbookItem  
Set CurrentItem = ActiveWorkbook.Root  
While (TypeName(CurrentItem) <> "Nothing")  
    Set CurrentItem = ActiveWorkbook.NextItem(CurrentItem)  
    If TypeName(CurrentItem) = "Nothing" Then Exit While  
    If CurrentItem.Type <> scWorkbookItemTypeFolder Then  
        CurrentItem.Name = CurrentItem.Name & " " & Now  
    End If  
Wend  
  
End Sub
```

After running this, our workbook items' names will look like this:



For another example, let us further customize documents inside of a workbook. Retrieving documents from a workbook is similar to retrieving documents from a report in that each node is returned as an object. From this, we can set a document variable to any given node and have full access to its properties. The only requirement is that we need to know the node's index and which document type it is.

Building on our previous example, let us perform some customizations to the graphs in our workbook.

Step 1:

First, create a `WorkbookItem` variable and set it to the last graph. To do this, set the `WorkbookItem` to the first folder in the workbook and then move to the last graph by calling `NextItem`. Add the following to the macro:

```
Set wkItem = ActiveWorkbook.Root
For i = 1 To 7
    Set wkItem = ActiveWorkbook.NextItem(wkItem)
Next i
```

Step 2:

Next, create a graph variable and set it to the embedded graph. This is accomplished by retrieving the object (in this case, a graph) from our `WorkbookItem` variable.

```
Dim gr As Graph
Set gr = wkItem.Object
```

Step 3:

Now that we have a graph variable pointing to the workbook's last graph, we can customize it using the `Graph` class. Recalling from our graph customization tutorials, we will set a layout object to the graph's content and then customize that. Here we will set a `Layout2D` object to the graph:

```
Dim graphLayout As Layout2D
Set graphLayout = gr.Content
```

Step 4:

Let us apply the same color to the histograms in the workbook. To do this, we will iterate with our `WorkbookItem` variable through the workbook's histograms and make this change to each one. Add the following code to do this:

```
'Reset to the first folder and move to the first histogram
Set wkItem = ActiveWorkbook.Root
```

```

For i = 1 To 3
    Set wkItem = ActiveWorkbook.NextItem(wkItem)
Next i

'Iterate through all three histograms
For i = 1 To 3
    Set wkItem = ActiveWorkbook.NextItem(wkItem)
    Set gr = wkItem.Object
    Set graphLayout = gr.Content

graphLayout.Plots(1).Attribute.Bar.Area.ForegroundColor.Color = RGB(0,200,200)
Next i

```

Our final macro should look like this:

```

Option Base 1
Sub Main

Dim newanalysis As Analysis
Set newanalysis = Analysis (scBasicStatistics,
ActiveDataSet)

With newanalysis.Dialog
    .Statistics = scBasDescriptives
End With

newanalysis.Run

With newanalysis.Dialog
    .Variables = "3 6 9"
    .PairwiseDeletionOfMD = True
    .DisplayLongVariableNames = False
    .ExtendedPrecisionCalculations = False
    .PlotMedianQuartileRange = False
    .PlotMeanSEAndSD = False
    .PlotMeanSD196TimesSD = True
    .PlotMeanSE196TimesSE = False
    .UserDefinedPercentiles = False
    .ValidN = True
    .Mean = True
    .Median = False

```

```

.Mode = False
.GeometricMean = False
.HarmonicMean = False
.ConfLimitsForMeans = False
.Sum = False
.StandardDeviation = True
.Variance = False
.StandardErrorOfMean = False
.MinimumMaximum = True
.LowerUpperQuartiles = False
.Range = False
.QuartileRange = False
.Skewness = False
.Kurtosis = False
.StandardErrorOfSkewness = False
.StandardErrorOfKurtosis = False
.UseNumberOfIntervals = True
.NumberOfIntervals = 10
.NormalExpectedFrequencies = False
.KSAndLillieforsTestForNormality = True
.ShapiroWilkWTest = False
.ConfidenceIntervalForMeansPlot = 95
.CompressedStemAndLeaf = False
End With

newanalysis.RouteOutput(newanalysis.Dialog.Summary).Visible
= True

newanalysis.RouteOutput(newanalysis.Dialog.Histograms).Visi
ble = True

newanalysis.Dialog.ResultsVariables = "6 | 9"
newanalysis.RouteOutput(newanalysis.Dialog.Scatterplot2D).V
isible = True

Dim CurrentItem As WorkbookItem
Set CurrentItem = ActiveWorkbook.Root
While (TypeName(CurrentItem) <> "Nothing")
Set CurrentItem = ActiveWorkbook.NextItem(CurrentItem)
If TypeName(CurrentItem) = "Nothing" Then Exit While
If CurrentItem.Type <> scWorkbookItemTypeFolder Then
CurrentItem.Name = CurrentItem.Name & " " & Now
End If

```

```

Wend

'Move to the scatterplot in the workbook
Dim wkItem As WorkbookItem
Set wkItem = ActiveWorkbook.Root
For i = 1 To 7
    Set wkItem = ActiveWorkbook.NextItem(wkItem)
Next i

Dim gr As Graph
Set gr = wkItem.Object
Dim graphLayout As Layout2D
Set graphLayout = gr.Content

'Reset to the first folder and move to the first histogram
Set wkItem = ActiveWorkbook.Root
For i = 1 To 3
    Set wkItem = ActiveWorkbook.NextItem(wkItem)
Next i

'Iterate through all three histograms
For i = 1 To 3
    Set wkItem = ActiveWorkbook.NextItem(wkItem)
    Set gr = wkItem.Object
    Set graphLayout = gr.Content

graphLayout.Plots(1).Attribute.Bar.Area.ForegroundColor.Color = RGB(0,200,200)
Next i

End Sub

```

STATISTICA Report

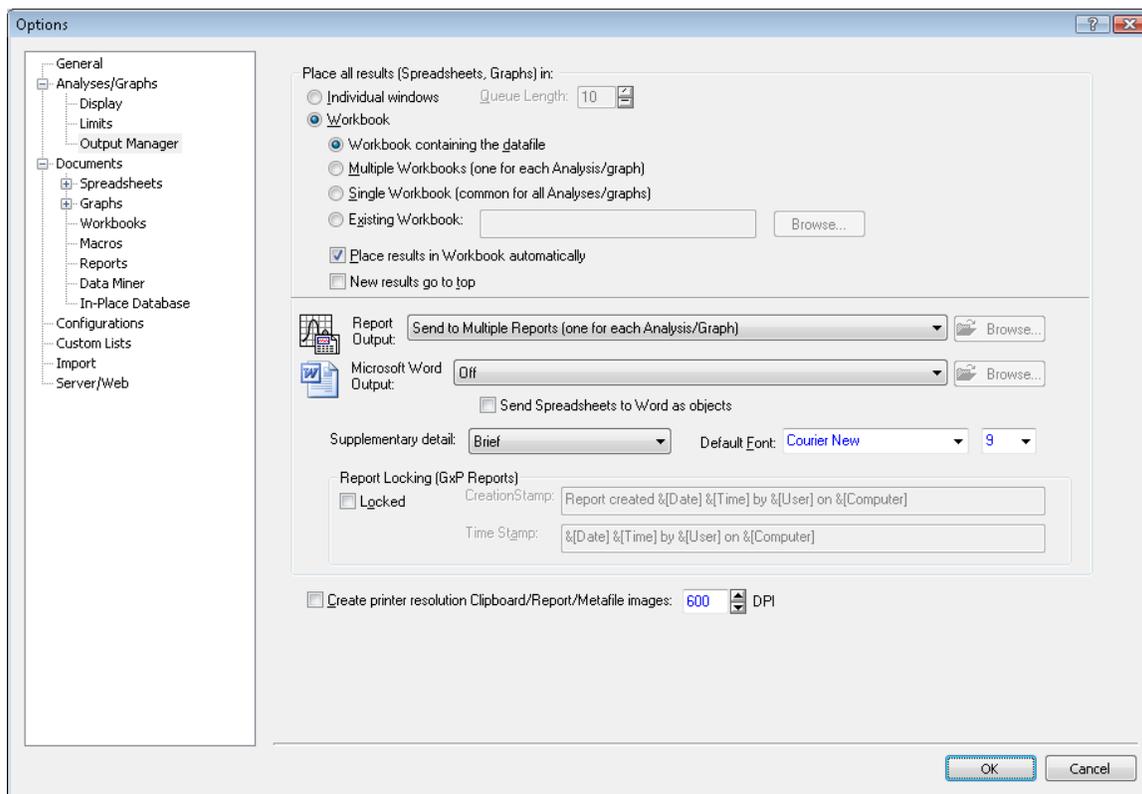
Along with workbooks, *STATISTICA* also offers the ability to organize your output to formatted reports. To set your analyses to also route their output to a report, set *Report Output* to *Send to Multiple Reports*. This is located on the *Output Manager* tab of the *Options* dialog (available from the *Options* button on the *Home* tab of the ribbon).

Although these reports will contain all the output from your analyses (along with supplementary information), *STATISTICA* also offers ability to further edit them. *STATISTICA*'s *Report* object grants you all the report-editing features available

interactively, making the customizing of a report a simple, automated task. Let us look at some useful examples of customizing a report via automation. The first example is how to add a custom printer header and footer.

Step 1:

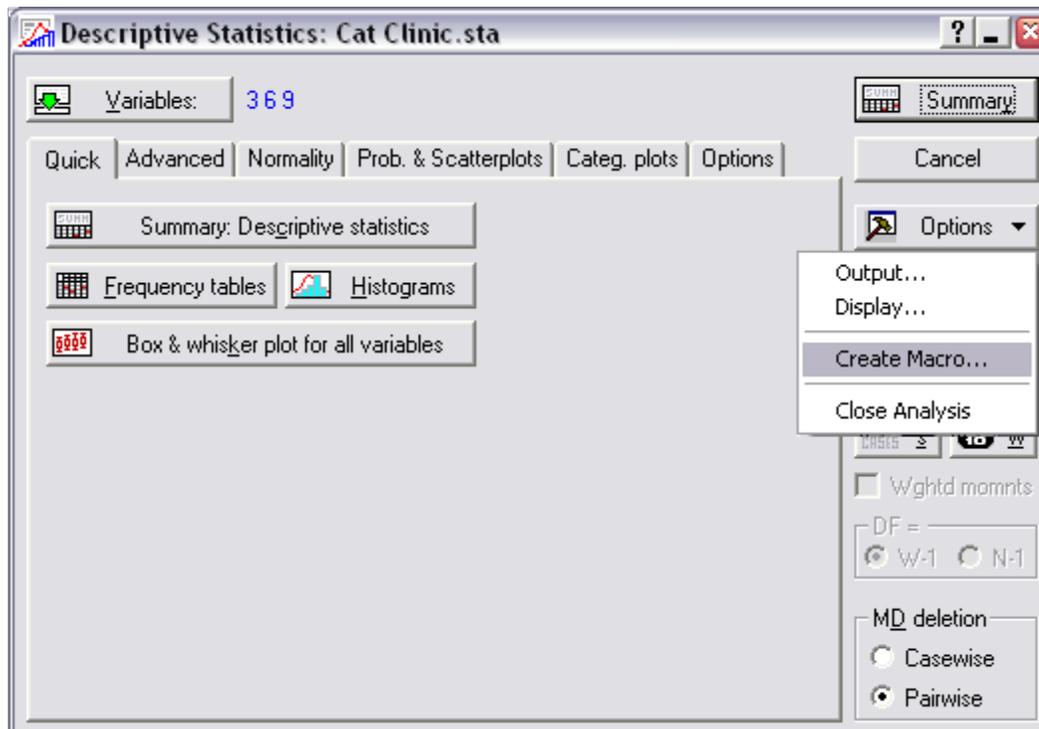
First, we will set our output to also be sent to a report. Go to the *Home* tab on the Ribbon and click the *Options* button. On the *Options* dialog, select the *Output Manager* page:



Set *Report Output* to *Send to Multiple Reports* and set *Supplementary detail* to *Brief*.

Step 2:

Perform a descriptive statistics on three variables and also generate some histograms. When finished, select the *Create Macro...* option from the *Options* button on the analysis dialog:



This is what our macro should look like:

```
Option Base 1
Sub Main
```

```
Dim newanalysis As Analysis
Set newanalysis = Analysis (scBasicStatistics, ActiveDataSet)
```

```
With newanalysis.Dialog
    .Statistics = scBasDescriptives
End With
```

```
newanalysis.Run
```

```
With newanalysis.Dialog
    .Variables = "3 6 9"
    .PairwiseDeletionOfMD = True
    .DisplayLongVariableNames = False
    .ExtendedPrecisionCalculations = False
    .PlotMedianQuartileRange = False
```

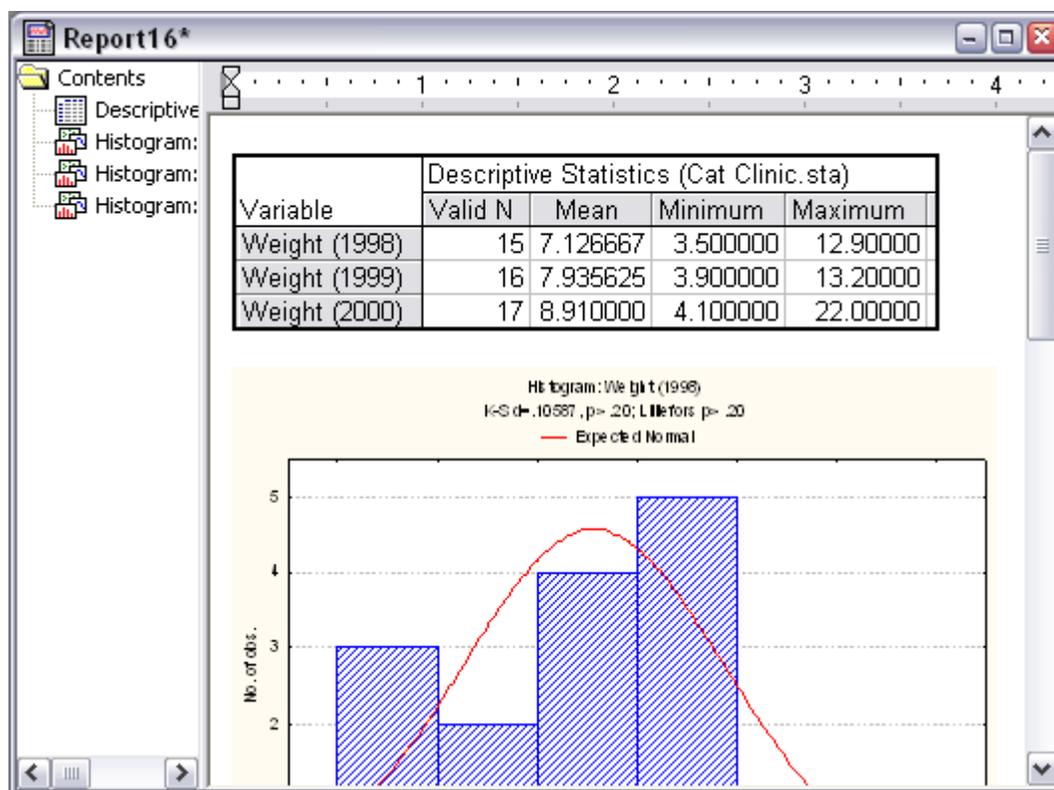
```
.PlotMeanSEAndSD = False
.PlotMeanSD196TimesSD = True
.PlotMeanSE196TimesSE = False
.UserDefinedPercentiles = False
.ValidN = True
.Mean = True
.Median = False
.Mode = False
.GeometricMean = False
.HarmonicMean = False
.ConfLimitsForMeans = False
.Sum = False
.StandardDeviation = True
.Variance = False
.StandardErrorOfMean = False
.MinimumMaximum = True
.LowerUpperQuartiles = False
.Range = False
.QuartileRange = False
.Skewness = False
.Kurtosis = False
.StandardErrorOfSkewness = False
.StandardErrorOfKurtosis = False
.UseNumberOfIntervals = True
.NumberOfIntervals = 10
.NormalExpectedFrequencies = False
.KSAndLillieforsTestForNormality = True
.ShapiroWilkWTest = False
.ConfidenceIntervalForMeansPlot = 95
.CompressedStemAndLeaf = False
End With

newanalysis.RouteOutput(newanalysis.Dialog.Summary).Visible =
True

newanalysis.RouteOutput(newanalysis.Dialog.Histograms).Visible
= True

End Sub
```

In addition, our output report should look something like this:



Step 3:

To add a header to our report, add the following to the end of the macro:

```
ActiveReport.PrintHeader(scPrintFooterCenter) = "CONFIDENTIAL:  
For Acme, inc. use only"
```

This will add a centered printer header saying "CONFIDENTIAL: For Acme, inc. use only" into the report. When you print the report, this header will appear on every page.

Step 4:

To add a footer to the report, add the following to the end of the macro:

```
ActiveReport.PrintFooter(scPrintFooterLeft) =
"&[Date]"ActiveReport.PrintFooter(scPrintFooterCenter) = "Page
&[Page] of &[Pages]"
```

This will add a printer footer with the date, current page, and total pages on every page. An interesting note here is that you can use the same header/footer syntax here that you can interactively. For example, the syntax "&[Page]" will display the current page number in the footer.

Another common feature is to add text to a report. Interactively, all that we have to do is type this information into the report; adding text to the report programmatically is almost as easy. Let us look at an example of adding some descriptive text throughout a report.

Step 1:

First, we should declare our string constants that we will be placing into the report. Declaring these as constants is a good idea because if we need to change them later, then we only need to change them in one place.

Add these lines to the beginning of the macro:

```
Const REPORT_HEADER As String = "Weight Analysis for 1998-
2000"

Const HISTOGRAM1_HEADER As String = "Weight Histogram
(1998)"
Const HISTOGRAM2_HEADER As String = "Weight Histogram
(1999)"
Const HISTOGRAM3_HEADER As String = "Weight Histogram
(2000)"
```

Step 2:

Next, we will set the selection of the report to the beginning of the document and then add our header to it. Add these following lines to the macro:

```
ActiveReport.SetSelection(1,1)
ActiveReport.SelectionText = REPORT_HEADER & vbCrLf
```

Step 3:

Next, we will insert some text above the first graph. To do this, we will “step over” the embedded spreadsheet and the text that we just inserted and set the selection to be right above the graph. Call `SetSelection` again with the length of the text that we just inserted (including the carriage return after it). We must also add another 2 to that so that we can step over the embedded spreadsheet and the carriage return after it. Add these lines of code to move the selection above the first graph and to add a title above it:

```
Dim TotalSizeOfInsertedText As Long
TotalSizeOfInsertedText = Len(REPORT_HEADER & vbCrLf)
ActiveReport.SetSelection(TotalSizeOfInsertedText+2,
TotalSizeOfInsertedText+2)
ActiveReport.SelectionText = vbCrLf & HISTOGRAM1_HEADER
```

Step 4:

Finally, do the same as you did in step 3 for the second and third graph by adding this code:

```
TotalSizeOfInsertedText = TotalSizeOfInsertedText +
Len(HISTOGRAM1_HEADER & vbCrLf)
ActiveReport.SetSelection(TotalSizeOfInsertedText+4,
TotalSizeOfInsertedText+4)
ActiveReport.SelectionText = vbCrLf & HISTOGRAM2_HEADER

TotalSizeOfInsertedText = TotalSizeOfInsertedText +
Len(HISTOGRAM2_HEADER & vbCrLf)
ActiveReport.SetSelection(TotalSizeOfInsertedText+6,
TotalSizeOfInsertedText+6)
ActiveReport.SelectionText = vbCrLf & HISTOGRAM3_HEADER
```

We could even add an RTF title to our report. Add this to the macro:

```
ActiveReport.SetSelection(1,1)
'add some space at the top of the report for our title
ActiveReport.SelectionText = vbCrLf & vbCrLf
ActiveReport.SetSelection(1,1)
```

```
ActiveReport.SelectionRTF = _
    "{\rtf1\ansi\ansicpg1252\deff0\deflang1029" & _
    "{\fonttbl{\f0\fnil\fprq4\fcharset0 Arial;}}" & _
    "{\colortbl ;\red0\green64\blue64;}" & _
    "\uc1\pard\cf1\lang1033\ul\b\f0\fs22 " & _
    "Weight Report (" & Trim(Str(Date)) & ")}"
```

As a final example, let us look at how to edit embedded objects within a report. If you know which document type an embedded object is, then set a temporary document variable to that object to edit it. Use the report's function `GetObject` to retrieve any embedded object.

Step 1:

First, get the spreadsheet in the report and sort the *Maximum* column:

```
Dim spr As Spreadsheet
Set spr = ActiveReport.GetObject(1)
spr.SortData(4)
```

Because we know that the first embedded object in the report is a spreadsheet, we set a spreadsheet variable to it by calling `GetObject` and telling it object 1.

Step 2:

Now, let us customize the second embedded object, which is a graph. First, set a `Graph` object to this embedded object:

```
Dim gr As Graph
Set gr = ActiveReport.GetObject(2)
```

Next, to customize the graph we need to create a layout object and set it to the graph's content. In this case, we know the graph is a 2D histogram and requires a `Layout2D` object.

```
Dim graphLayout As Layout2D
Set graphLayout = gr.Content
```

As an example, let us change the coordinate system of the graph to polar by using our Layout2D object:

```
graphLayout.CoordinateSystem = scgPolar
```

Finally, set the graph's background color to a shade of green:

```
gr.GraphWindow.Background.Color = RGB(50, 200, 50)
```

Our final analysis macro with numerous report customizations should appear like this:

```
Option Base 1
Sub Main

Dim newanalysis As Analysis
Set newanalysis = Analysis(scBasicStatistics,
ActiveDataSet)

With newanalysis.Dialog
.Statistics = scBasDescriptives
End With

newanalysis.Run

With newanalysis.Dialog
.Variables = "3 6 9"
.PairwiseDeletionOfMD = True
.DisplayLongVariableNames = False
.ExtendedPrecisionCalculations = False
.PlotMedianQuartileRange = False
.PlotMeanSEAndSD = False
.PlotMeanSD196TimesSD = True
.PlotMeanSE196TimesSE = False
.UserDefinedPercentiles = False
.ValidN = True
.Mean = True
```

```
.Median = False
.Mode = False
.GeometricMean = False
.HarmonicMean = False
.ConfLimitsForMeans = False
.Sum = False
.StandardDeviation = True
.Variance = False
.StandardErrorOfMean = False
.MinimumMaximum = True
.LowerUpperQuartiles = False
.Range = False
.QuartileRange = False
.Skewness = False
.Kurtosis = False
.StandardErrorOfSkewness = False
.StandardErrorOfKurtosis = False
.UseNumberOfIntervals = True
.NumberOfIntervals = 10
.NormalExpectedFrequencies = False
.KSAndLillieforsTestForNormality = True
.ShapiroWilkWTest = False
.ConfidenceIntervalForMeansPlot = 95
.CompressedStemAndLeaf = False
```

End With

```
newanalysis.RouteOutput(newanalysis.Dialog.Summary).Visible
= True
```

```
newanalysis.RouteOutput(newanalysis.Dialog.Histograms).Visi
ble = True
```

'add a header and footer

```
ActiveReport.PrintHeader(scPrintFooterCenter) = _
    "CONFIDENTIAL: For Acme, inc. use only"
```

```
ActiveReport.PrintFooter(scPrintFooterLeft) = _
    "&[Date]"
```

```
ActiveReport.PrintFooter(scPrintFooterCenter) = _
    "Page &[Page] of &[Pages]"
```

```
Const REPORT_HEADER As String = _
```

```

    "Weight Analysis for 1998-2000"
Const HISTOGRAM1_HEADER As String = _
    "Weight Histogram (1998)"
Const HISTOGRAM2_HEADER As String = _
    "Weight Histogram (1999)"
Const HISTOGRAM3_HEADER As String = _
    "Weight Histogram (2000)"

'add a header above the spreadsheet
ActiveReport.SetSelection(1,1)
ActiveReport.SelectionText = REPORT_HEADER & vbCrLf

'add a header above the first graph
Dim TotalSizeOfInsertedText As Long
TotalSizeOfInsertedText = Len(REPORT_HEADER & vbCrLf)
ActiveReport.SetSelection(TotalSizeOfInsertedText+2, _
    TotalSizeOfInsertedText+2)
ActiveReport.SelectionText = vbCrLf & HISTOGRAM1_HEADER

'add a header above the second graph
TotalSizeOfInsertedText = TotalSizeOfInsertedText + _
    Len(HISTOGRAM1_HEADER & vbCrLf)
ActiveReport.SetSelection(TotalSizeOfInsertedText+4, _
    TotalSizeOfInsertedText+4)
ActiveReport.SelectionText = vbCrLf & HISTOGRAM2_HEADER

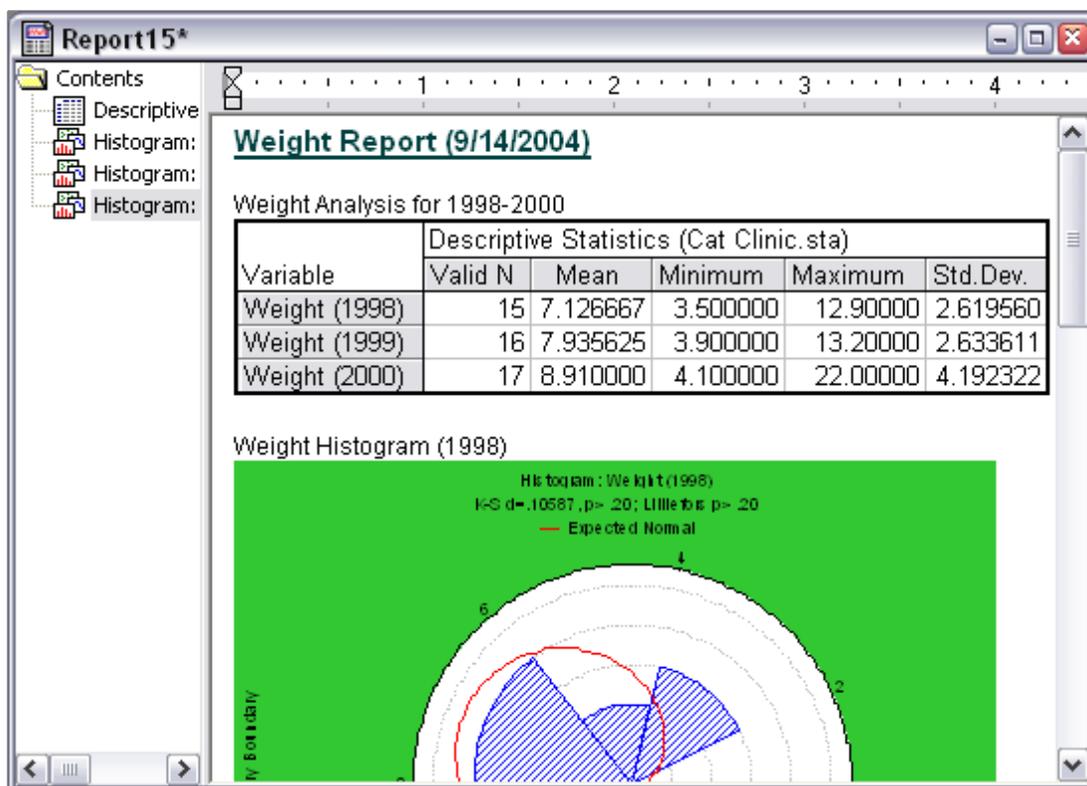
'add a header above the third graph
TotalSizeOfInsertedText = TotalSizeOfInsertedText + _
    Len(HISTOGRAM2_HEADER & vbCrLf)
ActiveReport.SetSelection(TotalSizeOfInsertedText+6, _
    TotalSizeOfInsertedText+6)
ActiveReport.SelectionText = vbCrLf & HISTOGRAM3_HEADER

'add an RTF-formatted header for the whole report
ActiveReport.SetSelection(1,1)
'add some space at the top of the report for our title
ActiveReport.SelectionText = vbCrLf & vbCrLf
ActiveReport.SetSelection(1,1)
ActiveReport.SelectionRTF = _
    "{\rtf1\ansi\ansicpg1252\deff0\deflang1029" & _
    "{\fonttbl{\f0\fnil\fprq4\fcharset0 Arial;}}}" & _
    "{\colortbl ;\red0\green64\blue64;}" & _

```

```
"\ucl\pard\cf1\lang1033\u1\b\f0\fs22 " & _  
"Weight Report (" & Trim(Str(Date)) & ")}"  
  
'sort the Maximum column of the spreadsheet  
Dim spr As Spreadsheet  
Set spr = ActiveReport.GetObject(1)  
spr.SortData(4)  
  
'change the coordinate system of the graph to polar  
Dim gr As Graph  
Set gr = ActiveReport.GetObject(2)  
Dim graphLayout As Layout2D  
Set graphLayout = gr.Content  
graphLayout.CoordinateSystem = scgPolar  
gr.GraphWindow.Background.Color = RGB(50, 200, 50)  
  
End Sub
```

Also, when we run this on the dataset *Cat Clinic* (located in *STATISTICA's examples/datasets* folder), then we should have an output report that looks something like this:

**NOTE**

After formatting your report, you can also save it as a PDF file by adding the following line:

ActiveReport.SaveAsPDF("C:\Data\AnalysisReport.pdf")

STATISTICA Macro

STATISTICA's macro document is similar to other STATISTICA documents in that all its features are available via automation. This enables us to open and run other macros from a macro, along with the ability to pass arguments between them. As an example, say that we have an SVB macro that subsets the active spreadsheet. It will base this subset on whether the values of the first variable contain text similar to "fail", and then exports it as an HTML file. This macro will look like this:

```

Option Base 1
Option Explicit
Sub Main
    Dim spr As Spreadsheet
    Set spr = ActiveSpreadsheet.Subset("", "v1 LIKE 'FAIL%'")
    spr.Header.Value = "Failure report"
    spr.SaveAs("C:\Data\Failure Report.htm", True)
    Set spr = Nothing
End Sub

```

Now, let us say that we have another macro that needs to perform this action with a particular spreadsheet. One option is to simply copy and paste the above into a new macro and rewrite it to be a separate function. However, the drawback to this approach is that future improvements made to the original macro will not be reflected in the newer one. A better method would be to run the original macro from the newer one and pass a spreadsheet to it. This way, we would always be using the current version of the original macro. To do this, we need to augment the first macro so that it can accept an argument.

Step 1:

Change the first macro to look like this:

```

Option Base 1
Option Explicit
Sub Main
    Dim spr As Spreadsheet
    Set spr = GetScriptArgument().Subset("", "v1 LIKE
'FAIL%'")
    spr.Header.Value = "Failure report"
    spr.SaveAs("C:\Failure Report.htm", True)
    Set spr = Nothing
End Sub

```

The only adjustment that we made was to change “ActiveSpreadsheet” to “GetScriptArgument()”. The function `GetScriptArgument` retrieves the argument passed to the macro. This argument is a variant, meaning that you can pass any data type to the macro. In this case, we are passing a spreadsheet object. Note that if nothing is passed to the macro, then `GetScriptArgument` will return the active

spreadsheet.

Step 2:

Now, in our second macro we will open the first macro to use it. In a new macro, type the following:

```
Option Base 1
Option Explicit
Sub Main
    Dim mc As Macro
    Set mc = Macros.Open("C:\Macros\SubsetFailures.svb")
End Sub
```

If the first macro was saved as "C:\Macros\SubsetFailures.svb", then this macro will create a `Macro` object and set it to the first macro. Note that we use the `Macros` collection to open the first macro.

Step 3:

Finally, we will use our `Macro` object, which is pointing to an open instance of "C:\Macros\SubsetFailures.svb", to run it. The `Macro` object provides a regular `Execute` function that simply runs the macro, but here we want to pass an argument to it. In this case, we will use the `ExecuteWithArgument` function to do this. Add the following line to the macro:

```
mc.ExecuteWithArgument(Spreadsheets.Open(Path &
"\Examples\Datasets\Adstudy.sta") )
```

This will open the example dataset *Adstudy* and pass it to *SubsetFailures.svb*. This will make *SubsetFailures.svb* use *Adstudy* as its input, rather than the active spreadsheet.

NOTE

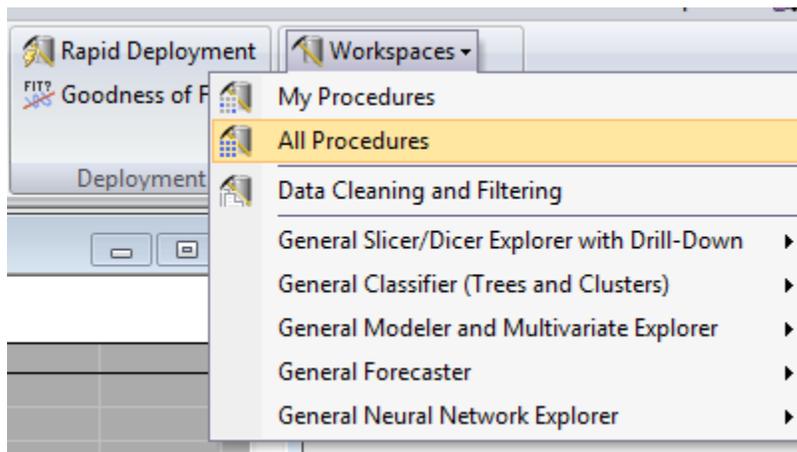
To execute a macro without arguments, simply use the `Execute` function instead. For example:

```
Dim mc As Macro
Set mc = Macros.Open("C:\Macros\SubsetFailures.svb")
mc.Execute()
```

Customizing Data Miner and ByGroup Nodes

Although the provided Data Miner and ByGroup nodes already offer many features, you can also enhance them. This enables you to configure analysis, graph, and data cleansing Data Miner nodes to better suit your needs.

Begin by opening a new Data Miner workspace. To create this new workspace from the Ribbon bar, select the *Data Mining* menu. Select *Workspaces, All Procedures*. A blank data miner workspace will be displayed.



Click the *Node Browser* button (or CTRL+B) to open the *Node Browser*. Ensure that the current selection in the box at the top is "All Procedures". Select the *User-Defined Subset* node in the *Data Cleaning and Filtering* folder and click *Insert into workspace*. Once inserted, close the *Node Browser* dialog.

Begin the customization of the node by renaming it. Right-click the *User-Defined Subset* node and select *Rename*. Name the node *Recode MD* and click *OK*. The node will be now be renamed. Right-click on the *Recode MD* node and select *Edit Code*. A macro window will open with the default code for the *User-Defined Subset* node:

```
'This function accepts as input an
'InputDescriptor (DataIn),
'applies a filter/subset operation, and returns the
'filtered data in DataOut
```

```

Private Sub SubsetNode( _
    DataIn() As InputDescriptor, _
    DataOut() As InputDescriptor)
ReDim DataOut(LBound(DataIn()) To UBound(DataIn())) As
InputDescriptor
For i=LBound(DataIn()) To UBound(DataIn())
    Set DataOut(i)=DataIn(i).Clone()
Next i

End Sub

```

Because this node is a subset node, it appears in the *Data Preparation, Cleaning and Transformation* pane of the Data Miner workspace. When the Data Miner project is ran, the `SubsetNode()` function is automatically called, instead of the `Main()` function. The `SubsetNode()` function takes as its arguments `DataIn()` and `DataOut()`.

`DataIn()` represents the collection of documents which are connected to this node in the Data Miner workspace. For example, if three spreadsheets were inserted into the Data Miner workspace and connected to this node, then when the node is executed, `DataIn()` will be an array of three `InputDescriptor` objects.

An `InputDescriptor` is an object which represents the dataset in a Data Miner project. The `InputDescriptor` object contains a `Datasource` property that returns the actual spreadsheet/IDP document that the node represents. The `InputDescriptor` object also contains configuration information for the node as well. This information includes variable selections, code selections, case selection conditions, case weights, etc.

The `DataOut()` parameter of the `SubsetNode()` function is the output from the node. Depending on the purpose of the node, the number of items in the `DataOut()` collection may match the number of items in the `DataIn()` collection. For example, if you are merging spreadsheets, you might have three inputs to a node and only one output. In our example, we are going to perform a missing data replacement operation on each input spreadsheet. For each member in the `DataIn()` collection, we will have a respective item in the `DataOut()` collection.

As written, the code for this node matches our needs. It makes a clone of each

`InputDescriptor`. When the `InputDescriptor.Clone()` function is called, a duplicate copy of the `InputDescriptor` is created, including the data source, variable selections, etc. We will need to add code after the clone process to modify the data source of the cloned `InputDescriptor`.

We will be using the `ReplaceMD()` function of the spreadsheet object to replace the missing data within the selected variables with the mean of the variable(s). We will make the assumption that the variables we will perform this operation on are selected as continuous dependent variables.

Modify the macro by adding the “`Dim spr As Spreadsheet`” line (as indicated below). Then, start typing the “`spr.ReplaceMD(`” line.

```
Private Sub SubsetNode( _
    DataIn() As InputDescriptor, _
    DataOut() As InputDescriptor)
ReDim DataOut(LBound(DataIn()) To UBound(DataIn())) _
    As InputDescriptor
Dim spr As Spreadsheet

For i=LBound(DataIn()) To UBound(DataIn())
    Set DataOut(i)=DataIn(i).Clone()
    Set spr = DataOut(i).DataSource
    spr.ReplaceMD(
Next i

End Sub
```

Once you type the left parenthesis, the macro editor will display tooltip text (Intellisense) that shows the expected parameters for the `ReplaceMD()` function. Notice that the parameters include the variable list, inclusion statement, includes list, exclusion statement, exclude list, and weight. Of all these parameters, only the variable list is a required parameter (as indicated by the brackets around the remaining parameters). For this example, we will use all parameters except the weight. Leave the “`spr.ReplaceMD(`” line of code incomplete and add the declarations for the new variables to store the needed values:

```

Private Sub SubsetNode( _
    DataIn() As InputDescriptor, _
    DataOut() As InputDescriptor)
ReDim DataOut(LBound(DataIn()) To UBound(DataIn())) _
    As InputDescriptor
Dim spr As Spreadsheet
Dim VariableString As String
Dim SCIncludeExpression As String
Dim SCIncludeList As String
Dim SCExcludeExpression As String
Dim SCExcludeList As String

For i=LBound(DataIn()) To UBound(DataIn())
    Set DataOut(i)=DataIn(i).Clone()
    Set spr = DataOut(i).DataSource
    spr.ReplaceMD(
Next i

End Sub

```

For each InputDescriptor, we will need to retrieve the values for the newly added variables before calling the ReplaceMD() function:

```

Private Sub SubsetNode( _
    DataIn() As InputDescriptor, _
    DataOut() As InputDescriptor)
ReDim DataOut(LBound(DataIn()) To UBound(DataIn())) _
    As InputDescriptor
Dim spr As Spreadsheet
Dim VariableString As String
Dim SCIncludeExpression As String
Dim SCIncludeList As String
Dim SCExcludeExpression As String
Dim SCExcludeList As String

For i=LBound(DataIn()) To UBound(DataIn())
    Set DataOut(i)=DataIn(i).Clone()
    Set spr = _
        DataOut(i).DataSource
    VariableString = _
        DataOut(i).ContinuousDependentVariablesString

```

```

    SCIncludeExpression = _
        DataOut(i).SelectionCondition.IncludeExpression
    SCIncludeList = _
        DataOut(i).SelectionCondition.IncludeList
    SCExcludeExpression = _
        DataOut(i).SelectionCondition.ExcludeExpression
    SCExcludeList = _
        DataOut(i).SelectionCondition.ExcludeList
    spr.ReplaceMD(
Next i

End Sub

```

The last change will be to pass these parameters into the `ReplaceMD()` function and reset them after each iteration:

```

Private Sub SubsetNode( _
    DataIn() As InputDescriptor, _
    DataOut() As InputDescriptor)
ReDim DataOut(LBound(DataIn()) To UBound(DataIn())) As
InputDescriptor
Dim spr As Spreadsheet
Dim VariableString As String
Dim SCIncludeExpression As String
Dim SCIncludeList As String
Dim SCExcludeExpression As String
Dim SCExcludeList As String

For i=LBound(DataIn()) To UBound(DataIn())
Set DataOut(i)=DataIn(i).Clone()
Set spr = DataOut(i).DataSource
VariableString = _
    DataOut(i).ContinuousDependentVariablesString
SCIncludeExpression = _
    DataOut(i).SelectionCondition.IncludeExpression
SCIncludeList = _
    DataOut(i).SelectionCondition.IncludeList
SCExcludeExpression = _
    DataOut(i).SelectionCondition.ExcludeExpression
SCExcludeList = _
    DataOut(i).SelectionCondition.ExcludeList

```

```

spr.ReplaceMD
(VariableString, SCIncludeExpression, _
    SCIncludeList, SCExcludeExpression, SCExcludeList)
VariableString = ""
SCIncludeExpression = ""
SCIncludeList = ""
SCExcludeExpression = ""
SCExcludeList = ""

Next i

End Sub

```

As written above, the selection conditions will always be applied in the ReplaceMD() function call. To change this, apply the assignment only if the selection conditions are enabled in the InputDescriptor:

```

Private Sub SubsetNode( _
    DataIn() As InputDescriptor, _
    DataOut() As InputDescriptor)
ReDim DataOut(LBound(DataIn()) To UBound(DataIn())) _
    As InputDescriptor
Dim spr As Spreadsheet
Dim VariableString As String
Dim SCIncludeExpression As String
Dim SCIncludeList As String
Dim SCExcludeExpression As String
Dim SCExcludeList As String

For i=LBound(DataIn()) To UBound(DataIn())
Set DataOut(i)=DataIn(i).Clone()
Set spr = DataOut(i).DataSource
VariableString = _
    DataOut(i).ContinuousDependentVariablesString
If DataOut(i).SelectionCondition.Enabled = True Then
    SCIncludeExpression = _
        DataOut(i).SelectionCondition.IncludeExpression
    SCIncludeList = _
        DataOut(i).SelectionCondition.IncludeList
    SCExcludeExpression = _
        DataOut(i).SelectionCondition.ExcludeExpression

```

```

        SCExcludeList = _
            DataOut(i).SelectionCondition.ExcludeList
    End If
    spr.ReplaceMD(VariableString, SCIncludeExpression, _
        SCIncludeList, SCExcludeExpression, SCExcludeList)
    VariableString = ""
    SCIncludeExpression = ""
    SCIncludeList = ""
    SCExcludeExpression = ""
    SCExcludeList = ""

    Next i

End Sub

```

The code is now complete. To test the node, insert a dataset with missing data into the workspace, connect it to the node, select the variable containing missing data as a continuous, dependent variable, and then run the project. With the variables selected as continuous dependents, their missing data will be replaced with their respective means. The node also accommodates multiple `InputDescriptors` and selection conditions.

Programming *STATISTICA* with Other Languages

Although *STATISTICA*'s SVB editor provides a powerful editing and debugging environment, there may be situations where you need to use *STATISTICA*'s object model from another environment. For example, you may need to incorporate a *STATISTICA* analysis into a C# or C++ project. Due to *STATISTICA*'s COM-based architecture, all of *STATISTICA*'s libraries are fully available to COM-compliant environments. Just some possible environments include Microsoft® Visual C++, Microsoft® C#, and Microsoft® Visual Basic.NET. Let us look at some of these environments and discuss how to import *STATISTICA*'s libraries.

Microsoft® Visual C++

To add any *STATISTICA* library to a Visual C++ project, following these steps:

Step1:

If your project already has COM support (e.g., an MFC project), then skip this step; otherwise, add the following to your main header (e.g., `StdAfx.h`):

```
#include <windows.h>
#include <comdef.h>
```

Step 2:

Use the Visual C++ preprocessor command *import* to import the COM interfaces of *STATISTICA* into your project. You will need to specify the EXE or DLL file to import. For example, to import *STATISTICA*'s object and graphics libraries, enter these lines to the top of your main header file (e.g. StdAfx.h):

```
#import "C:\Program Files\STATISTICA\statist.exe"
#import "C:\Program Files\STATISTICA\stl_tgra.dll"
```

NOTE

The “*#import*” preprocessor command is not an ANSI C command—it is specific to Microsoft® Visual C++. To import COM interfaces in a different C++ compiler environment, consult your compiler vendor’s documentation.

Step 3:

Before using any *STATISTICA* objects (or any other COM objects), you must call `CoInitialize` and `CoUninitialize`. It is recommended to call `CoInitialize` when your program first starts and call `CoUninitialize` when it exits. A common trick is to create a global object that calls `CoInitialize` in its constructor and `CoUninitialize` in its destructor. Add the following to the global area of your program:

```
//Initializes COM at start up and closes it at program exit
class ComInitializer
{
public:
    ComInitializer() { ::CoInitialize(NULL); }
    ~ComInitializer() { ::CoUninitialize(); }
} gComInitializer;
```

Step 4:

Next, create an instance of *STATISTICA*. We will first declare an `ApplicationPtr` object and then call `CoCreateInstance` to create an instance of *STATISTICA*.

```
STATISTICA::_ApplicationPtr pApp;
hr = ::CoCreateInstance(__uuidof(STATISTICA::Application),
                       NULL, CLSCTX_SERVER,
                       __uuidof(STATISTICA::_Application),
                       reinterpret_cast<void**>(&pApp));
if (FAILED(hr) )
    {
        ::MessageBox(NULL, "Unable to initialize STATISTICA",
                    "Initialization Failure", MB_ICONHAND);
    }
```

We could even use an already opened *STATISTICA*, or create our own instance if one cannot be found:

```
STATISTICA::_ApplicationPtr pApp;
CLSID clsid;
::CLSIDFromString(
    L"{C9E28000-3D45-11D4-9FF4-00C04FA0D540}", &clsid);
IUnknown* pUnk = NULL;

//look for an active STATISTICA
HRESULT hr = GetActiveObject(clsid, NULL,
                            reinterpret_cast<Iunknown**>(&pUnk));
if(SUCCEEDED(hr))
    {
        pUnk->QueryInterface
            (__uuidof(STATISTICA::_Application),
            reinterpret_cast<void**>(&pApp));
    }
else
    {
        //ST not open, so open our own copy
        hr = ::CoCreateInstance(
            __uuidof(STATISTICA::Application),
            NULL, CLSCTX_SERVER,
            __uuidof(STATISTICA::_Application),
            reinterpret_cast<void**>(&pApp));
        if (FAILED(hr) )
```

```
        {  
            ::MessageBox(NULL, "Unable to initialize  
STATISTICA",  
                "Initialization Failure", MB_ICONHAND);  
        }  
    }
```

Step 5:

Finally, use the *STATISTICA* application to create a spreadsheet:

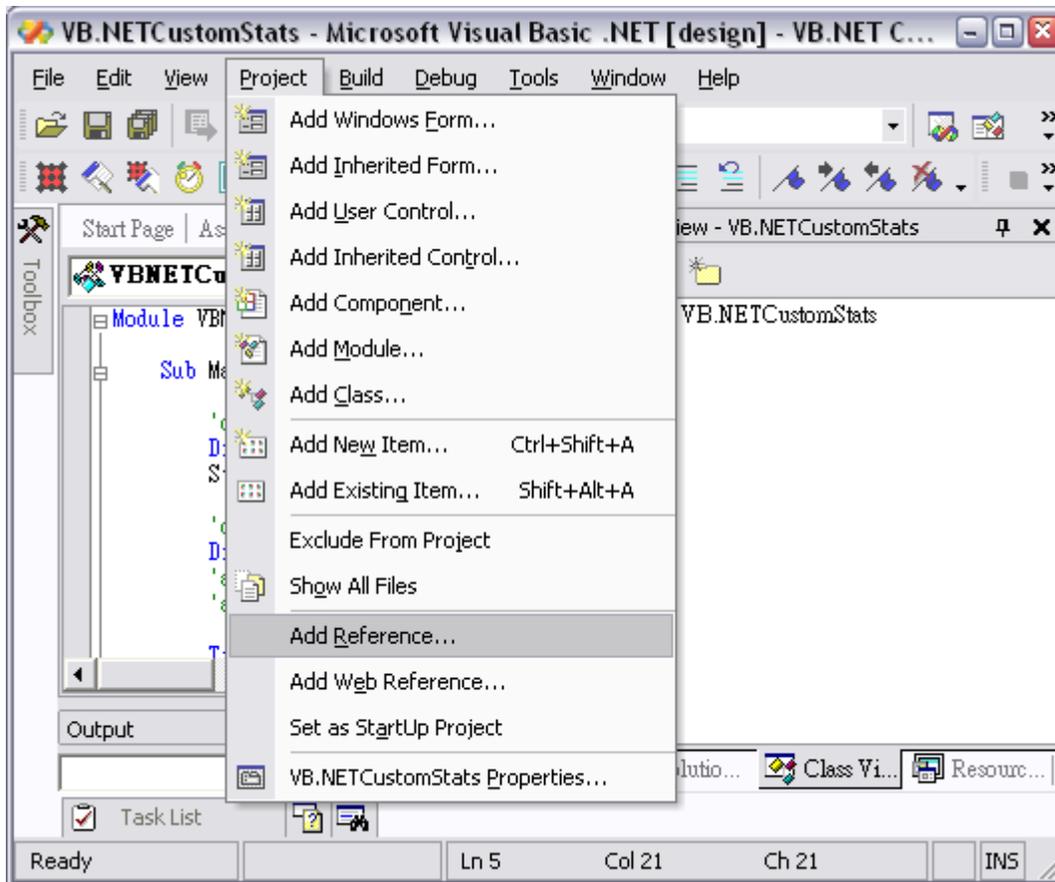
```
_SpreadsheetPtr pSpreadSheet = pApp->Spreadsheets-  
>New("MyDataset");
```

Microsoft® Visual Basic.NET

To add any *STATISTICA* library to a Visual Basic.NET project, follow these steps:

Step 1:

Select *Add Reference* from the *Project* menu:

**Step 2:**

On the *Add Reference* dialog, select the *COM* tab. Next, select the *STATISTICA* libraries that you want to include and click the *Select* button. Click *OK* when finished.

Note that the *STATISTICA Object Library* must be included to create a *STATISTICA* instance and work with its document types.

Step 3:

Now create an instance of *STATISTICA*. Add this text to `Sub Main` to create an instance of *STATISTICA* and a blank spreadsheet:

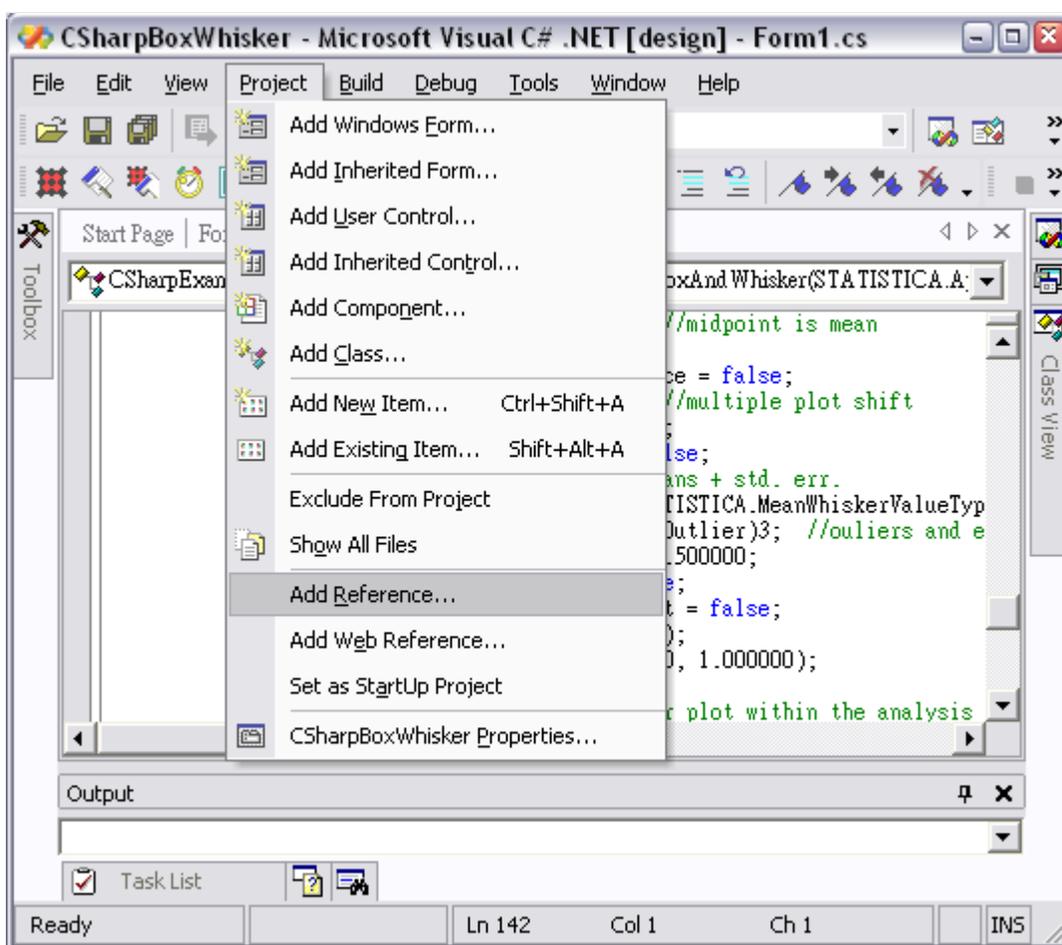
```
Dim Stat As New STATISTICA.Application
Stat.Visible = True
Dim spr As New STATISTICA.Spreadsheet
spr = Stat.Spreadsheets.New("MyDataset")
```

Microsoft ® C#

To add any STATISTICA library to a C# project, follow these steps:

Step 1:

Select *Add Reference* from the *Project* menu:



Step 2:

On the *Add Reference* dialog, select the *COM* tab. Next, select the *STATISTICA* libraries that you want to include and click the *Select* button. Click *OK* when finished.

Note that the *STATISTICA Object Library* must be included to create a *STATISTICA* instance and work with its document types.

Step 3:

Now create an instance of *STATISTICA*. Add this text to the main function to create an instance of *STATISTICA* and a blank spreadsheet:

```
//launch an instance of STATISTICA
STATISTICA.Application StatApp =
    new STATISTICA.Application();
StatApp.Visible = true;
/*open the spreadsheet--note that a cast is necessary
because Spreadsheets.Open() returns an Object*/
STATISTICA._Spreadsheet spr =
    (STATISTICA._Spreadsheet)StatApp.Spreadsheets.Open
        (FilePath,true);
```