IBM SPSS Statistics Programmability Extension Developer's Guide for Windows

Note

Before using this information and the product it supports, read the information in "Notices" on page 73.

Product Information

This edition applies to version 23, release 0, modification 0 of IBM SPSS Statistics and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 1. Overview

The *IBM SPSS Statistics Programmability Extension Developer's Guide* provides information about the tools available for working with the IBM® SPSS® Statistics Programmability Extension, which is available with the Core system.

Introduction to IBM SPSS Statistics

Before developing plug-ins and applications with the IBM SPSS Statistics Programmability Extension, it is useful to understand the basics of what IBM SPSS Statistics does and how it works.

Programmability is supported in three applications:
- IBM SPSS Statistics
- IBM SPSS Statistics Server
- IBM SPSS Statistics Batch Facility, a batch processing utility that is included with the IBM SPSS Statistics Server application.

All three applications provide a powerful statistical-analysis and data-management system. The statistics and data management facilities are controlled by the command syntax language.

IBM SPSS Statistics has a graphical user interface (GUI) of menus and dialog boxes that provides point and click access to the same features that are controlled by the command language; however the IBM SPSS Statistics Programmability Extension doesn't use the GUI.

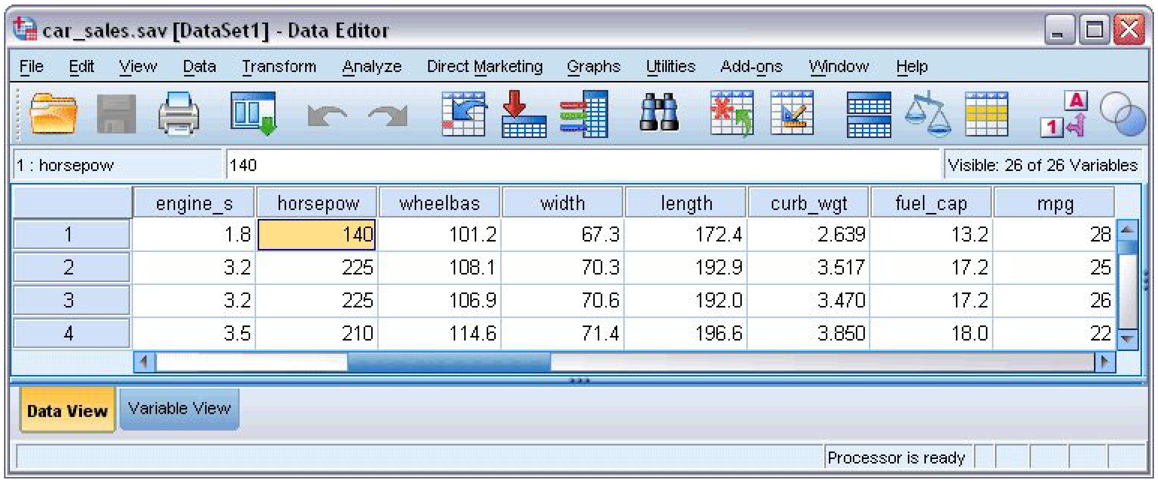
IBM SPSS Statistics Server is a distributed application. It has a client/server architecture. It distributes client requests for resource-intensive operations to powerful analytic server software. Requests are submitted by the IBM SPSS Statistics client and processed by IBM SPSS Statistics Server.

IBM SPSS Statistics Batch Facility is included with IBM SPSS Statistics Server. It is intended for automated data management and production of statistical output. Automated production provides the ability to submit commands without user intervention. The Batch Facility takes as its input a command syntax file with requests for data manipulation and statistics. It has no graphical user interface.

This section provides a quick overview of the basics of IBM SPSS Statistics. If you are not familiar with IBM SPSS Statistics, please refer to the *Command Syntax Reference*, which is installed with IBM SPSS Statistics and can be accessed from the Help menu. The Universals chapter contains information that is particularly useful for the reader interested in programmability. Also see the *Core System User's Guide*, which is on the manuals CD-ROM included with IBM SPSS Statistics.

Data Structure

IBM SPSS Statistics defines data as **variables** (similar to database fields) and **cases** (similar to database records). A variable contains information that you want to analyze—for example, an attribute like horsepower. A case is an observation—for example, a specific car.



car_sales.sav [DataSet1] - Data Editor

File Edit View Data Transform Analyze Direct Marketing Graphs Utilities Add-ons Window Help

1 : horsepow 140 Visible: 26 of 26 Variables

	engine_s	horsepow	wheelbas	width	length	curb_wgt	fuel_cap	mpg
1	1.8	140	101.2	67.3	172.4	2.639	13.2	28
2	3.2	225	108.1	70.3	192.9	3.517	17.2	25
3	3.2	225	106.9	70.6	192.0	3.470	17.2	26
4	3.5	210	114.6	71.4	196.6	3.850	18.0	22

Data View
Variable View
Processor is ready

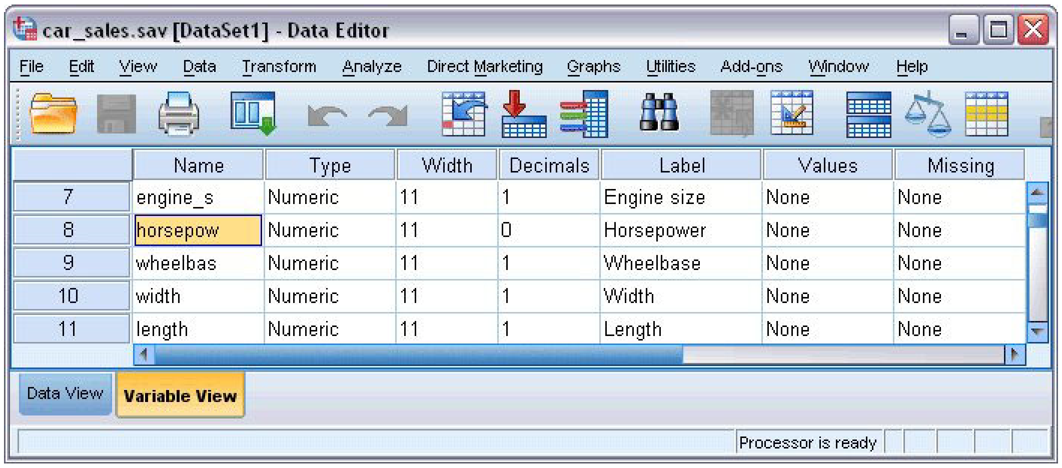
When a data file is open in IBM SPSS Statistics it becomes the **active dataset** and is displayed in the Data View tab of the Data Editor window, which is a spreadsheet-like system for entering and editing data. Variables are displayed in columns and cases are displayed in rows.

Data is saved in a proprietary format, the .sav file. It consists of the data and information about the data.

Metadata

Metadata are data about data, for example, information about the type, labels, and other characteristics. A **dictionary** contains some metadata at two levels—information about the data file and information about the variables.

The file-level dictionary information contains an optional file label, documents, and attributes that you define.



car_sales.sav [DataSet1] - Data Editor

File Edit View Data Transform Analyze Direct Marketing Graphs Utilities Add-ons Window Help

	Name	Type	Width	Decimals	Label	Values	Missing
7	engine_s	Numeric	11	1	Engine size	None	None
8	horsepow	Numeric	11	0	Horsepower	None	None
9	wheelbas	Numeric	11	1	Wheelbase	None	None
10	width	Numeric	11	1	Width	None	None
11	length	Numeric	11	1	Length	None	None

Data View
Variable View
Processor is ready

The variable-level dictionary information includes:

- Name
- Type
- Format
- Label
- Measurement level
- Missing values

- Value labels

You can also define your own variable metadata and assign attribute values to variables in the active dataset. Creating your own file-level and variable-level dictionary attributes can be quite useful in conjunction with Programmability because those attributes can then be used to control the flow of a command syntax job.

Syntax Commands

Syntax commands are used to indicate what type of data management and statistical analysis you want to perform. You use syntax commands to:

- Perform data transformations
- Run statistical procedures and produce output
- Create charts and other graphical output
- Display information about settings

For example if you want to calculate descriptive statistics for a variable, you could use the Descriptives statistical procedure. The command syntax could be:

```
DESCRIPTIVES
   VARIABLES=weight
   /STATISTICS=MEAN STDDEV MIN MAX.
```

Although syntax commands indicate what to do and what output to produce, it is difficult to use command syntax for **jobwise flow control**; for instance, to run different procedures for different variables based on each variable's metadata, or to determine which procedure to run next based on the output from the last procedure. This is where programmability can help. You can use programmability to branch or loop syntax commands based on:

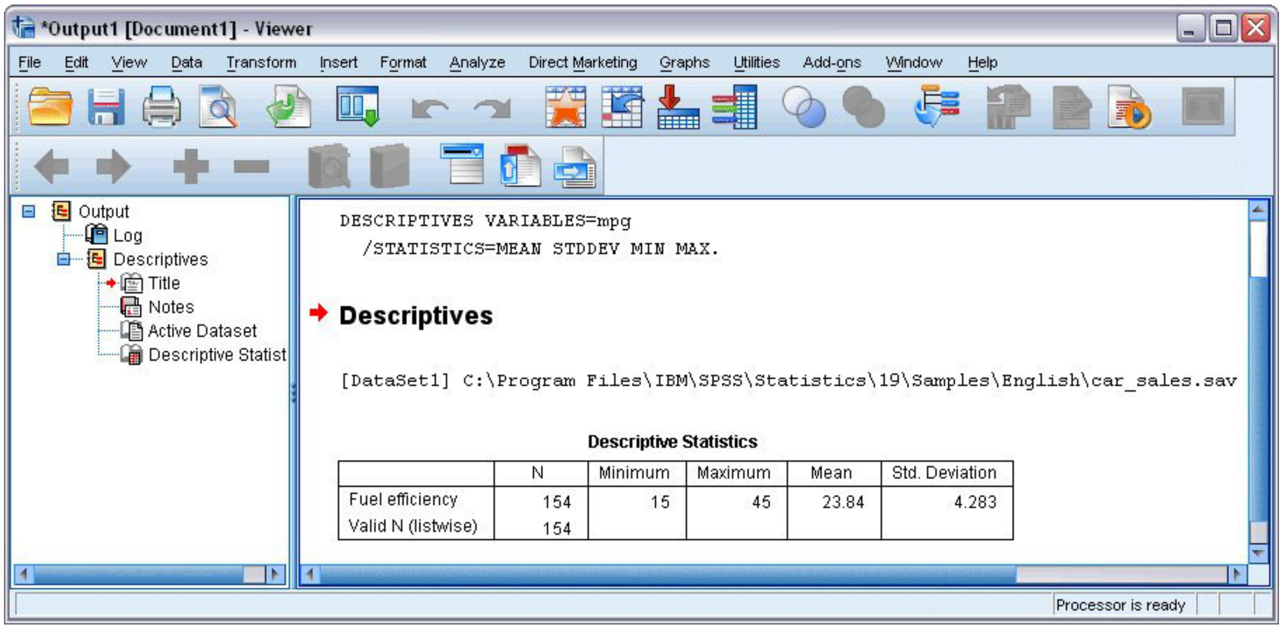
- The state of the active dataset's dictionary
- Values in the IBM SPSS Statistics output
- Case data from the active dataset
- Error level return codes from procedures

The code to control IBM SPSS Statistics is written in an external language, utilizing functions provided with an integration plug-in that are designed to interact with IBM SPSS Statistics.

Output

When you submit commands, IBM SPSS Statistics processes them and produces **output**. Output includes statistical results, tables, charts, trees, text, and other output.

For example, if you look at the Viewer window below you can see that the *Mean*, or average, fuel efficiency (mpg) of all cars in the active dataset is 23.84. This output was produced as a result of a `DESCRIPTIVES` request.


*Output1 [Document1] - Viewer
File Edit View Data Transform Insert Format Analyze Direct Marketing Graphs Utilities Add-ons Window Help
Output
Log
Descriptives
Title
Notes
Active Dataset
Descriptive Statist
Processor is ready

```
DESCRIPTIVES VARIABLES=mpg
  /STATISTICS=MEAN STDDEV MIN MAX.
```

Descriptives

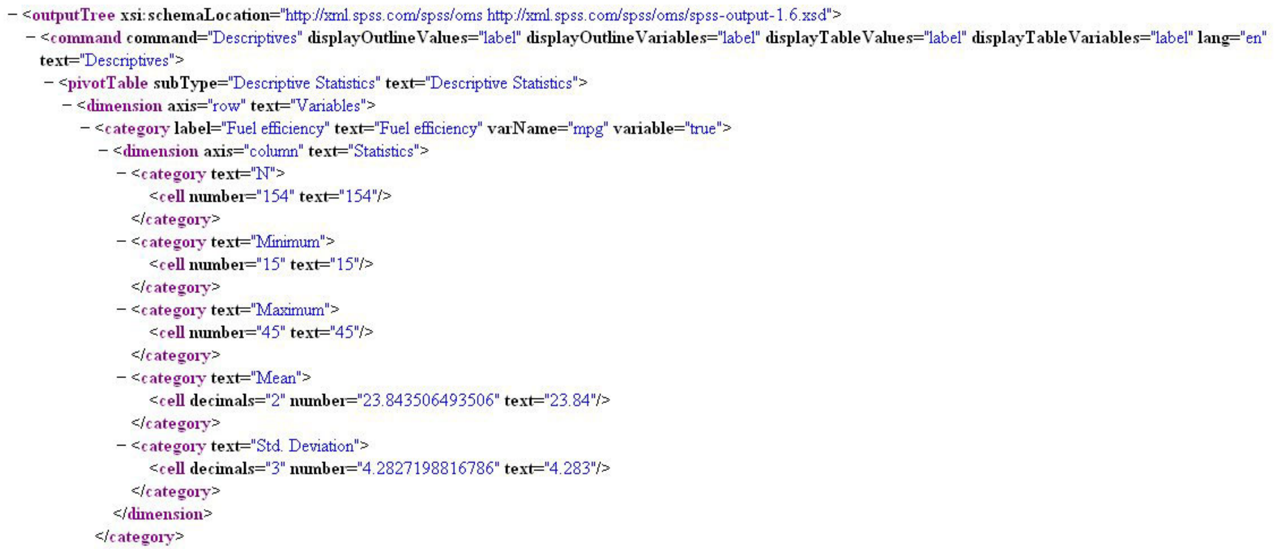
```
[DataSet1] C:\Program Files\IBM\SPSS\Statistics\19\Samples\English\car_sales.sav
```

Descriptive Statistics

	N	Minimum	Maximum	Mean	Std. Deviation
Fuel efficiency	154	15	45	23.84	4.283
Valid N (listwise)	154				

The **Output Management System** (OMS) provides the ability to automatically write selected categories of output to different destinations and in different formats. The **XML workspace** destination is designed for use with the IBM SPSS Statistics Programmability Extension, allowing values in the output to be accessed programmatically using XPath expressions.

For example, if you look at the XML produced by OMS for a `DESCRIPTIVES` request, you can see that it contains the same information as the Viewer output shown above. With the IBM SPSS Statistics Programmability Extension, this information is available through an API.



```
- <outputTree xsi:schemaLocation="http://xml.spss.com/spss/oms http://xml.spss.com/spss/oms/spss-output-1.6.xsd">
  - <command command="Descriptives" displayOutlineValues="label" displayOutlineVariables="label" displayTableValues="label" displayTableVariables="label" lang="en"
    text="Descriptives">
    - <pivotTable subType="Descriptive Statistics" text="Descriptive Statistics">
      - <dimension axis="row" text="Variables">
        - <category label="Fuel efficiency" text="Fuel efficiency" varName="mpg" variable="true">
          - <dimension axis="column" text="Statistics">
            - <category text="N">
                <cell number="154" text="154"/>
              </category>
            - <category text="Minimum">
                <cell number="15" text="15"/>
              </category>
            - <category text="Maximum">
                <cell number="45" text="45"/>
              </category>
            - <category text="Mean">
                <cell decimals="2" number="23.843506493506" text="23.84"/>
              </category>
            - <category text="Std. Deviation">
                <cell decimals="3" number="4.2827198816786" text="4.283"/>
              </category>
            </dimension>
          </category>
```

Basic Steps for Submitting IBM SPSS Statistics Commands

The basic steps to manipulate and analyze data with commands are:

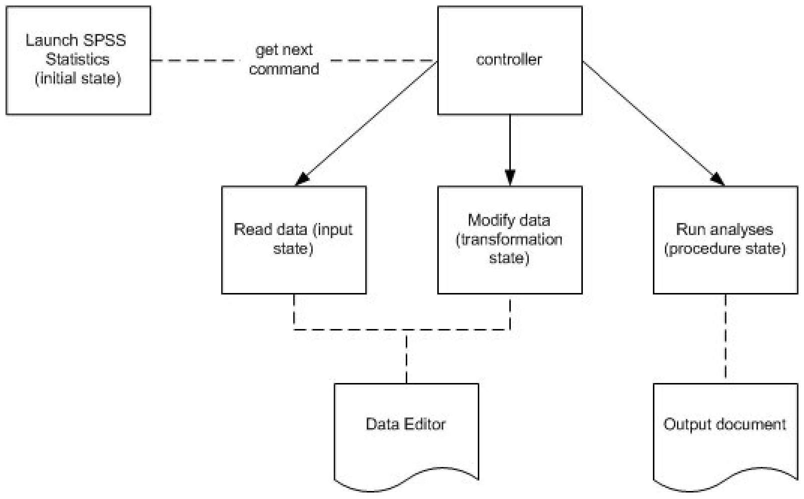
1. **Get data.** You can open a previously saved external IBM SPSS Statistics data file, read a spreadsheet, database, a text data file, and other file formats.
2. **Specify the commands and variables.** You can use syntax commands to transform data, calculate statistics, and to create charts and trees.

3. **Direct the output.** By default output appears in the Viewer window. You may want it there or you may choose to direct it to OMS.
4. **Submit the commands.** IBM SPSS Statistics processes the commands and produces output.
5. **Use the output.** Use the output to control the flow of additional syntax commands or as the final product.

How Commands Are Processed Normally

To process commands:

1. IBM SPSS Statistics gets a command from an input stream—commands from a file or the GUI, or a command typed at a prompt.
2. The command is evaluated. If it is valid for the current state, it is executed. If it is invalid for the current state, it is not executed.
3. Either way, IBM SPSS Statistics does something to the active data set or puts something into the output and gets the next command.


Launch SPSS Statistics (initial state)
get next command
controller
Read data (input state)
Modify data (transformation state)
Run analyses (procedure state)
Data Editor
Output document

With the IBM SPSS Statistics Programmability Extension, IBM SPSS Statistics breaks out of normal mode and responds to an external processor—that is when your code gets used.

Introduction to the Programmability Extension

The IBM SPSS Statistics Programmability Extension--included with the IBM SPSS Statistics Core system--provides generic support for extending IBM SPSS Statistics to work with external programming languages, each of which requires a separate integration plug-in. Integration with external languages is provided through the following means:

- **spssxd_p.dll and spssxd_p.dll.** In-process (spssxd_p.dll) and out-of-process (spssxd_p.dll) implementations of a C API, called the **XD API**, that allow an external processor to control IBM SPSS Statistics. The functions available in the API are detailed in Chapter 5, “Introduction to the XD API,” on page 53. Integration plug-ins interact with IBM SPSS Statistics through this library.
- A framework (installed with IBM SPSS Statistics) that enables IBM SPSS Statistics to execute external language code from within `BEGIN PROGRAM <language> - END PROGRAM` blocks in command syntax. Invocation of the processor for the external language is provided by a separate library that forms an optional part of an integration plug-in.

Programmability Integration Plug-in

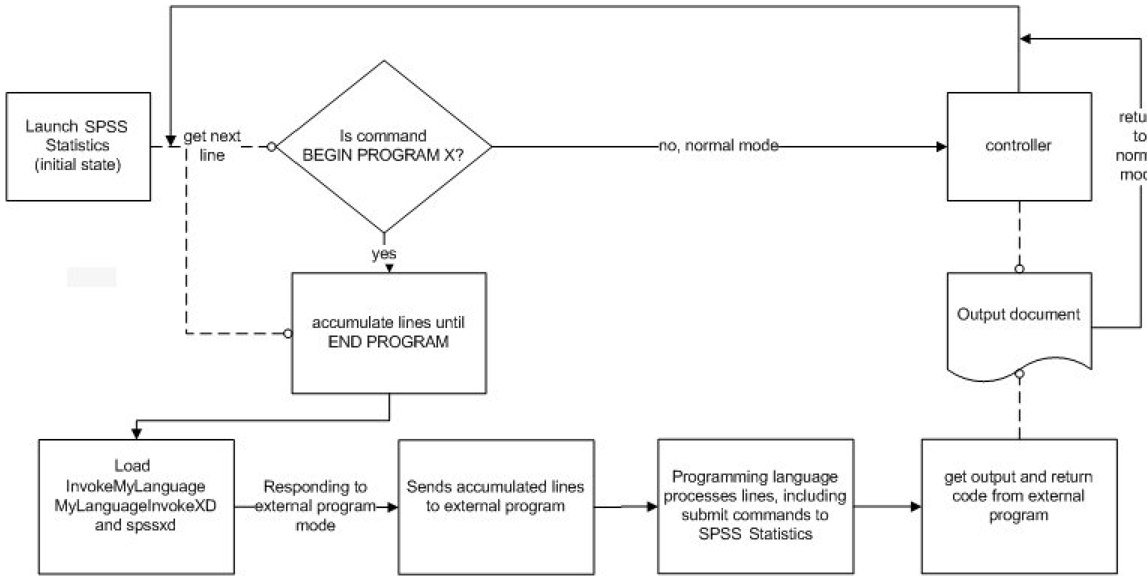
An integration plug-in contains code that can invoke and control IBM SPSS Statistics from an external processor through the XD API (and can optionally invoke an external processor from IBM SPSS Statistics) and support communication between the applications. A plug-in will also typically include a wrapper to the low level XD API functions, written in the language supported by the plug-in. Generically, a plug-in will include the following libraries:

- **MyLanguageInvokeXD.** Loads *spssxd_p.dll* or *spssxd_p.dll* and makes the XD API available in an external language (the API is implemented in C). This is a required piece of every plug-in.
- **InvokeMyLanguage.** An adapter to invoke an external processor from IBM SPSS Statistics and execute language statements (written in an external language) contained in a `BEGIN PROGRAM <language> - END PROGRAM` block. This is an optional component of a plug-in.

The IBM SPSS Statistics - Programmability SDK includes the source code for an example plug-in for Python that can be used as a model for creating your own plug-in. Language-specific plug-ins provided by IBM SPSS are available from the SPSS community at http://www.ibm.com/developerworks/spssdevcentral.

Operational Modes

The IBM SPSS Statistics Programmability Extension provides two modes for controlling IBM SPSS Statistics from an external language processor, both of which utilize *spssxd_p.dll* or *spssxd_p.dll* . In the more complex of the two, you invoke an external language processor from IBM SPSS Statistics through an *InvokeMyLanguage* library. The load sequence includes *MyLanguageInvokeXD* and *spssxd_p.dll* (or *spssxd_p.dll*), which then allows the processor to interact with IBM SPSS Statistics. This mode of operation is designed for end users who want to leverage the power of an external programming language from within a command syntax job. The following flowchart captures how IBM SPSS Statistics processes commands when responding to an external processor in this mode.


Launch SPSS Statistics (initial state)
get next line
Is command BEGIN PROGRAM X?
no, normal mode
controller
return to normal mode
yes
accumulate lines until END PROGRAM
Output document
Load InvokeMyLanguage MyLanguageInvokeXD and spssxd
Responding to external program mode
Sends accumulated lines to external program
Programming language processes lines, including submit commands to SPSS Statistics
get output and return code from external program

In a simpler mode of operation you invoke and control the IBM SPSS Statistics backend in a completely external manner through the use of *MyLanguageInvokeXD* and *spssxd_p.dll* (or *spssxd_p.dll*). This mode of operation is designed for applications developers who want to create applications that utilize IBM SPSS Statistics functionality, while leveraging the power of an external programming language to build the user interface and drive the program logic. For example, a .NET developer can use the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET to execute commands and capture appropriate output simply by calling functions provided with the plug-in.

Example

As an example of the kind of task suited for the IBM SPSS Statistics Programmability Extension, consider running a procedure for just those variables that have a particular property, as defined by the variable's metadata. Specifically, we'll instruct IBM SPSS Statistics to run the Descriptives procedure for the variables whose measurement level is scale. A scale variable is a numeric variable whose values represent ordered categories with a meaningful metric, so that distance comparisons between values are appropriate. Examples of scale variables include age in years and income in thousands of dollars.

Regardless of the operational mode used, the task is accomplished with code written in an external language, like Python or Visual Basic .NET, making use of functions in an integration plug-in that provide a wrapper to the low level XD API. Without tying ourselves to a particular language, a block of pseudo-code to accomplish this task might look like:

```
varcount=PlugInFunctionToGetVariableCount()
loop i to varcount:
   if PlugInFunctionToGetVariableMeasurementLevel(i)='scale'
      varname=PlugInFunctionToGetVariableName(i)
      append varname to list of scale variables (ScaleVarList)
PlugInFunctionToSubmitCommands
                ("DESCRIPTIVES VARIABLES= ", ScaleVarList, ".")
```

A comprehensive introduction to working with the example IBM SPSS Statistics - Integration Plug-in for Python provided with the IBM SPSS Statistics - Programmability SDK can be found in *Programming and Data Management for IBM SPSS Statistics* , available in PDF from the SPSS community at http://www.ibm.com/developerworks/spssdevcentral.

The Programmability Extension and Other IBM SPSS Statistics Programming Features

IBM SPSS Statistics includes other developer tools. The tools, and their relationship to the IBM SPSS Statistics Programmability Extension, are discussed in this section.

OLE Automation and Scripting. OLE Automation and the IBM SPSS Statistics Basic Script development environment makes it possible to write applications that incorporate, or extend, the functionality of IBM SPSS Statistics on Windows systems. OLE Automation allows an external application to access IBM SPSS Statistics output objects and manipulate them using methods and properties. Use OLE Automation when you need to integrate Viewer output into applications that support OLE automation, such as Microsoft PowerPoint, or when you need to control IBM SPSS Statistics from an application that supports Visual Basic, such as Microsoft Office or Visual Basic itself.

Macro. Program macros, which are blocks of syntax commands enclosed in `DEFINE` and `!ENDDEFINE` commands, can be used to parameterize repetitive tasks. Macros are named definitions. When IBM SPSS Statistics encounters a macro name, it expands the macro according to its definition. The IBM SPSS Statistics macro is a text substitution mechanism, not a scripting language. If you are working in Python, the `spss` module included in the IBM SPSS Statistics - Programmability SDK contains the function `SetMacroValue` which defines a macro that can be used just like a macro created with `DEFINE` and `!ENDDEFINE`. Likewise, the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET (available from the SPSS community at http://www.ibm.com/developerworks/spssdevcentral) includes the `SetMacroValue` method in the `Processor` class, available in the *SPSS.BackendAPI.Controller* assembly. We suggest that you use `SetMacroValue` instead of defining macros with `DEFINE` and `!ENDDEFINE`, or use the IBM SPSS Statistics - Programmability SDK to write your own macro definition function.

Matrix. Matrix programs, which are blocks of matrix algebra commands enclosed in `MATRIX` and `END MATRIX` commands, can be used to write customized statistical routines. With the IBM SPSS Statistics Programmability Extension you can retrieve case data from the active dataset, allowing you to perform custom calculations on your data or utilize the scientific programming libraries and packages available with external languages like Python or R.

Production Jobs. Production jobs allow you to run command syntax jobs in an automated fashion, producing output from regularly repeated, time-consuming statistical analyses. Consider using the IBM SPSS Statistics Programmability Extension and the Batch Facility, or the Batch Facility alone to meet your production needs.

Overview of Programmability SDK Contents

In addition to this Developer's Guide, the IBM SPSS Statistics - Programmability SDK includes binaries, source, projects, and other documentation to help you create plug-ins and applications for the IBM SPSS Statistics Programmability Extension. All file paths listed are relative to the directory where the IBM SPSS Statistics - Programmability SDK is installed.

Programmability Extension Files and Documentation

- For developers interested in building their own plug-in, the directory *\XD_API* includes the files-- *spssxd.h*, *spssxd.lib* , and *spssxd_p.lib* --needed to build an implementation of *MyLanguageInvokeXD*. You can choose to build an implementation that is either in-process (*spssxd.lib*) or out-of-process (*spssxd_p.lib*) with IBM SPSS Statistics. The in-process approach typically has better performance for executing large tasks, however, the out-of-process approach is recommended when using the nesting feature. Sample C++ code for loading *spssxd_p.dll* or *spssxd_p.dll* , and exercising the functions available in the API can be found in *\XD_API\in_process\examples* and *\XD_API\out_of_process\examples*.
- Documentation for the XD API is accessible from *\documentation\XDAPI\index.html*.
- The IBM SPSS Statistics Programmability Extension utilizes the configuration file *spssdxcfg.ini* when invoking an external language processor from IBM SPSS Statistics (an optional feature of an integration plug-in). An example of this file can be found in *\DX_API\config*. In addition, a simple example of invoking the Python language from within IBM SPSS Statistics can be found in *\DX_API\ MyInvokePython*.

IBM SPSS Statistics - Integration Plug-in for Python Source

The IBM SPSS Statistics - Programmability SDK includes the source code to build an example plug-in for Python consisting of an extension module that makes the XD API available from Python, a library for invoking Python from IBM SPSS Statistics, a Python module that provides a wrapper for the low level XD API, and other supporting files. The plug-in enables you to drive IBM SPSS Statistics from any separate Python process (such as a Python IDE or the Python interpreter) or to execute Python code contained within a `BEGIN PROGRAM-END PROGRAM` block in a command syntax job. The IBM SPSS Statistics - Programmability SDK includes the following items:

- **PyInvokeSpss.** A Python extension module which loads *spssxd_p.dll* and makes the XD API available from Python. Source files can be found in *\Statistics_Python\spss\src*.
- **InvokePython.** A Python-specific library that enables IBM SPSS Statistics to invoke Python. Source files and instructions for building this library can be found in *\DX_API\InvokePython*.
- **spssxdcfg.ini.** A configuration file which is used by Python to determine the location of *spssxd_p.dll* . A template can be found in *\Statistics_Python\spss*.

The main application files that constitute the wrapper for the low level XD API are available from *\Statistics_Python\spss*. They are:

- **__init__.py.** The Python Integration Package for IBM SPSS Statistics initialization code.
- **spss.py.** A Python module that wraps the low level XD API functions exposed to Python by *PyInvokeSpss*. It provides functions that invoke IBM SPSS Statistics and execute command syntax.
- **cursors.py.** A Python module that provides functions for getting case data, adding new variables, and appending cases to the active dataset.
- **dataStep.py.** A Python module that provides functions for creating and managing multiple concurrent datasets.

- **pivotTable.py.** A Python module that provides functions for creating custom pivot tables and text blocks.
- **preaction.py and postaction.py.** Python modules that provide functions for getting Python output into IBM SPSS Statistics.

IBM SPSS Statistics-Microsoft .NET Materials

The IBM SPSS Statistics - Programmability SDK includes sample projects and code examples for building .NET applications for the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET. The plug-in is distributed separately and available from the SPSS community at http://www.ibm.com/developerworks/spssdevcentral.

- **Visual Basic.** A sample Visual Basic project is available from \Statistics_NET\vbexample, and a template for creating Visual Basic classes is available from \Statistics_NET\classtemplates\VB.
- **C#.** A sample C# project is available from \Statistics_NET\csharpexample.
- **.NET Assemblies.** The .NET assemblies that enable a .NET application to interact with IBM SPSS Statistics are included in \Statistics_NET\libs.

Chapter 2. Creating .NET Applications for IBM SPSS Statistics

Getting Started

The IBM SPSS Statistics - Integration Plug-in for Microsoft .NET enables an application developer to create .NET applications that can invoke and control the IBM SPSS Statistics backend through the XD API. The plug-in includes two .NET assemblies, both of which are added to the Global Assembly Cache (GAC) during installation of the plug-in.

- **SPSS.BackendAPI.dll.** A .NET assembly which loads *spssxd_p.dll* and makes the XD API available from .NET.
- **SPSS.BackendAPI.Controller.dll.** A .NET assembly that wraps the low level XD API functions exposed to .NET by *SPSS.BackendAPI.dll*. It provides functions that invoke IBM SPSS Statistics, execute syntax commands, provide access to the data, and allow for the creation of custom pivot tables. This is the library intended for use by application developers.

Note: The IBM SPSS Statistics - Integration Plug-in for Microsoft .NET is strictly for use by an external .NET application that needs to access the IBM SPSS Statistics backend processor. It is not for use from within IBM SPSS Statistics.

You invoke the IBM SPSS Statistics backend with the `Processor` class from the *SPSS.BackendAPI.Controller* library. The constructor for this class is overloaded and provides two approaches for enabling the class instance to locate the IBM SPSS Statistics backend library *spssxd_p.dll* . *spssxd_p.dll* is located in the IBM SPSS Statistics install directory and should not be moved.

- **Processor().** Creates an instance of the `Processor` class and uses the location of IBM SPSS Statistics 23 stored in the Windows registry.
- **Processor(SpssXdPath).** Creates an instance of the `Processor` class and looks for *spssxd_p.dll* in the location specified by *SpssXdPath*.

To help get you started with writing applications for the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET, a sample Visual Basic Class (shown below) is provided that gives an example of creating an instance of the `Processor` class. How you access the sample class and use the plug-in in your development environment depends on whether you have Visual Studio.

Visual Studio users. The sample class is installed as the Visual Studio item template *SPSS_Statistics-.NET_Integration_Plug-In230Class* (accessible from the **Add New Item** choice on the Visual Studio **Project** menu). Adding the item template to a project creates references to the copies of *SPSS.BackendAPI.dll* and *SPSS.BackendAPI.Controller.dll* stored in the GAC so there is no need to manually add references to these libraries during development.

Users without Visual Studio. The sample class is available for download from *Example VB.NET Class* on the SPSS Community (http://www.ibm.com/developerworks/spssdevcentral). To use the sample class you'll need to manually create references to *SPSS.BackendAPI.dll* and *SPSS.BackendAPI.Controller.dll*. Both files can be found in the *NET* directory under the directory where IBM SPSS Statistics 23 is installed--for example, *C:\Program Files\IBM\SPSS\Statistics\23\NET*.

```
Imports SPSS.BackendAPI.Controller
Imports Microsoft.Win32

Public Class SPSS_Statistics_
    Sub PlugIn()
        Dim SPSS_Processor As Processor
        Dim SpssXdPath As String = "" 'Optional: Add your SPSS Statistics location
        If Not SpssXdPath Is String.Empty Then
            SPSS_Processor = New Processor(SpssXdPath)
        Else
            SPSS_Processor = New Processor()
        End If
        'Todo: Add your code here

        SPSS_Processor.StopSPSS()
    End Sub
End Class
```

The sample class includes the following items that should always be present in an application that uses the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET:

- An `Imports SPSS.BackendAPI.Controller` statement. It is up to you to decide how to deploy *SPSS.BackendAPI.Controller.dll* and *SPSS.BackendAPI.dll* with your application. See the topic "Deploying Your Application" on page 37 for more information.
- A statement to instantiate the `Processor` class, either with or without an argument specifying the location of *spssxd_p.dll* .

  In addition to instantiating the `Processor` class, you must start the IBM SPSS Statistics backend before utilizing most of the available API's. You can do this directly by calling the `StartSPSS` method, as in `SPSS_Processor.StartSPSS()`. Note that the IBM SPSS Statistics backend is automatically started when you call the `Submit` method, so an explicit call to `StartSPSS` may not be necessary.
- A call to the `StopSPSS` method of the `Processor` instance. This method should be called before exiting your application to ensure that all temporary files are removed.

Example

The following is a simple example of using the sample class to create a dataset in IBM SPSS Statistics, compute descriptive statistics and generate output.

```
Imports SPSS.BackendAPI.Controller
Imports Microsoft.Win32

Public Class SPSS_Statistics_
    Sub PlugIn()
        Dim SPSS_Processor As Processor
        SPSS_Processor = New Processor()

        SPSS_Processor.StartSPSS("c:\temp\test.txt")
        Dim cmdLines As System.Array = New String() _
        {"DATA LIST FREE /salary (F).", _
         "BEGIN DATA", _
         "21450", _
         "30000", _
         "57000", _
         "END DATA.", _
         "DESCRIPTIVES salary."}
        SPSS_Processor.Submit(cmdLines)

        SPSS_Processor.StopSPSS()
    End Sub
End Class
```

- The `StartSPSS` method is called with an argument specifying the path to a file where any output is written. In this example, output will be generated by the `DESCRIPTIVES` procedure and written to the file *c:\temp\test.txt*.
- A string array specifies IBM SPSS Statistics command syntax that creates a dataset and runs the `DESCRIPTIVES` procedure. The command syntax is submitted to IBM SPSS Statistics using the `Submit` method. See the topic "Running IBM SPSS Statistics Commands" on page 13 for more information.

You can instantiate the sample class and run the example code from the following module:

```
Module Module1
    Sub Main()
        Dim p As New SPSS_Statistics_
        p.PlugIn()
    End Sub
End Module
```

Documentation

Documentation for the *SPSS.BackendAPI.Controller* library can be found in the *NET* directory under the directory where IBM SPSS Statistics 23 is installed--for example, *C:\Program Files\IBM\SPSS\Statistics\23\NET*. For Visual Studio users, the documentation is integrated into the Visual Studio IDE, and available from the help contents and from the Dynamic Help window. The documentation includes simple code examples for each of the available methods.

Running IBM SPSS Statistics Commands

The `Submit` method from the `Processor` class is used to submit syntax commands to IBM SPSS Statistics for processing. It takes a string that resolves to one or more complete syntax commands, or an array of strings that resolves to one or more complete syntax commands. By default, output from syntax commands is written to the standard output stream. You can direct output to an in-memory workspace where it is stored as XML and can then be retrieved using XPath expressions. See the topic "Retrieving Output from Syntax Commands" on page 24 for more information.

Submitting a Single Command

You submit a single command to IBM SPSS Statistics by providing a string representation of the command as shown in this example. When submitting a single command in this manner the period (.) at the end of the command is optional.

```
Processor.Submit( _
"GET FILE='c:/data/Employee data.sav'.")
Processor.Submit("DESCRIPTIVES SALARY.")
```

- The `Submit` method is called twice; first to submit a `GET` command and then to submit a `DESCRIPTIVES` command.

Submitting Commands Using an Array

You can submit multiple commands as an array of strings where each array element is a string representation of a syntax command. The string for each command must be terminated with a period (.) as shown in this example.

```
Processor.Submit( _
"GET FILE='c:/data/Employee data.sav'.")
Dim cmdLines As System.Array = New String() _
    {"DESCRIPTIVES SALARY SALBEGIN.","FREQUENCIES EDUC JOBCAT."}
Processor.Submit(cmdLines)
```

- The `Submit` method is called with an array that specifies a `DESCRIPTIVES` and a `FREQUENCIES` command.

You can also use the elements of an array to represent parts of a command so that a single array specifies one or more complete syntax commands. When submitting multiple commands in this manner, each command must be terminated with a period (.) as shown in this example.

```
Processor.Submit( _
"GET FILE='c:/data/Employee data.sav'.")
Dim cmdLines As System.Array = New String() _
    {"OMS /SELECT TABLES ", _
     "/IF COMMANDS = ['Descriptives' 'Frequencies'] ", _
     "/DESTINATION FORMAT = HTML ", _
     "IMAGES = NO OUTFILE = 'c:/temp/stats.html'.", _
     "DESCRIPTIVES SALARY SALBEGIN.", _
     "FREQUENCIES EDUC JOBCAT.", _
     "OMSEND."}
Processor.Submit(cmdLines)
```

- The `Submit` method is called with an array that specifies an `OMS` command followed by a `DESCRIPTIVES` command, a `FREQUENCIES` command, and an `OMSEND` command. The first four elements of the array are used to specify the `OMS` command.

Submitting Multiple Commands In a Single String

You can specify multiple commands in a single string using the character sequence `"\n"` as a delimiter as shown in this example.

```
Processor.Submit( _
"GET FILE='c:/data/Employee data.sav'.")
Processor.Submit("DESCRIPTIVES SALARY.\nFREQUENCIES EDUC.")
```

- A single string is used to submit both a `FREQUENCIES` command and a `DESCRIPTIVES` command. IBM SPSS Statistics interprets `"\n"` as a newline character.

Displaying Command Syntax Generated by the Submit Method

For debugging purposes, it is convenient to see the completed syntax passed to IBM SPSS Statistics by any calls to the `Submit` method. This is enabled through command syntax with `SET PRINTBACK ON MPRINT ON`.

```
Processor.Submit("SET PRINTBACK ON MPRINT ON.\n" +  _
"GET FILE='c:/data/Employee data.sav'.")
Dim varName As String
varName = Processor.GetVariableName(1)
Processor.Submit("FREQUENCIES /VARIABLES=" + varName + ".")
```

The generated command syntax is written to the standard output stream and shows the completed `FREQUENCIES` command as well as the `GET` command. In the present example the variable with index value 1 in the dataset has the name *gender*.

```
4 M>  GET FILE='c:/data/Employee data.sav'.
6 M>  FREQUENCIES /VARIABLES=gender.
```

Retrieving Dictionary Information

The `Processor` class provides a number of methods for retrieving dictionary information from the active dataset. Variables are specified by their position in the dataset, starting with 0 for the first variable in file order. This is referred to as the **index value** of the variable. The following information is available:

- **GetDataFileAttributeNames().** Names of any datafile attributes for the active dataset.
- **GetDataFileAttributes(attrName).** The attribute values for the specified datafile attribute.
- **GetMultiResponseSet(mrsetName).** The details of the specified multiple response set.
- **GetMultiResponseSetNames().** The names of any multiple response sets for the active dataset.
- **GetSplitVariableNames().** Returns the names of the split variables, if any, in the active dataset.
- **GetVariableAttributeNames(index).** Names of any custom variable attributes for the specified variable.
- **GetVariableAttributes(index,attrName).** The attribute values for a specified attribute of a specified variable.
- **GetVariableCount().** The number of variables in the active dataset.
- **GetVariableCValueLabel(index).** The value labels for the specified string variable.
- **GetVariableFormat(index).** The display format for the specified variable; for example, *F8.2*.
- **GetVariableLabel(index).** The variable label for the specified variable.
- **GetVariableMeasurementLevel(index).** The measurement level for the specified variable: `"nominal"`, `"ordinal"`, `"scale"`, or `"unknown"`.
- **GetVariableMissingValues(index).** Any user-missing values for the specified variable.
- **GetVariableName(index).** The variable name for the specified variable.
- **GetVariableNValueLabel(index).** The value labels for the specified numeric variable.
- **GetVariableType(index).** The variable type (numeric or string) for the specified variable.
- **WeightVariable.** Property that returns the name of the weight variable, if any.

Example

Consider the common scenario of running a particular block of command syntax only if a specific variable exists in the dataset. For example, you are processing many datasets containing employee records and want to split them by gender--if a gender variable exists--to obtain separate statistics for the two gender groups. We will assume that if a gender variable exists, it has the name *gender*, although it may be spelled in upper case or mixed case. The following sample code illustrates the approach:

```
Dim name As String
Processor.Submit( _
"GET FILE='c:/data/Employee data.sav'.")
For i As Integer = 0 To Processor.GetVariableCount() - 1
     name = Processor.GetVariableName(i)
     If LCase(name) = "gender" Then
          Dim cmdLines As System.Array = New String() _
               {"SORT CASES BY " + name + ".", _
                "SPLIT FILE LAYERED BY " + name + "."}
          Processor.Submit(cmdLines)
          Exit For
     End If
Next i
```

Working With Case Data in the Active Dataset

The IBM SPSS Statistics - Integration Plug-in for Microsoft .NET provides the ability to read case data from the active dataset, create new variables in the active dataset, and append new cases to the active dataset. Three classes are provided: the `DataCursor` class allows you to read cases from the active dataset, the `DataCursorWrite` class allows you to add new variables (and their case values) to the active dataset, and the `DataCursorAppend` class allows you to append new cases to the active dataset.

The following rules apply to the use of data cursors:

- You cannot use the `Submit` method from the `Processor` class while a data cursor is open. You must close the cursor first using the `Close` method. In particular, if you need to save changes made to the active dataset to an external file, then use the `Submit` method to submit a `SAVE` command after closing the cursor.
- Only one data cursor can be open at any point in an application. To define a new data cursor, you must first close the previous one.

Note: For users of the 14.0.2 version of the plug-in who are upgrading to the 16.0.1 version, the `GetDataCursorInstance` method from the `Processor` class has been deprecated and replaced with the `GetReadCursorInstance` method from the `Processor` class.

Reading Case Data

To retrieve case data, you first create an instance of the `DataCursor` class using the `GetReadCursorInstance` method from the `Processor` class. To specify that case data are to be retrieved for all variables in the active dataset, use the form of the method without an argument, as in `GetReadCursorInstance()`.

You can also specify a selected set of variables to retrieve. Variables are specified by their position in the dataset, starting with 0 for the first variable in file order. For example, to create an instance of the `DataCursor` class for retrieving case data for just the 5th and 3rd variables in file order, use `GetReadCursorInstance(indexSet)`, where *indexSet* is a 1-dimension array with the elements 5 and 3.

The following rules apply to retrieved values:

- String values are right-padded to the defined width of the string variable.
- System-missing values are always returned as a value of *Nothing*. By default, user-missing values are returned as a value of *Nothing*. You can specify that user-missing values be treated as valid with the `UserMissingInclude` property from the `DataCursor` class.
- By default, data retrieved from a variable representing a date, or a date and a time, is given as the number of seconds from October 14, 1582. You can specify a set of IBM SPSS Statistics variables with date or datetime formats to convert to `Date` data type values when reading data from IBM SPSS Statistics. See the example on handling datetime values.
- The `DataCursor` class honors case filters specified with the `FILTER` or `USE` commands.

Retrieving Cases Sequentially

You can retrieve cases one at a time in sequential order using the GetRow method from the DataCursor class. For example:

```
Processor.Submit("GET FILE='c:/data/demo.sav'.")
Dim cur as DataCursor = Processor.GetReadCursorInstance()
Dim result as System.Array
'First case
result = cur.GetRow()
'Second case
result = cur.GetRow()
cur.Close()
```

- Each call to GetRow retrieves the next case in the active dataset. Calling GetRow after the last case has been read returns a value of *Nothing*.
- The case data is returned as a 1-dimension array where each element corresponds to the case data for a specific variable. When you invoke the DataCursor class with GetReadCursorInstance(), as in this example, the elements correspond to the variables in file order.

Retrieving All Cases

You can retrieve all cases at once using the GetAllRows method from the DataCursor class. For example:

```
Processor.Submit("GET FILE='c:/data/demo.sav'.")
Dim cur as DataCursor = Processor.GetReadCursorInstance()
Dim result as System.Array
result = cur.GetAllRows()
cur.Close()
```

- The case data is returned as a 2-dimension array where the first dimension refers to the case number (starting with 0 for the first case) and the second dimension indexes the variables in the active dataset. For example, *result(n,m)* is the value of the mth variable for the nth case. When you invoke the DataCursor class with GetReadCursorInstance(), as in this example, the elements in the second dimension correspond to the variables in file order.

Missing Data

By default, missing values (user and system) are returned as *Nothing*.

```
Dim cmdLines As System.Array = New String() _
{"DATA LIST LIST (',') /numVar (f) stringVar (a4).", _
"BEGIN DATA", _
"1,a", _
",b", _
"3,,", _
"9,d", _
"END DATA.", _
"MISSING VALUES numVar (9) stringVar (' ')."}
Processor.Submit(cmdLines)
Dim cur As DataCursor = Processor.GetReadCursorInstance()
Dim result as System.Array
result = cur.GetAllRows()
cur.Close()
```

The values of *result* are:

```
1        a
Nothing  b
3        Nothing
Nothing  d
```

You can specify that user-missing values be treated as valid data by setting the *UserMissingInclude* property of the DataCursor instance to *True*, as shown in the following reworking of the previous example.

```
Dim cmdLines As System.Array = New String() _
{"DATA LIST LIST (',') /numVar (f) stringVar (a4).", _
"BEGIN DATA", _
"1,a", _
",b", _
"3,,", _
"9,d", _
"END DATA.", _
"MISSING VALUES numVar (9) stringVar (' ')."}
Processor.Submit(cmdLines)
```

```
Dim cur As DataCursor = Processor.GetReadCursorInstance()
cur.UserMissingInclude = True
Dim result as System.Array
result = cur.GetAllRows()
cur.Close()
```

The values of *result* are:

```
1       a
Nothing  b
3
9       d
```

Handling Data with Splits

When reading datasets in which split file processing is in effect, you'll need to be aware of the behavior at a split boundary. Detecting split changes is necessary when you're creating custom pivot tables from data with splits and want separate results displayed for each split group (using the `SplitChange` method from the `DataCursor` class). The `IsEndSplit` method, from the `DataCursor` class, allows you to detect split changes when reading from datasets that have splits.

```
Dim cmdLines As System.Array = New String() _
{"DATA LIST FREE /salary (F) jobcat (F).", _
 "BEGIN DATA", _
 "21450 1", _
 "45000 1", _
 "30000 2", _
 "30750 2", _
 "103750 3", _
 "72500 3", _
 "57000 3", _
 "END DATA.", _
 "SPLIT FILE BY jobcat."}
Processor.Submit(cmdLines)
Dim cur As DataCursor = Processor.GetReadCursorInstance()
cur.UserMissingInclude = False
For i As Integer = 1 To cur.CaseCount()
    cur.GetRow()
    If cur.IsEndSplit() Then
        Console.WriteLine("A new split begins at case: {0}", i)
        cur.GetRow()
    End If
Next
cur.Close()
```

- `cur.IsEndSplit()` returns a Boolean—*true* if a split boundary has been crossed, and *false* otherwise. For the sample dataset used in this example, split boundaries are crossed when reading the 3rd and 5th cases.
- The value returned from the `GetRow` method is *Nothing* at a split boundary. In the current example, this means that *Nothing* is returned when attempting to read the 3rd and 5th cases. Once a split has been detected, you call `GetRow` again to retrieve the first case of the next split group, as shown in this example.
- Although not shown in this example, `IsEndSplit` also returns *true* when the end of the dataset has been reached. This scenario would occur if you replace the `For` loop with a `While True` loop that continues reading until the end of the dataset is detected. Although a split boundary and the end of the dataset both result in a return value of *true* from `IsEndSplit`, the end of the dataset is identified by a return value of *Nothing* from a subsequent call to `GetRow`.

Handling IBM SPSS Statistics Datetime Values

When retrieving values of IBM SPSS Statistics variables with date or datetime formats, you'll most likely want to convert the values to `Date` data type values. By default, date or datetime variables are not converted and are simply returned in the internal representation used by IBM SPSS Statistics (floating point numbers representing some number of seconds and fractional seconds from an initial date and time). To convert variables with date or datetime formats to `Date` data type values, you use the `GetReadCursorInstance(indexSet,dateindexSet)` form of the `GetReadCursorInstance` method.

```
 Dim cmdLines As System.Array = New String() _
{"DATA LIST FREE /case (F) value (DATE10).", _
 "BEGIN DATA", _
 "1 28-OCT-1990", _
 "END DATA."}
Processor.Submit(cmdLines)
Dim indexSet() As Integer = {0, 1}
Dim dateindexSet() As Integer = {1}
Dim cur As DataCursor = Processor.GetReadCursorInstance(indexSet, dateindexSet)
Dim result As System.Array
result = cur.GetRow()
cur.Close()
```

- The first argument to `GetReadCursorInstance` specifies the set of variables to retrieve and the second argument specifies the set of variables to convert to `Date` data type values. The values specified for the second argument must be a subset of those for the first argument. Variables are specified by their position in the dataset, starting with 0 for the first variable in file order.

The value of *result* is:

```
1       10/28/1990 12:00:00 AM
```

Creating New Variables in the Active Dataset

To add new variables along with their case values to the active dataset, you first create an instance of the `DataCursorWrite` class using the `GetWriteCursorInstance` method from the `Processor` class.

- All of the methods available with the `DataCursor` class are also available with the `DataCursorWrite` class.
- When adding new variables, the `CommitDictionary` method must be called after the statements defining the new variables and prior to setting case values for those variables. You cannot add new variables to an empty dataset.
- When setting case values for new variables, the `CommitCase` method must be called for each case that is modified. The `GetRow` method is used to advance the record pointer by one case, or you can use the `GetRows` method to advance the record pointer by a specified number of cases.
- Changes to the active dataset do not take effect until the cursor is closed.
- Write mode supports multiple data passes and allows you to add new variables on each pass. In the case of multiple data passes where you need to add variables on a data pass other than the first, you must call the `AllocNewVarsBuffer` method to allocate the buffer size for the new variables. When used, `AllocNewVarsBuffer` must be called before reading any data with `GetRow`, `GetRows`, or `GetAllRows`.
- The `SetVariableNameAndType` and `SetOneVariableNameAndType` methods, from the `DataCursorWrite` class, are used to add new variables to the active dataset. The `SetValueChar` and `SetValueNumeric` methods are used to set case values.

Example

In this example we create a new string variable and a new numeric variable and populate their case values for the first and third cases in the active dataset. A sample dataset is first created.

```
Dim cmdLines As System.Array = New String() _
{"DATA LIST FREE /case (A5).", _
 "BEGIN DATA", _
 "case1", _
 "case2", _
 "case3", _
 "END DATA."}
Processor.Submit(cmdLines)
Dim cur As DataCursorWrite = Processor.GetWriteCursorInstance()
Dim varName As String() = {"numvar", "strvar"}
Dim varType As Integer() = {0, 1}
Dim varLabel As String() = {"Sample numeric variable", _
                            "Sample string variable"}
cur.SetVariableNameAndType(varName, varType)
cur.SetVariableLabel(varName(0), varLabel(0))
cur.SetVariableLabel(varName(1), varLabel(1))
cur.CommitDictionary()
cur.GetRow()
cur.SetValueNumeric(varName(0), 1.0)
cur.SetValueChar(varName(1), "a")
cur.CommitCase()
cur.GetRows(2)
cur.SetValueNumeric(varName(0), 3.0)
cur.SetValueChar(varName(1), "c")
cur.CommitCase()
cur.Close()
```

- The first argument to the SetVariableNameAndType method is an array of strings that specifies the name of each new variable. The second argument is an array of integers specifying the variable type of each variable. Numeric variables are specified by a value of 0 for the variable type. String variables are specified with a type equal to the defined length of the string (maximum of 32767). In this example, we create a numeric variable named *numvar* and a string variable of length 1 named *strvar*.
- After calling SetVariableNameAndType you have the option of specifying variable properties (in addition to the variable type) such as the measurement level, variable label, and missing values. In this example variable labels are specified using the SetVariableLabel method.
- Specifications for new variables must be committed to the cursor's dictionary before case values can be set. This is accomplished by calling the CommitDictionary method, which takes no arguments. The active dataset's dictionary is updated when the cursor is closed.
- To set case values, you first position the record pointer to the desired case using the GetRow or GetRows method. GetRow advances the record pointer by one case and GetRows advances it by a specified number of cases. In this example we set case values for the first and third cases. *Note*: To set the value for the first case in the dataset you must call GetRow as shown in this example.
- Case values are set using the SetValueNumeric method for numeric variables and the SetValueChar method for string variables. For both methods, the first argument is the variable name and the second argument is the value for the current case. A numeric variable whose value is not specified, or specified as *Nothing*, is set to the system-missing value. A string variable whose value is not specified, or specified as *Nothing*, will have a blank value. When setting values of IBM SPSS Statistics date or date/time variables, you can specify a Date data type as the value for the SetValueNumeric method. If you specify an integer value, it will be interpreted as the number of seconds from October 14, 1582.
- The CommitCase method must be called to commit the values for each modified case. Changes to the active dataset take effect when the cursor is closed.

Note: To save the modified active dataset to an external file, use the Submit method (following the Close method) to submit a SAVE command, as in:

```
Processor.Submit("SAVE OUTFILE='c:/data/mydata.sav'.")
```

Example: Multiple Data Passes

Sometimes more than one pass of the data is required, as in the following example involving two data passes. The first data pass is used to read the data and compute a summary statistic. The second data pass is used to add a summary variable to the active dataset.

```
Dim cmdLines As System.Array = New String() _
{"DATA LIST FREE /var (F).", _
 "BEGIN DATA", _
 "57000", _
 "40200", _
 "21450", _
 "21900", _
 "END DATA."}
Processor.Submit(cmdLines)
Dim cur As DataCursorWrite = Processor.GetWriteCursorInstance()
Dim row As System.Array
Dim total As Integer = 0
cur.AllocNewVariablesBuffer(8)
For i As Integer = 1 To cur.CaseCount()
    row = cur.GetRow()
    total = total + row(0)
Next
Dim meanVal As Double = total / cur.CaseCount()
cur.Reset()
cur.SetOneVariableNameAndType("mean", 0)
cur.CommitDictionary()
For i As Integer = 1 To cur.CaseCount()
    row = cur.GetRow()
    cur.SetValueNumeric("mean", meanVal)
    cur.CommitCase()
Next
cur.Close()
```

- Because we'll be adding a new variable on the second data pass, the `AllocNewVarsBuffer` method is called to allocate the required space. In the current example we're creating a single numeric variable, which requires 8 bytes.
- The first `For` loop is used to read the data and total the case values.
- After the data pass, the `Reset` method must be called prior to defining new variables.
- The `SetOneVariableNameAndType` method is used to add a single new variable. The first argument is the variable name and the second argument is the variable type. In this example, we create a numeric variable named *mean*. The `CommitDictionary` method is called to commit this variable to the cursor.
- The second data pass (second `For` loop) is used to set the case values of the new variable.

Appending New Cases

To append new cases to the active dataset, you first create an instance of the `DataCursorAppend` class using the `GetAppendCursorInstance` method from the `Processor` class. The `DataCursorAppend` class cannot be used to add new variables or read case data from the active dataset. A dataset must contain at least one variable in order to append cases to it, but it need not contain any cases.

- The `CommitCase` method must be called for each case that is added.
- The `EndChanges` method must be called before the cursor is closed.
- Changes to the active dataset do not take effect until the cursor is closed.
- The `SetValueChar` and `SetValueNumeric` methods are used to set variable values for new cases.

Example

In this example two new cases are appended to the active dataset.

```
Dim cmdLines As System.Array = New String() _
{"DATA LIST FREE /case (F) value (A1).", _
 "BEGIN DATA", _
 "1 a", _
 "END DATA."}
Processor.Submit(cmdLines)
Dim cur As DataCursorAppend = Processor.GetAppendCursorInstance()
cur.SetValueNumeric("case", 2)
cur.SetValueChar("value", "b")
cur.CommitCase()
cur.SetValueNumeric("case", 3)
cur.SetValueChar("value", "c")
cur.CommitCase()
cur.EndChanges()
cur.Close()
```

- For both the `SetValueNumeric` and `SetValueChar` methods, the first argument is the variable name, as a string, and the second argument is the value for the current case. A numeric variable whose value is not specified, or specified as *Nothing*, is set to the system-missing value. A string variable whose value is not specified, or specified as *Nothing*, will have a blank value. The value will be valid unless you explicitly define the blank value to be missing for that variable. When setting values of IBM SPSS Statistics date or date/time variables, you can specify a `Date` data type as the value for the `SetValueNumeric` method. If you specify an integer value, it will be interpreted as the number of seconds from October 14, 1582.
- The `CommitCase` method must be called to commit the values for each new case. Changes to the active dataset take effect when the cursor is closed. When working in append mode, the cursor is ready to accept values for a new case (using `SetValueNumeric` and `SetValueChar`) once `CommitCase` has been called for the previous case.
- The `EndChanges` method signals the end of appending cases and must be called before the cursor is closed or the new cases will be lost.

Creating and Managing Multiple Datasets

Using a data step, you can create and work concurrently with multiple datasets. Data steps are initiated with the `GetDatastepInstance` method from the `Processor` class.

Once a data step has been initiated, you access existing datasets and create new datasets with the `DsDataset` class. `DsDataset` objects provide access to the case data and variable information contained in a dataset, allowing you to read from the dataset, add new cases, modify existing cases, add new variables, and modify properties of existing variables.

Limitations

- Within a data step you cannot create a cursor, a pivot table, or a text block, and you cannot call the `StartProcedure` method or the `Submit` method from the `Processor` class.
- You cannot start a data step if there are pending transformations. If you need to access case data in the presence of pending transformations, use a cursor.
- Only one data step can exist at a time.
- An instance of the `DsDataset` class cannot be used outside of the data step in which it was created. *Note*: `DsDataset` objects can also be created between `StartProcedure` and `EndProcedure` and not associated with a data step.
- A new dataset created with the `DsDataset` class is not set to be the active dataset. To make the dataset the active one, use the `SetActive` method of the associated `Datastep` object.
- The `DsDataset` class does not honor case filters specified with the `FILTER` or `USE` commands.

Example: Creating a New Dataset

```
Dim datastepObj As Datastep = Processor.GetDatastepInstance()
Dim dsObj As New DsDataset("newDataset", True)
Dim varsObj As DsVariableCollection = dsObj.VarList
Dim caseObj As DsCases = dsObj.Cases
Dim varObj As DsVariable
varsObj.Add("numvar", 0)
varsObj.Add("strvar", 1)
varObj = varsObj.Item("numvar")
varObj.Label = "Sample numeric variable"
varObj = varsObj.Item("strvar")
varObj.Label = "Sample string variable"
Dim values As Object() = {1, "a"}
caseObj.Add(values)
values(0) = 2
values(1) = "b"
caseObj.Add(values)
datastepObj.Dispose()
```

- Once a data step has been initiated, you create a new dataset with the `DsDataset` class. The first argument specifies the name of the new dataset, and setting the second argument to *True* specifies that this `DsDataset` object is a new dataset (you can also create `DsDataset` objects for existing datasets, allowing you to access and/or modify cases and variables).
- You add variables to a dataset using the `Add` (or `Insert`) method of the `DsVariableCollection` object associated with the dataset. The `DsVariableCollection` object is accessed from the `VarList` property of the `DsDataset` object, as in `dsObj.VarList`. The arguments to the `Add` method are the name of the new variable and the variable type (0 for numeric variables, and an integer equal to the defined length of the string for string variables).
- Variable properties, such as the variable label and measurement level, are set through properties of the associated `DsVariable` object, accessible from the `DsVariableCollection` object. For example, `varsObj.Item("numvar")` accesses the `DsVariable` object associated with the variable *numvar*.
- You add cases to a dataset using the `Add` (or `Insert`) method of the `DsCases` object associated with the dataset. The `DsCases` object is accessed from the `Cases` property of the `DsDataset` object, as in `dsObj.Cases`. The argument to the `Add` method is an array specifying the case values. Values specified in the array must match the corresponding variable type--a double for a numeric variable, and a string for a string variable.

Note: When setting case values of IBM SPSS Statistics date or date/time variables, you can specify the value as a `Date` data type. If you specify an integer value, it will be interpreted as the number of seconds from October 14, 1582.

- You end a data step with the `Dispose` method of the associated `Datastep` instance.

Example: Saving New Datasets

When creating new datasets that you intend to save, you'll want to keep track of the dataset names since the save operation is done outside of the associated data step. In this example, a sample dataset containing employee data for three departments is first created. Three new datasets, one for each department, are then created and saved.

```
Dim cmdLines As System.Array
cmdLines = New String() _
{"DATA LIST FREE /dept (F2) empid (F4) salary (F6).", _
 "BEGIN DATA", _
 "7  57  57000", _
 "5  23  40200", _
 "3  62  21450", _
 "3  18  21900", _
 "5  21  45000", _
 "5  29  32100", _
 "7  38  36000", _
 "3  42  21900", _
 "7  11  27900", _
 "END DATA.", _
 "DATASET NAME saldata.", _
 "SORT CASES BY dept."}
Processor.Submit(cmdLines)
Dim datastepObj As Datastep = Processor.GetDatastepInstance()
Dim dsObj As New DsDataset()
Dim caseObj As DsCases = dsObj.Cases
Dim newdsObj As DsDataset
Dim newvarsObj As DsVariableCollection
Dim newcaseObj As DsCases
Dim dept As Integer
Dim dsNames As New StringDictionary()
' Create the new datasets
newdsObj = New DsDataset(Nothing, True)
newvarsObj = newdsObj.VarList
newcaseObj = newdsObj.Cases
dept = caseObj.Item(0, 0)
dsNames.Add(dept, newdsObj.Name)
newvarsObj.Add("dept", 0)
newvarsObj.Add("empid", 0)
newvarsObj.Add("salary", 0)
For Each row As System.Array In caseObj
   If row(0) <> dept Then
      newdsObj = New DsDataset(Nothing, True)
      dept = row(0)
      dsNames.Add(dept, newdsObj.Name)
      newcaseObj = newdsObj.Cases
      newvarsObj = newdsObj.VarList
      newvarsObj.Add("dept", 0)
      newvarsObj.Add("empid", 0)
      newvarsObj.Add("salary", 0)
   End If
   newcaseObj.Add(row)
Next
datastepObj.Dispose()
' Save the new datasets
For Each de As DictionaryEntry In dsNames
    cmdLines = New String() _
    {"DATASET ACTIVATE " & de.Value & ".", _
     "SAVE OUTFILE='/mydata/saldata_" & de.Key & ".sav'."}
    Processor.Submit(cmdLines)
Next
cmdLines = New String() _
{"DATASET ACTIVATE saldata.", _
 "DATASET CLOSE ALL."}
Processor.Submit(cmdLines)
```

- Instantiating the `DsDataset` class without arguments, as in `DsDataset()`, creates a `DsDataset` object for the active dataset--in this case, the sample dataset.

- The code `newdsObj = New DsDataset(Nothing, True)` creates a new dataset. The name of the dataset is available from the *Name* property, as in `newdsObj.Name`. In this example, the names of the new datasets are stored to the dictionary *dsNames*.
- To save new datasets created with the `DsDataset` class, use the `SAVE` command after calling the `Dispose` method to end the data step. In this example, `DATASET ACTIVATE` is used to activate each new dataset, using the dataset names stored in *dsNames*.

Example: Modifying Case Values

```
Dim cmdLines As System.Array
cmdLines = New String() _
{"DATA LIST FREE /cust (F2) amt (F5).", _
 "BEGIN DATA", _
 "210 4500", _
 "242 6900", _
 "370 32500", _
 "END DATA."}
Processor.Submit(cmdLines)
Dim datastepObj As Datastep = Processor.GetDatastepInstance()
Dim dsObj As New DsDataset()
Dim caseObj As DsCases = dsObj.Cases
For I As Integer = 0 To caseObj.Count - 1
   caseObj.Item(I, 1) = 1.05 * caseObj.Item(I, 1)
Next
datastepObj.Dispose()
```

- The `DsCases` object, accessed from the `Cases` property of a `DsDataset` object, allows you to read, modify, and append cases. To access the value for a given variable within a particular case you can use the `Item` property of the `DsCases` object, specifying the case number and the index of the variable (index values represent position in the active dataset, starting with 0 for the first variable in file order, and case numbers start from 0). For example, `caseObj.Item(I, 1)` specifies the value of the variable with index 1 for case number `I`.

  Note: When setting case values of IBM SPSS Statistics date or date/time variables, you can specify the value as a `Date` data type. If you specify an integer value, it will be interpreted as the number of seconds from October 14, 1582.

Example: Comparing Datasets

`DsDataset` objects allow you to concurrently work with the case data from multiple datasets. As a simple example, we'll compare the cases in two datasets and indicate identical cases with a new variable added to one of the datasets.

```
Dim cmdLines As System.Array
cmdLines = New String() _
{"DATA LIST FREE /id (F2) salary (DOLLAR8) jobcat (F1).", _
 "BEGIN DATA", _
 "1 57000 3", _
 "3 40200 1", _
 "2 21450 1", _
 "END DATA.", _
 "SORT CASES BY id.", _
 "DATASET NAME empdata1.", _
 "DATA LIST FREE /id (F2) salary (DOLLAR8) jobcat (F1).", _
 "BEGIN DATA", _
 "3 41000 1", _
 "1 59280 3", _
 "2 21450 1", _
 "END DATA.", _
 "SORT CASES BY id.", _
 "DATASET NAME empdata2."}
Processor.Submit(cmdLines)
Dim datastepObj As Datastep = Processor.GetDatastepInstance()
Dim dsObj1 As New DsDataset("empdata1")
Dim dsObj2 As New DsDataset("empdata2")
Dim varsObj As DsVariableCollection = dsObj2.VarList
Dim caseObj1 As DsCases = dsObj1.Cases
Dim caseObj2 As DsCases = dsObj2.Cases
Dim nvars As Integer = varsObj.Count
varsObj.Add("match", 0)
For I As Integer = 0 To caseObj1.Count - 1
   caseObj2.Item(I, nvars) = 1
   For J As Integer = 0 To nvars - 1
      If caseObj1.Item(I, J) <> caseObj2.Item(I, J) Then
```

```
         caseObj2.Item(I, nvars) = 0
         Exit For
      End If
   Next
Next
datastepObj.Dispose()
```

- The two datasets are first sorted by the variable *id* which is common to both datasets.
- `dsObj1` and `dsObj2` are `DsDataset` objects associated with the two datasets *empdata1* and *empdata2* to be compared.
- The new variable *match*, added to *empdata2*, is set to 1 for cases that are identical and 0 otherwise.

Retrieving Output from Syntax Commands

To retrieve command output, you first route it via the Output Management System (OMS) to an area in memory referred to as the **XML workspace** where it is stored as an XPath DOM that conforms to the Output XML Schema. Output is retrieved from this workspace with functions that employ XPath expressions.

Constructing the correct XPath expression (IBM SPSS Statistics currently supports XPath 1.0) requires an understanding of the Output XML schema. The output schema spss-output-1.8.xsd is distributed with IBM SPSS Statistics. Documentation is included in the IBM SPSS Statistics Help system. It is also provided with the IBM SPSS Statistics - Programmability SDK (available from the SPSS Community) and accessible from *documentation**OutputSchema**oms_oxml_schema_intro.htm* within the IBM SPSS Statistics - Programmability SDK.

Example

In this example, we'll retrieve the mean value of a variable calculated from the Descriptives procedure.

```
Dim handle, context, xpath As String
'Route output to the XML workspace.
Dim cmdLines As System.Array = New String() _
{"GET FILE='c:/data/Employee data.sav'.", _
 "OMS SELECT TABLES ", _
 "/IF COMMANDS=['Descriptives'] SUBTYPES=['Descriptive Statistics'] ", _
 "/DESTINATION FORMAT=OXML XMLWORKSPACE='desc_table' ", _
 "/TAG='desc_out'.", _
 "DESCRIPTIVES VARIABLES=salary, salbegin, jobtime, prevexp ", _
 "/STATISTICS=MEAN.", _
 "OMSEND TAG='desc_out'."}
Processor.Submit(cmdLines)
'Get output from the XML workspace using XPath.
handle = "desc_table"
context = "/outputTree"
xpath = "//pivotTable[@subType='Descriptive Statistics']" + _
        "/dimension[@axis='row']" + _
        "/category[@varName='salary']" + _
        "/dimension[@axis='column']" + _
        "/category[@text='Mean']" + _
        "/cell/@text"
Dim result() As String = Processor.EvaluateXPath(handle,context,xpath)
Processor.DeleteXPathHandle(handle)
```

- The `OMS` command is used to direct output from a syntax command to the XML workspace. The `XMLWORKSPACE` keyword on the `DESTINATION` subcommand, along with `FORMAT=OXML`, specifies the XML workspace as the output destination. It is a good practice to use the `TAG` subcommand, as done here, so as not to interfere with any other OMS requests that may be operating. The identifiers used for the `COMMANDS` and `SUBTYPES` keywords on the `IF` subcommand can be found in the OMS Identifiers dialog box, available from the Utilities menu in IBM SPSS Statistics.
- The `XMLWORKSPACE` keyword is used to associate a name with this XPath DOM in the workspace. In the current example, output from the `DESCRIPTIVES` command will be identified with the name *desc_table*. You can have many XPath DOM's in the XML workspace, each with its own unique name.
- The `OMSEND` command terminates active `OMS` commands, causing the output to be written to the specified destination--in this case, the XML workspace.

- You retrieve values from the XML workspace with the `EvaluateXPath` method from the `Processor` class. The method takes an explicit XPath expression, evaluates it against a specified XPath DOM in the XML workspace, and returns the result as a 1-dimension array of string values.
- The first argument to the `EvaluateXPath` function specifies the XPath DOM to which an XPath expression will be applied. This argument is referred to as the handle name for the XPath DOM and is simply the name given on the `XMLWORKSPACE` keyword on the associated `OMS` command. In this case the handle name is *desc_table*.
- The second argument to `EvaluateXPath` defines the XPath context for the expression and should be set to `"/outputTree"` for items routed to the XML workspace by the `OMS` command.
- The third argument to `EvaluateXPath` specifies the remainder of the XPath expression (the context is the first part) and must be quoted. Since XPath expressions almost always contain quoted strings, you'll need to use a different quote type from that used to enclose the expression. For users familiar with XSLT for OXML and accustomed to including a namespace prefix, note that XPath expressions for the `EvaluateXPath` function should not contain the `oms:` namespace prefix.
- The XPath expression in this example is specified by the variable *xpath*. It is not the minimal expression needed to select the mean value of interest but is used for illustration purposes and serves to highlight the structure of the XML output.

  `//pivotTable[@subType='Descriptive Statistics']` selects the Descriptives Statistics table.

  `/dimension[@axis='row']/category[@varName='salary']` selects the row for the variable *salary*.

  `/dimension[@axis='column']/category[@text='Mean']` selects the *Mean* column within this row, thus specifying a single cell in the pivot table.

  `/cell/@text` selects the textual representation of the cell contents.
- When you have finished with a particular output item, it is a good idea to delete it from the XML workspace. This is done with the `DeleteXPathHandle` method, whose single argument is the name of the handle associated with the item.

If you're familiar with XPath, you might want to convince yourself that the mean value of *salary* can also be selected with the following simpler XPath expression:

```
//category[@varName='salary']//category[@text='Mean']/cell/@text
```

Note: To the extent possible, construct your XPath expressions using language-independent attributes, such as the variable name rather than the variable label. That will help reduce the translation effort if you need to deploy your code in multiple languages. Also consider factoring out language-dependent identifiers, such as the name of a statistic, into constants. You can obtain the current language used for pivot table output with the syntax command `SHOW OLANG`.

You may also consider using `text_eng` attributes in place of `text` attributes in XPath expressions. `text_eng` attributes are English versions of `text` attributes and have the same value regardless of the output language. The `OATTRS` subcommand of the `SET` command specifies whether `text_eng` attributes are included in OXML output.

Retrieving Images Associated with an Output XPath DOM

You can retrieve images associated with output routed to the XML workspace. In this example, we'll retrieve a bar chart associated with output from the Frequencies procedure.

```
Dim cmdLines As System.Array = New String() _
{"GET FILE='c:/data/Employee data.sav'.", _
 "OMS SELECT CHARTS ", _
 "/IF COMMANDS=['Frequencies'] ", _
 "/DESTINATION FORMAT=OXML IMAGES=YES", _
 "CHARTFORMAT=IMAGE IMAGEROOT='myimages' IMAGEFORMAT=JPG XMLWORKSPACE='demo'.", _
 "FREQUENCIES VARIABLES=jobcat", _
 "  /BARCHART PERCENT", _
 "  /ORDER=ANALYSIS.", _
 "OMSEND."}
Processor.Submit(cmdLines)
handle = "demo"
context = "/outputTree"
```

```
xpath = "//command[@command='Frequencies']" + _
        "/chartTitle[@text='Bar Chart']" + _
        "/chart/@imageFile"
Dim result() As String = Processor.EvaluateXPath(handle, context, xpath)
Dim imageName As String = result(0)
Dim imageSize As Integer = 0
Dim imageType As String = String.Empty
Dim imageObj As Byte() = Processor.GetImage(handle, imageName, imageSize, imageType)
Dim fs As New FileStream("c:\temp\result.jpg", FileMode.Create, FileAccess.Write)
fs.Write(imageObj, 0, imageSize)
fs.Flush()
fs.Close()
Processor.DeleteXPathHandle(handle)
```

- The `OMS` command routes output from the `FREQUENCIES` command to an output XPath DOM with the handle name of *demo*.
- To route images along with the OXML output, the `IMAGES` keyword on the `DESTINATION` subcommand (of the `OMS` command) must be set to `YES`, and the `CHARTFORMAT`, `MODELFORMAT`, or `TREEFORMAT` keyword must be set to `IMAGE`.
- The `EvaluateXPath` function is used to retrieve the name of the image associated with the bar chart output from the `FREQUENCIES` command. In the present example, the value returned by `EvaluateXPath` is a list with a single element, which is then stored to the variable *imageName*.
- The `GetImage` function retrieves the image, which is then written to an external file.

  The first argument to the `GetImage` function specifies the particular XPath DOM and must be a valid handle name defined by a previous IBM SPSS Statistics `OMS` command.

  The second argument to `GetImage` is the filename associated with the image in the OXML output--specifically, the value of the `imageFile` attribute of the `chart` element associated with the image.

  The third argument to `GetImage` is the amount of memory required for the image and is a value that is returned by the function.

  The fourth argument to `GetImage` is a string specifying the image type and is a value that is returned by the function. The possible values are: “PNG”, “JPG”, “BMP".

Writing XML Workspace Contents to a File

When writing and debugging XPath expressions, it is often useful to have a sample file that shows the XML structure. This is provided by the `GetXmlUtf16` method from the `Processor` class, as well as by an option on the `OMS` syntax command. The following program block recreates the XML workspace for the preceding example and writes the XML associated with the handle *desc_table* to the file *c:\temp\descriptives_table.xml*.

```
'Route output to the XML workspace.
Dim cmdLines As System.Array = New String() _
{"GET FILE='c:/data/Employee data.sav'.", _
 "OMS SELECT TABLES ", _
 "/IF COMMANDS=['Descriptives'] SUBTYPES=['Descriptive Statistics'] ", _
 "/DESTINATION FORMAT=OXML XMLWORKSPACE='desc_table' ", _
 "/TAG='desc_out'.", _
 "DESCRIPTIVES VARIABLES=salary, salbegin, jobtime, prevexp ", _
 "/STATISTICS=MEAN.", _
 "OMSEND TAG='desc_out'."}
Processor.Submit(cmdLines)
'Write an item from the XML workspace to a file.
Processor.GetXmlUtf16("desc_table","c:/temp/descriptives_table.xml")
Processor.DeleteXPathHandle("desc_table")
```

The section of *c:\temp\descriptives_table.xml* that specifies the Descriptive Statistics table, including the mean value of *salary*, is as follows (the output is written in Unicode (UTF-16)):

```
<pivotTable subType="Descriptive Statistics" text="Descriptive Statistics">
  <dimension axis="row" text="Variables">
    <category label="Current Salary" text="Current Salary"
    varName="salary" variable="true">
      <dimension axis="column" text="Statistics">
        <category text="N">
          <cell number="474" text="474"/>
        </category>
        <category text="Mean">
          <cell decimals="2" format="dollar" number="34419.567510548"
          text="$34,419.57"/>
        </category>
      </dimension>
    </category>
```

Note: The form `GetXmlUtf16(String)` of the `GetXmlUtf16` method returns the content associated with a specified handle, without writing the content to a file.

Retrieving a DOM as an XmlDocument Object

As an alternative to the `GetXmlUtf16` method for retrieving an entire XPath DOM, you can use the `GetXmlObject` method to return the contents of an output DOM as an `XmlDocument` object. This allows you to process the contents using the powerful methods available for .NET `XmlDocument` objects. The root element of the associated DOM has a default namespace with the URI *http://xml.spss.com/spss/oms*, so depending on which `XmlDocument` methods you use, you may have to add this URI and an associated arbitrary prefix to the `XmlNamespaceManager`. XPath references will then need to include this prefix, as shown in the following code sample that extracts the `text` attribute from each `cell` element, for an XPath DOM specified by *handle*.

```
Dim result As Xml.XmlDocument = Processor.GetXmlObject(handle)
Dim nsmgr As XmlNamespaceManager = New XmlNamespaceManager(result.NameTable)
nsmgr.AddNamespace("spss", "http://xml.spss.com/spss/oms")
Dim root As XmlElement = result.DocumentElement
Dim nodeList As XmlNodeList
nodeList = root.SelectNodes("//spss:cell/@text", nsmgr)
```

Running Python and R Programs from .NET

The IBM SPSS Statistics `BEGIN PROGRAM` command provides the ability to integrate the capabilities of external programming languages with IBM SPSS Statistics command syntax. The supported languages are the Python programming language and R. This enables you to utilize the extensive set of scientific and statistical programming libraries available with the Python language and R to create custom algorithms that you can apply to your data. Results can be written to a dataset or the XML workspace and then retrieved by your .NET program.

Python Programs

Once you have installed the IBM SPSS Statistics - Integration Plug-in for Python, you have full access to the Python programming language by including Python code in a `BEGIN PROGRAM PYTHON-END PROGRAM` block (within command syntax) and submitting the command syntax with the `Submit` method from the `Processor` class. For example:

```
Dim cmdLines As System.Array
cmdLines = New String() _
{"BEGIN PROGRAM PYTHON.", _
 "import spss", _
 "<Python code>", _
 "END PROGRAM."}
Processor.Submit(cmdLines)
```

- The Python statement `import spss` imports the Python modules (installed with the IBM SPSS Statistics - Integration Plug-in for Python) that allow the Python processor to interact with IBM SPSS Statistics. Your Python language code follows the `import` statement.
- You can omit the keyword `PYTHON` on `BEGIN PROGRAM`--it is the default.

Complete documentation on the functionality available with the IBM SPSS Statistics - Integration Plug-in for Python is available in the SPSS Statistics Help system under Integration Plug-in for Python.

R Programs

Once you have installed the IBM SPSS Statistics - Integration Plug-in for R, you have full access to the R programming language by including R code in a `BEGIN PROGRAM R`-`END PROGRAM` block (within command syntax) and submitting the command syntax with the `Submit` method from the `Processor` class. For example:

```
Dim cmdLines As System.Array
cmdLines = New String() _
{"BEGIN PROGRAM R.", _
 "<R code>", _
 "END PROGRAM."}
Processor.Submit(cmdLines)
```

Complete documentation on the functionality available with the IBM SPSS Statistics - Integration Plug-in for R is available in the SPSS Statistics Help system under Integration Plug-in for R.

Inserting Programs From Command Syntax Files

You can use the IBM SPSS Statistics `INSERT` command to include `BEGIN PROGRAM`-`END PROGRAM` blocks contained in command syntax files. This allows you to store your programs as separate code files and include them as needed. For example:

```
Processor.Submit("INSERT FILE='/myprograms/program_block.sps'.")
```

The file */myprograms/program_block.sps* would contain a `BEGIN PROGRAM` block, as in:

```
BEGIN PROGRAM PYTHON.
import spss
<Python code>
END PROGRAM.
```

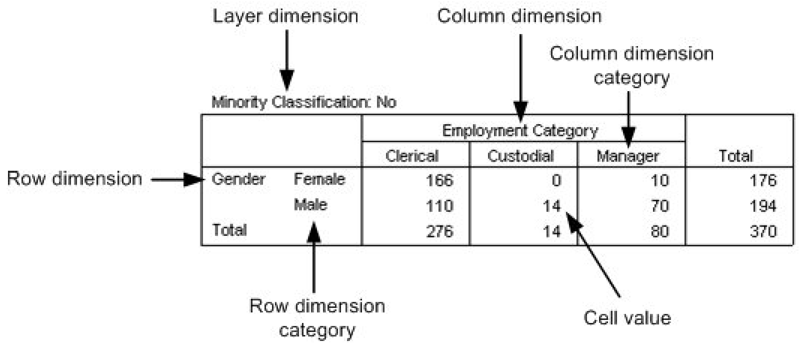
Creating Custom Output

The IBM SPSS Statistics - Integration Plug-in for Microsoft .NET provides the ability to create output in the form of custom pivot tables and text blocks. Using the Output Management System (OMS), the output can be rendered in a variety of formats such as HTML, text, or XML that conforms to the Output XML Schema (*xml.spss.com/spss/oms*).

Creating Pivot Tables

The following figure shows the basic structural components of a pivot table. Pivot tables consist of one or more dimensions, each of which can be of the type row, column, or layer. In this example there is one dimension of each type. Each dimension contains a set of categories that label the elements of the dimension--for instance, row labels for a row dimension. A layer dimension allows you to display a separate two dimensional table for each category in the layered dimension--for example, a separate table for each value of minority classification, as shown here. When layers are present the pivot table can be thought of as stacked in layers, with only the top layer visible.

Each cell in the table can be specified by a combination of category values. In the example shown here, the indicated cell is specified by a category value of *Male* for the *Gender* dimension, *Custodial* for the *Employment Category* dimension, and *No* for the *Minority Classification* dimension.



Minority Classification: No

		Employment Category			
		Clerical	Custodial	Manager	Total
Gender	Female	166	0	10	176
	Male	110	14	70	194
Total		276	14	80	370

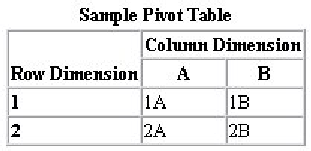
Figure 1. Pivot table structure

The SimplePivotTable Method

Pivot tables are created with the `BasePivotTable` class. For the common case of creating a pivot table with a single row dimension and a single column dimension, the `BasePivotTable` class provides the `SimplePivotTable` method. The arguments to the method provide the dimensions, categories, and cell values. No other methods are necessary in order to create the table structure and populate the cells. See if you require more functionality than the `SimplePivotTable` method provides.

```
Dim cmdLines As System.Array = New String() _
{"GET FILE='c:/examples/data/demo.sav'.", _
 "OMS SELECT TABLES", _
 "/IF SUBTYPES=['SimpleTableDemo']", _
 "/DESTINATION FORMAT=HTML OUTFILE='c:/temp/simpletable.htm'."}
Processor.Submit(cmdLines)
Dim title As String = "Sample Pivot Table"
Dim templateName As String = "SimpleTableDemo"
Dim table As BasePivotTable = _
Processor.GetBasePivotTableInstance(title, templateName, Nothing)
table.SimplePivotTable( _
                       "Row Dimension", _
                       New Object() {"1", "2"}, _
                       "Column Dimension", _
                       New Object() {"A", "B"}, _
                       New Object() {"1A", "1B", "2A", "2B"})
table.Close()
Processor.Submit("OMSEND.")
```

Result



Sample Pivot Table

	Column Dimension	
Row Dimension	**A**	**B**
1	1A	1B
2	2A	2B

Figure 2. Output of Simple Pivot Table

- The `BasePivotTable` class requires an active dataset, so a dataset is opened with the `GET` syntax command.
- In order to produce something other than a textual representation of a pivot table, you route the output with OMS. To route pivot table output to OMS, you include the `TABLES` keyword on the `SELECT` subcommand of the `OMS` command, as in this example. As specified on the `OMS` command, the pivot table in this example will be rendered in HTML and routed to the file *c:/temp/simpletable.htm*.
- To create a pivot table, you create an instance of the `BasePivotTable` class with the `GetBasePivotTableInstance` method from the `Processor` class. The first argument to the `GetBasePivotTableInstance` method is a required string that specifies the title that appears with the table.

- The second argument to the `GetBasePivotTableInstance` method is a string that specifies the OMS (Output Management System) table subtype for this table. This value is required (even if you don't route the output with OMS), must begin with a letter, and have a maximum of 64 bytes. The table subtype can be used on the `SUBTYPES` keyword of the `OMS` command, as done here, to include this pivot table in the output for a specified set of subtypes.

  Note: By creating the pivot table instance within a `StartProcedure-EndProcedure` block, you can associate the pivot table output with a command name, as for pivot table output produced by syntax commands. The command name is the argument to the `StartProcedure` function and can used on the `COMMANDS` keyword of the `OMS` command to include the pivot table in the output for a specified set of commands (along with optionally specifying a subtype as discussed above).
- The third argument to the `GetBasePivotTableInstance` method is a string that specifies an optional outline title for the table, or *Nothing*, as in this example. When output is routed to OMS in OXML format, the outline title specifies a heading tag that will contain the output for the pivot table.

Once you've created an instance of the `BasePivotTable` class, you use the `SimplePivotTable` method to create the structure of the table and populate the table cells. The method has four different calling signatures. The signature used in this example makes use of all possible arguments. The arguments used in this example, are in order:

- **rowdim.** A label for the row dimension, given as a string. If empty, the row dimension label is hidden.
- **rowcats.** A 1-dimensional System.Array of categories for the row dimension. Labels can be given as numeric values or strings. They can also be specified as a `CellText` object. `CellText` objects allow you to specify that category labels be treated as variable names or variable values, or that cell values be displayed in one of the numeric formats used in pivot tables, such as the format for a mean. When you specify a category as a variable name or variable value, pivot table display options such as display variable labels or display value labels are honored.

  Note: The number of rows in the table is equal to the number of elements in *rowcats*, when provided. If *rowcats* is omitted, the number of rows is equal to the number of elements in the argument *cells*.
- **coldim.** A label for the column dimension, given as a string. If empty, the column dimension label is hidden.
- **colcats.** A 1-dimensional System.Array of categories for the column dimension. The array can contain the same types of items as *rowcats* described above.

  Note: The number of columns in the table is equal to the number of elements in *colcats*, when provided. If *colcats* is omitted, the number of columns is equal to the length of the first element of *cells*.
- **cells.** This argument specifies the values for the cells of the pivot table, and can be given as a 1- or 2-dimensional System.Array. In the current example, *cells* is given as the 1-dimensional array `{"1A", "1B", "2A", "2B"}`. It could also have been specified as the 2-dimensional array `{{"1A", "1B"}, {"2A", "2B"}}`.

  Elements in the pivot table are populated in row-wise fashion from the elements of *cells*. In the current example, the table has two rows and two columns (as specified by the row and column labels), so the first row will consist of the first two elements of *cells* and the second row will consist of the last two elements. When *cells* is 2-dimensional, each 1-dimensional element specifies a row. For example, with *cells* given by `{{"1A", "1B"}, {"2A", "2B"}}`, the first row is `{"1A", "1B"}` and the second row is `{"2A", "2B"}`.

  Cells can be given as numeric values or strings, or `CellText` objects (as described for *rowcats* above).
- Numeric values specified for cell values, row labels, or column labels, are displayed using the default format for the pivot table. Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays values rounded to the nearest integer. You can change the default format using the `SetDefaultFormatSpec` method from the `BasePivotTable` class. You can also override the default format for specific cells or labels by specifying the value as a `CellText.Number` object and providing a format, as in `CellText.Number(22,FormatSpec.GeneralStat)`.

- The `Close` method of the `BasePivotTable` class closes the table object. If the pivot table is not part of a `StartProcedure-EndProcedure` block, the output will be produced when the last open pivot table or text block is closed. If the pivot table is part of a `StartProcedure-EndProcedure` block, the output will be produced when `EndProcedure` is called.

Note: If you're creating a pivot table from data with splits, you'll probably want separate results displayed for each split group. See the topic "Creating Pivot Tables from Data with Splits" on page 35 for more information.

General Approach to Creating Pivot Tables

The `BasePivotTable` class provides methods for creating pivot tables that can't be created with the `SimplePivotTable` method. The basic steps for creating a pivot table are:

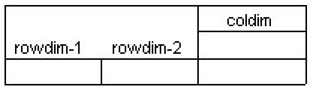
1. Create an instance of the `BasePivotTable` class
2. Add dimensions
3. Define categories
4. Set cell values

Step 1: Adding Dimensions: You add dimensions to a pivot table with the `Append` or `Insert` method.

Example: Using the Append Method

```
Dim table As BasePivotTable = _
    Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim coldim As Dimension = table.Append(Dimension.Place.column, "coldim")
Dim rowdim1 As Dimension = table.Append(Dimension.Place.row, "rowdim-1")
Dim rowdim2 As Dimension = table.Append(Dimension.Place.row, "rowdim-2")
```

- The first argument to the `Append` method specifies the type of dimension, using a member of the `Dimension.Place` Enumeration: `Dimension.Place.row` for a row dimension, `Dimension.Place.column` for a column dimension, and `Dimension.Place.layer` for a layer dimension.
- The second argument to `Append` is a string that specifies the name used to label this dimension in the table.



		coldim
rowdim-1	rowdim-2	

Figure 3. Resulting table structure

The order in which the dimensions are appended determines how they are displayed in the table. Each newly appended dimension of a particular type (row, column, or layer) becomes the current innermost dimension in the displayed table. In the example above, *rowdim-2* is the innermost row dimension since it is the last one to be appended. Had *rowdim-2* been appended first, followed by *rowdim-1*, *rowdim-1* would be the innermost dimension.

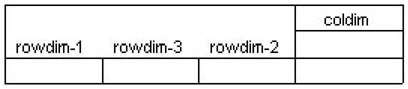
Note: Generation of the resulting table requires more code than is shown here.

Example: Using the Insert Method

```
Dim table As BasePivotTable = _
    Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim rowdim1 As Dimension = table.Append(Dimension.Place.row, "rowdim-1")
Dim rowdim2 As Dimension = table.Append(Dimension.Place.row, "rowdim-2")
Dim rowdim3 As Dimension = table.Insert(2, Dimension.Place.row, "rowdim-3")
Dim coldim As Dimension = table.Append(Dimension.Place.column, "coldim")
```

- The first argument to the `Insert` method specifies the position within the dimensions of that type (row, column, or layer). The first position has index 1 and defines the innermost dimension of that type in the displayed table. Successive integers specify the next innermost dimension and so on. In the current example, *rowdim-3* is inserted at position 2 and *rowdim-1* is moved from position 2 to position 3.

- The second argument to `Insert` specifies the type of dimension, using a member of the `Dimension.Place` Enumeration: `spss.Dimension.Place.row` for a row dimension, `spss.Dimension.Place.column` for a column dimension, and `spss.Dimension.Place.layer` for a layer dimension.
- The third argument to `Insert` is a string that specifies the name used to label this dimension in the displayed table.



			coldim
rowdim-1	rowdim-3	rowdim-2	

Figure 4. Resulting table structure

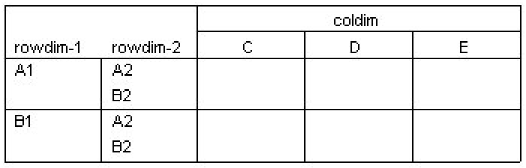
Note: Generation of the resulting table requires more code than is shown here.

Step 2: Defining Categories: Categories for a dimension are defined using the `SetCategory` method of the associated `Dimension` object (created with the `Append` or `Insert` method).

Example

```
Dim table As BasePivotTable = _
    Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim coldim As Dimension = table.Append(Dimension.Place.column, "coldim")
Dim rowdim1 As Dimension = table.Append(Dimension.Place.row, "rowdim-1")
Dim rowdim2 As Dimension = table.Append(Dimension.Place.row, "rowdim-2")
coldim.SetCategory(New CellText.NString("C"))
coldim.SetCategory(New CellText.NString("D"))
coldim.SetCategory(New CellText.NString("E"))
rowdim1.SetCategory(New CellText.NString("A1"))
rowdim1.SetCategory(New CellText.NString("B1"))
rowdim2.SetCategory(New CellText.NString("A2"))
rowdim2.SetCategory(New CellText.NString("B2"))
```

- You set categories after you add dimensions, so the `SetCategory` method calls follow the `Append` or `Insert` method calls.
- The argument to `SetCategory` is a single category expressed as a `CellText` object (one of `CellText.Number`, `CellText.NString`, `CellText.VarName`, or `CellText.VarValue`). When you specify a category as a variable name or variable value, pivot table display options such as display variable labels or display value labels are honored. In the present example, we use string objects whose single argument is the string specifying the category.
- For a given dimension, the order of the categories displayed in the table, is the order in which they are created. For instance, the first column has a category value of `"C"`, the second column has a category value of `"D"`, etc.



		coldim		
rowdim-1	rowdim-2	C	D	E
A1	A2			
	B2			
B1	A2			
	B2			

Figure 5. Resulting table structure

Notes

- Generation of the resulting table (shown above) requires more code than is shown here.
- When specifying numeric category values with `CellText.Number` objects, values will be formatted using the pivot table's default format, unless a specific format is supplied, as in `CellText.Number(22,FormatSpec.GeneralStat)`. Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays values rounded to the nearest integer. You can change the default format using the `SetDefaultFormatSpec` method from the `BasePivotTable` class.

Step 3: Setting Cell Values: There are two methods for setting cell values: setting values one cell at a time using the `SetCell` method, or setting entire rows or columns using the `SetCellsByRow` or `SetCellsByColumn` method.

Example: Using the SetCell Method

This example reproduces the table created in the SimplePivotTable example.

```
Dim table As BasePivotTable = _
    Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim rowdim As Dimension = table.Append(Dimension.Place.row, "Row Dimension")
Dim coldim As Dimension = table.Append(Dimension.Place.column, _
                                       "Column Dimension")
rowdim.SetCategory(New CellText.NString("1"))
rowdim.SetCategory(New CellText.NString("2"))
coldim.SetCategory(New CellText.NString("A"))
coldim.SetCategory(New CellText.NString("B"))
table.SetCell(New Object() {New CellText.NString("1"), _
                            New CellText.NString("A")}, _
                            New CellText.NString("1A"))
table.SetCell(New Object() {New CellText.NString("1"), _
                            New CellText.NString("B")}, _
                            New CellText.NString("1B"))
table.SetCell(New Object() {New CellText.NString("2"), _
                            New CellText.NString("A")}, _
                            New CellText.NString("2A"))
table.SetCell(New Object() {New CellText.NString("2"), _
                            New CellText.NString("B")}, _
                            New CellText.NString("2B"))
```

- The `Append` method is used to add a row dimension and then a column dimension to the structure of the table. The table specified in this example has one row dimension and one column dimension.
- The `SetCategory` method is used to create the four categories needed for the table--two categories for the row dimension and two categories for the column dimension.
- Cell values are set with the `SetCell` method. The first argument is a 1-dimensional System.Array of categories (one for each dimension) that specifies the cell. The first element specifies a category in the first appended dimension (what we have named "Row Dimension"), the second element specifies a category in the second appended dimension (what we have named "Column Dimension"), etc. In this example, the array`{New CellText.NString("1"), New CellText.NString("A")}` specifies the cell whose "Row Dimension" category is "1" and "Column Dimension" category is "A".
- The second argument to the `SetCell` method is the cell value. Cell values must be specified as `CellText` objects (one of `CellText.Number`, `CellText.NString`, `CellText.VarName`, or `CellText.VarValue`).

Example: Setting Cell Values by Row or Column

The `SetCellsByRow` and `SetCellsByColumn` methods allow you to set cell values for entire rows or columns with one method call. To illustrate the approach we'll use the `SetCellsByRow` method to reproduce the table created in the SimplePivotTable example. It is a simple matter to rewrite the example to set cells by column.

Note: You can only use the `SetCellsByRow` method with pivot tables that have one column dimension and you can only use the `SetCellsByColumn` method with pivot tables that have one row dimension.

```
Dim table As BasePivotTable = _
    Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim rowdim As Dimension = table.Append(Dimension.Place.row, "Row Dimension")
Dim coldim As Dimension = table.Append(Dimension.Place.column, _
                                       "Column Dimension")
rowdim.SetCategory(New CellText.NString("1"))
rowdim.SetCategory(New CellText.NString("2"))
coldim.SetCategory(New CellText.NString("A"))
coldim.SetCategory(New CellText.NString("B"))
table.SetCellsByRow(New Object() {New CellText.NString("1")}, _
                    New Object() {New CellText.NString("1A"), _
                                  New CellText.NString("1B")})
table.SetCellsByRow(New Object() {New CellText.NString("2")}, _
                    New Object() {New CellText.NString("2A"), _
                                  New CellText.NString("2B")})
```

- The `SetCellsByRow` method is called for each of the two categories in the row dimension.

- The first argument to the SetCellsByRow method is a 1-dimensional System.Array of categories (one for each row dimension) that specifies a row. Each element of the array must be specified as a CellText object (one of CellText.Number, CellText.NString, CellText.VarName, or CellText.VarValue).
- The second argument to the SetCellsByRow method is a 1-dimensional System.Array of cell values that specifies the cells in the row, one element for each column category in the single column dimension. Each element of the array must be specified as a CellText object. The first element in the array will populate the first column category, the second will populate the second column category, and so on.

Note

When specifying numeric cell values with CellText.Number objects, values will be formatted using the pivot table's default format, unless a specific format is supplied, as in CellText.Number(22,FormatSpec.GeneralStat). Instances of the BasePivotTable class have an implicit default format of Count, which displays values rounded to the nearest integer. You can change the default format using the SetDefaultFormatSpec method from the BasePivotTable class.

Generating the Pivot Table Output: The Close method of the BasePivotTable class closes the table object. If the pivot table is not part of a StartProcedure-EndProcedure block, the output will be produced when the last open pivot table or text block is closed. If the pivot table is part of a StartProcedure-EndProcedure block, the output will be produced when EndProcedure is called.

Using Cell Values in Expressions: Once a cell's value has been set it can be accessed and used to specify the value for another cell. Cell values are stored as a CellText object (one of CellText.Number, CellText.NString, CellText.VarName, or CellText.VarValue). To use a cell value in an expression, you obtain a string or numeric representation of the value using the ToString or ToNumber method.

Example: Numeric Representations of Cell Values

```
Dim table As BasePivotTable = _
    Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim rowdim As Dimension = table.Append(Dimension.Place.row, "Row Dimension")
Dim coldim As Dimension = table.Append(Dimension.Place.column, _
                                       "Column Dimension")
rowdim.SetCategory(New CellText.NString("1"))
rowdim.SetCategory(New CellText.NString("2"))
coldim.SetCategory(New CellText.NString("A"))
coldim.SetCategory(New CellText.NString("B"))
table.SetCell(New Object() {New CellText.NString("1"), _
                            New CellText.NString("A")}, _
                            New CellText.Number(11))
Dim cell As Object = table.GetCell(New Object() _
                                   {New CellText.NString("1"), _
                                    New CellText.NString("A")})
Dim cellValue As Double = cell.ToNumber()
table.SetCell(New Object() {New CellText.NString("1"), _
                            New CellText.NString("B")}, _
                            New CellText.Number(2 * cellValue))
```

- The GetCell method is used to obtain the cell object for a specified cell. The argument is a 1-dimensional System.Array of categories (one for each dimension) that specifies the cell. Each element of the array must be specified as a CellText object. The first element of the array specifies a category in the first appended dimension (what we have named "Row Dimension"), the second element specifies a category in the second appended dimension (what we have named "Column Dimension"), etc.
- In this example, the ToNumber method is used to obtain a numeric representation of the cell object with category values {"1","A"}. The numeric value of the cell is stored in the variable *cellValue* and used to specify the value of another cell.
- Character representations of numeric values stored as CellText.NString objects, such as CellText.NString("11"), are converted to a numeric value by the ToNumber method.

Example: String Representations of Cell Values

```
Dim table As BasePivotTable = _
    Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim rowdim As Dimension = table.Append(Dimension.Place.row, "Row Dimension")
Dim coldim As Dimension = table.Append(Dimension.Place.column, _
                                        "Column Dimension")
rowdim.SetCategory(New CellText.NString("1"))
rowdim.SetCategory(New CellText.NString("2"))
coldim.SetCategory(New CellText.NString("A"))
coldim.SetCategory(New CellText.NString("B"))
table.SetCell(New Object() {New CellText.NString("1"), _
                           New CellText.NString("A")}, _
                           New CellText.NString("abc"))
Dim cell As Object = table.GetCell(New Object() _
                                 {New CellText.NString("1"), _
                                  New CellText.NString("A")})
Dim cellValue As String = cell.ToString()
table.SetCell(New Object() {New CellText.NString("1"), _
                           New CellText.NString("B")}, _
                           New CellText.NString(cellValue & "d"))
```

- The `GetCell` method is used to obtain the cell object for a specified cell. The argument is a 1-dimensional System.Array of categories (one for each dimension) that specifies the cell. Each element of the array must be specified as a `CellText` object. The first element of the array specifies a category in the first appended dimension (what we have named "Row Dimension"), the second element specifies a category in the second appended dimension (what we have named "Column Dimension"), etc.
- In this example, the `ToString` method is used to obtain a string representation of the cell with category values `{"1","A"}`. The string value is stored in the variable *cellValue* and used to specify the value of another cell.
- Numeric values stored as `CellText.Number` objects are converted to a string value by the `ToString` method.

Creating Pivot Tables from Data with Splits

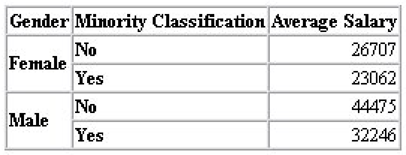
When generating pivot tables from data with splits, you may want to produce separate results for each split group. This is accomplished with the `SplitChange` method from the `DataCursor` class (also available with the `DataCursorWrite` class).

Example

In this example, a split is created and separate averages are calculated for the split groups. Results for different split groups are shown in a single pivot table. In order to understand the example, you will need to be familiar with creating pivot tables using the `BasePivotTable` class.

```
Dim cmdLines As System.Array = New String() _
    {"GET FILE='c:/data/employee data.sav'.", _
     "SORT CASES BY GENDER.", _
     "SPLIT FILE LAYERED BY GENDER.", _
     "OMS SELECT TABLES", _
     "/IF SUBTYPES=['SplitChange']", _
     "/DESTINATION FORMAT=HTML OUTFILE='c:/temp/splitchange.htm'."}
Processor.Submit(cmdLines)
Dim table As BasePivotTable = Processor.GetBasePivotTableInstance( _
                            "Table Title", "SplitChange", Nothing)
table.Append(Dimension.Place.row, "Minority Classification")
table.Append(Dimension.Place.column, "coldim", True, False)
Dim cur As DataCursor = Processor.GetReadCursorInstance()
Dim salary As Integer = 0, salarym As Integer = 0
Dim n As Integer = 0, m As Integer = 0
Dim minorityIndex As Integer = 9
Dim salaryIndex As Integer = 5
Dim row As System.Array = cur.GetRow()
cur.SplitChange()
While True
    If cur.IsEndSplit() Then
        If n > 0 Then
            salary = salary / n
        End If
        If m > 0 Then
            salarym = salarym / m
        End If
        'Populate the pivot table with values for the previous split group
        table.SetCell(New Object() _
            {New CellText.NString("No"), New CellText.NString("Average Salary")}, _
             New CellText.Number(salary, FormatSpec.Count))
        table.SetCell(New Object() _
            {New CellText.NString("Yes"), New CellText.NString("Average Salary")}, _
             New CellText.Number(salarym, FormatSpec.Count))
        salary = 0
        salarym = 0
        n = 0
        m = 0
        'Try to get the first case of the next split group
        row = cur.GetRow()
        If Not row Is Nothing Then
            cur.SplitChange()
        Else
            'There are no more cases, so quit
            Exit While
        End If
    End If
    If row(minorityIndex) = 1 Then
        salarym += row(salaryIndex)
        m += 1
    ElseIf row(minorityIndex) = 0 Then
         salary += row(salaryIndex)
         n += 1
    End If
    row = cur.GetRow()
End While
cur.Close()
table.Close()
```

Result



Gender	Minority Classification	Average Salary
Female	No	26707
	Yes	23062
Male	No	44475
	Yes	32246

Figure 6. Pivot table displaying results for separate split groups

- Command syntax is first submitted to create a split on a gender variable. The LAYERED subcommand on the SPLIT FILE command indicates that results for different split groups are to be displayed in the same table.
- The SplitChange method is called after getting the first case from the active dataset. This is required so that the pivot table output for the first split group is handled correctly.
- Split changes are detected using the IsEndSplit method from the DataCursor class (also available with the DataCursorWrite class). Once a split change is detected, the pivot table is populated with the results from the previous split.
- The value returned from the GetRow method is *Nothing* at a split boundary. Once a split has been detected, you will need to call GetRow again to retrieve the first case of the new split group, followed by SplitChange.

  Note: IsEndSplit returns *true* when the end of the dataset has been reached. Although a split boundary and the end of the dataset both result in a return value of *true* from IsEndSplit, the end of the dataset is identified by a return value of *Nothing* from a subsequent call to GetRow, as shown in this example.

Creating Text Blocks

Text blocks are created with the TextBlock class.

```
Dim cmdLines As System.Array = New String() _
{"GET FILE='c:/examples/data/demo.sav'.", _
 "OMS SELECT TEXTS", _
 "/IF LABELS = ['Text block name']", _
 "/DESTINATION FORMAT=HTML OUTFILE='c:/temp/textblock.htm'."}
Processor.Submit(cmdLines)
Dim tb As TextBlock = Processor.GetTextBlockInstance( _
                      "Text block name", "A single line of text.", Nothing)
tb.Close()
Processor.Submit("OMSEND.")
```

- The TextBlock class requires an active dataset, so a dataset is opened with the GET syntax command.
- In the common case that you're creating text blocks along with other output such as pivot tables, you'll probably be routing the output with OMS. To route text block output to OMS, you include the TEXTS

keyword on the SELECT subcommand of the OMS command, as in this example. As specified on the OMS command, the text block in this example will be rendered in HTML and routed to the file *c:/temp/textblock.htm*.

- You create an instance of the TextBlock class with the GetTextBlockInstance method from the Processor class. The first argument to the GetTextBlockInstance method is a required string that specifies the title that appears with the table. The title can be used on the LABELS keyword of the OMS command, as done here, to limit the textual output routed to OMS.
- The second argument to the GetTextBlockInstance method is the content of the text block as a string. Additional lines can be appended using the Append method of the TextBlock instance.
- The third argument to the GetTextBlockInstance method is a string that specifies an optional outline title for the text block, or *Nothing*, as in this example. When output is routed to OMS in OXML format, the outline title specifies a heading tag that will contain the output for the text block.

Deploying Your Application

You can deploy a .NET application that you have developed, but your application can only be used with a licensed IBM SPSS Statistics application. For information about licensing or distribution arrangements, please contact IBM Corp. directly.

In order for your application to run properly, the target computer must have:

- An IBM SPSS Statistics application. Support for the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET was introduced in version 14.0.2.
- The .NET Integration assemblies for IBM SPSS Statistics.
- A means for the .NET Integration assemblies to locate *spssxd_p.dll* .

.NET Integration Assemblies for IBM SPSS Statistics

Copies of these libraries are included with the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET and are located in the *NET* directory under the directory where IBM SPSS Statistics 23 is installed--for example, *C:\Program Files\IBM\SPSS\Statistics\23\NET*.

Deploy versions of these libraries that match the version of *spssxd_p.dll* on the target machine. You can check the version of *spssxd_p.dll* from the value of *SpssdxVersion* in *spssdxcfg.ini*, located in the IBM SPSS Statistics application's root directory.

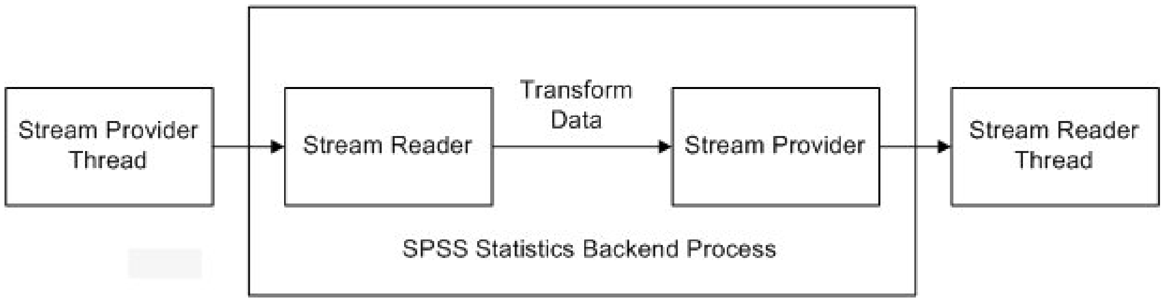
Locating spssxd_p.dll

You must provide the means for instances of the Processor class, from *SPSS.BackendAPI.Controller.dll*, to locate the IBM SPSS Statistics backend API library *spssxd_p.dll* . See the topic “Getting Started” on page 11 for more information.

Chapter 3. Streaming Binary Data in IBM SPSS Statistics

Through command syntax you can instruct the IBM SPSS Statistics backend to read binary data from a pipe or socket, and you can independently instruct the backend to write binary data from the active dataset to a pipe or socket. This allows you to eliminate the time-consuming step of writing data to a disk file when reading data into IBM SPSS Statistics or when passing data from IBM SPSS Statistics to an external program for further processing.

The two independent functions of streaming data into IBM SPSS Statistics and streaming data from IBM SPSS Statistics require separate threads on the parts of the external processes that are involved in providing a stream to IBM SPSS Statistics and/or reading a stream from IBM SPSS Statistics. The IBM SPSS Statistics backend process is single-threaded. For example, you might stream data into IBM SPSS Statistics from an external processor that collects real-time data, and then use IBM SPSS Statistics to analyze the data and produce the results. Or, you might start with an IBM SPSS Statistics dataset, transform it, and stream it to an external processor for further analysis. You can also combine the capabilities of streaming data into and out of IBM SPSS Statistics in the same IBM SPSS Statistics backend process, as shown in the following figure. This illustrates the use of IBM SPSS Statistics as an intermediate data transformation step in a larger analysis which involves one or more external processors (the two external threads shown in the figure may or may not belong to the same processor).



You use the `GET DATA` command with `/TYPE=BINARYSTREAM` to read from a stream, and you use the `SAVE` or `XSAVE` command with `/TYPE=BINARYSTREAM` to write to a stream (details of the command syntax are provided in the sections that follow). Remotely instructing IBM SPSS Statistics to read from and/or write to a binary stream is done by submitting command syntax to an instance of the IBM SPSS Statistics backend. This can be done in one of the following ways:

- Use the IBM SPSS Statistics - Integration Plug-in for Python or IBM SPSS Statistics - Integration Plug-in for Microsoft .NET , and pass the necessary command syntax to IBM SPSS Statistics, from an external Python or .NET program, using the `Submit` function (available with all integration plug-ins). The `Submit` function starts up an instance of the IBM SPSS Statistics backend if there isn't one already. Plug-ins are freely available from the SPSS community at http://www.ibm.com/developerworks/spssdevcentral.
- Use the IBM SPSS Statistics Batch Facility (a separate executable provided with IBM SPSS Statistics Server) with a command syntax (.sps) file containing the necessary syntax. The IBM SPSS Statistics Batch Facility starts up an instance of the IBM SPSS Statistics backend.

For help getting started with the IBM SPSS Statistics - Integration Plug-in for Python, see *Python Integration Package for IBM SPSS Statistics.pdf*, available from *\documentation\PythonPackage*. You may also want to consult *Programming and Data Management for IBM SPSS Statistics* available in PDF from the Articles page at http://www.ibm.com/developerworks/spssdevcentral.

For help getting started with the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET, see .

For information about using the Batch Facility, see the *IBM SPSS Statistics Batch Facility User's Guide*, provided in PDF with the IBM SPSS Statistics Server product DVD/CD.

GET DATA /TYPE=BINARYSTREAM

The following syntax diagram shows all of the subcommands, keywords, and specifications available for the `GET DATA` command with `/TYPE=BINARYSTREAM`.

```
GET DATA /TYPE=BINARYSTREAM

 { /PIPE='name' [REWINDABLE='name2' [SEEKABLE] ]                 }

 { /SOCKET=portnum [HOST={localhost**} ] [REWINDABLE [SEEKABLE] ] }
                         {'hostname' }

 /VARIABLES = varname format ...
```

** Default if the keyword is omitted.

Example

```
GET DATA /TYPE=BINARYSTREAM
   /PIPE='\\.\pipe\datapipe'
   /VARIABLES=V1 F3.2 V2 F6.4 V3 DOLLAR9.2 V4 ADATE10 V5 A33.
```

Overview

The `GET DATA` command with `/TYPE=BINARYSTREAM` reads records of cases from a pipe or socket, allowing you to stream data into IBM SPSS Statistics from an external processor. In the discussion that follows, the provider of the data is referred to as the **stream provider** and **data source** refers to the data that it provides. You can stream data values from the active dataset in IBM SPSS Statistics to a pipe or socket using the `SAVE` or `XSAVE` commands with `/TYPE = BINARYSTREAM`. See the topic "SAVE, XSAVE /TYPE=BINARYSTREAM" on page 43 for more information.

OptionsBasic Specification

Pipe. The source of case records can be a named pipe. The pipe must be created by an external program prior to executing `GET DATA /TYPE=BINARYSTREAM`. The pipe option assumes that the stream provider and IBM SPSS Statistics are on the same machine.

Socket. The source of case records can be the port number of a TCP socket. The TCP socket must exist in the listening state and be ready to accept a connection prior to executing `GET DATA /TYPE=BINARYSTREAM`. The socket option allows for the stream provider and IBM SPSS Statistics to be on different machines.

The basic specification for `GET DATA /TYPE=BINARYSTREAM` is a named pipe or the port number and hostname of a socket, and the name and format for at least one variable.

- In the absence of the `REWINDABLE` keyword the data source provides only one pass of the data.

Syntax Rules

- Subcommand names, keywords, and options must be spelled in full.
- Either the `PIPE` or `SOCKET` subcommand, but not both, must be specified.
- An error occurs if the `SEEKABLE` keyword is specified in the absence of the `REWINDABLE` keyword.

Operations

- `GET DATA /TYPE=BINARYSTREAM` does not initiate the reading of data from the specified pipe or socket. Data are read as needed by procedures that follow `GET DATA /TYPE=BINARYSTREAM`, or by a following `SAVE` command.
- If `REWINDABLE` is specified, IBM SPSS Statistics will send a rewind command to the stream provider when it requires that the provider rewind to the beginning of the data source. In the absence of

REWINDABLE, IBM SPSS Statistics will not send rewind commands to the stream provider and will be unable to rewind the data source; any subsequent data requests to this data source will yield errors.
- If SEEKABLE is specified, IBM SPSS Statistics will send a seek command to the stream provider when it requires data for a specific case number. In the absence of SEEKABLE, IBM SPSS Statistics will not send seek commands to the stream provider.
- An error occurs if the pipe or socket connection cannot be established or cannot provide case records when needed.
- An error occurs if REWINDABLE is specified and the pipe specified for rewind or the full duplex socket connection cannot be established or cannot be maintained through the life of the data source.

Examples

Streaming Data From a Pipe

```
GET DATA /TYPE=BINARYSTREAM
   /PIPE='\\.\pipe\datapipe' REWINDABLE='\\.\pipe\rewpipe'
   /VARIABLES=V1 F3.2 V2 F6.4 V3 DOLLAR9.2 V4 ADATE10 V5 A33.
```

- The PIPE subcommand specifies that the stream provider will provide cases of data on the pipe named \\.*pipe**datapipe*.
- REWINDABLE specifies that the stream provider allows the data source to be rewound and is listening for rewind commands on the pipe named \\.*pipe**rewpipe*.
- The VARIABLES subcommand defines the variables that will be populated by the streamed data.

Streaming Data From a Socket

```
GET DATA /TYPE=BINARYSTREAM
   /SOCKET=5764 HOST='myhost.mycompany.com' REWINDABLE SEEKABLE
   /VARIABLES=V1 F3.2 V2 F6.4 V3 DOLLAR9.2 V4 ADATE10 V5 A33.
```

- The SOCKET subcommand specifies that the stream provider will provide cases of data on the socket on port 5764 on the host *myhost.mycompany.com*.
- REWINDABLE specifies that the socket is full duplex and that the stream provider allows the data source to be rewound and is listening for rewind commands on the specified socket.
- SEEKABLE specifies that the stream provider will listen for, and honor, a seek command from IBM SPSS Statistics specifying a particular case number.

PIPE Subcommand

The PIPE subcommand specifies a named pipe on which the stream provider will provide case data to IBM SPSS Statistics. Use of a pipe assumes that the stream provider and IBM SPSS Statistics are on the same machine.
- The pipe must be created by an external program prior to executing this command.
- The name of the pipe must be enclosed in quotes. You cannot assign a file handle to a pipe.
- Since, for pipes, IBM SPSS Statistics and the stream provider are on the same machine, they are both assumed to be in the same endian. The binary format of the pipe stream is as described in .

REWINDABLE. *Option to specify that the stream provider supports multiple data passes.* Use of this option requires a second named pipe (enclose the name in quotes) on which the stream provider is ready to listen for, and honor, a rewind command from IBM SPSS Statistics. In the absence of this option, IBM SPSS Statistics will not attempt to rewind the data source; only one pass of the data is provided and the stream provider is free to close the named pipe after the last case.
- The rewind command is the null-terminated character message: "RWND". After receiving a rewind command, the next case that the stream provider provides must be the first case in the data source.
- The stream provider must also listen on the rewind pipe for, and honor, a close command from IBM SPSS Statistics. The close command is the null-terminated character message: "STOP". IBM SPSS Statistics sends the close command when it is finished with the data source.
- The rewind pipe must be created by an external program prior to executing this command.

SEEKABLE. *Option to specify that the stream provider supports the seek operation in conjunction with multiple data passes.* This option specifies that the stream provider will listen for, and honor, a seek command from IBM SPSS Statistics on the rewind pipe. In the absence of this option, IBM SPSS Statistics will not send seek commands to the stream provider.

- To specify `SEEKABLE`, `REWINDABLE` must also be specified.
- The seek command is the null-terminated character string message: "SEEK n" and is sent to the stream provider on the rewind pipe. After receiving a seek command, the next case provided must be case number n, where the first case is defined as case number 1.

SOCKET Subcommand

The `SOCKET` subcommand specifies the port number of a TCP socket on which the stream provider will provide case data to IBM SPSS Statistics.

- The TCP socket must already be in the listening state, ready to accept a connection, prior to executing this command.
- For endian detection by IBM SPSS Statistics, the socket stream provider must write a 32-bit integer value (between 1 and 255) to the socket prior to sending any data, and only send this integer once.
- Upon executing this command, IBM SPSS Statistics will read the first 4 bytes of data from the socket to determine if the host is big or little endian. IBM SPSS Statistics will perform byte swapping, if necessary. The remainder of the stream is as described in .
- You cannot assign a file handle to the port number of a TCP socket.

HOST. *Optional host name of the stream provider.* This option specifies the server machine on which the TCP socket exists. In the absence of the `HOST` option, the local computer, also referred to as *localhost*, is assumed.

REWINDABLE. *Option to specify that the stream provider supports multiple data passes.* This option specifies that the socket is full duplex and that the stream provider is ready to listen for, and honor, a rewind command from IBM SPSS Statistics. In the absence of this option, IBM SPSS Statistics will not attempt to rewind the data source; only one pass of the data is provided and the stream provider is free to close the socket after the last case.

- The rewind command is the null-terminated character message: "RWND". After receiving a rewind command, the next case that the stream provider provides must be the first case in the data source.
- The stream provider must also listen on the socket for, and honor, a close command from IBM SPSS Statistics. The close command is the null-terminated character message: "STOP". IBM SPSS Statistics sends the close command when it is finished with the data source.

SEEKABLE. *Option to specify that the stream provider supports the seek operation in conjunction with multiple data passes.* This option specifies that the stream provider will listen for, and honor, a seek command from IBM SPSS Statistics. In the absence of this option, IBM SPSS Statistics will not send seek commands to the stream provider.

- To specify `SEEKABLE`, `REWINDABLE` must also be specified.
- The seek command is the null-terminated character string message: "SEEK n". After receiving a seek command, the next case provided must be case number n, where the first case is defined as case number 1.

VARIABLES Subcommand

The `VARIABLES` subcommand is required and specifies the properties of the variables in the active dataset to be created from the streamed data. At least one variable must be specified.

- The order of the variables specified on the `VARIABLES` subcommand is the order in which they will be populated from the streaming data.
- Each variable specified on the `VARIABLES` subcommand must have a name and a format (used to specify type of variable, numeric or string, and the input format). Variable names must conform to IBM

SPSS Statistics variable naming rules and must be unique. All IBM SPSS Statistics input formats are supported. Variable naming rules as well as IBM SPSS Statistics variable formats are described in the "Universals" chapter of the *Command Syntax Reference*, available in the Help system or in PDF form from the Help menu.

Common Binary Format of the Stream

Case data is written as a packed stream of 8 byte entities such that each case is represented by the same number of bytes. Cases and variables within cases are written in the order specified by the active data set, honoring the effects of any `KEEP` and `DROP` subcommands.

- Each numeric variable is a double, 8 bytes wide, and in the endian of the IBM SPSS Statistics backend. The IBM SPSS Statistics backend does not perform byte swapping. Examples of numeric variables include, but are not limited to, those with formats `Fn.d`, `DATEn`, `ADATEn`, or `DOLLARn.d`.
- Variables with a date or date and time format, represent the number of seconds from 0 hours, 0 minutes, and 0 seconds on Oct. 14, 1582.
- Missing values for numeric variables are specified by setting the value to the most negative normalized double supported by IEEE double floating point (big endian representation: X'FFEFFFFF FFFFFFFF') or to an invalid IEEE floating point double.
- String variables in IBM SPSS Statistics can have widths from 1 to 32767 characters, however, they are written to the stream in widths that are integer multiples of 8 bytes, large enough to hold the defined length of the string variable, and padded with spaces if needed. For example, the width of the buffer provided for a string variable defined as `A7` will be 8 bytes with the last byte set to a space (ASCII code decimal 32), whereas the buffer for a string variable defined as `A12` will be 16 bytes with the last 4 bytes set to spaces. The strings are multibyte strings and in the same locale as the IBM SPSS Statistics backend.
- Missing values for string variables are specified by setting all characters in the string to spaces (ASCII code decimal 32).

SAVE, XSAVE /TYPE=BINARYSTREAM

The following syntax diagrams show all of the subcommands, keywords, and specifications available for the `SAVE` and `XSAVE` commands with `/TYPE=BINARYSTREAM`.

```
SAVE /TYPE = BINARYSTREAM

 {/PIPE='name'   }
 {/SOCKET=portnum}

 [/UNSELECTED = [ {RETAIN**} ] ]
                  {DELETE  }

 [/KEEP = {ALL**  } ] [/DROP = varlist]
          {varlist}
```

```
XSAVE /TYPE = BINARYSTREAM

 {/PIPE='name'   }
 {/SOCKET=portnum}

 [/KEEP = {ALL**  } ] [/DROP = varlist]
          {varlist}
```

** Default if the subcommand is omitted.

Example

```
SAVE /TYPE=BINARYSTREAM
    /PIPE='\\.\pipe\datapipe'.
```

Overview

The `SAVE` and `XSAVE` commands with `/TYPE=BINARYSTREAM` write cases from the active dataset to a specified pipe or socket. The `SAVE` command reads the active dataset, causes execution of any pending commands, and causes the data to be written to the specified pipe or socket. The `XSAVE` command does not read the active dataset; it is stored, pending execution with the next command that reads the dataset. In the discussion that follows, the consumer of the case data is referred to as the **stream reader**. You can stream data values into IBM SPSS Statistics from a pipe or socket using `GET DATA /TYPE = BINARYSTREAM`. See the topic "GET DATA /TYPE=BINARYSTREAM" on page 40 for more information.

OptionsBasic Specification

Pipe. The destination for case records can be a named pipe to be created by IBM SPSS Statistics. The pipe option assumes that the stream reader and IBM SPSS Statistics are on the same machine.

Socket. The destination for case records can be the port number of a TCP socket to which IBM SPSS Statistics binds, and waits for a connection. The socket option allows for the stream reader and IBM SPSS Statistics to be on different machines.

The basic specification for `SAVE /TYPE=BINARYSTREAM` or `XSAVE /TYPE=BINARYSTREAM` is a pipe name or socket port number.

Syntax Rules
- Subcommand names, keywords, and options must be spelled in full.
- Either the `PIPE` or `SOCKET` subcommand, but not both, must be specified.
- `XSAVE /TYPE=BINARYSTREAM` cannot appear within a `DO REPEAT—END REPEAT` structure.
- Multiple `XSAVE /TYPE=BINARYSTREAM` commands writing to the same pipe or socket are not allowed.

Operations
- The `PIPE` subcommand creates the named pipe, writes all of the specified cases to the pipe, and then closes the pipe.
- The `SOCKET` subcommand creates one listening socket, waits for one TCP connection, writes all of the specified cases to the socket, and then closes the socket.

Example

```
SAVE /TYPE=BINARYSTREAM
   /PIPE='\\.\pipe\datapipe'
   /DROP=GRADE STORE
   /KEEP=LNAME NAME TENURE JTENURE ALL.
```

- The `PIPE` subcommand specifies that IBM SPSS Statistics will write cases from the active dataset to the pipe named *\\.\pipe\datapipe*.
- `DROP` excludes variables *GRADE* and *STORE* from the written stream. `KEEP` specifies that *LNAME*, *NAME*, *TENURE*, and *JTENURE* are the first four variables written to the stream (for each case), followed by all remaining variables not specified on *DROP*. These remaining variables are written to the stream in the same order as they occur in the active dataset.

PIPE Subcommand

The `PIPE` subcommand specifies a named pipe to be created by IBM SPSS Statistics and to which IBM SPSS Statistics will write case data. Use of a pipe assumes that IBM SPSS Statistics and the stream reader are on the same machine.
- The name of the pipe must be enclosed in quotes. You cannot assign a file handle to a pipe.
- Both forward slashes and backslashes can be used as path separators in the pipe name.
- Since, for pipes, IBM SPSS Statistics and the stream reader are on the same machine, they are both assumed to be in the same endian. The binary format of the pipe stream created by IBM SPSS Statistics is as described in .

SOCKET Subcommand

The `SOCKET` subcommand specifies the port number for which IBM SPSS Statistics will create a TCP socket in the listening state. IBM SPSS Statistics will wait for a connection, write a 32-bit integer number (between 1 and 255) for endian detection, and then write case data to the socket with no byte swapping. The stream reader is required to detect endian and perform byte-swapping, if necessary.

- You cannot assign a file handle to the port number of a TCP socket.
- Following the initial 32-bit integer used for endian detection, the remainder of the stream created by IBM SPSS Statistics is as described in .

UNSELECTED Subcommand

The `UNSELECTED` subcommand, which applies to `SAVE` but not `XSAVE`, determines whether cases excluded by a previous `FILTER` or `USE` command are to be retained or deleted in the output stream. The default is `RETAIN`.

- The `UNSELECTED` subcommand has no effect when the active dataset does not contain unselected cases.

RETAIN. *Retain the unselected cases*. All cases in the active dataset are written to the stream. This setting is the default when `UNSELECTED` is specified by itself.

DELETE. *Delete the unselected cases*. Only cases that meet the `FILTER` or `USE` criteria currently in effect are written to the stream.

DROP and KEEP Subcommands

`DROP` and `KEEP` are used to specify a subset of variables to be written to the stream. `DROP` specifies the variables that are not to be included in the output stream. `KEEP` specifies the variables that are to be included in the output stream. When `KEEP` is specified, variables that are not named on `KEEP` are dropped.

- Variables can be specified in any order. The order of variables on `KEEP` determines the order of variables in the output stream. The order on `DROP` does not affect the order of variables in the stream.
- The keyword `ALL` on `KEEP` refers to all remaining variables that were not previously specified on `KEEP`. `ALL` must be the last specification on `KEEP`.
- If a variable is specified twice on the same `DROP` or `KEEP` subcommand, only the first mention is recognized.
- Multiple `DROP` and `KEEP` subcommands are allowed. If a variable is specified that is not in the active dataset or that has been dropped because of a previous `DROP` or `KEEP` subcommand, an error results, and the `SAVE` or `XSAVE` command is not executed.
- The keyword `TO` can be used to specify a group of consecutive variables in the active dataset.

Common Binary Format of the Stream

Case data is written as a packed stream of 8 byte entities such that each case is represented by the same number of bytes. Cases and variables within cases are written in the order specified by the active data set, honoring the effects of any `KEEP` and `DROP` subcommands.

- Each numeric variable is a double, 8 bytes wide, and in the endian of the IBM SPSS Statistics backend. The IBM SPSS Statistics backend does not perform byte swapping. Examples of numeric variables include, but are not limited to, those with formats `Fn.d`, `DATEn`, `ADATEn`, or `DOLLARn.d`.
- Variables with a date or date and time format, represent the number of seconds from 0 hours, 0 minutes, and 0 seconds on Oct. 14, 1582.
- Missing values for numeric variables are specified by setting the value to the most negative normalized double supported by IEEE double floating point (big endian representation: X'FFEFFFFF FFFFFFFF') or to an invalid IEEE floating point double.
- String variables in IBM SPSS Statistics can have widths from 1 to 32767 characters, however, they are written to the stream in widths that are integer multiples of 8 bytes, large enough to hold the defined length of the string variable, and padded with spaces if needed. For example, the width of the buffer

provided for a string variable defined as `A7` will be 8 bytes with the last byte set to a space (ASCII code decimal 32), whereas the buffer for a string variable defined as `A12` will be 16 bytes with the last 4 bytes set to spaces. The strings are multibyte strings and in the same locale as the IBM SPSS Statistics backend.

- Missing values for string variables are specified by setting all characters in the string to spaces (ASCII code decimal 32).

Chapter 4. Creating an Integration Plug-in

An **integration plug-in** is code that you write that:

- Invokes IBM SPSS Statistics from an external processor.
- Supports communication between IBM SPSS Statistics and the external processor.
- Optionally invokes an external processor from IBM SPSS Statistics and passes language statements (written in the external language), contained in a `BEGIN PROGRAM <language> - END PROGRAM` block of command syntax, to the external processor. For example, if you create a plug-in for the Basic language then you can wrap Basic statements in a `BEGIN PROGRAM BASIC - END PROGRAM` block in command syntax. When you run the syntax, the Basic statements will be executed by the Basic processor.

We recommend that your plug-ins be at the same release level as the IBM SPSS Statistics application software, although they may work with versions as low as 14.0.

Planning

Before you start writing code, consider:

- **Language type.** You can create plug-ins for the IBM SPSS Statistics Programmability Extension in any language that can call a C API. The optional feature of invoking an external language from IBM SPSS Statistics is designed for interpreted or dynamically compiled languages.
- **Processes and threads.** If you are building a plug-in that will only invoke IBM SPSS Statistics from an external processor and are using the in-process mode, then you should be aware that there is one process with two threads, one for the IBM SPSS Statistics application and one for your external processor. The IBM SPSS Statistics application is single-threaded—only one instance of IBM SPSS Statistics can be running in the process. Because the IBM SPSS Statistics application and your implementation operate in the same process, they may share the runtime library. Be sure that your implementation does not make changes to the state of the runtime that would affect IBM SPSS Statistics. For example, a C `setlocale` call to set the process locale would change the locale in which IBM SPSS Statistics is running, but it would not completely effect this change because IBM SPSS Statistics needs to do additional tasks when the locale changes.
- **Synchronization.** When using the in-process mode, the two threads execute synchronously. Only one thread is executing at a given time. The other thread waits for execution to complete.
- **Resource utilization.** The IBM SPSS Statistics application can have a large footprint. The amount of computing resources (for example RAM, CPU time and disc space) that it requires in order to operate varies according to what it is doing. The XD API has a small footprint.
- **Functions.** Decide which XD API functions you need to use. For an introduction to the available functions, see Chapter 5, "Introduction to the XD API," on page 53.

Invoking IBM SPSS Statistics from an External Processor

You invoke IBM SPSS Statistics from an external processor through a library that calls the XD API. For example, this can be a C++ interface to the low-level functions that the C API in *spssxd_p.dll* or *spssxd_p.dll* implements. We generically refer to this library as *MyLanguageInvokeXD*. The sequence of actions for invoking IBM SPSS Statistics in this manner is as follows:

1. *MyLanguageInvokeXD* loads.
2. *MyLanguageInvokeXD* determines the path to *spssxd_p.dll* or *spssxd_p.dll* and loads the library.

The mechanism for determining the path might involve reading from a configuration file or may involve reading from the registry. Ultimately the path must initially be determined when a user installs a plug-in or an application.

The code that you write to launch IBM SPSS Statistics from an external processor needs to include, at a minimum, functions to:

- Load and unload *spssxd_p.dll* or *spssxd_p.dll*.
- Submit command syntax to IBM SPSS Statistics for processing.
- Get information from your external processor to the IBM SPSS Statistics application.

Building MyLanguageInvokeXD

To build *MyLanguageInvokeXD* you use the *spssxd.h* and *spssxd.lib* (or *spssxd_p.lib*) files which are available from \XD_API in the directory where the IBM SPSS Statistics - Programmability SDK is installed. An implementation of *MyLanguageInvokeXD* can only be used with a version of *spssxd_p.dll* (or *spssxd_p.dll*) that matches the version of IBM SPSS Statistics associated with the IBM SPSS Statistics - Programmability SDK used to build *MyLanguageInvokeXD*. For more information, see the section labeled *Versions* in .

Python Example

The IBM SPSS Statistics - Programmability SDK includes *PyInvokeSpss* as an example implementation of how to invoke IBM SPSS Statistics from Python. The source files and script to build *PyInvokeSpss* are included in the IBM SPSS Statistics - Programmability SDK in \Statistics_Python. The readme file in that directory provides further information about building *PyInvokeSpss*.

The Path to spssxd_p.dll or spssxd_p.dll

The Python example utilizes the configuration file *spssxdcfg.ini* (a template is available in \Statistics_Python\spss) to determine the path to *spssxd_p.dll* or *spssxd_p.dll*. Specifically, the path is stored as the value of *spssxd_path* in this file. Regardless of the language and implementation of the plug-in, it is the responsibility of the program that installs the plug-in to query the user for the path to the desired installation of IBM SPSS Statistics (*spssxd_p.dll* and *spssxd_p.dll* are located in the IBM SPSS Statistics application installation directory) and to store that value so that it can be retrieved at run-time.

Invoking an External Processor from IBM SPSS Statistics

To invoke the processor for an external language from IBM SPSS Statistics, you will need to create the following:

- A library that launches and initializes the processor. We generically refer to this library as *InvokeMyLanguage*. An external processor invoked from IBM SPSS Statistics should provide access to all the functionality of the external programming language—in some programming languages this is called **embedding**. Details on creating the library are provided in the topic “Creating an InvokeMyLanguage library.”
- A section in the configuration file *spssdxcfg.ini* (located in the directory where IBM SPSS Statistics is installed) that provides the details needed to invoke the external language processor, such as the name of the library and the path to the home directory for the external processor. Details are provided in the topic “Modifying the DX configuration file” on page 49.

Note: The framework for invoking an external language from IBM SPSS Statistics is designed for interpreted or dynamically compiled languages.

Creating an InvokeMyLanguage library

The InvokeMyLanguage library should include a set of functions that are called by the framework that invokes external language processors from IBM SPSS Statistics. Following are the functions:

int init_embedded_x (int argc, char **argv)

The *init_embedded_x* function initializes the external language. It is only called once during an IBM SPSS Statistics session. The argument *argv* is an array of strings containing the set of values, if any, specified by the ARGS keyword in the *spssdxcfg.ini* configuration file (See the topic "Modifying the DX configuration file" for more information.). The argument *argc* is an integer specifying the number of elements in the string array. A return value of 0 indicates success. *init_embedded_x* is a required function.

int stop_embedded_x ()

The *stop_embedded_x* function stops the external language. It is only called once during an IBM SPSS Statistics session. A return value of 0 indicates success. *stop_embedded_x* is a required function.

int pre_action ()

The *pre_action* function is called prior to the beginning of each `BEGIN PROGRAM <language> - END PROGRAM` block and should handle any actions that are required prior to executing the block. For example, if you intend to capture text output from the external processor then you would typically use the *pre_action* function to handle redirecting the output to a temporary file. A return value of 0 indicates success. *pre_action* is a required function.

int post_action ()

The *post_action* function is called at the completion of each `BEGIN PROGRAM <language> - END PROGRAM` block and should handle any actions that are required after executing the block. For example, if you intend to capture text output from the external processor then you would typically use the *post_action* function to retrieve any output directed to a temporary file and post it to IBM SPSS Statistics. A return value of 0 indicates success. *post_action* is a required function.

int execute_x (char *cmd)

The *execute_x* function accepts a string containing the contents of a `BEGIN PROGRAM <language> - END PROGRAM` block (i.e. statements written in the external language) and passes it to the external processor for execution. The string is null-terminated with multiple lines separated by `"\n"`. A return value of 0 indicates success. *execute_x* is a required function.

void set_post_spss_output (FPPostSpssOutput fpSpssOutput)

The *set_post_spss_output* function is called by the framework in order to pass a pointer to the function that should be used to post text output to IBM SPSS Statistics. Output is posted to a log item in the IBM SPSS Statistics Viewer. The argument *fpSpssOutput* is a function pointer and its function prototype is:

```
typedef int  (*FPPostSpssOutput)(const char* text, int length);.
```

Note that *set_post_spss_output* is an optional function. You can also post text output to IBM SPSS Statistics with the *PostSpssOutput* function in the XD API.

Source files and instructions for building example implementations of *InvokeMyLanguage* for Python can be found in *\DX_API\InvokePython* and *\DX_API\MyInvokePython*, located in the directory where the IBM SPSS Statistics - Programmability SDK is installed. The example in *\DX_API\MyInvokePython* is very basic and simply illustrates how to invoke an external language processor, execute external language statements from `BEGIN PROGRAM <language> - END PROGRAM` blocks and post output to IBM SPSS Statistics; but it does not support utilizing the XD API to interact with IBM SPSS Statistics. The example in *\DX_API\InvokePython* supports invoking the Python processor from IBM SPSS Statistics as well as utilizing the XD API to interact with IBM SPSS Statistics.

Modifying the DX configuration file

The configuration file *spssdxcfg.ini* specifies how to invoke the processor for an external language. The file contains an entry for each external processor that can be invoked from IBM SPSS Statistics. *spssdxcfg.ini* is part of the IBM SPSS Statistics Programmability Extension and is located in the IBM SPSS Statistics application installation directory. It is the responsibility of the installation program for a plug-in to populate this file if the plug-in supports invoking the associated language from IBM SPSS Statistics.

To add support for a new external language:

1. Add the name of the language to the existing list in the `[SUPPORTED_LANG]` section, using a semicolon as a delimiter. The name is case insensitive but cannot contain blank characters. For example, if the existing list is specified as `X=PYTHON;R`, then you add a language named `MYLANGUAGE` by updating the list to:

   `X=PYTHON;R;MYLANGUAGE`.

   Note that the name you specify for the language is the name that users will specify when creating `BEGIN PROGRAM <language> - END PROGRAM` blocks. For the current example, users will create `BEGIN PROGRAM MYLANGUAGE - END PROGRAM` blocks.
2. Add a section for your language to *spssdxcfg.ini*. A template for the contents of the section is as follows:

```
[MYLANGUAGE]
LIB_NAME=
HOME=
ARGS=
```

[MYLANGUAGE]. The name of the external language enclosed in square brackets.

LIB_NAME. The name of the library that provides the adapter for your external processor--what we have generically called *InvokeMyLanguage*. This is required. For example, if the name of the library is *InvokeMyLanguage* then specify: `LIB_NAME=InvokeMyLanguage`.

HOME. An absolute path to the external processor. This is required.

ARGS. A blank separated list of arguments, if any, to pass to the *init_embedded_x* function in the *InvokeMyLanguage* library. Values should be specified as unquoted strings, as in: `ARGS=arg1 arg2 arg3`. The `ARGS` keyword is optional.

You can also add key/value pairs specifying any environment variables that may be needed by your plug-in. In that regard, note that for UNIX platforms (including MacOS), the process in which the external processor runs only inherits the following environment variables: *SPSS_HOME*, *SPSSDXTEMP*, *SPSS_EXTENSIONS_PATH* and the library path (*LD_LIBRARY_PATH* for Linux and Solaris, *LIBPATH* for AIX, and *DYLD_LIBRARY_PATH* for MacOS).

For example, if your plug-in requires libraries distributed with the external processor and the path to those libraries is not in the current library search path, then you would add a library path environment variable to the `[MYLANGUAGE]` section, as in the following for Windows:

```
PATH=<library path>;%PATH%
```

Likewise, for Linux or Solaris you would add:

```
LD_LIBRARY_PATH=<library path>:$LD_LIBRARY_PATH
```

Language-specific Wrappers

For ease of use by applications developers and end users, you may want to consider providing a wrapper to the library that loads and exposes the XD API to the external language, i.e. a wrapper to *MyLanguageInvokeXD*. In this regard, the IBM SPSS Statistics - Programmability SDK includes an example module for Python, *spss.py*, that wraps *PyInvokeSpss*. The source files are included in the IBM SPSS Statistics - Programmability SDK in *\Statistics_Python\spss*.

Setting the IBM SPSS Statistics Locale

When writing language-specific wrappers, you should be aware that the `StartSPSS` function at the level of the XD API does not set the IBM SPSS Statistics locale. In the example Python and .NET plug-ins available from http://www.ibm.com/developerworks/spssdevcentral, the locale for IBM SPSS Statistics is set by the *StartSPSS* function at the level of the wrapper to *MyLanguageInvokeXD*--that is, at the level of *spss.py* for Python, and *SPSS.BackendAPI.Controller.dll* for .NET.

In designing your own plug-ins and optional wrappers, you'll have to decide where best to set the IBM SPSS Statistics locale when starting an IBM SPSS Statistics session. And if you decide to write your own wrapper for *PyInvokeSpss*, or the .NET equivalent *SPSS.BackendAPI.dll*, you'll need to set the IBM SPSS Statistics locale as part of the process of starting an IBM SPSS Statistics session.

Deploying An Integration Plug-in

You can deploy a plug-in that you have developed using the IBM SPSS Statistics - Programmability SDK. Your plug-in can only be used with a licensed IBM SPSS Statistics application. For information about licensing or distribution arrangements, please contact IBM Corp. directly.

In order for your plug-in to run properly on the target computer you must have:
- An IBM SPSS Statistics application
- Your chosen programming language
- Your plug-in files
- Updated configuration files

Programming Language

The programming language that you used to develop your plug-in must be installed on the computer where you will run it. Note that if you are using your plug-in from an IBM SPSS Statistics client which is connected to IBM SPSS Statistics Server, the programming language and plug-in must be installed on the computer on which IBM SPSS Statistics Server is running.

Integration Plug-in

Your plug-in files must be available on the computer where you will run your plug-in.

MyLanguageInvokeXD. The library that loads *spssxd_p.dll* or *spssxd_p.dll* and makes the XD API available in an external language should be installed in a location where your programming language can find it (for example, *C:\Python27\Lib\site-packages\MyPackage*).

InvokeMyLanguage. The optional library that invokes an external processor from IBM SPSS Statistics must be installed in the IBM SPSS Statistics application’s root directory. Note that the IBM SPSS Statistics Programmability Extension files—*spssxd_p.dll*, *spssxd_p.dll*, and *spssdxcfg.ini*—are installed in this directory as part of the IBM SPSS Statistics application installation.

Language-specific wrappers. Wrappers to *MyLanguageInvokeXD* should be installed in a location where your programming language can find them (for example, *C:\Python27\Lib\site-packages\MyPackage*).

Configuration File Updates

If your plug-in invokes an external language processor from IBM SPSS Statistics, make sure to update the configuration file *spssdxcfg.ini* so that the appropriate libraries can be located by your plug-in. Be sure to update the path settings for the value of *HOME* in *spssdxcfg.ini* to be an absolute path to the external processor.

Versions

To build *MyLanguageInvokeXD* you use the *spssxd.h* and *spssxd.lib* (or *spssxd_p.lib*) files which are included in the IBM SPSS Statistics - Programmability SDK. The version of *spssxd_p.dll* (or *spssxd_p.dll*) on the target machine needs to be the same as the version of *spssxd.h* and *spssxd.lib* (or *spssxd_p.lib*) used to build *MyLanguageInvokeXD*. You can check the version of *spssxd_p.dll* and *spssxd_p.dll*, on the target machine, from the value of *SpssdxVersion* in *spssdxcfg.ini*, located in the IBM SPSS Statistics application's root directory. The copies of *spssxd.h* , *spssxd.lib*, and *spssxd_p.lib* included with the IBM SPSS Statistics - Programmability SDK are compatible with the version of IBM SPSS Statistics given in the readme file located at the root of the IBM SPSS Statistics - Programmability SDK installation directory.

Documentation

You can deploy IBM SPSS Statistics Programmability Extension documentation with your plug-in, including:

Dictionary schema documentation. The dictionary schema is distributed with the IBM SPSS Statistics application, in *dictionary-1.xsd*, and is also available from http://xml.spss.com/spss/data/. Documentation is included in the IBM SPSS Statistics - Programmability SDK and accessible from *\documentation\DictionarySchema\dictionary_schema_intro.htm*.

Output schema documentation. The output schema is distributed with your IBM SPSS Statistics application, in *spss-output-1.8.xsd*, and is also available from http://xml.spss.com/spss/oms/. Documentation is included in the IBM SPSS Statistics - Programmability SDK and accessible from *\documentation\OutputSchema\oms_oxml_schema_intro.htm*.

Plug-in documentation. You may want to write documentation for your plug-in. You can use *\documentation\PythonPackage\Python Integration Package for IBM SPSS Statistics.pdf* as an example.

Programmability SDK documentation. You can distribute this document with your plug-in.

Chapter 5. Introduction to the XD API

An overview of the functions available with the XD API, along with examples of using them to accomplish basic tasks follows. Complete documentation of the XD API is included in the IBM SPSS Statistics - Programmability SDK and accessible from *\documentation\XDAPI\index.html*.

The XD API in *spssxd_p.dll* contains low-level functions which you can sequence to:
- Submit requests for actions to IBM SPSS Statistics.
- Submit requests for information to IBM SPSS Statistics.

Typically, the return value from an action request is an error code. Typically, the return value from an information request is the information requested.

Functions operate on IBM SPSS Statistics, the variable dictionary, the XML workspace, and the active dataset.

Controlling IBM SPSS Statistics

With programmability you can use an external programming language to control IBM SPSS Statistics. Functions include:

StartSPSS. Action request that starts the IBM SPSS Statistics backend.
- Note that this function does not set the IBM SPSS Statistics locale. You must set the appropriate locale (after starting the backend) with `set locale` command syntax executed with the `Submit` function. An example of command syntax to set the locale is: `set locale = 'English_United States.1252'`.
- Starting with version 21, IBM SPSS Statistics operates in Unicode mode by default. If you need to work in code page mode, you can use the `Submit` function with `SET UNICODE=NO` syntax.

StopSPSS. Action request that causes the backend to shutdown when it was started from an external application. The function has no effect when IBM SPSS Statistics is the main program.

Submit. Action request that queues a line of syntax command text and executes all of the currently queued lines.

QueueCommandPart. Action request that queues a line of syntax command text but does not execute it.

PostSpssOutput. Action request that adds a line to the IBM SPSS Statistics log.

IsBackendReady. Information request used to determine if IBM SPSS Statistics is ready to receive commands.

IsXDriven. Information request used to determine whether the backend is being driven by an external processor.

IsUTF8mode. Information request to determine whether IBM SPSS Statistics is running in Unicode mode or code page mode.

GetGraphic. Action request to display an R-style graphic in the IBM SPSS Statistics Viewer.

GetSPSSLocale. Information request to get the current locale of the IBM SPSS Statistics processor.

GetOutputLanguage. Action request to get the output language of the IBM SPSS Statistics processor.

SetOutputLanguage. Action request to set the output language of the IBM SPSS Statistics processor.

GetFileHandles. Action request to get any defined file handles.

GetOMStagList. Information request that returns a list of tags associated with any active OMS requests.

GetSetting. Information request that returns the value of an options setting, as set from the `SET` command.

EncryptTempFiles. Action request that specifies whether temp files are encrypted. By default, temp files are not encrypted.

Variable Dictionary Information

A variable dictionary describes the contents of a dataset. Two sets of functions are available for retrieving dictionary information; one set for retrieving information from the active dataset, and another set for retrieving information from a cursor. While a cursor is open, both sets of functions return information about the current cursor and give identical results. Functions that only retrieve information from the current cursor have a suffix of *InProcDS*. Cursors are created with the `MakeCaseCursor` function and provide both read and write access to the active dataset.

GetCValueLabels. Information request that returns the value labels for a specified string variable.

GetNValueLabels. Information request that returns the value labels for a specified numeric variable.

GetRowCount (GetRowCountInProcDS). Information request that returns the number of cases in the active dataset (or current cursor).

GetVarAttributeNames (GetVarAttributeNamesInProcDS). Information request that returns the names of all variable attributes for a specified variable.

GetVarAttributes (GetVarAttributesInProcDS). Information request that returns the attribute values for a specified variable and variable attribute.

GetVarCMissingValues (GetVarCMissingValuesInProcDS). Information request that returns the user-missing values for a specified string variable.

GetVarNMissingValues (GetVarNMissingValuesInProcDS). Information request that returns the user-missing values for a specified numeric variable.

GetVariableCount (GetVariableCountInProcDS). Information request that returns the number of variables in the active dataset (or current cursor).

GetVariableFormat (GetVariableFormatInProcDS). Information request that returns the display format of a specified variable.

GetVariableLabel (GetVariableLabelInProcDS). Information request that returns the Label of a specified variable.

GetVariableMeasurementLevel (GetVariableMeasurementLevelInProcDS). Information request that returns the measurement level (nominal, ordinal, scale, or unknown) of a specified variable.

GetVariableName (GetVariableNameInProcDS). Information request that returns the name of a specified variable. Variables are specified by an index value representing position in the active dataset (or current cursor), starting with 0 for the first variable in file order.

GetVariableRole (GetVariableRoleInProcDS). Information request that returns the variable role for a specified variable. Variables are specified by an index value representing position in the active dataset (or current cursor), starting with 0 for the first variable in file order.

GetVariableType (GetVariableTypeInProcDS). Information request that returns the variable type (numeric or string) for a specified variable.

GetWeightVar. Information request that returns the name of the weight variable, if any, in the active dataset.

GetSPSSLowHigh. Information request that returns the values IBM SPSS Statistics uses for LO and HI.

GetSystemMissingValue. Information request that returns the system-missing value.

GetMultiResponseSetNames (GetMultiResponseSetNamesInProcDS). Information request that returns the names of any multiple response sets for the active dataset.

GetMultiResponseSet (GetMultiResponseSetInProcDS). Information request that returns the details of a specified multiple response set.

SetVarRole. Action request that sets the variable role for a specified variable.

XML Workspace

The **XML workspace** is an area in memory where you can store and then retrieve both dictionary information and syntax command output. A variable dictionary describes the contents of a dataset. To get dictionary information into the XML workspace, use the `CreateXPathDictionary` function. Use this when you want access to dictionary information that isn't directly accessible with the dictionary information request functions listed above. IBM SPSS Statistics output includes statistical results, tables, charts, trees, and text. To get output into the XML workspace, use the `OMS` syntax command with the `XMLWORKSPACE` destination. See the topic "Retrieving Output from Syntax Commands" on page 69 for more information.

XML workspace functions include:

CreateXPathDictionary. Action request that creates an XPath DOM from the active dataset's dictionary and adds it to the XML workspace with the given handle.

EvaluateXPath. Information request that evaluates the XPath expression in the given context against a specified XPath DOM--either a dictionary DOM or an output DOM. Dictionary DOM's have a structure that conforms to the dictionary schema published at http://xml.spss.com/spss/data/, and output DOM's have a structure that conforms to the output schema published at http://xml.spss.com/spss/oms/. Documentation for the dictionary and output schemas is included in the IBM SPSS Statistics - Programmability SDK and accessible from *\documentation\DictionarySchema\dictionary_schema_intro.htm* and *\documentation\OutputSchema\oms_oxml_schema_intro.htm* respectively.

GetStringFromList. Information request that gets a single string from the list returned from EvaluateXPath.

GetStringListLength. Information request that returns the length of the list found with EvaluateXPath.

GetHandleList. Information request that gets the list of handles of the objects currently in the XML workspace.

RemoveStringList. Action request that deletes the storage associated with the string list found with EvaluateXPath.

RemoveXPathHandle. Action request that removes an object from the XML workspace, freeing the storage.

GetXmlUtf16. Information request that gets XML from the XML workspace and returns a Utf16 representation of it.

GetImage. Information request that returns a pointer to an image generated by a syntax command and stored in the XML workspace.

Getting Case Data, Adding New Variables, and Appending Cases to the Active Dataset

The active dataset contains variables (columns) and cases (rows). Case data functions for the active dataset include:

MakeCaseCursor. Action request that creates a data cursor. Cursors have three modes: read, write, and append.

HasCursor. Information request that returns an integer indicating if a cursor is running.

NextCase. Action request that moves the cursor to the next case.

IsEndSplit. Information request that returns an integer indicating if the cursor position has crossed a split boundary.

ResetDataPass. Action request to reset the cursor to the first case.

GetCursorPosition. Information request that gets the row index of the cursor.

GetNumericValue. Information request that gets a numeric data value and missing status for a specific variable and case.

GetStringValue. Information request that gets a string data value and missing status for a specific variable and case.

RemoveCaseCursor. Action request to free the storage for a case.

AllocNewVarsBuffer. Action request to specify the buffer size, in bytes, to use when adding new variables in the context of multiple data passes.

CommitHeader. Action request used in write mode to commit the dictionary information for new variables to the current cursor.

CommitCaseRecord. Action request used in write mode to commit changes to the current case in the current cursor.

CommitNewCase. Action request used in append mode to commit data for a new case to the current cursor.

EndChanges. Action request used in append mode to signal the end of appending new cases.

SetOneVarNameAndType. Action request used in write mode to create a single new variable in the active dataset. This function is intended to be used in conjunction with the AllocNewVarsBuffer function.

SetValueChar. Action request used in write mode or append mode to set the value for the current case for a string variable.

SetValueNumeric. Action request used in write mode or append mode to set the value for the current case for a numeric variable.

SetVarAlignment. Action request used in write mode to set the variable alignment property for a new variable.

SetVarAttributes. Action request used in write mode to set a value in an attribute array for a new variable.

SetVarCMissingValues. Action request used in write mode to specify user-missing values for a new string variable.

SetVarCValueLabel. Action request used in write mode to specify value labels for a new string variable.

SetVarFormat. Action request used in write mode to specify the display format for a new variable.

SetVarLabel. Action request used in write mode to specify the variable label for a new variable.

SetVarMeasureLevel. Action request used in write mode to specify the measurement level (nominal, ordinal, or scale) for a new variable.

SetVarNameAndType. Action request used in write mode to create new variables in the active dataset.

SetVarNMissingValues. Action request used in write mode to specify user-missing values for a new numeric variable.

SetVarNValueLabel. Action request used in write mode to specify value labels for a new numeric variable.

ValidateVarName. Information request used to validate a potential variable name.

Creating and Accessing Multiple Datasets

In addition to accessing the active dataset, you can concurrently access multiple open datasets as well as create new datasets. This allows you to create one or more new datasets from existing datasets, combining the data from the existing datasets in any way you choose. It also allows you to concurrently modify multiple datasets. The environment that allows you to create new datasets and access multiple datasets is initiated with the `StartDataStep` function. Functions for creating and accessing multiple datasets include:

StartDataStep. Action request that initiates a data step.

EndDataStep. Action request that ends a data step.

CreateDataset. Action request that provides access to an open dataset or creates a new empty dataset.

SetDatasetName. Action request to rename a dataset. Acts on a dataset object created by the `CreateDataset` function.

GetNewDatasetName. Information request to get the name of a newly created dataset.

GetActive. Information request to get the name of the active dataset.

SetActive. Action request to set the active dataset.

CopyDataset. Action request to create a deep copy of a dataset.

GetSpssDatasets. Information request to return a list of the names of the currently open datasets.

GetDatastepDatasets. Information request to return a list of the names of the datasets that are available in the current datastep. These are the datasets accessed with the `CreateDataset` function.

CloseDataset. Action request to close a dataset accessed with the `CreateDataset` function.

InsertVariable. Action request to insert a new variable into a specified dataset. Acts on a dataset object created by the `CreateDataset` function.

DeleteVariable. Action request to delete a specified variable from a specified dataset. Acts on a dataset object created by the `CreateDataset` function.

GetVarCountInDS. Information request to return the number of variables in a specified dataset.

GetVarNameInDS. Information request to return the name of the variable at a specified index position in a specified dataset.

SetVarNameInDS. Action request to set the name of the variable at a specified index position in a specified dataset.

GetVarLabelInDS. Information request to get the label of the variable at a specified index position in a specified dataset.

SetVarLabelInDS. Action request to set the label of the variable at a specified index position in a specified dataset.

GetVarTypeInDS. Information request to get the variable type (numeric or string) of the variable at a specified index position in a specified dataset.

SetVarTypeInDS. Action request to set the variable type of the variable at a specified index position in a specified dataset.

GetVarFormatInDS. Information request to get the display format of the variable at a specified index position in a specified dataset.

SetVarFormatInDS. Action request to set the display format of the variable at a specified index position in a specified dataset.

GetVarAlignmentInDS. Information request to get the alignment (left, right, or center) of the variable at a specified index position in a specified dataset.

SetVarAlignmentInDS. Action request to set the alignment of the variable at a specified index position in a specified dataset.

GetVarMeasurementLevelInDS. Information request to get the measurement level (nominal, ordinal, scale, or unknown) of the variable at a specified index position in a specified dataset.

SetVarMeasurementLevelInDS. Action request to set the measurement level (nominal, ordinal, or scale) of the variable at a specified index position in a specified dataset.

GetVarNMissingValuesInDS. Information request to get the missing values of the numeric variable at a specified index position in a specified dataset.

SetVarNMissingValuesInDS. Action request to set the missing values of the numeric variable at a specified index position in a specified dataset.

GetVarCMissingValuesInDS. Information request to get the missing values of the string variable at a specified index position in a specified dataset.

SetVarCMissingValuesInDS. Action request to set the missing values of the string variable at a specified index position in a specified dataset.

GetVarAttributesNameInDS. Information request to get the names of the custom variable attributes associated with a specified variable in a specified dataset.

GetVarAttributesInDS. Information request to get the values associated with a particular custom variable attribute for a specified variable in a specified dataset.

SetVarAttributesInDS. Action request to set the values associated with a particular custom variable attribute for a specified variable in a specified dataset.

DelVarAttributesInDS. Action request to delete the values associated with a particular custom variable attribute for a specified variable in a specified dataset.

GetVarNValueLabelInDS. Information request to get the value labels associated with the numeric variable at a specified index position in a specified dataset.

SetVarNValueLabelInDS. Action request to set a single value label for the numeric variable at a specified index position in a specified dataset.

GetVarCValueLabelInDS. Information request to get the value labels associated with the string variable at a specified index position in a specified dataset.

SetVarCValueLabelInDS. Action request to set a single value label for the string variable at a specified index position in a specified dataset.

DelVarValueLabelInDS. Action request to delete all value labels associated with a specified variable in a specified dataset.

DelVarNValueLabelInDS. Action request to delete a given value label associated with a specified numeric variable in a specified dataset.

DelVarCValueLabelInDS. Action request to delete a given value label associated with a specified string variable in a specified dataset.

GetVarColumnWidthInDS. Information request to get the column width of the variable at a specified index position in a specified dataset. This is the column width for data values displayed in the Data Editor.

SetVarColumnWidthInDS. Action request to set the column width of the variable at a specified index position in a specified dataset. This is the column width for data values displayed in the Data Editor.

InsertCase. Action request to insert a case at a specified position in a specified dataset.

DeleteCase. Action request to delete the case at a specified position in a specified dataset.

GetCaseCountInDS. Information request to get the number of cases in a specified dataset.

GetNCellValue. Information request to get the value for a specified case for a particular numeric variable in a specified dataset.

SetNCellValue. Action request to set the value for a specified case for a particular numeric variable in a specified dataset.

GetCCellValue. Information request to get the value for a specified case for a particular string variable in a specified dataset.

SetCCellValue. Action request to set the value for a specified case for a particular string variable in a specified dataset.

GetMultiResponseSetNamesInDS. Information request to get the names of any multiple response sets for a specified dataset.

GetMultiResponseSetInDS. Information request to get the details of a specified multiple response set for a specified dataset.

SetMultiResponseSetInDS. Action request to create a multiple response set for a specified dataset.

DelMultiResponseSetInDS. Action request to delete a specified multiple response set from a specified dataset.

GetDataFileAttributeNamesInDS. Information request to get the names of any datafile attributes for a specified dataset.

GetDataFileAttributesInDS. Information request to get the value of a specified datafile attribute in a specified dataset.

SetDataFileAttributesInDS. Action request to create a datafile attribute in a specified dataset.

DelDataFileAttributesInDS. Action request to delete a specified datafile attribute from a specified dataset.

GetVarRoleInDS. Information request that returns the variable role for a specified variable in a specified dataset.

SetVarRoleInDS. Action request to set the variable role for a specified variable in a specified dataset.

Creating Pivot Tables and Text Blocks

Pivot tables and text blocks are generated within the context of user procedures created with the `StartProcedure` function. Functions for working with pivot tables and text blocks include:

StartProcedure. Action request that creates a user procedure and signals the beginning of pivot table or text block output.

EndProcedure. Action request that ends a user procedure and signals the end of pivot table or text block output.

HasProcedure. Information request that returns an integer indicating if a user procedure is running.

SplitChange. Action request to process a split change when creating a pivot table from data that have splits. Split changes are detected with the *IsEndSplit* function.

StartPivotTable. Action request to create a pivot table. Dimensions, categories, and cell values are specified separately.

MinDataColumnWidth. Action request to set the minimum column width for a pivot table.

MaxDataColumnWidth. Action request to set the maximum column width for a pivot table.

PivotTableCaption. Action request to add a caption to a pivot table.

AddCellFootnotes. Action request to add a footnote, associated with a specified cell, to a pivot table.

HidePivotTableTitle. Action request to hide the title of a pivot table.

AddDimension. Action request that adds a row, column, or layer dimension to a pivot table.

AddNumberCategory. Action request that adds a category with a numeric value to a specified dimension in a pivot table.

AddStringCategory. Action request that adds a category with a string value to a specified dimension in a pivot table. The string will not be translated.

AddVarNameCategory. Action request that adds a category representing a variable name to a specified dimension in a pivot table. Display settings for variable names in pivot tables (names, labels, or both) will be honored.

AddVarValueDoubleCategory. Action request that adds a category representing the value of a numeric variable to a specified dimension in a pivot table. Display settings for variable values in pivot tables (values, labels, or both) will be honored.

AddVarValueStringCategory. Action request that adds a category representing the value of a string variable to a specified dimension in a pivot table. Display settings for variable values in pivot tables (values, labels, or both) will be honored.

SetNumberCell. Action request to set a pivot table cell to a numeric value.

SetStringCell. Action request to set a pivot table cell to a string value. The string will not be translated.

SetDateCell. Action request to set a pivot table cell to a date value.

SetVarNameCell. Action request to set a pivot table cell to a variable name. Display settings for variable names in pivot tables (names, labels, or both) will be honored.

SetVarValueDoubleCell. Action request to set a pivot table cell to the value of a numeric variable. Display settings for variable values in pivot tables (values, labels, or both) will be honored.

SetVarValueStringCell. Action request to set a pivot table cell to the value of a string variable. Display settings for variable values in pivot tables (values, labels, or both) will be honored.

SetFormatSpec<format name>. Action request to set the format used for numeric values of cells and categories in a pivot table--for instance, `SetFormatSpecGeneralStat`. Each of the available formats has an associated `Set` function. The default format for numeric values of cells and categories in a pivot table is `Count`, which rounds values to the nearest integer.

AddTextBlock. Action request to create and populate a text block item.

Examples

Running IBM SPSS Statistics Commands

You execute a single IBM SPSS Statistics command by providing a string representation of the command as shown in this example. When submitting a single command in this manner the period (.) at the end of the command is optional.

```
// Open a data file
string filename = "/data/Employee data.sav";
string cmd = "GET FILE='" + filename + "'.";
Submit(cmd.c_str(),(int)cmd.length());

//Queue multiple command parts for a multi-line command
cmd = "OMS /SELECT TABLES";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "/IF COMMANDS = ['Descriptives' 'Frequencies']";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "/DESTINATION FORMAT = HTML";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "IMAGES = NO OUTFILE = '/temp/stats.html'.";
QueueCommandPart(cmd.c_str(),(int)cmd.length());

// Execute a Descriptives command and call OMSEND to write the output
cmd = "DESCRIPTIVES jobcat.";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "OMSEND.";
Submit(cmd.c_str(),(int)cmd.length());
```

- The first `Submit` function executes a `GET` command to open a data file.
- The `QueueCommandPart` function is used to queue a set of lines specifying the syntax for an `OMS` command and a `DESCRIPTIVES` command. In this example, the `OMS` command routes output from the `DESCRIPTIVES` command to an html file.
- The `Submit` function is called again to execute an `OMSEND` command as well as the queued commands.

Notes

- You can direct output to the XML workspace and subsequently retrieve it using XPath expressions. See the topic "Retrieving Output from Syntax Commands" on page 69 for more information.
- When submitting syntax that includes a `BEGIN PROGRAM-END PROGRAM` block, each of the lines in the block (including `BEGIN PROGRAM`) must be queued using the `QueueCommandPart` function. The `END PROGRAM` statement, however, can be submitted with the `Submit` function. In addition, note that you cannot submit an empty `BEGIN PROGRAM-END PROGRAM` block. The block must contain at least one statement in the associated language (Python or R).

Getting Variable Dictionary Information

Consider the common scenario of running a particular block of command syntax only if a specific variable exists in the dataset. For example, you are processing many datasets containing employee records and want to split them by gender--if a gender variable exists--to obtain separate statistics for the two gender groups. We will assume that if a gender variable exists, it has the name *gender*, although it may be spelled in upper case or mixed case. The following sample code illustrates the approach:

```
string filename = "/data/Employee data.sav";
string cmd = "GET FILE='" + filename + "'.";
Submit(cmd.c_str(),(int)cmd.length());

int errLevel;
unsigned int varNum = GetVariableCount(errLevel);
for (unsigned int n=0; n < varNum; n++) {
   string varName = GetVariableName(n,errLevel);
   for (unsigned int m=0; m < varName.length(); m++) {
      varName[m] = toupper(varName[m]);
   }
   if (varName == "GENDER")
   {
      string cmd = "SORT CASES BY " + varName + ".";
      QueueCommandPart(cmd.c_str(),(int)cmd.length());

      cmd = "SPLIT FILE LAYERED BY " + varName + ".";
      Submit(cmd.c_str(),(int)cmd.length());
      break;
   }
}
```

The example makes use of the `GetVariableCount` and `GetVariableName` functions for retrieving the number of variables in the active dataset and the name of a specified variable respectively. Variables are specified by their position in the dataset, starting with 0 for the first variable in file order. This is referred to as the **index value** of the variable.

Getting Case Data, Adding New Variables, and Appending Cases to the Active Dataset

You can read case data from the active dataset, create new variables in the active dataset, and append new cases to the active dataset using cursors. Cursors are created with the `MakeCaseCursor` function. The function takes a single argument that specifies the type of cursor: `"r"` allows you to read cases from the active dataset and is the default; `"w"` allows you to add new variables (and their case values) to the active dataset; and `"a"` allows you to append new cases to the active dataset.

The following rules apply to the use of cursors:

- You cannot use the `Submit` function while a cursor is open. You must close the cursor first using the `RemoveCaseCursor` function. In particular, if you need to save changes made to the active dataset to an external file, then use the `Submit` function to submit a `SAVE` command after closing the cursor.
- Only one cursor can be open at any point in an application. To define a new cursor, you must first close the previous one. You can use the `HasCursor` function to determine if a cursor is already open.

Reading Case Data

To read case data you create a cursor in read mode by calling `MakeCaseCursor("r")` (the argument can be omitted for read mode). Cases can only be read sequentially, although variable values within a case can be accessed directly by specifying the index of the variable. Index values represent position in the dataset, starting with 0 for the first variable in file order. When reading case data, the following rules apply to retrieved values:

- String values are right-padded to the defined width of the string variable.
- Data retrieved from a variable representing a date, or a date and a time, is given as the number of seconds from October 14, 1582.

Example

This example demonstrates reading case data from the active dataset.

```
string filename = "/data/mydata.sav";
string cmd = "GET FILE='" + filename + "'.";
Submit(cmd.c_str(),(int)cmd.length());

int errLevel, isMissing;
double dResult;
char *cResult;

int varCount = GetVariableCount(errLevel);
int *varType = new int[varCount];

for(int i = 0; i < varCount; i++) {
    varType[i] = GetVariableType(i,errLevel);
}

errLevel = MakeCaseCursor();
int rowCount = GetRowCount(errLevel);
for(int i = 0; i < rowCount; i++) {
    errLevel = NextCase();
    for(int j = 0; j < varCount; j++) {
        if(varType[j] == 0) {
            errLevel = GetNumericValue(j,dResult,isMissing);
        }
        else {
            cResult = new char[varType[j] + 1];
            memset(cResult,'\0',varType[j] + 1);
            errLevel = GetStringValue(j,cResult,varType[j] + 1,isMissing);
        }
    }
}

errLevel = RemoveCaseCursor();
```

- The `GET` command is used to get the data to be read.
- The `GetVariableType` function gets the type of the specified variable: 0 for numeric variables and the length in bytes for string variables. Variables are specified by their index.
- The `NextCase` function moves the cursor to the next case.
- Values for numeric variables are retrieved using the `GetNumericValue` function and values for string variables are retrieved using the `GetStringValue` function. The parameter *isMissing* indicates if the value is missing: 0 if the value is not missing; 1 if the value is user-missing; and 2 if the value is system-missing.
- The `RemoveCaseCursor` function closes the cursor.

Note: When reading datasets in which split file processing is in effect, you'll need to be aware of the behavior at a split boundary. Detecting split changes is necessary when you're creating custom pivot tables from data with splits and want separate results displayed for each split group (using the `SplitChange` function). The `IsEndSplit` function allows you to detect split changes when reading from datasets that have splits.

Creating New Variables in the Active Dataset

To add new variables along with their case values to the active dataset, you create a cursor in write mode by calling `MakeCaseCursor("w")`.

- When adding new variables, the `CommitHeader` function must be called after the statements defining the new variables and prior to setting case values for those variables. You cannot add new variables to an empty dataset. If you need to create a dataset from scratch, use a data step. See the topic “Creating and Accessing Multiple Datasets” on page 67 for more information.
- When setting case values for new variables, the `CommitCaseRecord` function must be called for each case that is modified. The `NextCase` function is used to advance the record pointer by one case.
- Changes to the active dataset do not take effect until the cursor is closed.
- Write mode supports multiple data passes and allows you to add new variables on each pass. In the case of multiple data passes where you need to add variables on a data pass other than the first, you must call the `AllocNewVarsBuffer` function to allocate the buffer size for the new variables. When used, `AllocNewVarsBuffer` must be called before reading any data with `NextCase`.
- The `SetVarNameAndType` and `SetOneVarNameAndType` functions are used to add new variables to the active dataset. The `SetValueChar` and `SetValueNumeric` functions are used to set case values.

Example

In this example we create a new string variable and a new numeric variable and populate their case values. A sample dataset is first created.

```
string cmd = "DATA LIST FREE /case (A5).";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "BEGIN DATA";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "case1";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "case2";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "END DATA.";
Submit(cmd.c_str(),(int)cmd.length());

char *varName[] = {"numvar","strvar"};
int varType[] = {0,8};
unsigned int nVars = 2;
char *varLabel_0 = "Sample numeric variable";
char *varLabel_1 = "Sample string variable";

int errLevel = MakeCaseCursor("w");
```

```
errLevel = SetVarNameAndType(varName,varType,nVars);
errLevel = SetVarLabel(varName[0],varLabel_0);
errLevel = SetVarLabel(varName[1],varLabel_1);
errLevel = CommitHeader();

char *cValue = "cheese";
int rowCount = GetRowCount(errLevel);

for(int i = 0; i < rowCount; i++) {
   errLevel = NextCase();
   errLevel = SetValueNumeric(varName[0],i);
   errLevel = SetValueChar(varName[1],cValue,strlen(cValue));
   errLevel = CommitCaseRecord();
}

errLevel = RemoveCaseCursor();
```

- The first argument to the `SetVarNameAndType` function is an array of strings that specifies the name of each new variable. The second argument is an array of integers specifying the variable type of each variable. Numeric variables are specified by a value of 0 for the variable type. String variables are specified with a type equal to the defined length of the string (maximum of 32767). In this example, we create a numeric variable named *numvar* and a string variable of length 8 named *strvar*.
- After calling `SetVarNameAndType` you have the option of specifying variable properties (in addition to the variable type) such as the measurement level, variable label, and missing values. In this example, variable labels are specified using the `SetVarLabel` function.
- Specifications for new variables must be committed to the cursor's dictionary before case values can be set. This is accomplished by calling the `CommitHeader` function, which takes no arguments. The active dataset's dictionary is updated when the cursor is closed.
- To set case values, you first position the record pointer to the desired case using the `NextCase` function, which advances the record pointer by one case. *Note*: To set the value for the first case in the dataset you must call `NextCase` as shown in this example.
- Case values are set using the `SetValueNumeric` function for numeric variables and the `SetValueChar` function for string variables. For both functions, the first argument is the variable name and the second argument is the value for the current case. A numeric variable whose value is not specified is set to the system-missing value. A string variable whose value is not specified will have a blank value. The value will be valid unless you explicitly define the blank value to be missing for that variable.
- The `CommitCaseRecord` function must be called to commit the values for each modified case. Changes to the active dataset take effect when the cursor is closed.

Note: To save the modified active dataset to an external file, use the `Submit` function (following the `RemoveCaseCursor` function) to submit a `SAVE` command, as in:

```
cmd = "SAVE OUTFILE='/data/mydata.sav'.";
Submit(cmd.c_str(),(int)cmd.length());
```

Example: Multiple Data Passes

Sometimes more than one pass of the data is required, as in the following example involving two data passes. The first data pass is used to read the data and compute a summary statistic. The second data pass is used to add a summary variable to the active dataset.

```
string cmd = "DATA LIST FREE /var (F6).";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "BEGIN DATA";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "57000";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "40200";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "21450";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "21900";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "END DATA.";
Submit(cmd.c_str(),(int)cmd.length());

int isMissing;
double dResult;
double total = 0;
```

```
int errLevel = MakeCaseCursor("w");
errLevel = AllocNewVarsBuffer(8);
int rowCount = GetRowCount(errLevel);

for(int i = 0; i < rowCount; i++) {
   errLevel = NextCase();
   errLevel = GetNumericValue(0,dResult,isMissing);
   total = total + dResult;
}
double meanVal = total/rowCount;

ResetDataPass();
SetOneVarNameAndType("mean",0);
errLevel = CommitHeader();

for(int i = 0; i < rowCount; i++) {
   errLevel = NextCase();
   errLevel = SetValueNumeric("mean",meanVal);
   errLevel = CommitCaseRecord();
}

errLevel = RemoveCaseCursor();
```

- Because we'll be adding a new variable on the second data pass, the `AllocNewVarsBuffer` function is called to allocate the required space. In the current example we're creating a single numeric variable, which requires 8 bytes.
- The first `for` loop is used to read the data and total the case values.
- After the data pass, the `ResetDataPass` function must be called prior to defining new variables.
- The `SetOneVarNameAndType` function is used to add a single new variable. The first argument is the variable name and the second argument is the variable type. In this example, we create a numeric variable named *mean*. The `CommitHeader` function is called to commit this variable to the cursor.
- The second data pass (second `for` loop) is used to set the case values of the new variable.

Appending New Cases

To append new cases to the active dataset, you create a cursor in append mode by calling `MakeCaseCursor("a")`. A cursor in append mode cannot be used to add new variables or read case data from the active dataset. A dataset must contain at least one variable in order to append cases to it, but it need not contain any cases. If you need to create a dataset from scratch, use a data step. See the topic "Creating and Accessing Multiple Datasets" on page 67 for more information.

- The `CommitNewCase` function must be called for each case that is added.
- The `EndChanges` function must be called before the cursor is closed.
- Changes to the active dataset do not take effect until the cursor is closed.
- The `SetValueChar` and `SetValueNumeric` functions are used to set variable values for new cases.

Example

In this example two new cases are appended to the active dataset.

```
string cmd = "DATA LIST FREE /case (F) value (A1).";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "BEGIN DATA";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "1 a";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "END DATA.";
Submit(cmd.c_str(),(int)cmd.length());

int errLevel = MakeCaseCursor("a");
errLevel = SetValueNumeric("case",2);
errLevel = SetValueChar("value","b",strlen("b"));
errLevel = CommitNewCase();
errLevel = SetValueNumeric("case",3);
errLevel = SetValueChar("value","c",strlen("c"));
errLevel = CommitNewCase();
errLevel = EndChanges();
errLevel = RemoveCaseCursor();
```

- For both the `SetValueNumeric` and `SetValueChar` functions, the first argument is the variable name, as a string, and the second argument is the value for the current case. A numeric variable whose value is

not specified is set to the system-missing value. A string variable whose value is not specified will have a blank value. The value will be valid unless you explicitly define the blank value to be missing for that variable.
- The `CommitNewCase` function must be called to commit the values for each new case. Changes to the active dataset take effect when the cursor is closed. When working in append mode, the cursor is ready to accept values for a new case (using `SetValueNumeric` and `SetValueChar`) once `CommitNewCase` has been called for the previous case.
- The `EndChanges` function signals the end of appending cases and must be called before the cursor is closed or the new cases will be lost.

Note: To save the modified active dataset to an external file, use the `Submit` function (following the `RemoveCaseCursor` function) to submit a `SAVE` command, as in:

```
cmd = "SAVE OUTFILE='/data/mydata.sav'.";
Submit(cmd.c_str(),(int)cmd.length());
```

Creating and Accessing Multiple Datasets

Using a data step, you can create and work concurrently with multiple datasets. Data steps are initiated with the `StartDataStep` function. Once a data step has been initiated, you access existing datasets and create new datasets with the `CreateDataset` function.

Limitations
- Within a data step you cannot create a cursor, a pivot table, or a text block, and you cannot call the `StartProcedure` function or the `Submit` function.
- You cannot start a data step if there are pending transformations or if a user procedure exists. If you need to access case data in the presence of pending transformations, use a cursor.
- Only one data step can exist at a time.
- A new dataset created with the `CreateDataset` function is not automatically set to be the active dataset, unless explicitly specified as such with `CreateDataset("*",true)`. To make a dataset the active one, use the `SetActive` function.

Example: Creating and Saving a New Dataset

```
StartDataStep();
int errLevel = CreateDataset("newds", true);
errLevel = InsertVariable("newds",0,"numvar",0);
errLevel = InsertVariable("newds",1,"strvar",1);
errLevel = SetVarLabelInDS("newds",0,"Sample Numeric Variable");
errLevel = SetVarLabelInDS("newds",1,"Sample String Variable");
errLevel = InsertCase("newds",0);
errLevel = SetNCellValue("newds",0,0,0);
errLevel = SetCCellValue("newds",0,1,"a");
errLevel = InsertCase("newds",1);
errLevel = SetNCellValue("newds",1,0,1);
errLevel = SetCCellValue("newds",1,1,"b");
errLevel = SetActive("newds");
EndDataStep();
string cmd = "SAVE OUTFILE='/data/mynewdata.sav'.";
Submit(cmd.c_str(),(int)cmd.length());
```

- Once a data step has been initiated, you create a new dataset with the `CreateDataset` function. The first argument specifies the name of the new dataset, and setting the second argument to *true* specifies a new dataset (you can also use the `CreateDataset` function to access existing datasets).
- You add variables to a dataset using the `InsertVariable` function. The arguments are the name of the dataset, the position (0-based) at which to add the variable, the name of the variable, and the variable type (0 for numeric variables, and an integer equal to the defined length for string variables).
- You can set variable properties, such as the variable label and measurement level, using additional functions. For example, you can set the variable label using the `SetVarLabelInDS` function, whose arguments are the name of the dataset, the position of the variable, and the variable label. Functions for setting variable properties are easily identified by the common suffix `InDs`.

- You add cases to a dataset using the `InsertCase` function. The arguments are the name of the dataset and the position of the case (0-based). The `InsertCase` function inserts an empty case. To populate the case values you use the `SetNCellValue` (for numeric values) and `SetCCellValue` (for string values) functions, which allow you to set the case value for a single variable. The arguments to each of these functions are the same and as follows: the name of the dataset, the position of the case for which a value will be set, the position of the variable within the case, and the value of the variable.
- The `SetActive` function is used to make the new dataset the active one. This is not necessary, but simplifies the process of saving the dataset--an action which must be performed outside of the data step.
- You end a data step with the `EndDataStep` function.
- The new dataset is saved by submitting a `SAVE` command outside of the data step.

Example: Modifying Case Values

```
string cmd = "DATA LIST FREE /cust (F2) amt (F5).";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "BEGIN DATA";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "210 4500";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "242 6900";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "370 32500";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "END DATA.";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "DATASET NAME ds1.";
Submit(cmd.c_str(),(int)cmd.length());

StartDataStep();
int err, isMissing;
int errLevel = CreateDataset("ds1");
int rowCount = GetCaseCountInDS("ds1",errLevel);
for(int i = 0; i < rowCount; i++) {
   errLevel = SetNCellValue("ds1",i,1,1.05*GetNCellValue("ds1",i,1,isMissing,err));
}
EndDataStep();
```

- A sample dataset is created and assigned the name *ds1*.
- To modify case values in an existing dataset, you first access the dataset using the `CreateDataset` function, specifying the name of the dataset as the argument.
- Case values are modified using the `SetNCellValue` (for numeric values) and `SetCCellValue` (for string values) functions, which allow you to set the case value for a single variable. The arguments to each of these functions are the same and as follows: the name of the dataset, the position of the case for which a value will be set, the position of the variable within the case, and the value of the variable.

Example: Comparing Datasets

Data steps allow you to concurrently work with the case data from multiple datasets. As a simple example, we'll compare the cases in two datasets and indicate identical cases with a new variable added to one of the datasets.

```
//Create the first dataset
string cmd = "DATA LIST FREE /id (F2) salary (DOLLAR8) jobcat (F1).";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "BEGIN DATA";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "1 57000 3";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "3 40200 1";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "2 21450 1";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "END DATA.";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "SORT CASES BY id.";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "DATASET NAME ds1.";
QueueCommandPart(cmd.c_str(),(int)cmd.length());

//Create the second dataset
```

```
cmd = "DATA LIST FREE /id (F2) salary (DOLLAR8) jobcat (F1).";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "BEGIN DATA";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "3 41000 1";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "1 59280 3";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "2 21450 1";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "END DATA.";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "SORT CASES BY id.";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "DATASET NAME ds2.";
Submit(cmd.c_str(),(int)cmd.length());

//Perform the comparison
StartDataStep();
int err, isMissing;
int errLevel = CreateDataset("ds1");
int rowCount = GetCaseCountInDS("ds1",errLevel);
errLevel = CreateDataset("ds2");
int varCount = GetVarCountInDS("ds2",errLevel);
errLevel = InsertVariable("ds2",varCount,"match",0);

for(int i = 0; i < rowCount; i++) {
   errLevel = SetNCellValue("ds2",i,varCount,1);
   for(int j = 0; j < varCount; j++) {
      if (GetNCellValue("ds1",i,j,isMissing,err) != \
          GetNCellValue("ds2",i,j,isMissing,err)){
             errLevel = SetNCellValue("ds2",i,varCount,0);
             break;
      }
   }
}
EndDataStep();
```

- The two sample datasets are sorted by the variable *id* which is common to both datasets.
- The new variable *match*, added to *ds2*, is set to 1 for cases that are identical and 0 otherwise.

Retrieving Output from Syntax Commands

To retrieve command output, you first route it via the Output Management System (OMS) to the **XML workspace** where it is stored as an XPath DOM that conforms to the Output XML Schema (*xml.spss.com/spss/oms*). Textual output is retrieved from this workspace with functions that employ XPath expressions. Charts generated by syntax commands can be retrieved with the `GetImage` function. See the topic "Retrieving Chart Output" on page 71 for more information.

Constructing the correct XPath expression (IBM SPSS Statistics currently supports XPath 1.0) requires an understanding of the Output XML schema. The output schema spss-output-1.8.xsd is distributed with IBM SPSS Statistics and is also available from http://xml.spss.com/spss/oms/. Documentation is included in the IBM SPSS Statistics - Programmability SDK and accessible from *documentation*\ *OutputSchema**oms_oxml_schema_intro.htm*.

Example

In this example, we'll retrieve the mean value of a variable calculated from the Descriptives procedure.

```
string cmd = "GET FILE='/data/Employee data.sav'.";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "OMS SELECT TABLES";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "/IF COMMANDS=['Descriptives'] SUBTYPES=['Descriptive Statistics']";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "/DESTINATION FORMAT=OXML XMLWORKSPACE='desc_table'";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "/TAG='desc_out'.";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "DESCRIPTIVES VARIABLES=salary, salbegin, jobtime, prevexp ";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "/STATISTICS=MEAN.";
QueueCommandPart(cmd.c_str(),(int)cmd.length());
cmd = "OMSEND TAG='desc_out'.";
Submit(cmd.c_str(),(int)cmd.length());
```

```
int err;
const char* handle = "desc_table";
const char* context = "/outputTree";
const char* xpath = "//pivotTable[@subType='Descriptive Statistics']" \
                    "/dimension[@axis='row']" \
                    "/category[@varName='salary']" \
                    "/dimension[@axis='column']" \
                    "/category[@text='Mean']" \
                    "/cell/@text";
void* result = EvaluateXPath(handle,context,xpath,err);
int len = GetStringListLength(result);
if(len > 0){
   const char* str = GetStringFromList(result,0);
}
RemoveStringList(result);
RemoveXPathHandle(handle);
```

- The `OMS` command is used to direct output from a syntax command to the XML workspace. The `XMLWORKSPACE` keyword on the `DESTINATION` subcommand, along with `FORMAT=OXML`, specifies the XML workspace as the output destination. It is a good practice to use the `TAG` subcommand, as done here, so as not to interfere with any other OMS requests that may be operating. The identifiers used for the `COMMANDS` and `SUBTYPES` keywords on the `IF` subcommand can be found in the OMS Identifiers dialog box, available from the Utilities menu in IBM SPSS Statistics.
- The `XMLWORKSPACE` keyword is used to associate a name with this XPath DOM in the workspace. In the current example, output from the `DESCRIPTIVES` command will be identified with the name *desc_table*. You can have many XPath DOM's in the XML workspace, each with its own unique name.
- The `OMSEND` command terminates active `OMS` commands, causing the output to be written to the specified destination--in this case, the XML workspace.
- You retrieve values from the XML workspace with the `EvaluateXPath` function. The function takes an explicit XPath expression, evaluates it against a specified XPath DOM in the XML workspace, and returns the result as a list of string values.
- The first argument to the `EvaluateXPath` function specifies the XPath DOM to which an XPath expression will be applied. This argument is referred to as the handle name for the XPath DOM and is simply the name given on the `XMLWORKSPACE` keyword on the associated `OMS` command. In this case the handle name is *desc_table*.
- The second argument to `EvaluateXPath` defines the XPath context for the expression and should be set to `"/outputTree"` for items routed to the XML workspace by the `OMS` command.
- The third argument to `EvaluateXPath` specifies the remainder of the XPath expression (the context is the first part). For users familiar with XSLT for OXML and accustomed to including a namespace prefix, note that XPath expressions for the `EvaluateXPath` function should not contain the `oms:` namespace prefix.
- The XPath expression in this example is specified by the variable *xpath*. It is not the minimal expression needed to select the mean value of interest but is used for illustration purposes and serves to highlight the structure of the XML output.

  `//pivotTable[@subType='Descriptive Statistics']` selects the Descriptives Statistics table.

  `/dimension[@axis='row']/category[@varName='salary']` selects the row for the variable *salary*.

  `/dimension[@axis='column']/category[@text='Mean']` selects the *Mean* column within this row, thus specifying a single cell in the pivot table.

  `/cell/@text` selects the textual representation of the cell contents.
- The `RemoveStringList` function frees the storage associated with the specified result from `EvaluateXPath`, and `RemoveXPathHandle` frees the storage associated with the specified XPath DOM.

If you're familiar with XPath, note that the mean value of *salary* can also be selected with the following simpler XPath expression:

```
//category[@varName='salary']//category[@text='Mean']/cell/@text
```

Notes

To the extent possible, construct your XPath expressions using language-independent attributes, such as the variable name rather than the variable label. That will help reduce the translation effort if you need to deploy your code in multiple languages. Also consider factoring out language-dependent identifiers, such as the name of a statistic, into constants. You can obtain the current language used for pivot table output from the `GetOutputLanguage` function.

You may also consider using `text_eng` attributes in place of `text` attributes in XPath expressions. `text_eng` attributes are English versions of `text` attributes and have the same value regardless of the output language. The `OATTRS` subcommand of the `SET` command specifies whether `text_eng` attributes are included in OXML output.

When writing and debugging XPath expressions, it is often useful to inspect the associated XPath DOM. This is provided by the `GetXmlUtf16` function, as well as by an option on the `OMS` syntax command.

Retrieving Chart Output

Charts generated by syntax commands whose output has been routed to the **XML workspace** can be retrieved with the `GetImage` function.

Example

In this example, we'll retrieve a control chart generated by the `SPCHART` command.

```
int err = 0;
const char* cmd1 = "GET FILE='/data/clothing_defects.sav'.";
const char* cmd2 = "OMS /SELECT CHARTS /IF COMMANDS=['SPchart'] " \
                   "/DESTINATION FORMAT=OXML XMLWORKSPACE ='spchart' " \
                   "IMAGES=YES IMAGEFORMAT=JPG CHARTFORMAT=IMAGE.";
const char* cmd3 = "SPCHART /P=COUNT(defects) N(sampled) BY lot /RULES=All /SIGMAS=3.";
const char* cmd4 = "OMSEND.";
QueueCommandPart(cmd1,strlen(cmd1));
QueueCommandPart(cmd2,strlen(cmd2));
QueueCommandPart(cmd3,strlen(cmd3));
Submit(cmd4,strlen(cmd4));
int real_image_size = 0;
char* imageType = 0;
const char* handle = "spchart";
const char* context = "/outputTree";
const char* xpath = "//command[@command='SPchart']" \
                    "/chartTitle[@text='p of defects by lot']/chart/@imageFile";
void* result = EvaluateXPath(handle,context,xpath,err);
const char* image_name = GetStringFromList(result,0);
const char* image = GetImage(handle,image_name,real_image_size,&imageType,err);
assert(err == 0);
fstream ftemp;
string output_image = "/images/";
output_image.append(image_name);
ftemp.open(output_image.c_str(),ios::out|ios::binary);
ftemp.write(image,real_image_size);

ftemp.close();
RemoveStringList(result);
RemoveXPathHandle(handle);
```

- The `OMS` command is used to direct output from the `SPCHART` command to the XML workspace. See the topic "Retrieving Output from Syntax Commands" on page 69 for more information.
- To route images along with the OXML output, the `IMAGES` keyword on the `DESTINATION` subcommand (of the `OMS` command) must be set to `YES`, and the `CHARTFORMAT`, `MODELFORMAT`, or `TREEFORMAT` keyword must be set to `IMAGE`.
- The `EvaluateXPath` function is used to retrieve the name of the image associated with the control chart output from the `SPCHART` command. In the present example, the value returned by `EvaluateXPath` is a list with a single element, whose value is extracted with the `GetStringFromList` function and stored to the variable *image_name*.
- The `GetImage` function returns a pointer to the image.

  The first argument to `GetImage` specifies the XPath DOM that contains the output from the command that generated the image.

The second argument to `GetImage` is the filename associated with the image in the OXML output--specifically, the value of the `imageFile` attribute of the `chart`, `modelView` or `treeView` element associated with the image.

On return, the third argument to `GetImage` is set to the amount of memory required for the image.

On return, the fourth argument to `GetImage` is set to a string specifying the image type: “PNG”, “JPG”, “EMF”, “BMP”, or “VML”.

- In the present example, the retrieved image is written to an external file.

Notices

Trademarks

Index

Printed in USA