



TIBCO SPOTFIRE S+[®] 8.2 Application Developer's Guide

November 2010

TIBCO Software Inc.

IMPORTANT INFORMATION

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN *TIBCO SPOTFIRE S+® LICENSES*). USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIBCO Software Inc., TIBCO, Spotfire, TIBCO Spotfire S+, Insightful, the Insightful logo, the tagline "the Knowledge to Act," Insightful Miner, S+, S-PLUS, TIBCO Spotfire Axum, S+ArrayAnalyzer, S+EnvironmentalStats, S+FinMetrics, S+NuoOpt, S+SeqTrial, S+SpatialStats, S+Wavelets, S-PLUS Graphlets, Graphlet, Spotfire S+ FlexBayes, Spotfire S+ Resample, TIBCO Spotfire Miner, TIBCO Spotfire S+ Server, TIBCO Spotfire Statistics Services, and TIBCO Spotfire Clinical Graphics are either registered trademarks or trademarks of TIBCO Software Inc. and/or subsidiaries of TIBCO Software Inc. in the United States and/or other countries. All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. This

software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. Please see the readme.txt file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

Copyright © 1996-2010 TIBCO Software Inc. ALL RIGHTS RESERVED. THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

TIBCO Software Inc. Confidential Information

Reference

The correct bibliographic reference for this document is as follows:

TIBCO Spotfire S+® 8.2 Application Developer's Guide, TIBCO Software Inc.

Technical Support

For technical support, please visit <http://spotfire.tibco.com/support> and register for a support account.

TIBCO SPOTFIRE S+ BOOKS

Note about Naming

Throughout the documentation, we have attempted to distinguish between the language (S-PLUS) and the product (Spotfire S+).

- “S-PLUS” refers to the engine, the language, and its constituents (that is objects, functions, expressions, and so forth).
- “Spotfire S+” refers to all and any parts of the product beyond the language, including the product user interfaces, libraries, and documentation, as well as general product and language behavior.

The TIBCO Spotfire S+[®] documentation includes books to address your focus and knowledge level. Review the following table to help you choose the Spotfire S+ book that meets your needs. These books are available in PDF format in the following locations:

- In your Spotfire S+ installation directory (**\$HOME/help** on Windows, **\$HOME/doc** on UNIX/Linux).
- In the Spotfire S+ Workbench, from the **Help ► Spotfire S+ Manuals** menu item.
- In Microsoft[®] Windows[®], in the Spotfire S+ GUI, from the **Help ► Online Manuals** menu item.

Spotfire S+ documentation.

Information you need if you...	See the...
Must install or configure your current installation of Spotfire S+; review system requirements.	<i>Installation and Administration Guide</i>
Want to review the third-party products included in Spotfire S+, along with their legal notices and licenses.	<i>Licenses</i>

Spotfire S+ documentation. (Continued)

Information you need if you...	See the...
Are new to the S language and the Spotfire S+ GUI, and you want an introduction to importing data, producing simple graphs, applying statistical models, and viewing data in Microsoft Excel [®] .	<i>Getting Started Guide</i>
Are a new Spotfire S+ user and need how to use Spotfire S+, primarily through the GUI.	<i>User's Guide</i>
Are familiar with the S language and Spotfire S+, and you want to use the Spotfire S+ plug-in, or customization, of the Eclipse Integrated Development Environment (IDE).	<i>Spotfire S+ Workbench User's Guide</i>
Have used the S language and Spotfire S+, and you want to know how to write, debug, and program functions from the Commands window.	<i>Programmer's Guide</i>
Are familiar with the S language and Spotfire S+, and you want to extend its functionality in your own application or within Spotfire S+.	<i>Application Developer's Guide</i>
Are familiar with the S language and Spotfire S+, and you are looking for information about creating or editing graphics, either from a Commands window or the Windows GUI, or using Spotfire S+ supported graphics devices.	<i>Guide to Graphics</i>
Are familiar with the S language and Spotfire S+, and you want to use the Big Data library to import and manipulate very large data sets.	<i>Big Data User's Guide</i>
Want to download or create Spotfire S+ packages for submission to the Comprehensive S-PLUS Archive Network (CSAN) site, and need to know the steps.	<i>Guide to Packages</i>

Spotfire S+ documentation. (Continued)

Information you need if you...	See the...
Are looking for categorized information about individual S-PLUS functions.	<i>Function Guide</i>
If you are familiar with the S language and Spotfire S+, and you need a reference for the range of statistical modelling and analysis techniques in Spotfire S+. Volume 1 includes information on specifying models in Spotfire S+, on probability, on estimation and inference, on regression and smoothing, and on analysis of variance.	<i>Guide to Statistics, Vol. 1</i>
If you are familiar with the S language and Spotfire S+, and you need a reference for the range of statistical modelling and analysis techniques in Spotfire S+. Volume 2 includes information on multivariate techniques, time series analysis, survival analysis, resampling techniques, and mathematical computing in Spotfire S+.	<i>Guide to Statistics, Vol. 2</i>

CONTENTS

Chapter 1 Introduction to the Application Developer's Guide	1
Developing Applications	2
Chapter 2 The Spotfire S+ Command Line and the System Interface	3
Using the Command Line	4
Command Line Parsing	7
Working With Projects	24
Enhancing Spotfire S+	26
The System Interface	28
Chapter 3 CONNECT/C++	35
Introduction	36
Examples: An Application and a Called Routine	38
CONNECT/C++ Class Overview	46
CONNECT/C++ Architectural Features	49
A Simple Spotfire S+ Interface in Windows	60
Chapter 4 CONNECT/Java	65
Introduction	66
Calling Java from Spotfire S+	67
Calling Spotfire S+ from Java Applications	81

Chapter 5 Interfacing with C and FORTRAN Code	97
Overview	99
When to Use the C and Fortran Interfaces	100
Using C and Fortran Code with Spotfire S+ for Windows	102
Calling C Routines from Spotfire S+ for Windows	109
Writing C and Fortran Routines Suitable for Use with Spotfire S+ for Windows	114
Common Concerns in Writing C and Fortran Code for Use with Spotfire S+ for Windows	119
Using C Functions Built into Spotfire S+ for Windows	134
Calling Spotfire S+ Functions from C Code (Windows)	138
The .Call Interface (Windows)	146
Debugging Loaded Code (Windows)	151
A Simple Example: Filtering Data (Unix)	155
Calling C or Fortran Routines From Spotfire S+ for Unix	157
Writing C and Fortran Routines Suitable for Use in Spotfire S+ for Unix	161
Compiling and Dynamically Linking your Code (Unix)	162
Common Concerns in Writing C and Fortran Code for Use with Spotfire S+ for Unix	166
Using C Functions Built into Spotfire S+ for Unix	177
Calling S-PLUS Functions From C Code (Unix)	180
The .Call Interface (Unix)	187
Debugging Loaded Code (Unix)	192
A Note on StatLib (Windows and Unix)	195

Chapter 6 Automation	197
Introduction	198
Using Spotfire S+ as an Automation Server	199
Using Spotfire S+ as an Automation Client	224
Automation Examples	235
Chapter 7 Calling Spotfire S+ Using DDE	241
Introduction	242
Working With DDE	243
Chapter 8 Extending the User Interface	251
Overview	253
Menus	255
Toolbars and Palettes	264
Dialogs	276
Dialog Controls	290
Callback Functions	327
Class Information	333
Style Guidelines	341
Chapter 9 Libraries	369
Introduction	370
Creating a Library	372
Distributing the Library	379
Chapter 10 Spotfire S+ Dialogs in Java	381
Overview	383
Classes	387
Layout	391
Actions	394

Calling The Function	397
Modifying Menus	399
Style Guidelines	402
Example: Correlations Dialog	423
Example: Linear Regression Dialog	427
Chapter 11 User-Defined Help	439
Introduction to Creating Help Files in Windows	440
Creating, Editing, and Distributing a Help File in Windows	444
Introduction to Creating Help Files in UNIX	451
Creating, Editing, and Distributing a Help File in UNIX	453
Common Text Formats	458
Contents of Help Files	460
Chapter 12 Globalization	477
Introduction	478
Working With Locales	479
Using Extended Characters	484
Importing, Exporting, and Displaying Numeric Data	486
Chapter 13 Verbose Logging	491
Overview	492
Verbose Batch Execution	494
Example	501
Chapter 14 XML Generation	507
XML Overview	508
XML and SPXML Library Overview	509

The SPXML Library	510
Reading and Writing XML Using the SPXML Library	511
Examples of XSL Transformations	512
Chapter 15 XML Reporting	525
Overview	526
What is XSL?	527
Custom Reports	529
Summary Reports	533
Character Substitutions	545
Index	45

Contents

INTRODUCTION TO THE APPLICATION DEVELOPER'S GUIDE

1

Developing Applications

2

DEVELOPING APPLICATIONS

As an application developer, you can use Spotfire S+[®] for the following three scenarios:

- Customizing the Spotfire S+ user interface.
- Embedding Spotfire S+ functionality (that is, the Spotfire S+ engine and libraries) in your application, so your users can call S-PLUS functions from within your application.
- Creating your own S-PLUS functions, methods, classes, libraries, and help files.

TIBCO Software Inc. offers a package for creating customized or embedded OEM applications for the first two scenarios described above. For more information about the OEM kit, contact TIBCO.

A Spotfire S+ application developer's interest can include:

- Extending the functionality of Spotfire S+ by creating libraries and graphical objects.
- Automating Spotfire S+ routines to be called from the system interface.
- Calling S-PLUS functions to or from other languages.
- Customizing the graphical user interface (GUI) and other aspects of the Spotfire S+ environment.
- Writing scripts and validation routines.
- Creating logs and reports.

This guide provides guidance in these areas.

THE SPOTFIRE S+ COMMAND LINE AND THE SYSTEM INTERFACE

2

Using the Command Line	4
Command Line Parsing	7
Variables	8
Switches	18
Checking Default Environment Settings	22
Working With Projects	24
The Preferences Directory	24
The Data Directory	25
Enhancing Spotfire S+	26
Adding Functions and Data Sets to Your System	26
The System Interface	28
Using the Windows Interface	28
Using the DOS Interface	32

USING THE COMMAND LINE

Spotfire S+ accepts a number of optional commands on the **splus.exe** executable's command line that allow users significant control over the operation of Spotfire S+.

These facilitate running Spotfire S+ in an automated or batch environment. They also make it possible to readily alter the behavior of Spotfire S+ on a session by session basis. Some users may find it handy to have several shortcuts (or program manager icons if running older versions of Windows), each of which starts Spotfire S+ with specific projects and options selected by default. Figure 2.1 shows an example of the command line with the optional **/BATCH** switch.

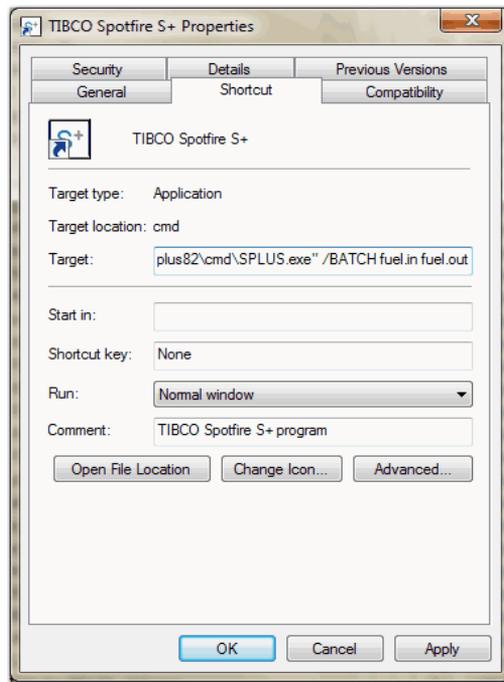


Figure 2.1: Example of a command line with optional **/BATCH** switch. You can save the settings on the **Target** field and run the shortcut without starting up Spotfire S+ from your desktop.

Note

This chapter refers to the **splus.exe** command line that is used to start execution of Spotfire S+, not the **Commands** window that is used to enter commands once Spotfire S+ has started. The Spotfire S+ command line refers to anything that follows the name of the executable (**splus.exe** by default) in the shortcut, program manager icon, or batch file from which Spotfire S+ may be started. On the Spotfire S+ command line only certain switches are permitted and have their own syntax as discussed in this section.

Command line processing of variables is implemented in both **splus.exe** and **sqpe.exe**, although variables that apply to the **Commands** window or **Commands history** (S_CMDFILE, S_CMDLINEBUF, S_CMDSAVE, S_PRINT_COMMAND, and S_SCRSAVE) are ignored in **sqpe.exe**. Command line processing of switches is only implemented for **splus.exe**, not for **sqpe.exe**.

Filenames that follow a @ symbol on the command line are expanded in place. The command line is then tokenized, with the following classes of tokens identified:

- Variables: Any environment variable is identified by the trailing equal sign and uses a Variable=Value syntax. Spotfire S+ recognizes certain variables (Table 2.1), and user-written routines might also query and react to these.
- Switches: The predefined switches listed below can be specified on the command line. They use a /Switch [Value1 [Value2 [Value3 [...]]]] syntax and are parsed based on the leading symbol (/ or -).
- Script Files: Remaining tokens are considered script files to be run by the Spotfire S+ interpreter.

An example command line might be:

```
SPLUS.EXE ScriptFile1 /REGKEY SPLUS1 S_PROJ=c:\Project1
```

Most options set on the command line are for advanced users. Some more generally useful options are the BATCH switch, Script file processing, and for intermediate users, S_TMP, S_FIRST, and the

ability to set up Spotfire S+ to run with different project directories using the S_PROJ variable. See the following section for more information about using multiple projects.

COMMAND LINE PARSING

The operating system passes the command line unaltered to Spotfire S+.

File Expansion

Spotfire S+ expands files specified in the command line. Anything between an '@' character and the first subsequent delimiter (@ sign, double quote, single quote, or the standard delimiters: space, tab, newline, linefeed) is considered a filename and the entire file will be expanded in place.

The @ token can be escaped by placing a backslash before it, for example, in “EnvVar1=EnvValueWith@Filename” the @ sign will be active, and in “EnvVar2=EnvValueWith\@NoFilename” it will be ignored. The escape character is removed during this stage.

Multiple file names in the command line are fine, as are further filenames embedded within a file. Files that use a relative path are normally located relative to the current working directory; if they are not found, the search will continue in the same directory where **splus.exe** is found.

There is no way to specify a filename with spaces in it, nor to avoid a trailing delimiter after the filename, nor to avoid a trailing delimiter after the expanded file contents. As a result, keep the filenames as simple and intuitive before the expansion.

Tokenizing

The command line is then broken into tokens. Standard command line delimiters are space, tab, newline, and linefeed and any combination of these are ignored at the start of the command line, and between tokens.

If the first character of a token is a single or double quote then it will be matched with another of the same type of quote, and anything between will be considered part of the token but nothing thereafter.

Otherwise, a token begins with any non-delimiter and goes to the first delimiter or equal sign (the only way to "escape" a delimiter or equal sign is to place the entire token in quotes).

Variables

If the token is followed by an equal sign, it is considered to be part of a variable-value pair. (This is true even if the token begins with "-" or "/".) If a delimiter is found trailing the equal sign, the variable is assigned an empty string for the value. (This can be used to cancel or override environment variables in the process environment.) Variables are then assigned the specified value.

Switches

Any token (not followed by an equal sign) that has either "-" or "/" as its first character is considered a switch. Each switch takes a variable number of successive tokens. Switches are evaluated in a case-insensitive manner. Switches are separated from successive tokens by the normal delimiters. Unknown switches are ignored.

Script Files

Remaining tokens are then considered script files and their contents sent to the Spotfire S+ interpreter. (Also see the /BATCH switch for an alternative mechanism for automating Spotfire S+ sessions.)

Variables

The following is a list of the variables recognized by Spotfire S+. You are *not* required to set them.

Table 2.1: *Variables.*

Variable	Description
HOME	Deprecated, replaced by the synonymous S_PROJ .
S_CMDFILE	Name of the file used to initialize, and optionally append to, the Commands History window.
S_CMDLINEBUF	Sets the maximum number of characters that can be entered to specify a command in the Commands window. By default, this is 32767.
S_CMDSAVE	Number of commands to save for Commands History recall.

Table 2.1: Variables.

S_CWD	Directs Spotfire S+ to set the current working directory to this directory at startup. Subsequent file I/O will be done relative to this directory.
S_DATA	A series of semicolon-separated directory paths, which is searched for a suitable database 1 (which stores user Spotfire S+ functions and data sets).
S_DISPLAY_MODE	Specifies whether Spotfire S+ is running in the graphical user interface ("s+gui"), the Spotfire S+ Console ("console") or the Spotfire S+ Workbench ("s+java"). (The <code>interactive()</code> function can address this query as well.)
S_ECLIPSE	Specifies whether Spotfire S+ is running as the Spotfire S+ Workbench (an Eclipse plug-in) or in BATCH (local).
S_ENGINE_LOCATION	No longer supported.
S_FIRST	S-PLUS function evaluated at start-up. See section <code>S_FIRST=function</code> (page 14).
SHOME	Specifies the directory where Spotfire S+ is installed.
S_INTERACTIVE_STATES	Has the value "yes" in all contexts except BATCH (TIBCO Spotfire Statistics Services).
S_LOAD_BIGDATA	Specifies that the bigdata library should be loaded.
S_NOAUDIT	Tells Spotfire S+ not to write the audit file. Set by default in splus.exe . Not set by default in sqpe.exe .
S_NO_RECLAIM	No longer supported.
S_NOSYMLOOK	No longer supported.
S_PATH	No longer supported; see <code>SV4_SEARCH_PATH</code> .

Table 2.1: *Variables.*

S_PREFS	Directory to use for storing user preferences.
S_PRINT_COMMAND	Windows command to use for printing the Commands window. By default, the command is “ Notepad /p ”.
S_PROJ	Sets default values for S_CWD , S_DATA , and S_PREFS . See the section Working With Projects (page 24).
S_PS2_FORMAT	Tells Spotfire S+ to put a CTRL-D at the end of any PostScript file it generates. By default, Spotfire S+ does not do this.
S_SCRSAVE	KB of Commands window output to save for scrollbar.
S_SILENT_STARTUP	Disable printing of copyright/version messages.
S_TMP	Specifies the directory where Spotfire S+ creates temporary scratch files.
S_USER_APPDATA_DIR	A version-, platform-, and user-specific directory, where you can keep platform-dependent and version-dependent data for a specific user.
S_WORK	Deprecated; replaced by S_DATA .
SV4_SEARCH_PATH	A list of semicolon-separated directories used for the initial Spotfire S+ search path, set by default to the Spotfire S+ system databases.

Many of the variables in this section take effect if you set them to any value, and do not take effect if you do not set them, so you may leave them unset without harm. For example, to set **S_NOAUDIT** you can enter:

```
SPLUS.EXE S_NOAUDIT=X
```

on the command line and Spotfire S+ will not write an audit file, because the variable **S_NOAUDIT** has a value (any value); this is the default for that variable in **splus.exe**. If you want Spotfire S+ to begin

writing the audit file again during your next Spotfire S+ session, unset **S_NOAUDIT** on the command line. To unset a variable that has previously been set in some way, enter no value (or a space) after typing the equal sign:

```
SPLUS.EXE S_NOAUDIT=
```

Now, **S_NOAUDIT** is not set, and Spotfire S+ writes the audit file.

Variables are currently implemented using environment variables. Therefore, advanced users can specify these variables by altering their system or process environment variables using standard operating system specific techniques (for example, via the Control Panel's System applet). Variables specified on the command line are placed in the process environment at start-up and hence take precedence over any previously defined for the process.

User code can check the current values for these variables by using `getenv` from C or S code.

Note

We recommend placing variables on the command line. If you want to have multiple shortcuts use some of the same variables or switches, we recommend you place those common settings in a file and place the file name on the command line preceded with the @ sign. For specifics, see the File Expansion section above.

S_CMDFILE=filePath

By using the **S_CMDFILE** variable you can initialize the **Commands History** dialog to contain the commands found in a named text file. Optionally, the commands from your current session are appended to this file. Below are several examples illustrating the use of the **S_CMDFILE** variable. These lines would be placed on the Spotfire S+ start-up command line.

```
S_CMDFILE=d:\splus\cmdhist.q  
S_CMDFILE=d:\splus\cmdhist.q+  
S_CMDFILE==d:\splus\cmdhist.q  
S_CMDFILE==+history.q
```

In all cases, a path and filename are specified, and any commands found in the named file are placed in the **Commands History** dialog at startup. In the first example, new commands from the current

session will not be appended to the file. Placing a "+" immediately after the path and filename, as in the second example, causes the commands from the current session to be appended to the named file. Placing a "+" immediately before the path and filename, as in the third example, causes the commands from the current session to be appended to the named file and causes Spotfire S+ to create a new file if the named file does not exist. (The directory must already exist; only the file is created automatically.) In the final case, Spotfire S+ uses the file **history.q** in the start-up directory; it creates the file if it does not already exist. If you later change your start-up directory, another **history.q** will be created in that directory. You can also use the auditing facility in Spotfire S+ to automatically save your commands history. For this to work, you must turn on auditing and set the **S_CMDFILE** variable to **.AUDIT** by placing the following on the Spotfire S+ start-up command line:

```
SPLUS.EXE S_NOAUDIT= S_CMDFILE=+.AUDIT
```

You need the "+" to avoid an error message when you start up in a new working directory. When you use auditing, Spotfire S+ saves more than commands in the **.audit** file. However, the **Commands History** window will show you only the Spotfire S+ commands when you use the auditing facility for your commands history. The **.audit** file used by the auditing facility is found in the data directory. The **Commands History** window will look in that directory for **.audit** when the variable **S_CMDFILE** is set to **.AUDIT**.

S_CMDLINEBUF=integer

Set the **S_CMDLINEBUF** variable to increase or decrease the maximum number of characters that can be entered for any one command in the **Commands** window. The default is 32767.

S_CMDSAVE=integer

Set the **S_CMDSAVE** variable if you want to limit the number of commands saved for command line/**Commands History** recall. For example, to limit the number of commands stored to the most recent 100, set this variable as follows:

```
S_CMDSAVE=100
```

S_CWD=directoryPath

Every operating system process, including Spotfire S+, has a *current working directory* that determines where file input/output occurs (or is relative to). The Spotfire S+ process is assigned a directory when it is started from a shortcut (or a program manager icon in earlier versions of Windows) or from a batch file or DOS prompt. Specifying the **S_CWD** variable causes Spotfire S+ to ignore the default directory assigned by the parent process and use a specific one instead. Note that **S_CWD** defaults to **S_PROJ**.

Note

Previous versions of Spotfire S+ used the **S_WORK** variable to refer to the “working directory.” To avoid confusion with the term “current working directory,” the terminology has changed and now we use the **S_DATA** variable to refer to the “data directory.”

S_DATA=directoryPath[;directoryPath[...]]

S_DATA specifies a list of semicolon-separated directories that is searched for a suitable database 1. Thus the first valid directory in the list is used by Spotfire S+ to store user data and functions. It traditionally is named **.Data**.

S_DATA defaults to **.Data;%S_PROJ%\Data**, so Spotfire S+ seeks a **.Data** directory under the current working directory (see **S_CWD**), and otherwise seek a **.Data** directory under the project directory. If that then fails, a dialog will ask the user for the directory path. **S_DATA** replaces **S_WORK** that was used in previous versions of Spotfire S+.

S_DISPLAY_MODE=mode

S_DISPLAY_MODE can specify either the Windows GUI, the console (for BATCH), or the Spotfire S+ Workbench.

- The setting “s+gui” is useful if you need to run code that requires the Spotfire S+ GUI, such as displaying plots in a graphsheet device or if you are creating and accessing menu items and toolbar buttons.

- The setting “console” is useful if you are running a script in BATCH mode using TIBCO Spotfire Statistics Services, and you need to write graphic files to disk or write data to a database.
- The setting “s+java” is useful if you need to run code that requires the Spotfire S+ Workbench.

S_ECLIPSE=[value]

This variable is useful for working with sample code or libraries that include calls that work only in the Spotfire S+ Workbench or only in the Spotfire S+ GUI.

If you develop code in the Spotfire S+ Workbench and deploy to other than BATCH (local), you can use this variable to distinguish test and production modes. (See S_INTERACTIVE_STATES.)

If you develop code for the Spotfire S+ GUI, you can use this to determine the environment. For example, the nSurvival library includes calls to guiCreate and guiRemove functions in their .First.lib and .Last.lib objects, respectively. You can wrap these functions in conditional statements that check whether Spotfire S+ Workbench is running, and then behave appropriately. See Chapter 5 of the *Spotfire S+ Workbench User’s Guide* for more information.

S_FIRST=function

S_FIRST specifies a function that will be executed upon start-up, immediately after Spotfire S+ finishes its initialization. It can be used to execute routine tasks related to setting up your work environment, for department wide functions or other initialization. If set, it overrides .First. See the section Enhancing Spotfire S+ (page 26) for specifics.

SHOME=directoryPath

SHOME refers to the directory where Spotfire S+ is installed, which contains the Spotfire S+ application files. Spotfire S+ libraries, data sets, and other related files are stored in subdirectories under the top-level directory specified by **SHOME**. Spotfire S+ determines the location of the Spotfire S+ installation by referring to the parent

directory where the executable is stored. You should never need to change **\$HOME** in Spotfire S+ but expert users can explicitly define it if they move their Spotfire S+ files to another directory.

S_INTERACTIVE_STATES=[value]

Has the value "yes" in all contexts except BATCH (TIBCO Spotfire Statistics Services).

You might have routines that access different data or acquire different parameter inputs in test mode versus production mode, where production mode is BATCH (TIBCO Spotfire Statistics Services). In this case, the logic of the routines could determine whether to access test or production data and parameters, and the same code could be used in both test and production modes.

S_NOAUDIT=[value]

If you set this variable (to any value), Spotfire S+ does not write an audit file. This is useful if you do not need a record of the commands you've typed into Spotfire S+. (In **splus.exe**, the default is to not write an audit file.) If this variable is not set, Spotfire S+ maintains a record of your Spotfire S+ commands (and other information) in a file called **.Audit** in your data directory. The audit file accumulates the commands from each Spotfire S+ session, so it may naturally grow large. The following setting causes Spotfire S+ not to maintain this record:

```
S_NOAUDIT=YES
```

If **S_NOAUDIT** is set to any value, the **.Audit** file will not be opened or written into.

If you keep an audit file, it can grow very large. To reduce the size of the audit file, use the **/TRUNC_AUDIT** command line switch. See Page 21 for details.

S_PATH=directoryPath[;directoryPath[...]]

No longer supported; see **SV4_SEARCH_PATH**.

S_PREFS=directoryPath

Directory to use for storing user preferences. It defaults to %S_PROJ%\Prefs. See the section Working With Projects (page 24).

S_PRINT_COMMAND=WindowsCommand

Windows command to use for printing the **Commands** window. By default, the command is “**Notepad /p**”.

S_PROJ=directoryPath

S_PROJ sets the defaults for **S_PREFS**, **S_DATA**, and **S_CWD**. By default, **S_PROJ** is set as follows:

On Microsoft® Windows XP®:

**C:\Documents and Settings\username\
My Documents\Spotfire S+ Projects\Project1.**

On Microsoft Vista™:

**C:\Users\username\Documents\
Spotfire S+ Projects\Project1**

You can change **S_PROJ** if you want to move your Spotfire S+ data files to another directory, or if you begin a new project. For example, if your name is Jay and you want to create a home directory for your personal use, you could create the directory **C:\JAY**, and then set **S_PROJ** to **C:\JAY** by setting it as a command line variation:

SPLUS.EXE S_PROJ=C:\JAY

Note
S_PROJ has replaced the HOME variable used in previous versions of Spotfire S+. Internally HOME remains a synonym for S_PROJ for compatibility with previous versions.

S_PS2_FORMAT=[value]

If you set the **S_PS2_FORMAT** variable, to any value, Spotfire S+ puts a CTRL-D character at the end of any PostScript file it generates. This is for compatibility with older PostScript formats. By default, Spotfire S+ does not put the CTRL-D character in the file.

S_SCRSAVE=integer

Set the **S_SCRSAVE** variable if you want to limit the amount of output saved for rollback in the **Commands** window. For example, to limit the number of characters saved to a maximum of 100KB, set this variable as follows:

```
S_SCRSAVE=100
```

Note

This variable also imposes a limit on the number of commands available for recall.

S_SILENT_STARTUP=value

If you set the **S_SILENT_STARTUP** variable (to any value), the Spotfire S+ copyright and version information are displayed, and the location of **S_DATA** is *not* displayed when the **Commands** window is opened. The Spotfire S+ **Commands** window then appears with a prompt.

S_TMP=directoryPath

Set the **S_TMP** variable to the name of the directory where you want Spotfire S+ to create temporary scratch files. By default **S_TMP** is unset, so temporary files are created in **S_CWD**, the process current working directory.

If the directory specified by **S_TMP** does not exist or cannot be accessed, it will be ignored. If you want Spotfire S+ to create temporary scratch files in the **C:\TEMP** directory, first create the directory **C:\TEMP**. Then, set **S_TMP** to **C:\TEMP**:

```
SPLUS.EXE S_TMP=C:\TEMP
```

S_USER_APPDATA_DIR=[directory]

Under interactive and BATCH cases, **S_USER_APPDATA_DIR** points to **%APPDATA%\TIBCO\splus major/minor version _ platform**.

Under server cases (that is, TIBCO Spotfire Statistics Services and TIBCO Spotfire Statistics Services Local Adapter), it points to the **appdata** directory inside the **data** directory. In both cases, it specifies

the location containing the **library** directory, where binary packages are found by default when you invoke the `library()` and `module()` functions.

For example, you can use this variable to specify a binary package build target when you run a batch script inside the Spotfire S+ Workbench to automate building a user's binary packages.

SV4_SEARCH_PATH=directoryPath[;directoryPath[...]]

Set the `SV4_SEARCH_PATH` variable only if you want to override the standard Spotfire S+ search list every time you start Spotfire S+. By default, `SV4_SEARCH_PATH` is set by Spotfire S+, and it includes the built-in Spotfire S+ functions and data set libraries. You can display these libraries using the `S` command `search`. As with other variables and their values, enclose the value with matched quotes if the directory paths include spaces.

Switches

Note
Unlike variables, switches do not use equal signs between the switch and any necessary values. If you need to include an equals sign, use quotes around the entire token.

/BATCH stdin [stdout [stderr]]

BATCH may be followed with one, two, or three file names indicating where **stdin**, **stdout**, and **stderr**, respectively, should be redirected. Specify “**stdin**,” “**stdout**,” or “**stderr**” to maintain the default input and output processing for any of these values. **stdin** is typically the name of a text file containing valid Spotfire S+ commands.

When Spotfire S+ is run in batch mode, a small dialog appears to notify the user that it is running rather than the normal Spotfire S+ window. Once completed running a **BATCH** session, Spotfire S+ automatically terminates. Use a script file for running Spotfire S+ commands automatically without automatically terminating Spotfire S+ when done, although that does not allow one to redirect `stdin`, `stdout`, or `stderr`.

For example, save the Spotfire S+ function

```
test.splus.func()
```

to a file named test. Now, simply create a shortcut and specify the SHOME and S_PROJ settings and the input/output names in the start-up command line (in the Target field):

```
C:\Program Files\TIBCO\splus82\cmd\splus.exe  
SHOME="C:\Program Files\TIBCO\splus82"  
S_PROJ="C:\Documents and Settings\username\My Documents\  
S-PLUS Projects\Project1"  
/BATCH  
"C:\Documents and Settings\username\My Documents\  
S-PLUS Projects\Project1\test"  
"C:\Documents and Settings\username\My Documents\  
S-PLUS Projects\Project1\testout"
```

Note that test is not limited to a single line. It could include a number of functions to run in the background.

/BATCH_PROMPTS

BATCH_PROMPTS specifies whether any progress, non-fatal warning or error, and/or exit status dialog should be displayed. By default, whenever Spotfire S+ runs in batch mode, it displays a “Pre-processing” and then a “thermometer” dialog to indicate the batch session’s progress. When batch mode is completed, the progress dialogs disappear, but no completion dialog is displayed to indicate the batch session’s success or failure. Also, if a non-fatal warning or error occurs that would have displayed a dialog in interactive mode, that dialog is suppressed and its message (and default response) is written to the batch output file. Use the **/BATCH_PROMPTS** switch to change this default behavior. The value “Yes” turns on all dialogs, “No” turns off all dialogs. To apply these values to a particular dialog, use the prefixes “progress:”; “non-fatal:”; and “exitstatus:”. For example, to display the exit status dialog but suppress the progress dialogs, use the following switch:

```
/BATCH_PROMPTS progress:no,exitstatus:yes
```

When specifying the prefix/word pairs, use either “,” or “;”, but no spaces, to separate them.

If an error occurs in one of the batch files while Spotfire S+ is in batch mode, a dialog is displayed indicating the error, unless both progress and exitstatus dialogs are suppressed.

/COMMANDS_WINDOW_ONLY

Starts Spotfire S+ with only the **Commands** window displayed, overriding any “Open at Startup” preferences.

Note

The next two switches below provide a convenient mechanism for developers to run the utility file of the same name (**/CHAPTER** runs **CHAPTER.EXE** and so on). All values that follow the command line switch are passed directly to the batch file, once the **SHOME** environment variable is set in the spawned process.

/CHAPTER [-b] [-d directoryPath] [-m] [-o] [-r filePath] [-s] [-u]

Creates a Spotfire S+ chapter, optionally compiling and linking C/C++ and/or FORTRAN code to be dynamically loaded, and optionally sourcing S code. This is the standard way to compile and link your C/C++ and/or FORTRAN code for use with Spotfire S+. The following additional switches are recognized by the **CHAPTER** utility:

-b

A shortcut for specifying both the **-m** and **-s** switches.

-d directoryPath

Specifies the directory in which to create the Spotfire S+ chapter. Remember to enclose **directoryPath** in quotes if it contains spaces. By default, the Spotfire S+ chapter is created in the Spotfire S+ startup directory, which can be set with **S_CWD**.

-m

Makes an **S.dll** in the chapter directory, which is loaded automatically when the chapter is attached by Spotfire S+. Any file with a **.c**, **.cxx**, **.cpp**, **.f**, or **.for** extension will be compiled if it does not have (or is newer than) an associated **.obj** file. The associated **.obj** files will be linked with the module definition file **S.def** (which will be created from the **.obj** files if it does not exist) to create **S.dll**.

-o

Creates a single file (**__objects**) containing all the S objects that are to be part of the Spotfire S+ chapter. This is an alternative to the typical user-created Spotfire S+ chapter in which each S object is contained in its own separate file. Note that if additional objects are assigned to the chapter in subsequent sessions of Spotfire S+, they cannot be contained in the **__objects** file. The files **all.Sdata** and **meta.Sdata** must have been previously created, using the function `dumpForObjects` in the chapter directory; if the chapter already contains a **.Data** directory, that directory must not contain any objects or meta data.

-r filePath

Specifies the rules for creating **S.dll**. By default, this is **SHOME\cmd\mrules.mak**; the default can be changed in the **Plus.ini** file, located in the **SHOME\cmd** directory.

-s

Starts up Spotfire S+ to source any files with a **.q**, **.s**, or **.ssc** extension (S source code files) and assign the resulting objects to the chapter.

-u

Creates the necessary subdirectories that allow the Spotfire S+ function `undo` to operate on the chapter.

/TRUNC_AUDIT integer

Spawns the **TRUNC_AUDIT** program that truncates the **.Audit** file (found in the data directory), removing the oldest part of the file. The integer argument specifies the maximum size (in characters) to keep in the audit file and defaults to 100,000; **TRUNC_AUDIT** only removes entire commands, not partial commands.

/MULTIPLEINSTANCES

Specifies that multiple instances of Spotfire S+ are allowed. For example, more than one session of Spotfire S+ or different versions of Spotfire S+ can be run at once if this is enabled; however, you must specify different working databases when using multiple instances.

/Q

Causes Spotfire S+ to automatically quit after any script file processing. This is automatically set when the **/BATCH** switch is specified.

/REGISTEROLEOBJECTS

Registers Spotfire S+'s Automation components in the registry. This permits other programs to access Spotfire S+ functions programmatically via ActiveX Automation (previously known as OLE Automation). See Chapter 6, Automation, for more information.

/REGKEY KeyName

Specify alternative registry key under:
HKEY_CURRENT_USER\Software\TIBCO.

This is useful for expert users who wish to maintain multiple versions of Spotfire S+ on one system. KeyName defaults to Spotfire S+.

/UNREGISTEROLEOBJECTS

Removes the Automation components (installed by **/REGISTEROLEOBJECTS**) from the registry.

/UTILITY ProgramName values

Runs the specified ProgramName, passing it all values that follow, once the **\$HOME** environment variable is set in the spawned process.

Script Files

See the Windows version of *User's Guide*, Chapter 9, for information about using script files.

Checking Default Environment Settings

The following table describes six environments in which you can run Spotfire S+, and describes the combination of environment settings they return. To see these result, at the command prompt, type:

```
getenv(c("S_DISPLAY_MODE", "S_ECLIPSE",  
        "S_INTERACTIVE_STATES", "S_USER_APPDATA_DIR"))
```

Table 2.2: Default environment settings for six possible Spotfire S+ environment.

Environment	S_DISPLAY_MODE	S_ECLIPSE	S_INTERACTIVE_STATES	S_USER_APPDATA_DIR
Spotfire S+ Workbench	"s+java"	"T"	"yes"	"%APPDATA%\TIBCO\ <plus major /minor version_ platform"
Spotfire S+ GUI	s+gui	""	"yes"	"%APPDATA%\TIBCO\ <plus major /minor version_ platform"
TIBCO Spotfire Statistics Services	"s+java"	""	"yes"	SPSERVER_HOME/data/ appdata
TIBCO Spotfire Statistics Services Local Adapter	"s+java"	""	"yes"	"%APPDATA%\TIBCO\ Spotfire Statistics Services Local Adapter/ <i>version</i> / data/appdata"
BATCH (local)	"s+java"	"T"	"yes"	"%APPDATA%\TIBCO\ <i>splus major</i> <i>/minor version_</i> <i>platform</i> "
BATCH (TIBCO Spotfire Statistics Services)	"console"	""	""	"%AppData%\Local\TIB CO\ <i>splus major</i> <i>/minor version_</i> <i>platform</i> "

WORKING WITH PROJECTS

This section illustrates ways intermediate users can use the various switches discussed above to customize Spotfire S+ on a project by project basis.

Spotfire S+ now makes it quite easy to maintain distinct data and preferences directories for each of your projects. For each project:

1. Make a copy of the Spotfire S+ desktop shortcut by holding the CTRL key down as you drag a copy of the shortcut to its new location,
2. Edit the command line (right-click, or, for older versions of Windows, select it and press ALT-ENTER) to set **S_PROJ**:

```
SPLUS.EXE S_PROJ=c:\Banana\Survey
```

When you use the new shortcut, Spotfire S+ will use **c:\Banana\Survey\Data** for the project data and **c:\Banana\Survey.Prefs** for the project preferences.

There are many ways to customize this to fit your particular needs. For instance you may want to use various **.Data** directories and a common **.Prefs** directory.

S_PROJ is used to set the default values for **S_CWD** (to %S_PROJ%), **S_PREFS** (to %S_PROJ%\Prefs), and **S_DATA** (to %S_CWD%\Data;%S_PROJ%\Data).

The Preferences Directory

Spotfire S+ is very customizable. Your preferences are all stored in a variety of files in a directory with the name **.Prefs**.

Note

The only exception to this is that the Options object is stored in the **S_DATA** directory for backward compatibility reasons.

Upon creation of a new **.Prefs** directory, it is populated with “template” preference files from the **%SHOME%\MasterPrefs** directory, so expert users who wish to permanently set a particular preference may edit the templates. You should make a copy of the original templates for future reference.

The Data Directory

Whenever you assign the results of a Spotfire S+ expression to an object, using the `<-` operator within a Spotfire S+ session, Spotfire S+ creates the named object in your data directory. The data directory occupies position 1 in your Spotfire S+ search list, so it is also the first place Spotfire S+ looks for a Spotfire S+ object. You specify the data directory with the variable **S_DATA**, which can specify one directory or a colon-separated list of directories. The first valid directory in the list is used as the data directory, and the others are placed behind it in the search list.

Like other variables, **S_DATA** is referenced only at the start of a Spotfire S+ session. To change the data directory during a session, use the `attach` function with the optional argument `pos=1`, as in the following example that specifies **MYSPLUS\FUNCS** as the data directory:

```
attach("C:\MYSPLUS\FUNCS", pos=1)
```

If **S_DATA** is not set, Spotfire S+ sets the data directory, to one of two directories according to the following rules:

1. If a subdirectory named **.Data** exists in **S_CWD**, the current working directory, Spotfire S+ sets the data directory to this **.Data** subdirectory.
2. Otherwise Spotfire S+ checks to see if the **%S_PROJ%\Data** directory exists. If so, it will be used as the default location for objects created in Spotfire S+. If not, it will be created.

Note

Although **S_DATA** may be used to provide alternative directory names (other than **.Data**), in practice some code depends on this being set to **.Data**. Therefore it is recommended that **S_DATA** be used primarily to set the path to a particular directory named **.Data**, and not to change the name of the directory itself.

ENHANCING SPOTFIRE S+

With the instructions in this section, you can:

- Add functions or modify system functions to change default values or use different algorithms.

Note

Keep a careful log of how you modify Spotfire S+, so that you can restore your changes when you receive the next update.

Adding Functions and Data Sets to Your System

You may need to add or modify Spotfire S+ functions. This section describes how to add or modify functions.

1. Start Spotfire S+.
2. Create a version of the function or data set you want to add or modify with a command such as the one below, where *my.function* is the name of the function or data set you want to modify:

```
> fix(my.function)
```

3. Run *my.function* to make sure that it works properly (you don't want to install a bad version).
4. Create a directory for your new and modified functions:

```
new.database ("modfuncs")
```

where *modfuncs* is the name of your directory.

5. Use the search command to see a list of available Spotfire S+ system directories:

```
> search()
```

The position of a directory in this list is called its *index*. Each row in the list displays the index of the first directory in the row.

6. Create a function `.First.local` in the directory in search position 2, as follows:

```
> setDBStatus(2,T)
> assign(" .First.local", function()
+ attach(modfuncs, pos = 2),
+ where=2)
```

Each time you start Spotfire S+, Spotfire S+ executes `.First.local` if it exists. This `.First.local` attaches your new and modified functions directory ahead of all built-in Spotfire S+ functions, but behind your data.

7. Attach your *modfuncs* directory to your current session:

```
> attach("modfuncs", pos=2)
```

8. Assign your new or modified function to its permanent home:

```
> assign("my.function", my.function, where=2)
```

Warning

Be careful when you modify system functions, because you may have to repeat the installation procedure if you make a mistake. You should keep a careful change log, both to guide your own troubleshooting and to assist support staff in solving any problems you report.

THE SYSTEM INTERFACE

Using the Spotfire S+ interfaces to Windows[®] and DOS, you can run your favorite Windows and DOS applications from the Spotfire S+ prompt, or incorporate those applications in your Spotfire S+ functions. For example, the `fix` and `edit` functions use the Windows interface to start an editor on a file containing a Spotfire S+ object to be edited. Similarly, the `objdiff` function uses the DOS interface to run the `fc` command on ASCII dumps of two Spotfire S+ objects to be compared.

This chapter describes the DOS and Windows interfaces and provides several examples of Spotfire S+ functions you can write to incorporate spreadsheets, databases, and word processors into your Spotfire S+ programming environment.

Using the Windows Interface

To run a Windows application from Spotfire S+, use the `system`, `systemOpen`, or `systemPrint` functions, which all require one argument: for `system`, a character string containing a command suitable for the Windows “Run” command line, and for `systemOpen` or `systemPrint`, a character string containing a filename or URL. For example, to run the Windows Calculator, you could call `system` as follows:

```
> system("calc")
```

The Windows Calculator accessory pops up on your screen, ready for your calculations. By default, Spotfire S+ waits for the Windows application to complete before returning a Spotfire S+ prompt. To return to Spotfire S+, close the application window.

To run Windows applications concurrently with Spotfire S+, so that you can type Spotfire S+ expressions while the Windows applications are running, use `system` with the `multi=T` argument:

```
> system("calc", multi=T)
```

The Windows Calculator accessory pops up on your screen, and the Spotfire S+ prompt immediately appears in your Spotfire S+ **Commands** window. (You can, of course, always start Windows applications as usual.)

Commonly used calls should be written into function definitions:

```
calc <- function() { system("calc", multi=T) }  
notepad <- function() {system("notepad", multi=T) }
```

The command argument to `system` can be generated using the `paste` and `cat` functions. For example, the `ed` function, used by both `fix` and `edit` to actually call an editor, pastes together the name of an editor and the name of a file to create the command used by `system`. (The `ed` function distributed with Spotfire S+ actually uses `win3`, a wrapper for `system`, instead of calling `system` directly. The `win3` function is now deprecated, but it predates `system` on the Windows platform):

```
ed <- function(data, file=tempfile("ed."), editor="ed",  
error.expr)  
{  
  ...  
  system(paste(editor, file), trans = T)  
  ...  
}
```

The argument `trans=T` is useful for converting strings containing UNIX-type directory names (e.g., `“/betty/users/rich”`) to strings containing DOS-type directory names (e.g., `“\\betty\\users\\rich”`). It can also save you typing, since it allows you to substitute forward slashes for the double backslashes required to represent a single backslash in Spotfire S+. If `trans=T`, literal forward slashes must be enclosed in single quotes. For example, Notepad uses the flag `/p` to print a file. To print a file in `infile.txt` in the directory `c:\rich`, you could use `system` as follows:

```
> system("notepad '/p' c:/rich/infile.txt", trans = T)
```

Note that the single quotes can surround the entire flag, not just the forward slash; in fact, one set of quotes can surround all necessary flags.

If you try the above example on one of your own files, you will notice that the Notepad window appears on your screen with the text of the file while Notepad is printing. You can force Notepad to run in a minimized window by using the `minimize=T` argument:

```
> system("notepad '/p' c:/rich/infile.txt", trans=T,  
        minimize=T)
```

There are two arguments to `system` that control how it behaves when an error occurs in starting or exiting the application. The more commonly used is the `on.exec.status` argument, which controls how `system` behaves when an error occurs in starting the application. If the application specified in the `command` argument could not be started, Spotfire S+ queries the operating system for a character string that briefly describes the error that occurred. It then calls the function specified (as a character string) in the `on.exec.status` argument, passing it the error string. The default for the argument is “stop”, so that all current Spotfire S+ expressions are terminated. For example, if you wanted to run the Wordpad application, but the directory in which it resides is not in your `PATH` environment variable, you would get the following result:

```
> system("wordpad")  
Problem in eval(expression(system("wordpad"))): Unable to  
execute 'wordpad', exec.status = 2 (The system cannot find  
the file specified. Full path needed?)  
Use traceback() to see the call stack
```

Specifying the full path to the Wordpad application successfully starts it:

```
> system("\\Program Files\\Accessories\\wordpad")
```

You may substitute the name of another function for the `on.exec.status` argument, so long as the function’s only required argument is a character string. For example, suppose you wanted to open a file for jotting down some notes in an ASCII editor, but you weren’t particular as to which editor you opened. You could write a `whiteboard` function to call Wordpad, but use `on.exec.status` to try Notepad if Wordpad wasn’t available, as follows:

```
> whiteboard  
function()  
{  
  system("wordpad", multi = T,  
        on.exec.status = "trying.notepad")  
}
```

The `trying.notepad` function is defined as follows:

```
> trying.notepad
function(message = NULL)
{
  print(message)
  print("Trying to start notepad.\n")
  system("notepad", multi = T,
        on.exec.status = "trying.edit")
}
```

As in the initial whiteboard function, `trying.notepad` calls `system` with an alternative function as its `on.exec.status` argument. The `trying.edit` function uses a call to the `dos` function to start the MS-DOS editor, and uses the default `on.exec.status` behavior, that is, if you can't find any of Wordpad, Notepad or MS-DOS editor, whiteboard fails.

A less commonly used argument, because most Windows applications do not return a useful or documented exit status, is `on.exit.status`, which controls how `system` behaves when the application returns a non-zero exit status. The default for this argument is "", so that no action is taken by `system`; you may substitute the name of any function for this argument, again so long as its only required argument is a character string. For example, if you had knowledge that a particular application returned a non-zero exit status when some condition was (or was not) met, you could have this condition reported as follows:

```
> system("myapp", on.exit.status="my.report")
```

If `myapp` returned a non-zero exit status, the function `my.report` would be called with the argument "'myapp' returned with exit.status=n", where `n` is the exit status value.

The `.16.bit` argument, which was useful when Spotfire S+ supported the Win32s API, is now deprecated.

To automatically run the application associated with a particular file type or URL, use the `systemOpen` or `systemPrint` functions. For example, to display a web page in your default browser, you could call `systemOpen` as follows:

```
> systemOpen("http://www.tibco.com")
```

The arguments available to the system function (other than the deprecated `olb.bit`) are also available to the `systemOpen` and `systemPrint` functions.

The argument `with` is also available to the `systemOpen` and `systemPrint` functions; this allows you to temporarily override the association for the particular file type or URL and run the application specified by `with` instead.

Using the DOS Interface

While the Windows interface allows you to run Windows applications from Spotfire S+, it cannot be used to run internal DOS commands (such as **dir** and **copy**), nor can it return a command's output as a Spotfire S+ vector. The DOS interface provides a way to perform these tasks.

To run internal DOS commands from Spotfire S+, use the `dos` function. For example, to get a listing of files in your home directory, use `dos` with the **dir** command as follows:

```
> dos("dir")
[1] ""
[2] " Volume in drive C has no label"
[3] " Volume Serial Number is 6146-07CB"
[4] " Directory of C:\\RICH"
[5] ""
[6] ".          <DIR>    12-07-92  5:01p"
[7] "..         <DIR>    12-07-92  5:01p"
[8] "__DATA"   <DIR>    12-07-92  5:02p"
[9] "DUMP      Q          74 01-14-93  2:51p"
[10] "WINWORK  TEX        10053 12-13-92  4:08p"
...
```

By default, the output from the DOS command is returned to Spotfire S+ as a character vector, one element per line of output. In the case of the **dir** command, the first five to seven lines of output will seldom change. A Spotfire S+ function that strips off this repetitive information may be of more use than the simple call to `dos`:

```
dir <- function(directory="") {
  dos(paste("dir", directory))[-(1:5)]
}
```

Including the `directory` argument allows using the function to get a listing of an arbitrary directory.

If you don't want the output from the DOS command returned to Spotfire S+, use the argument `output.to.S=F` in the call to `dos`:

```
> dos("copy file1 b:", output = F)
```

The output from the DOS command is displayed in a "DOS box", which in this example will close automatically when the DOS command is completed. As with the `system` function, you can specify `minimize=T` to force the DOS box to run minimized. You can run DOS applications concurrently with Spotfire S+ by combining `output=F` with the `multi=T` argument. For example, to open the DOS text editor concurrently with Spotfire S+, use `dos` as follows:

```
> dos("edit", output = F, multi = T)
```

A DOS box opens on your screen with the DOS text editor loaded.

Warning
When you use <code>dos</code> with <code>multi=T</code> , you must explicitly close the DOS box when you're done with it. It does not close when the DOS command finishes executing.

Forward slashes can be translated to backslashes using the `trans=T` argument; this can save you typing (since one forward slash equals two backslashes), and is also useful if you are sharing files on a UNIX file system:

```
> dos("edit c:/rich/infile.txt", output = F, trans = T)
```

Two other arguments to `dos` are used less frequently---`input` and `redirection`. The `input` argument can be used to write data to a file that can then be passed as input to the DOS command. More often, however, such data is simply pasted into the command specified by the `command` argument. The `redirection` argument is a flag that can be used with `input`; if `redirection=T`, the input is passed to the command using the DOS redirection operator `<`, otherwise the input is passed as an argument to `command`. See the `dos` help file for more information.

Introduction	36
Resources	36
Examples: An Application and a Called Routine	38
Creating a Simple Application	38
Example of Calling a C Function Via <code>.Call</code>	41
Compiling and Executing C++ on UNIX	44
Compiling and Executing C++ on Windows	45
CONNECT/C++ Class Overview	46
Data Object Classes	46
Function Evaluation Classes	46
Client-to-Engine Connection Classes	47
Evaluator Classes	47
CONNECT/C++ Architectural Features	49
CSPobject	49
Constructors and Generating Functions	49
Constructing From an Existing Object	50
Assignment Operators	51
Overloading Operators	51
Converting C++ Objects to S-PLUS Objects	52
Subscripting Operators	53
Subscript and Replacement Operations	54
Subscript and Arithmetic Operations	54
Matrix Computations	55
Printing to Standard Output	56
Named Persistent Objects	56
Storage Frames For Unnamed Objects	58
A Simple Spotfire S+ Interface in Windows	60
Creating a Dialog-Based Application	60
Connecting to Spotfire S+	61

INTRODUCTION

CONNECT/C++ is a tool used for interfacing C++ with the S language. It is a convenient tool for integrating the S-PLUS engine inside other programs written in C++, but it can also be used for integrating C++ code into the Spotfire S+ environment.

To enable communication between the GUI (Graphical User Interface) and Spotfire S+, CONNECT/C++ was developed to provide a framework for the S language version 4-based engine used in Spotfire S+ for Windows[®]. In fact, the Spotfire S+ GUI provides the most comprehensive example of using CONNECT/C++ to integrate the S-PLUS engine with C++ applications. Similarly, C++ developers could create their own GUI to interface with Spotfire S+ using the same technique.

CONNECT/C++ is a class library providing C++ classes with member functions that operate on S-PLUS objects similar to S methods in the S language. Users can use these classes and their member functions to create and manipulate persistent as well as local S objects.

CONNECT/C++ provides various mechanisms for evaluating S expressions inside a C++ program and module. Spotfire S+ ships with several examples that illustrate how to use this library. Some of these examples contain pairs of equivalent S and C++ functions that perform the same tasks. The speed of the C++ functions can be many times faster than the S code depending on the code's complexity and the data sizes. The examples are located in the **SHOME/sconnect** directory, where **SHOME** is your Spotfire S+ installation directory.

Resources

For more information on CONNECT/C++:

- **On Windows[®]**: go to **SHOME/help/ConnectC++ Class library.htm**.
- **On Linux[®] or Solaris[®]**: go to **SHOME/sconnect/help/ConnectC++.Class.library.htm**.

This HTML file is a guide to the CONNECT/C++ class library for C++ developers, and it discusses how to connect to the S-PLUS engine, how to create data objects, call S-PLUS functions, and evaluate S-PLUS syntax.

EXAMPLES: AN APPLICATION AND A CALLED ROUTINE

CONNECT/C++ can be used for two distinct purposes: to create C++ applications that can access Spotfire S+ functionality, and to create C++ functions that can be called via the S-PLUS `.Call` interface. We begin our investigation of CONNECT/C++ with a simple example of each.

Creating a Simple Application

The CONNECT/C++ application used in this example is a console application that creates two S-PLUS vectors. It then uses Spotfire S+ to compute a linear model relating the two vectors.

The code begins with the inclusion of `sconnect.h`, the CONNECT/C++ library which all CONNECT/C++ code must reference at the start. It then declares a global S-PLUS connection object, with the CONNECT/C++ class `CSPengineConnect`, before beginning the main application function. The `CSPengineConnect` class generates a connection between the client application and Spotfire S+, allowing you to create S-PLUS objects in the permanent frame, notifying you when the databases are attached or detached to the client, and evaluating S language expressions. Here's what the code looks like so far:

```
#include "sconnect.h"

// A global connection object
CSPengineConnect g_engineConnect;

int main(int argc, char* argv[])
{
```

The first step in the main function is to create the actual connection object, which opens a connection to Spotfire S+:

```
// Create the connection to Spotfire S+
g_engineConnect.Create( argc, argv);
```

We then create the variables `x` and `y` to use in the regression. The CONNECT/C++ class `CSPnumeric` is used to store S-PLUS numeric vectors. The `CSPnumeric` class is one of many in CONNECT/C++ that are used to represent S-PLUS objects within C++. Similar classes

exist for most of the standard atomic objects in Spotfire S+ (see Table 3.1). The `Create` method creates instances of the class; the `Assign` method assigns the class to a Spotfire S+ database:

```
// Create S object with name "x" in the current database.
// Same as x<-1:10 at the command line.
CSPnumeric sx;
sx.Create("1:10","x");

// Squaring sx, which is the same as S expression
// sy <- x*x in a local frame, but here we set it to local
// C++ variable sy.

CSPnumeric sy = sx * sx;

// Assign the result as S object with name "y" in the
// current database.
sy.Assign("y");
```

Finally, we fit the linear model, passing the appropriate call to Spotfire S+ via the `CONNECT/C++` method `SyncParseEval`:

```
// Evaluate z<-lm(y~x)
g_engineConnect.SyncParseEval("z<-lm(y~x)");

return 1;
}
```

The complete code for this example is in the directory **SHOME/samples/spllm** (Windows) and **SHOME/sconnect/samples/splm** (UNIX). The C++ code for both platforms is in the file **spllm.cxx**.

To run the application, open a Command Prompt or MS-DOS window (Windows) or compile (UNIX):

1. Change the current directory to the directory containing the code:

```
cd SHOME/samples/spllm
```

if you are on Windows or

```
cd /sconnect/samples/splm
```

on UNIX, where **SHOME** is your Spotfire S+ installation directory.

2. Build the program:

```
devenv spl1m.dsp /make
```

on Windows or

```
Splus CHAPTER -sconnectapp *.cxx
```

```
Splus make
```

on UNIX.

Note

If you are using Microsoft Visual Studio Express on Windows, the build command is `vcbuild`.

3. If you are on Windows, check the PATH environment variable to make sure it includes `%SHOME%\cmd`, and that it follows `%SHOME%`. This path is not added by default to the Spotfire S+ installer. You must add it before running the next step.

4. Run the program:

```
spl1m.exe S_PROJ=.
```

on Windows or

```
Splus EXEC S.app
```

on UNIX.

To verify the results, start the Spotfire S+ console version in the same directory (Windows) or start Spotfire S+ (UNIX):

```
sqpe.exe S_PROJ=.
```

on Windows and Spotfire S+ returns the following:

```
S-PLUS : Copyright (c) 1988, 2010 TIBCO Spotfire Inc.  
S: Copyright TIBCO Spotfire Inc.  
Version 8.2.0 for Microsoft Windows : 2010  
Working data will be in C:/Program Files/splus82/users/  
username
```

or enter

```
Splus
```

on UNIX to return this:

```
S-PLUS : Copyright (c) 1988, 2010 TIBCO Spotfire Inc.
```

Version 8.2.0 for Sun SPARC, SunOS 5.8 : 2010
Working data will be in .Data

and look at the objects x, y, and z:

```
> x
[1] 1 2 3 4 5 6 7 8 9 10
> y
[1] 1 4 9 16 25 36 49 64 81 100
> z
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)  x
          -22  11

Degrees of freedom: 10 total; 8 residual
Residual standard error: 8.124038
```

Example of Calling a C Function Via

The Gauss-Seidel method is a familiar technique for solving systems of linear equations. The algorithm is straightforward and easy to implement in Spotfire S+:

```
.Call
gaussSeidel<-
# gaussSeidel solves a linear system using Gauss-Seidel
# iterative method.
# REQUIRED ARGUMENTS:
#     A and b are numeric matrix and vector respectively.
# VALUE:
#     a vector x, solution of A x = b
#
# Usage:
# A<-matrix(rnorm(100),nrow=10)
# diag(A)<-seq(ncol(A),ncol(A)) #Make it diagonally
#                               # dominant
# b<-rnorm(ncol(A))
# sys.time({x1<-gaussSeidel(A,b)})
function(A,b)
{
  # Hard-coded relative tolerance and max iterations
  tol<-1.0e-4
```

```
maxItr<-1e4

# Validating
A <- as.matrix(A)
b <- as.numeric(b)
if(nrow(A)!=ncol(A) || ncol(A)!=length(b))
  stop("nrow(A)!=ncol(A) || ncol(A)!=length(b)")

# Begin Gauss-Seidel step
x<-b
for(k in 1:maxItr)
{
  xOld<-x
  for(i in 1:nrow(A))
  {
    s<- A[i,i]*x[i]
    for(j in 1:ncol(A))
      s <- s - A[i,j]*x[j]
    x[i] <- (b[i]+s)/A[i,i]
  }
  # Check convergence; continue if necessary
  if(max(abs((x-xOld)/x)) < tol)
    return(x);
}
warning("Solution does not converge\n")
return(x)
}
```

This code, which involves a nested loop, could be made more efficient, but the intention is to illustrate the Gauss-Seidel iteration in its most familiar form. An example including the implementation of CONNECT/C++ is shown below, and notice that by using the classes and methods of CONNECT/C++, this code closely resembles the equivalent computation in Spotfire S+.

The code begins by including the **sconnect.h** header file to give us access to the CONNECT/C++ library. Next, it includes the header file required for the Gauss-Seidel code itself:

```
# include "sconnect.h"
# include "gausssdl.h"
```

We then declare the `gaussSeidel` object as an object of class `s_object`, as required by the `.Call` interface:

```
s_object* gaussSeidel(s_object* ps_A, s_object* ps_b)
```

As is typical for S-PLUS code, we declare the `S_EVALUATOR` and then embed the implementation in a `try-catch` block. Within the `try` block, the tolerances are hard-coded. We then construct the C++ equivalents to the S-PLUS objects `A` and `b`:

```
{
  S_EVALUATOR
  try
  {
    // Hard-coded relative tolerance and max iterations
    double tol = 1e-4;
    long maxItr = 1000;

    // Constructing and validating C++ objects
    CSPnumericMatrix A(ps_A);
    CSPnumeric b(ps_b);
    if(A.nrow()!=A.ncol() || A.ncol()!=b.length())
      PROBLEM "A.nrow()!=A.ncol() || A.ncol()!=b.length()"
      ERROR;
  }
}
```

The actual Gauss-Seidel step follows:

```
// Begin Gauss-Seidel step
CSPnumeric x=b;
for(long k =1; k<= maxItr; k++)
{
  CSPnumeric xOld = x;
  for(long i= 1; i <= A.nrow(); i++)
  {
    double s = A(i,i) * x(i);
    for(long j = 1; j <= A.ncol(); j++)
      s = s - A(i,j) * x(j);
    x(i) = (b(i)+s)/A(i,i);
  }
  // Check convergence; continue if necessary
  if(Max(abs((x-xOld)/x)) < tol)
    return(x);
}
```

```
        PROBLEM "Solution does not converge" WARN;  
        return(x);  
    }  
    catch(...)  
    {  
    }  
    return(blt_in_NULL); // return the built-in NULL object  
}
```

Compiling and Executing C++ on UNIX

The complete code for this example is in the directory **SHOME/samples/gausssdl**, with the C++ code in the file **gausssdl.cxx**.

To compile and execute the C++ code:

1. Change the current directory to the directory containing the code:

```
cd SHOME/samples/gausssdl
```

2. Build the share library:

```
Splus CHAPTER -sconnectlib *.cxx  
Splus make
```

3. Run Spotfire S+:

```
Splus
```

With the makefile created by CHAPTER, compiling your code is simple: just run the make command as a Spotfire S+ utility as shown in step 2.

The Splus in front of make allows Spotfire S+ to set its environment variables appropriately before calling the standard make utility; in particular it defines the **SHOME** environment variable used in the makefile.

The make utility executes the necessary commands to compile and link the C++ code into the shared object `S.so`. Note that `-sconnectlib` is required to include the CONNECT/C++ library.

CONNECT/C++ called via `.Call` runs considerably faster than the Spotfire S+ code. The following is a comparison for a 100 column by 100 row matrix A using a Pentium III with 512MB of RAM on Windows:

```
> A<-matrix(rnorm(10000),nrow=100); diag(A)<-seq(ncol(A),  
+ ncol(A)) # Make it diagonally dominant  
> b<-rnorm(100);
```

```
> sys.time({x1<-gaussSeidel(A,b)})  
[1] 19.328 19.354
```

Here is a comparison for a matrix A with 100 columns and 100 rows on a Solaris machine:

```
[1] 37.00 39.35
```

If we compare `sys.time` on both platforms:

```
> sys.time({x2<- .Call('gaussSeidel',A,b)})  
[1] 0.07 0.07
```

is the Windows output, while

```
[1] 0.04 0.04
```

is the UNIX output.

The CONNECT/C++ version ran over 250 times faster in Windows and about 1000 times in UNIX than the pure Spotfire S+ version!

Compiling and Executing C++ on Windows

1. Browse to the folder **SHOME/samples**. This directory contains all samples and solutions for building them.
2. Read and follow the instructions in the file `readme.txt`.
3. Browse to the folder **SHOME/samples/GaussSDL** and read the **readme.txt** for more information about using the interfaces with the Connect C++ classes.

CONNECT/C++ CLASS OVERVIEW

The class library provides a set of classes that can be used to create and manipulate persistent data objects, run S-PLUS functions, parse and evaluate S-PLUS expressions, and receive output and notification when objects are changed or when databases are attached and detached.

The following sections provide an overview of specific categories of classes used to accomplish these operations.

Data Object Classes

Data object classes provide methods to create and operate on arrays, matrices, and vectors. To use these classes to create a data object, simply call the object constructor or call the `Create()` method. For a persistent object, specify the name of the object and an S language expression you want to parse, evaluate, and assign the result in order to initialize it with data. Alternatively, a data object can be constructed using a form of the constructor that takes an optional S language expression as an argument. This is useful if named (persistent) objects are not required, but initialization is required. Once the object is created, methods can be used to operate on the object.

To receive notification in a client application when a data object changes, create a new class in the client application derived from the appropriate base class and override the virtual methods for handling object notification. When a named object is modified or removed, those virtual methods in the client are called.

Function Evaluation Classes

The `CSPcall` class allows S-PLUS functions to be evaluated with arguments passed to the function. Arguments are any `S_object` as well as objects derived from `CSPobject`, which may include data objects and other S-PLUS objects. Results are returned as a `CSPobject` to the client. To use this class, simply call the object constructor with the name of the function to run and any arguments you wish to pass from the client to the function.

Client-to-Engine Connection Classes

The `CSPengineConnect` class creates a connection between the client and the S-PLUS engine. This connection permits creation of objects in the permanent frame, creation of persistent unnamed objects outside of `.Call` routines, notification in the client when databases are attached or detached, output routing to the client, and evaluation of S language expressions.

To use `CSPengineConnect`, create a new class derived from `CSPengineConnect` in the client, override the virtual methods for receiving database attach/detach notification, and output notification, and add a member variable to the client application class object to record a reference to a single instance of this derived class.

Use of the `CSPengineConnect` class is only necessary when one or more of the following features is desired in the client program:

- Integrate S+ engine DLLs (Windows) or the shared library **libSqpe.so** (UNIX) with another application (client).
- Notification in the client when databases are attached or detached and when changes are made in persistent objects.
- Output redirected to the client.

For more information on using `CSPengineConnect`, please see the section on this class by going to **SHOME/help/Connect/C++ Library Help** in Windows, or **SHOME/sconnect/help/ConnectC++.Class.library.htm** in UNIX.

Evaluator Classes

The `CSPevaluator` class manages memory resources, errors, the top-evaluation frame, and a set of local evaluation frames. Although it is optional, instantiating an object of `CSPevaluator` class at the top of a try block can speed up the code, and the corresponding catch block receives an exception error when an unexpected error occurs in the S-PLUS engine.

To use `CSPevaluator`, create an instance of this class at the top of a try block as shown below:

```
double minValue = 0;
try
{
    // Open top-level-evaluator (frame 1) if it is closed
    CSPevaluator sEvaluator;
    CSPnumeric myNumeric = sEvaluator.eval("1:10");
}
```

```
        minValue = myNumeric.Min(); //minValue = 1

    } // Close top-level evaluator when sEvaluator is out of
      // scope
    catch(...)
    {
        // Unexpected error occurred in the engine
    }
}
```

For more information on using `CSPevaluator`, please see the section on this class, please see the section on this class by going to **SHOME/help/Connect/C++ Library Help** in Windows, or **SHOME/sconnect/help/ConnectC++.Class.library.htm** in UNIX.

CONNECT/C++ ARCHITECTURAL FEATURES

The following sections describe the basic architectural features in the class library and some of the specific programming features available in the library that make it possible to perform S-PLUS operations efficiently in client programs and modules written in C++. Classes and methods discussed in this section are fully documented in the reference sections for the classes in the online help for CONNECT/C++.

CSPobject

CSPobject is the base class of most of the classes that represent S-PLUS classes. It provides common functionality to its derived classes, and its most important data member is:

```
s_object* CSPobject::m_ps_object
```

A class that represents a S-PLUS class inherits `m_ps_object` because CSPobject is its base class. As a smart pointer, a derived class of CSPobject provides safer methods to manipulate the data pointed by `m_ps_object` as compared to using global C functions. For example, the constructor, the destructor, and the assignment operators automatically increment and decrement reference counts whenever appropriate to provide the same data sharing mechanism as that of the SV4 language.

All CSPobject-derived classes have a method called `IsValid()` which allows you to test whether the member `m_ps_object` is valid or not.

Constructors and Generating Functions

Often, S generating functions are more convenient than the S method `new`. Similarly, constructors of CONNECT/C++ classes can provide the same convenience. They have the following form:

```
CSPclass::CSPclass(const char* pszExpression);
// pszExpression is a string representing valid S code.
```

where `class` is a CSPobject-derived object.

This form of the object constructor parses and evaluates `pszExpression` and uses the resultant S-PLUS object as its value. Normally, `pszExpression` should contain a S-PLUS expression that calls to an appropriate generating function. However, it works for any S-PLUS expression that returns a valid S-PLUS object, and the

constructor automatically coerces the returned object to the class that it represents. It increments the reference count upon completion, as well. In case of errors, the constructor throws an exception in the client application.

For example:

```
CSPevaluator s;

CSPinteger x("1:4");           // x<-1:4
CSPnumeric y("fuel.frame[,1]"); // y<-as(fuel.frame[,1],
// 'numeric')
CSPnumeric z("new('numeric')"); // z<- new('numeric')
CSPmatrix A("matrix(1:4, nrow=2)"); // A<-matrix(1:4,
// nrow=2)
CSPmatrix B("1:4");           // B<-as(1:4,'matrix')

// Do something with x,y,z,A, and B
```

Constructing From an Existing Object

You can construct new objects from existing objects using one of the following forms:

```
CSPclass::CSPclass(const CSPclass& sObject); //copy
//constructor
CSPclass::CSPclass(s_object* ps_object); //construct
//from s_object
```

where class is a CSPobject-derived object.

The copy constructor of a CONNECT/C++ class behaves like a S-PLUS assignment operator when the S-PLUS object name is first used. They both share the same data with the object names used to construct them. However, for the CONNECT/C++ classes, sharing is not possible if the classes are incompatible. It increments the reference count upon completion.

An example of creating new objects from existing objects follows:

```
CSPevaluator s;

CSPnumeric x("1:4"); // x<-1:4
CSPnumeric u(x);     // u<-x # u shares data with x
CSPmatrix A(x);     // A<-as(x,'matrix') # A shares data with x
```

```

CSPcharacter v(x); // v<-as(x,'character') # no sharing

s_object* ps_object = x.GetPtr();//Get pointer to s_object*
CSPnumeric U(ps_object); // U shares data with x
CSPmatrix a(ps_object); // a shares data with x

```

Assignment Operators

The assignment operator of an CONNECT/C++ class behaves like a S-PLUS assignment operator when the S-PLUS object name is already used. However, the left-hand-side object of the operator = is an existing and valid object. The assignment operator decrements the reference count on the old object and increments the reference count on the new object before swapping the two object pointers:

```
CSPclass& CSPclass::operator=(const CSPclass& sObject);
```

where class is a CSPobject-derived object.

An example of the assignment operator follows:

```

CSPevaluator s;

CSPnumeric x("1:4"); // x<-1:4
CSPnumeric u = x; // u<-new('numeric'); u<-x # u shares
// data with x
CSPmatrix A = x; // A<-new('matrix'); A<-as(x,'matrix')
// # no sharing

CSPnumeric y; // y<-new("numeric")
u = y; // u<-y # u switches to share data with y
A = y;//A<-as(y,'matrix') # A switches to share data with y

```

Overloading Operators

CONNECT/C++ contains some useful overloading operators such as +, -, * and /. These operators perform element-by-element operations in the same way as in the S language. However, for the matrix class, the * operator is different. The operator for CSPmatrix is a real matrix multiplication operator equivalent to the S %** operator.

```

CSPclass& CSPclass::operator+(const CSPclass& sObject);
CSPclass& CSPclass::operator-(const CSPclass& sObject);
CSPclass& CSPclass::operator*(const CSPclass& sObject);
CSPclass& CSPclass::operator/(const CSPclass& sObject);

```

where class is a CSPobject-derived object.

An example using the `CSPmatrix` follows:

```
CSPevaluator s;

CSPnumeric x("1:4"); // x<-1:4
CSPnumeric y ("4:1"); // y<-4:1
y = y+x*x; // y<-y+x*x

CSPmatrix A("matrix(1:4,nrow=2)");//A <- matrix(1:4,nrow=2)
CSPmatrix B("matrix(4:1,nrow=2)");//B <- matrix(4:1,nrow=2)
CSPmatrix D = A*A + B*B; //D <- A %*% A + B %*% B
```

Converting C++ Objects to S-PLUS Objects

Objects derived from class `CSPobject` are C++ representations of S-PLUS objects; within Spotfire S+, S-PLUS objects are represented as C objects of type `s_object*`. Sometimes, an application needs to access the `s_object*` directly. For example, the arguments and the return value of all `.Call` interfaces must be of type `s_object*`.

The `CSPobject` class provides a convenient way to automatically convert to `s_object*`. Simply use a `CSPobject` wherever a `s_object*` is required. It automatically invokes a conversion operator that returns the `s_object*` as appropriate.

```
s_object* CSPobject::operator*();
s_object* CSPobject::operator&();
```

For example:

```
s_object* myCall()
{
    CSPnumeric x("1:10");
    return x;
}
s_object *pReturn = myCall();
```

The return statement, `return x`, first typecasts `x` to type `s_object*`. This invokes the conversion operator `s_object *`(`)` of the `CSPnumeric` class (derived from `CSPobject`) which ensures that the destructor of `x` does not delete the object, even if the reference count drops to zero.

Subscripting Operators

CONNECT/C++ contains some useful overloading subscripting operators () for the derived classes of CSPvector and CSParray such as CSPnumeric and CSPmatrix. The proxy class of the returned object provides supports for read/write and mixed-mode operations:

```
const double CSPnumeric::operator()(long lIndex); const
    // Fortran style indexing starting from index 1
    // rvalue only

CSPproxy CSPnumeric::operator()(long lIndex);
    // Fortran style indexing and ordering
    // lvalue and rvalue
```

An example using the subscripting operators:

```
CSPevaluator s;

CSPnumeric x("c(0.1, 0.2, 0.8, 0.9)"); // x<- c(0.1, 0.2,
                                         //      0.8, 0.9)
double d = x(1); // d <-x[1] # d is 0.1
d = d + x(2); // d<- d+x[1] # d is 0.3
double e = (long) x(1); // e<-as.integer(x[2]) # e is 0
long n = x(1); // n <-as.integer(x[1]) # n is 0
n = n + x(2); // n <- n+as.integer(x[2]) # n is still 0
```

The following is another example using the subscripting operator for a matrix:

```
CSPevaluator s;

CSPmatrix A("matrix(c(0.1, 0.2, 0.8, 0.9), 2)");
    // A<- matrix(c(0.1, 0.2, 0.8, 0.9), 2)

double d = A(1,1); // d <-A[1,1] # d is 0.1
d = d + A(2,1); // d<- d+A[2,1] # d is 0.3
long e = (long) A(2,1); // e<-as.integer(A[2,1]) # e is 0
long n = A(1,1); // n <-as.integer(A[1,1]) # n is 0
n = n + A(2,1); //n <- n+as.integer(A[2,1]) # n is still 0
```

Subscript and Replacement Operations

If a subscript operator of a CSPobject-derived class returns an lvalue object of CSPproxy, the operation involves replacing an element of the S-PLUS object. Since writing data is not possible for a shared S-PLUS object, CSPproxy must determine whether to copy data before replacing its elements. This action occurs in one of its overloaded assignment operations:

```
CSPproxy& CSPproxy::operator=(long);
CSPproxy& CSPproxy::operator=(double);
CSPproxy& CSPproxy::operator=(const CSPproxy&);
```

For example:

```
CSPevaluator s;

CSPnumeric x("1:4"); // x<- 1:4
x(1) = 0.0;          // x[1]<- 0 # x is not share,
                    // simply set x[1] to 0.0
x(2) = x(1);        // x[2]<- x[1] # x is not share, simply
                    // set x[2] to 0.0
CSPnumeric y(x);    // y<- x # y shares data with x
y(1)= 10.0;         // y[1]<- 10 #copy and replace:
                    // y[1] is 10 and x[1] is 0
```

Subscript and Arithmetic Operations

Some overloaded operators are available to support mixed-mode arithmetic operations involving subscripting objects of classes derived from CSPobject. These operators, +, -, *, and /, perform mixed-mode operations following the same rules as Spotfire S+:

```
long CSPproxy::operator+(long)
double CSPproxy::operator+(double)
...
```

An example using the arithmetic operators:

```
CSPevaluator s;

CSPnumeric x("1:4"); // x<- 1:4
CSPnumeric y(x);     // y<- x # y shares data with x
// A <- matrix(1:4,nrow=2)
CSPmatrix A("matrix(1:4,nrow=2)");
```

```

// e <- A[1,1] + A[1,2]
double e = A(1,1)+A(1,2);
// A[1,2] <- e*(A[1,1]+A[2,1])
A(1,2) = e*(A(1,1)+A(2,1));
// A[2,2] <- x[1]*A[1,1]+y[2]*A[2,1]
A(2,2) = x(1)*A(1,1)+y(2)*A(2,1);

// X<-array(1:16, c(2,2,2,2))
CSParray X("array(1:16, c(2,2,2,2))");
// X[1,1,1,1] <- X[2,1,1,1]+e;
X(1,1,1,1) = X(2,1,1,1) + e;
// X[2,1,1,1] <- y[1] - X[2,1,1,1];
X(2,1,1,1) = y(1) - X(2,1,1,1);
// X[1,2,1,1] = A[1,1] * X[2,1,1,1];
X(1,2,1,1) = A(1,1) * X(2,1,1,1);

```

Matrix Computations

Some overloaded functions are available for matrix computations, such as the example below (in UNIX). These computations are multi-threaded on some platforms (currently Windows 2000[®], NT, and XP on Intel multi-processor machines).

```

double CSPmatrix::ConditionNumber(void);
CSPmatrix SPL_Multiply(const CSPmatrix& A,
                      const CSPmatrix& B);
CSPnumeric SPL_Multiply(const CSPmatrix& A,
                      const CSPnumeric& x);
...

```

For example:

```

CSPEvaluator s;

CSPmatrix A("matrix(5:8, nrow=2)");
// A<- matrix(5:8, nrow=2)
CSPmatrix B(A); // B<- A
CSPmatrix D = SPL_Multiply(A, B); // D<-A %*% B

CSPnumeric x("1:2"); // x<- rnorm(2)
CSPnumeric y = SPL_Multiply(A, x); // y<- A %*% x

```

Printing to Standard Output

You can use the following CONNECT/C++ method to print to the S-PLUS standard output stream:

```
void CSPObject::Print(void);
```

For example:

```
CSPevaluator s;
```

```
CSPcharacter message("'hello'"); //message <- 'hello'  
message.Print();                //print(message)
```

```
CSPmatrix M("matrix(1:4,nrow=2)");//M<-matrix(1:4, nrow=2)  
M.Print();                       //print(M)
```

Named Persistent Objects

All CSPObject-derived objects are placeholders for an `s_object` that exists in the engine. So, this C++ object can reference an `s_object` or none at all, depending on whether the member `s_object` pointer points to a valid `s_object`. All CSPObject-derived classes have a method called `IsValid()` which allows you to test whether it is pointing to a valid `s_object` or not.

All named objects are created in a permanent frame associated with a Spotfire S+ database, and are thus persistent between calls and between sessions in the S engine. When you create a new CSPObject in your client program, a new `s_object` is created in the S engine. When you delete this CSPObject, the `s_object` is also released in the engine. However, when you execute S-PLUS expressions to remove the `s_object` that your CSPObject points to, such as by using `rm(myObject)`, or you call the `Remove()` method on the object, the CSPObject is not deleted in your client. The `OnRemove()` method of the CSPObject in your client is called and the base class version of this method “disconnects” your CSPObject from the now released `s_object` by setting the member `s_object` pointer to NULL. After this event, calling `IsValid()` on the CSPObject returns FALSE.

Deleting the CSPObject in your client program does not automatically remove the permanent frame `s_object` in the S-PLUS engine that this CSPObject refers to. You must call the method `Remove()` to remove the `s_object` from the engine.

You can create named objects using the `Create()` method of the various object classes derived from CSPObject, such as `CSPnumeric`. Whenever these objects are modified, the `OnModify()` method is

called in your client program. Whenever these objects are removed, the `OnRemove()` method is called in your client program. Only named objects support this kind of client program notification.

To create a named object in your client, first derive a new class from the appropriate `CSPObject`-derived class, such as `CSPnumeric`. Then, construct an instance of this derived class using the constructor, then call the `Create()` method to specify the name you wish to give the object. It is important to derive a new class from the `CSPObject`-derived class instead of just using the base class directly in your client because the `OnModify()` and `OnRemove()` methods are virtual and must be overridden in your derived class in the client in order to be notified when these events occur.

A `CSPObject` can be modified in one of two ways. It can be modified in the client program by using the operators available for the object to assign and operate on the elements of the object. When this kind of modification is done, it is necessary to call the `Commit()` method on the object to commit it to the S-PLUS engine before any changes to the object are reflected in the persistent `s_object` that is referenced by the object in the client.

Another way it can be modified is by evaluating S-PLUS expressions, such as by using `CSPengineConnect::SyncParseEval()`. When this kind of modification is done, it is not necessary to call `Commit()` on the object, as the `s_object` is automatically updated by the S-PLUS engine. For both kinds of modification, the `OnModify()` method of the `CSPObject` is called in the client program. It is important to call the base class `OnModify()` in your override of `OnModify()`. This allows the base class to update the member `s_object` pointer to point to the newly modified `s_object` in the engine.

The `s_object` member of a `CSPObject` can be removed (invalidated) in one of two ways:

1. It can be removed in the client program by calling the `Remove()` method on the `CSPObject`. This method removes the `s_object` from the permanent frame and triggers a call to the `OnRemove()` method of the `CSPObject` in the client program. The base class version of `OnRemove()`, which should be called at the end of the overridden version in the client, releases the member `s_object` from the `CSPObject`.

2. It can be removed by evaluating S-PLUS expressions, such as by calling `CSPengineConnect::SyncParseEval()`. This also triggers a call to the `OnRemove()` method of the `CSPobject` in the client program.

In Windows, for examples of using `CSPobject`-derived classes in a client program and responding to `OnModify()` and `OnRemove()` notifications, see the example C++ client program called **SSC** located in **SHOME/samples/SSC** in the subdirectory.

Storage Frames For Unnamed Objects

Normally, when you create an unnamed `CSPobject` in a client routine that you call via `.Call`, the `s_object` corresponding to this `CSPobject` is “alive” or is valid until the routine ends and scope changes out of the routine.

If you create an unnamed `CSPobject` when the S-PLUS evaluator is not open, the `s_object` corresponding to this `CSPobject` may not be valid. For most client applications, this is usually inadequate. Therefore, you need to do the following to ensure that an unnamed `CSPobject` created in a client application does not get released until the end of the client routine:

- Create an instance of a `CSPevaluator` at the top of the scope “{.”
- Create and use any unnamed `CSPobject`-derived objects in the client.

For example:

```
{
    CSPevaluator s;
    CSPnumeric x("1:10");
    ...
}
```

For named objects, you do not have to use the above approach: simply create named `CSPobject`-derived objects using the constructor and a call to `CSPobject::Create()`. For further information, see the online help for the classes `CSPengineConnect::OpenTopLevelEval()`, `CSPengineConnect::CloseTopLevelEval()`, and the `Create()` method for the object type to be created.

Table 3.1: *CONNECT/C++ classes and their S-PLUS counterparts.*

S-PLUS Class	CONNECT/C++ Class	Example
any	CSPobject	CSPobject x("2")
numeric	CSPnumeric	CSPnumeric x("2.32")
integer	CSPinteger	CSPinteger x("2")
logical	CSPlogical	CSPlogical x("c(T,F)")
character	CSPcharacter	CSPcharacter("abcd")
named	CSPnamed	CSPnamed("c(a=1,b=2, d=3)")
matrix	CSPmatrix CSPnumericMatrix CSPcharacterMatrix	CSPmatrix A("matrix(1:4,2)") CSPnumericMatrix A("matrix(rnorm(12,6)") CSPcharacterMatrix A("matrix(letters[1:12],6)")
array	CSParray	CSParray B("array(1:8,c(2,2,2))")
list	CSPlist	CSPlist("list(1:2,6:700)")
function	CSPfunction	CSPfunction ("function(x) x^2")
call	CSPcall	CSPcall("lm")

A SIMPLE SPOTFIRE S+ INTERFACE IN WINDOWS

In this section, we build a small client application to accept S-PLUS expressions, send these expressions to Spotfire S+ for evaluation, and then return the output from Spotfire S+. As part of this process, we use numerous features of CONNECT/C++.

Before continuing, you must perform the steps described in the file **SHOME\samples\readme.txt**.

Warning

Most of this example was generated automatically using Visual C++, and it uses Microsoft Foundation Classes (MFC). If you are not familiar with MFC, you can ignore the uses of MFC where they occur.

Creating a Dialog-Based Application

To keep the example application as simple as possible, create it as a dialog-based application in Visual C++.

Create the basic interface:

1. Open Microsoft Visual Studio[®].
2. From the menu, click **File > New > Project**.
3. Under **Project types**, expand **Other Languages** and, from the list, click **Visual C++**.
4. In the **Templates** pane, click **MFC Application**.
5. For **Name**, provide the project name **spint**.
6. Click **OK**. The **MFC Application Wizard** appears.
7. In the dialog **Application Type**, select **Dialog based**. You can accept all other defaults.
8. Click **Finish** to create the application skeleton.

The **Solution Explorer** displays the project's header files, resource files, and source files, along with a **ReadMe.txt**, which contains a brief description of the other files in the skeleton. For this example,

we work with the files **spint.rc**, **spint.cpp**, **spint.h**, **spintDlg.cpp**, and **spintDlg.h**. First, edit **spint.rc**, the dialog resource displayed for editing when you created the application skeleton.

1. Delete the **TODO** static text displayed in the skeleton dialog resource.
2. Open the **Toolbox** palette and drag two **Static Text** fields and two **Edit Controls** onto the dialog.
3. Rename the **Static Text** fields to **S-PLUS Commands** and **S-PLUS Output**, respectively. (Hint: You can edit these names in the **Properties** dialog in the controls' **Caption** fields. The **Properties** dialog is available from the right-click menu.)
4. In the **Properties** dialog for the **S-PLUS Output** edit control, set the **Read Only** property to **True**.
5. Rename the **OK** button to **Run Commands**.
6. Reposition and resize the controls in the dialog to resemble Figure 3.1.

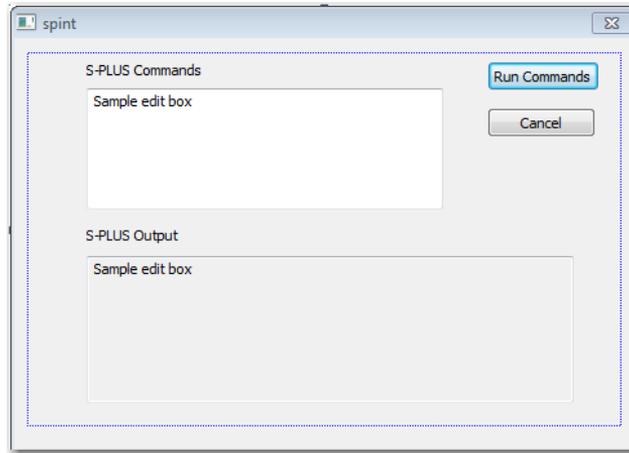


Figure 3.1: *Our simple Spotfire S+ interface.*

Connecting to Spotfire S+

To establish a connection between Spotfire S+ and the dialog, you must edit the main header file, **spint.h**, and the main source file, **spint.cpp**.

Change spint.h

- At the top of the file, include the header file **sconnect.h** immediately after the resource header file:

```
// spint.h : main header file for the
// PROJECT_NAME application
//
#pragma once

#ifdef __AFXWIN_H__
    #error "include 'stdafx.h' before including
    #this file for PCH"
#endif

#include "resource.h" // main symbols
#include "sconnect.h"
```

Change spint.cpp

1. Add the following code immediately *after* the line reading CspintApp theApp:

```
// The S-PLUS Engine Connection object
CSPengineConnect g_engineConnect;
```

In the section of the code titled CSpintApp initialization, add the following code immediately *before* the line reading CSpintDlg dlg::

```
// Create and connect to S+ engine
int argc =1;
char *argv[1];
argv[0]="spint";

g_engineConnect.Create( argc, argv);
```

Now you have an interface and a connection to the engine. All that remains is to define the code that reads the input from the **S-PLUS Commands** edit control and writes the output to the **S-PLUS Output** edit control. The following sections address this task.

Add and edit an event handler to the Run Commands button

1. Reopen the dialog resource **spint.rc**.

2. Right click the button **Run Command** and, from the menu, click **Add Event Handler**.
3. Make sure the message type is BN_CLICKED, the classed used is CspintDlg and the function handler name is OnBnClickedOk.
4. Click **Add and Edit**. The function skeleton appears.
5. Change the function to the following:

```
void CspintDlg::OnBnClickedOk()
{
    CWnd* pEdit = GetDlgItem(IDC_EDIT1);
    CString Commands1;
    pEdit->GetWindowText(Commands1);
    CSPevaluator sEvaluator;
    CSPobject returnVals =
        sEvaluator.Eval((LPCTSTR)Commands1);
    CSPcharacter outputText = returnVals.Deparse();
    CString outputText2 = outputText[0];
    CWnd* pEdit2 = GetDlgItem(IDC_EDIT2);
    pEdit2->SetWindowText(outputText2);
}
```

This function:

- a. Takes the input
- b. Reads it into a CString object
- c. Sends it to Spotfire S+ for evaluation
- d. Returns the output as an array of character strings
- e. Takes the first output string and puts it into the output field.

For the project to compile, you must change the character set used by the project.

Set the project Properties page

1. Click **Configuration Properties > General** and change the **Character Set** option to **Use Multi-Byte Character Set**.
2. Click **Linker > Input** and, in the field **Additional Dependencies**, add **sconnect.lib**.

3. Click **C/C++ > General**, and to the field **Additional Include Directories**, add **[SHOME]\sconnect** and **[SHOME]\include**

This implementation has one significant limitation: we get only the first string in the array of strings that forms the return value. That is, the output pane displays only the first line of output.

A solution to this problem is provided by defining a new class derived from `CSPengineConnect` that includes a new method for `OnOutput`. You can view this solution by exploring the code in **SHOME/samples/ssc**, a more sophisticated "simple" interface to Spotfire S+.

Introduction	66
Java Tools	66
Example Files	66
Calling Java from Spotfire S+	67
Static Fields	67
Static Methods	70
Class Files	72
Instance Methods	73
Managing Java Object Instances	74
Calling Spotfire S+ from Java Applications	81
Running the Java Program	81
Evaluating S-PLUS Commands	82
Using SplusDataResult Objects	83
Example Applications	86

INTRODUCTION

CONNECT/Java is a powerful programming language with a variety of attractive features. Its strengths include a clean object-oriented design, a rich set of class libraries, and cross-platform capabilities.

Spotfire S+ may be used with Java in a variety of ways. Just as Spotfire S+ can call C and Fortran routines, it can call Java methods to perform computations. Alternatively, a Java program can call Spotfire S+ to perform computations and create graphs.

Java Tools

The Spotfire S+ installation includes a copy of the Java 2 Runtime Environment (JRE). This runtime environment includes the Java Virtual Machine and related classes necessary to run any Java 2-compliant compiled Java code. Users with compiled Java code need no additional tools to use their code with Spotfire S+.

Users writing new code need a Java 2 development environment such as the Java 2 JDK. For Spotfire S+, the desired JDK is version 1.6.

This is freely available from Sun[®] for Solaris[®], Linux[®], and Windows[®] platforms, and from other vendors on other platforms. Licensing restrictions prevent TIBCO Software Inc. from redistributing the Java 2 JDK with Spotfire S+.

Example Files

In Windows, the Java code samples for the examples discussed in this chapter are included in ***.java** files in the **SHOME\library\winjava\examples** directory. In UNIX, these files are located in the **SHOME/library/example5/java** directory.

The **java** directory distributed as part of the **example5/java** library in UNIX contains the file **examples.jar**. This file includes the byte-compiled ***.class** files for all of the ***.java** files. As these classes are available to Java by default, it is not necessary to compile or install the Java code to run the examples.

CALLING JAVA FROM SPOTFIRE S+

Spotfire S+ can exchange information with Java via static fields and static methods. Currently, Spotfire S+ does not have a way to directly call non-static methods, which would require the S-PLUS engine to keep track of references to particular instances of Java objects.

Functionality that is available through instance methods (non-static methods) may be accessed by creating static Java methods that take care of the details of creating Java instances, tracking them, and calling methods on them. These static methods may then be called from Spotfire S+.

Static Fields

Java objects may be passed from Java to Spotfire S+ through static fields. The S-PLUS function `.JavaField` returns a field value given the following information:

- The name of the Java class containing the field. The package name may be forward slash or period delimited.
- The name of the field.
- The Java Native Interface (JNI) field descriptor for the field.
- The optional client argument.

For example, the following call will return the value of PI:

```
> .JavaField("java/lang/Math", "PI", "D")
```

The `.JavaField` function is the primary mechanism for transferring values from Java into Spotfire S+.

Field Descriptors

The JNI is the standard interface for passing information between Java and native C code. In JNI the type of a field is declared using a *field descriptor*. For example, an `int` field is represented with "I", a `float` field with "F", a `double` field with "D", a `boolean` field with "Z", and so on.

The descriptor for a reference type such as `java.lang.String` begins with the letter "L", is followed by the *class descriptor*, and is terminated by a semicolon. (The class descriptor is typically the class name delimited with slashes.) The field descriptor for a string is "Ljava/lang/String;".

Descriptors for array types consist of the "[" character, followed by the descriptor of the component type of the array. For example, "[I" is the descriptor for the `int[]` field type.

Spotfire S+ uses the field descriptors to communicate field types to Java. Spotfire S+ can access values of type `void`, `int`, `short`, `long`, `boolean`, `float`, `double`, `byte`, `char`, and `String`. It can also access arrays of these types. Table 4.1 presents each supported Java type, the JNI field descriptor, and the corresponding S-PLUS type. All S-PLUS atomic types except `complex` have Java equivalents.

The latest version of Spotfire S+ extends the connection to allow access of any Java object. If the object is an array of objects or an object whose class implements `java.util.Collection` (such as `Vector` or `ArrayList`), then the result will be a S-PLUS list. For any other type of object, the `toString()` method will be used to return a string.

Table 4.1: *JNI Field Descriptors*

JNI Field Descriptor	Java Type	Spotfire S+ Type
V	<code>void</code>	NULL
I	<code>int</code>	integer
S	<code>short</code>	integer
J	<code>long</code>	integer
Z	<code>boolean</code>	logical
F	<code>float</code>	single
D	<code>double</code>	double (numeric)
B	<code>byte</code>	raw
C	<code>char</code>	character
Ljava/lang/String;	<code>String</code>	character

The javap utility included with the Java 2 SDK will print out the JNI field descriptors for a class. For example:

```
% javap -s -p java.lang.Math
Compiled from Math.java
public final class java.lang.Math extends java.lang.Object
{
    public static final double E;
        /*    D    */
    public static final double PI;
        /*    D    */
    private static java.util.Random randomNumberGenerator;
        /*    Ljava/util/Random;    */
    private static long negativeZeroFloatBits;
        /*    J    */
    private static long negativeZeroDoubleBits;
        /*    J    */
    ...
}
```

Integer Conversions

S-PLUS integers are equivalent to C longs. They are either 32 bit or 64 bit in size depending upon whether the operating system is 32 bit or 64 bit. In Java, shorts are 16 bit, ints are 32 bit, and longs are 64 bit. All three of these types are converted to S-PLUS integers.

Special Values

Spotfire S+ has special values NA, Inf, and -Inf to represent missing values, positive infinity, and negative infinity, respectively. Java has special values NaN, POSITIVE_INFINITY, and NEGATIVE_INFINITY for these cases. These special values are mapped appropriately when values are transferred.

Client Argument

Within a Java virtual machine (JVM) process, a static field will have a uniquely determined value. In client/server mode, we have two separate JVMs available. Sometimes we will be interested in determining the field value for the server JVM that shares a process with the S-PLUS engine, while at other times we are interested in getting the field value from the client JVM.

The `client` argument to `.JavaField()` is used to specify the JVM to use. The default is `client=F`, which means to use the server JVM. Specify `client=T` to use the client JVM.

If `client=T`, the Java object returned must be serializable.

Field Examples The `java.lang.Math` class contains a few interesting static fields that we can access from Spotfire S+:

```
> .JavaField("java/lang/Math", "PI", "D")
[1] 3.141593
> .JavaField("java/lang/Math", "E", "D")
[1] 2.718282
> .JavaField("java/lang/Double", "NaN", "D")
[1] NA
```

Static Methods Scalars and arrays of primitives may be passed from Spotfire S+ to a static Java method, and a Java object may be returned from the Java method to Spotfire S+. The S-PLUS function `.JavaMethod` takes the following arguments:

- The name of the Java class. The package name may be forward slash (/) or period (.) delimited.
- The name of the method.
- The JNI method descriptor indicating the types of the arguments.
- Optionally, one or more values used as the arguments to the method.
- Optional client argument, which is used similarly to the argument in `.JavaField()`.

For example, the following call will return 2 to the power of 10:

```
> .JavaMethod("java/lang/Math", "pow", "(DD)D", 2, 10)
[1] 1024
```

Note that `.JavaMethod` automatically converts the values to doubles based on the signature of the method. Unlike with the S-PLUS functions `.C` and `.Fortran`, the programmer does not need to assure that the values are of a particular type via calls to `as.double()`.

Method Descriptors

The JNI method descriptors are formed from the field descriptors for each method argument and the field descriptor for the return value. The argument types appear first and are surrounded by one pair of parentheses. Argument types are listed in the order in which they

appear in the method declaration. If a method takes no arguments, this is represented by empty parentheses. The method's return type is placed immediately after the right closing parenthesis.

For example, "(I)V" denotes a method that takes an `int` argument and has return type `void`, which is returned in Spotfire S+ as a `NULL`.

A method that takes a `String` argument and returns a `String` is denoted by "(Ljava/lang/String;)Ljava/lang/String;".

Arrays are again indicated with "[" character, followed by the descriptor of the array element type. A method that takes a `String` array and returns `void` has the method descriptor "([Ljava/lang/String;)V".

The `javap` utility included with the Java 2 SDK will print out the JNI method descriptors for a class. For example:

```
% javap -s -p java.lang.Math
Compiled from Math.java
public final class java.lang.Math extends java.lang.Object
{
    ...
    public static double toDegrees(double);
        /* (D)D */
    public static double toRadians(double);
        /* (D)D */
}
```

Client Argument In client/server mode, we have two separate JVMs available. Sometimes we will need to call a Java method on the server, and at other times we will want to call the method on the client.

The `client` argument to `.JavaMethod()` is used to specify the JVM to use. The default is `client=F`, which means to use the server JVM. Specify `client=T` to use the client JVM.

If `client=T`, the Java object returned must be serializable.

Simple Examples Here are some examples using `.JavaMethod`:

```
> .JavaMethod("java/lang/Math", "round", "(D)J", pi)
[1] 3
> .JavaMethod("java/lang/Math", "round", "(F)I", pi)
[1] 3
```

```
> .JavaMethod("java/lang/Math", "random", "()D")
[1] 0.6300195
> .JavaMethod("java.lang.System", "getProperty",
+ "(Ljava/lang/String;)Ljava/lang/String;", "os.name")
[1] "SunOS"
```

The `StaticMethodsExample` class provides various examples of how to retrieve values from Java. In Windows, the Java code for this class is in `$SHOME/library/winjava/examples`, and for UNIX, it is in `$SHOME/library/example5/java`.

Class Files

In order for the Java virtual machine to find Java classes, the class files must be in one of the locations where Java looks for files. The location may be specified using either the Java classpath, or the Java extensions mechanism.

The main difference between these two approaches appears to be a difference in strictness of security such that any extension class can be instantiated via reflection, while objects in the classpath may not always be instantiated with reflection. This is unlikely to be an issue for most users. However, if you get a “Class Not Found” exception when using the classpath mechanism and reflection, you may want to try the extension mechanism instead.

Java Classpath

Java finds class files that are in a location specified in the Java classpath. If the class files are in a **jar** file, the location will be the full path to the **jar** file including the name of the file. If the class files are in a directory, the location will be the full path to the directory.

Two locations you should always specify in your classpath are the **Splus.jar** file, located in `$SHOME/java/jre/lib/ext` (Windows and UNIX), and your current directory (or the directory in which you create your Java code).

The classpath may be modified from the UNIX shell by setting the `CLASSPATH` environment variable. An item may be added to the `CLASSPATH` using syntax such as the following at a UNIX prompt:

```
setenv CLASSPATH {$CLASSPATH}:/homes/user/examples
```

for `csh` and `tcsh`, and

```
CLASSPATH = {$CLASSPATH}:/homes/user/examples
export CLASSPATH
```

for sh, ksh, and bash.

The classpath may be specified in Windows by setting the CLASSPATH environment variable or using the `-classpath` argument to Java. As a general rule, avoid spaces in the `classpath` on Windows.

Java Extensions Java will automatically find any jar files placed in the **java/jre/lib/ext**. This is the default location for Java extensions.

Other directories may be added to the list of extension directories with the `-Djava.ext.dirs` argument to Java. If this is specified it must be a path which also lists the standard **java/jre/lib/ext** directory.

Instance Methods Instance methods are methods that reference a particular Java object rather than static class information. Instance methods may be called by writing static methods that manage creation and tracking of the necessary instances.

Random Numbers Example The class `java.util.Random` provides objects that represent pseudorandom number streams. We can obtain random numbers from Java by creating a class that instantiates a `java.util.Random` object and provides methods for getting values from the stream.

```
import java.util.Random;
public class RandomWrapperExample {
    static Random ran = new Random();

    public static double nextDouble(){
        return ran.nextDouble();
    }

    public static int nextInt(){
        return ran.nextInt();
    }
}
```

After compiling this class (typically by invoking `javac` on the Java source file) and adding its location to the classpath, we can call these methods from Spotfire S+:

```
> .JavaMethod("RandomWrapperExample", "nextDouble", "()D")
[1] 0.9606592
```

```
> .JavaMethod("RandomWrapperExample", "nextInt", "()I")  
[1] 4026360078
```

File Chooser Example

The `JFileChooser` class provides a file chooser dialog that is useful for locating and selecting a file. We can create a Java class that launches this dialog and returns the path to the selected file. The S-PLUS function for invoking the file chooser will let the user specify the starting location for the browser.

The S-PLUS function is:

```
fileChooserExample <- function(startPath = getenv("PWD")){  
  .JavaMethod("FileChooserExample", "showFileChooser",  
    "(Ljava/lang/String;)Ljava/lang/String;",  
    startPath, client = T)  
}
```

Note the use of the `client = T` argument to show the `JFileChooser` is on the client side.

The Java class definition is:

```
import javax.swing.JFileChooser;  
  
public class FileChooserExample {  
  public static String showFileChooser(String startPath){  
    JFileChooser fileChooser = new JFileChooser(startPath);  
    fileChooser.setDialogTitle("Select File");  
    int exitStatus = fileChooser.showDialog(null, "OK");  
  
    String selectedFileName = "";  
    if (exitStatus == JFileChooser.APPROVE_OPTION)  
      selectedFileName =  
        fileChooser.getSelectedFile().getAbsolutePath();  
  
    return selectedFileName;  
  }  
}
```

Managing Java Object Instances

Spotfire S+ does not directly attempt to track instances of Java objects. However, instances may be created using Java code, and identifiers to these instances may be passed back to Spotfire S+ for use in tracking the objects for further manipulation.

The basic technique is to use a static `Vector`, `HashTable`, or other type of collection in the Java class. When a new instance is created, it is placed in the collection and a key into the table is passed back to Spotfire S+. This key may be passed into a static method that then finds the object in the table and applies the relevant method to the object.

In this example, we will create a Java class representing a person and family relationship information about the person. When we create an object for the person, we specify the person's first and last name. We then have methods to indicate family relationships between individuals. In particular, we can indicate an individual's mother and father using `setMother()` and `setFather()` methods. These methods modify the individual to note the parenting information and also modify the parent's object to note that the individual is their child. We can retrieve information about the individual using the `getInfo()` method.

Before showing the Java code, let's see how these methods would be used from within Spotfire S+. In this example, we will use `.JavaMethod()` directly at the Spotfire S+ prompt. We could create S-PLUS functions to call these routines in order to avoid having to specify the class name and method signature each time we want to use a method.

First we will create three `FamilyMember` objects representing a mother, father, and son. The new `FamilyMember()` method creates an object in Java and returns an integer ID, which we can use to refer to the object.

```
> momId <- .JavaMethod("FamilyMember", "newFamilyMember",  
+ "(Ljava/lang/String;Ljava/lang/String;)I",  
+ "Sue", "Jones")  
> dadId <- .JavaMethod("FamilyMember", "newFamilyMember",  
+ "(Ljava/lang/String;Ljava/lang/String;)I",  
+ "Tom", "Jones")  
> sonId <- .JavaMethod("FamilyMember", "newFamilyMember",  
+ "(Ljava/lang/String;Ljava/lang/String;)I",  
+ "Skip", "Jones")
```

Next we will use the `setMother()` and `setFather()` methods to establish the relationship between the parents and the son. Note that we are using the identifiers returned above.

```
> .JavaMethod("FamilyMember", "setMother", "(II)Z",
+   sonId, momId)
[1] T
> .JavaMethod("FamilyMember", "setFather", "(II)Z",
+   sonId, dadId)
[1] T
```

Now that we have created the objects and specified their relationship, we can use `getInfo()` to examine the objects. The `getInfo()` method uses the family relationship information to determine the names of parents and children for the individual.

```
> .JavaMethod("FamilyMember", "getInfo",
+   "(I)[Ljava/lang/String;", sonId)
[1] "Name: Skip Jones" "Mother: Sue Jones"
[3] "Father: Tom Jones"
> .JavaMethod("FamilyMember", "getInfo",
+   "(I)[Ljava/lang/String;", dadId)
[1] "Name: Tom Jones" "Mother: Unknown"
[3] "Father: Unknown" "Child: Skip Jones"
```

The Java code for the `FamilyMember` class is straightforward. We present it here with comments describing the key points.

```
import java.util.Vector;

public class FamilyMember {
    /* This class is an example of creating and modifying a
       dynamic collection of instances using static methods.
       */

    // Track instances by keeping a static Vector of
    // instances. We will add each FamilyMember object to
    // this Vector when it is created, and return its index
    // in this Vector as the key for accessing the object.

    static Vector members = new Vector();

    // Instance variables. We get each person's first and
    // last name when the object is created. Methods are then
    // used to specify their mother and father. When the
    // person is specified as a mother or father, we know
    // they have a child, which we also track.
```

```
String firstName, lastName;
FamilyMember mother, father;
Vector children = new Vector();

/* Constructor */

public FamilyMember (String first, String last){
    firstName = first;
    lastName = last;
}

/* Instance methods */
/* Java routines would call these */

// public methods to get names

public String getFirstName(){
    return firstName;
}

public String getLastName(){
    return lastName;
}

// public methods to set and get parents

public void setMother(FamilyMember mom){
    mother = mom;
    mother.addChild(this);
}

public void setFather(FamilyMember dad){
    father = dad;
    father.addChild(this);
}

public FamilyMember getMother(){
    return mother;
}
```

```
public FamilyMember getFather(){
    return father;
}

// private method to add child when parent set

private void addChild(FamilyMember kid){
    children.add(kid);
}

// public method to get children

public Vector getChildren(){
    return children;
}

/* Static methods */
/* S-PLUS would call these */

// static method to create a family member and return
// an ID to track them

public static int newFamilyMember(String first,
    String last){
    FamilyMember newMember = new FamilyMember(first, last);

    // Add new instance to list of members
    members.add(newMember);

    // Use the position in the members vector as an ID
    return (members.size() -1);
}

// private method to check that ID in legal range

private static boolean checkId(int id){
    boolean status = true;

    if (id < 0 || id > (members.size()-1)){
        // Could throw exception. we'll just print a message
        System.out.println("Error: ID out of range");
    }
}
```

```
        status = false;
    }
    return status;
}

// static methods to specify mother and father based on ID

// The basic steps in these methods are:
// 1) Check that the ID is within the range of ID's.
// 2) Get the object from the members Vector.
// 3) Cast the object to a FamilyMember object.
// 4) Apply the relevant non-static method to the object
//
// If the ID is out of range we return false. Otherwise
// we return true.

public static boolean setMother(int personId, int momId){
    boolean status = true;
    if (checkId(personId) && checkId(momId)){
        FamilyMember person =
            (FamilyMember) members.get(personId);
        FamilyMember mom = (FamilyMember) members.get(momId);
        person.setMother(mom);
    }
    else
        status = false;
    return status;
}

public static boolean setFather(int personId, int dadId){
    boolean status = true;
    if (checkId(personId) && checkId(dadId)){
        FamilyMember person =
            (FamilyMember) members.get(personId);
        FamilyMember dad = (FamilyMember) members.get(dadId);
        person.setFather(dad);
    }
    else
        status = false;
    return status;
}
}
```

```
// static method to get information about a family member

public static String [] getInfo(int id){
    if (!checkId(id))
        return new String [] {"Name: Unknown",
                               "Mother: Unknown", "Father: Unknown"};

    FamilyMember person = (FamilyMember) members.get(id);
    FamilyMember mom = person.getMother();
    FamilyMember dad = person.getFather();
    Vector kids = person.getChildren();

    String [] info = new String [3 + kids.size()];

    info[0] = "Name: " + person.getFirstName() + " " +
        person.getLastName();

    if (mom==null)
        info[1] = "Mother: Unknown";
    else
        info[1] = "Mother: " + mom.getFirstName() + " " +
            mom.getLastName();

    if (dad==null)
        info[2] = "Father: Unknown";
    else
        info[2] = "Father: " + dad.getFirstName() + " " +
            dad.getLastName();

    for (int i = 0; i < kids.size(); i++){
        FamilyMember aKid = (FamilyMember) kids.get(i);
        if (!(aKid==null)){
            info[3+i] = "Child: " + aKid.getFirstName() + " " +
                aKid.getLastName();
        }
    }

    return info;
}
}
```

CALLING SPOTFIRE S+ FROM JAVA APPLICATIONS

Spotfire S+ can be called from Java to perform computations and create graphics. This section describes the primary Java classes for communicating between Java and Spotfire S+. The discussion of classes is followed by examples.

The `SplusUserApp` class provides a simple way for a Java application to connect to Spotfire S+. It contains static methods that Java applications can call to generate results and graphics. Spotfire S+ graphs created with the `java.graph` graphics device can be embedded within Java application windows. This section describes using `SplusUserApp` to call Spotfire S+ locally.

The `SplusSession` interface provides another way for a Java application to connect to Spotfire S+. Developers can program directly to this interface when they need more control than the `SplusUserApp` class provides.

Documentation on `SplusUserApp`, `SplusSession`, and all the other Java classes used by Spotfire S+ can be found at `$SHOME/java/javadoc/index.html` on both Windows and UNIX.

Running the Java Program

The Java program is invoked in different ways on UNIX and Windows

UNIX

As various environment variables must be set at start-up for the S-PLUS engine to function properly, the `Splus` or `SplusClient` script must be used to run the Java program. These scripts set various environment variables and then use the Java virtual machine included with Spotfire S+ to run the Java program.

Use the `-userapp` flag to indicate that Spotfire S+ is being run as part of a user-written Java application. This flag should be followed by the full name of the directory containing the class file, and the class name. For example:

```
Splus -userapp /homes/user/examples TextOutputExample
```

You should only access methods from user applications started as above with the `-userapp` option. They should not be called when running with the `-g` or `-j` option. Attempting to do so throws a Java exception.

Windows

To run a Java program in Microsoft Windows, specify the full path to the java executable and the name of the Java class. This may be done at a Windows Command prompt or in a Windows shortcut.

To call Spotfire S+, the location of **SHOME** must also be specified using the Java property `splus.shome`. For example:

```
"D:\Program Files\TIBCO\splus82\java\jre\bin\java"  
-Dsplus.shome="D:\Program Files\TIBCO\splus82"  
TextOutputExample
```

Evaluating S-PLUS Commands

S-PLUS expressions can be evaluated with the following `SplusUserApp` methods:

```
public static SplusDataResult eval(String cmd);  
public static SplusDataResult eval(String cmd,  
    boolean Output, boolean Result, boolean Errors,  
    boolean Warnings, boolean Expr);
```

The `cmd` parameter contains an expression to be evaluated by Spotfire S+. The resulting `SplusDataResult` object contains the result of evaluating this expression.

The expression should be a syntactically complete S-PLUS expression. An `SplusIncompleteExpressionException` is thrown if the expression is not complete.

The additional parameters `Output`, `Result`, `Errors`, `Warnings`, and `Expr` allow the user to specify what elements to include in the `SplusDataResult`. These may be specified as `false` to avoid the overhead of passing unnecessary elements of the output over the data channel. The default is to include the `Result`, `Errors`, and `Warnings`.

Connection Threads

The first time `SplusUserApp.eval()` is called, it starts the S-PLUS engine process and several Java threads. It takes approximately 10 seconds for the process to complete.

These Java threads remain active until the S-PLUS engine is shut down, so it is typically necessary to call `System.exit(0)` to exit the user Java application. Even if the application's main thread is finished, Java will wait forever for the S-PLUS connection threads to finish.

Using SPLUSDATARESULT Objects

An object deriving from `SplusDataResult` is produced by a call to `evalDataQuery()` when using the `SplusSession` interface, described later in this chapter.

The results of a query are formalized to have the following fields:

- `expression`: a string representing the query that was processed.
- `output`: a string representing the output that the query produced. This is the output that would be printed if the command were evaluated as standard input. This output may be printed to standard out or to a Java text area.
- `error`: a string representing the error output that the query produced.
- `warning`: an array of strings representing warnings that the query produced.

If the query produced a data result, it will have an additional field:

- `result`: an array of values of some primitive type determined by the query. This will be an array of a standard atomic type, for example, `boolean`, `byte`, `double`, `float`, `long`, or `String`. This is the mechanism for passing results of computations back to Java.

The `SplusDataResult` parent class contains the first four fields but lacks the `result` field. The following classes inherit from the `SplusDataResult` class and contain a `result` field that is an array of the appropriate type:

```
SplusBooleanDataResult  
SplusByteDataResult  
SplusDoubleDataResult  
SplusFloatDataResult  
SplusLongDataResult  
SplusStringDataResult
```

These are the only types presently supported in accessing the S-PLUS

analytic engine from Java. The correspondence between Java types and S-PLUS types is as follows:

Table 4.2: *Java and S-PLUS correspondence.*

Java	S-PLUS
boolean	logical
byte	raw
double	numeric
float	single
long	integer
string	character

You must ensure that queries produce only a single vector of one of these primitive types. The analytic engine then automatically constructs a data result object with a class that matches the query result type.

If the query fails to produce a result, an `SplusDataResult` object is returned that does not contain a result field. If the query returns a more complex object than a vector, a warning message is printed to the engine's standard output stream and an `SplusDataResult` object is returned that does not contain a result field.

The fields in data result objects are private and are available only via accessor methods. Hence, all data result objects include the following methods:

```
public String getExpression();
public String getOutput();
public String getError();
public String [] getWarning();

public Object getData();
public boolean [] getBooleanData() throws
    SplusBadDataException;
```

```
public byte [] getByteData() throws  
    SplusBadDataException;  
public double [] getDoubleData() throws  
    SplusBadDataException;  
public float [] getFloatData() throws  
    SplusBadDataException;  
public long [] getLongData() throws  
    SplusBadDataException;  
public String [] getStringData() throws  
    SplusBadDataException;
```

Usually, only one of the `getxxxData` methods returns correct data, depending on the type of the data result that is available. If the type of the data result available does not match the type of data requested in the `get` operation, an `SplusBadDataException` is thrown with a string describing the condition. For instance, suppose you are expecting a vector of `longs` as the result of a query. Then in a simplified situation, you would use the code:

```
SplusDataResult result = evalDataQuery("/* some query */");  
long [] values = result.getLongData();
```

Now suppose further that by some mistake your query produced a vector of `doubles` instead of a vector of `longs`. Then at runtime, the call to `result.getLongData` would generate an `SplusBadDataException` with the string:

```
"long" data was requested, but existing data is "double"
```

Under some circumstances, you may not care initially about the type of return data. For example, you may be constructing a collection of result vectors which you want to store as a hash-table of Java objects. The actual type of the data would be handled at a later time, using Java's `instanceof` operator.

For such situations, the `SplusDataResult` class (and its derivatives) have the method

```
public Object getData();
```

which returns the result vector as a generic Java Object. The `getData()` method returns `null` if the object has no data and an array of primitives otherwise. This is useful when you intend to put the result in a Java collection object such as a `Vector`, rather than immediately using the result as a primitive array of a known type.

In addition to the methods discussed above, the following boolean methods are available to determine whether the fields of the `SpplusDataResult` object have non-null entries:

```
public boolean hasExpression();  
public boolean hasOutput();  
public boolean hasError();  
public boolean hasWarning();
```

The `SpplusDataResult` class and the classes that inherit from it are intended to form a complete set of result classes reflecting the capabilities currently offered in interacting with the S-PLUS analytic engine.

Graph Components

The S-PLUS `java.graph()` device displays Spotfire S+ graphics within a Java component. By calling `SpplusUserApp` methods, a Java application can embed a `java.graph()` display component within a Java window. Spotfire S+ graphs are then displayed in this component by evaluating Spotfire S+ graphics commands.

Typically, Spotfire S+ allows multiple graphics devices to be open at once, and multiple graphics devices can be opened while evaluating a single expression. When you run the Spotfire S+ graphical user interface, notice that these appear in different windows. Currently, `SpplusUserApp` only supports a single graph window embedded in a specific `JComponent`.

The following `SpplusUserApp` method returns the `JComponent` containing the graph window:

```
public static JComponent getGraph();
```

This component contains a multi-page graph display that can be embedded within another window. The component corresponds to the last `java.graph()` device that was opened. When a new `java.graph()` device is opened, this graph is cleared and no longer has any association with the previous device.

Example Applications

In each of the following examples we create a simple Java class that communicates with Spotfire S+ using `SpplusUserApp`.

In Windows, the Java code for these examples is in

`$$HOME/library/winjava/examples`

and in UNIX, it is located at

\$SHOME/library/example5/java

This UNIX directory also contains the file **examples.jar**. This file includes the ***.class** files for all of the ***.java** files other than those that are copies of the files for the **Correlations** dialog and **Linear Regression** dialog in the standard Spotfire S+ Java GUI.

Java requires that each class be stored in a file named by appending **.java** to the class name. To run each example, perform the following steps:

1. Create a text file containing the example. Name the file based on the class, for example, **TextOutputExample.java**.
2. Compile the file using a Java compiler such as `javac`. This will create a file **TextOutputExample.class**.
3. Start the Java application.

On UNIX, use the `Splus` or `SplusClient` script to start the application, specifying the directory containing the class file (which will be added to the Java classpath) and the name of the class:

```
Splus -userapp /user/examples TextOutputExample
```

```
SplusClient -userapp /user/examples TextOutputExample
```

On Windows, place the class file in a location that Java can find, as described in section “Class Files” on page 72. Then use the `java` executable to start the application:

```
"D:\Program Files\TIBCO\Splus82\java\jre\bin\java"  
-Dsplus.client.mode=true  
-Dsplus.shome="D:\Program Files\TIBCO\Splus82"  
TextOutputExample
```

Text Output

The `TextOutputExample` application generates the numbers 1 to 100 and returns them formatted for printing.

```
import com.insightful.splus.*;  
  
public class TextOutputExample {  
    public static void main(String [] args){  
        try {  
            String expression = "1:100";
```

```
        System.out.println("Sending expression " +
            expression);

        // Get just the text output back
        SplusDataResult result =
            SplusUserApp.eval(expression + "\n",
                true, false, false, false, false);

        System.out.println("Result from S-PLUS:");
        System.out.println(result.getOutput());
    }
    catch (Exception e){
        System.out.println(e);
    }

    System.exit(0);
}
}
```

Returning Values The `RandomNormalExample` application generates 100 random normal values and passes them back to Java. It then prints the number of values, first value, and last value.

```
import com.insightful.splus.*;

public class RandomNormalExample {
    public static void main(String [] args){
        try {
            String expression = "rnorm(100)";

            // Get just the result output back
            SplusDataResult result =
                SplusUserApp.eval(expression + "\n",
                    false, true, false, false, false);

            // Get the double values. We know the S-PLUS function
            // rnorm() returns doubles. If it did not, the
            // try/catch mechanism would catch the
            // SplusBadDataException.

            double [] randomValues = result.getDoubleData();
```

```
        System.out.println("Generated " +
            randomValues.length +
            " random normal values.");
        System.out.println("First value: " + randomValues[0]);
        System.out.println("Last value: " +
            randomValues[randomValues.length - 1]);
    }
    catch (Exception e){
        System.out.println(e);
    }

    System.exit(0);
}
}
```

Passing Values to Spotfire S+

The `SplusDataResult` class provides a way to get values from Spotfire S+. We can pass values needed for a computation from Java to Spotfire S+ in two ways.

The first way is to include the values in the string passed to `SplusUserApp.eval()`. This is the simplest approach when we are specifying a few parameters to a function, as in the previous examples.

The other way is to store the values in static fields and have the S-PLUS function query the values of these fields. The `.JavaField()` function will access the field value directly. Alternatively, we might use a `.JavaMethod()` call to a static method that extracts the field value. The `CorrelationExample` application uses the `.JavaField()` approach.

```
import com.insightful.splus.*;

public class CorrelationExample {

    // static fields to pass values to S-PLUS
    static double [] xValue;
    static double [] yValue;

    // Java function calling S-PLUS to
    // compute correlation of two double arrays.
    // This can be called from other
    // classes.
```

```
//
// Throw an IllegalArgumentException if x or y
// inappropriate.
// Pass on other exceptions if they are thrown.

public static double getCorrelation(double [] x,
    double [] y) throws IllegalArgumentException,
    SplusBadDataException {

    if (x == null || x.length == 0 ||
        y == null || y.length == 0)
        throw (new IllegalArgumentException(
            "Argument is null or length zero."));

    xValue = x;
    yValue = y;

    // Define the S-PLUS expression to use.
    // Note we use \" for quotes within the expression.

    String expression = "cor(" +
        ".JavaField(\"CorrelationExample\", " +
        "\"xValue\", \"[D\", client = T),\" +
        ".JavaField(\"CorrelationExample\", " +
        "\"yValue\", \"[D\", client = T))";

    SplusDataResult result =
        SplusUserApp.eval(expression + "\n");

    return result.getDoubleData()[0];
}

public static void main(String [] args){
    // Create some double arrays and get their correlations.

    double [] a = {2.1, 3.5, 8.4, 5.3, 6.8};
    double [] b = {9.8, 3.7, 2.1, 8.3, 6.5};

    System.out.println(
        "Getting correlation of two double arrays.");
}
```

```
try {
    double d = CorrelationExample.getCorrelation(a, b);
    System.out.println("Correlation is: " + d);
}
catch (Exception e){
    System.out.println(e);
}
System.exit(0);
}
```

Embedding Graphs in a Custom GUI

The GraphButtonsExample application launches a window containing a graph region and buttons that generate graphs when pressed.

```
import com.insightful.splus.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class GraphButtonsExample {

    static ActionListener buttonActionListener;

    public static void main(String [] args){

        // Get the graph component
        JComponent splusGraph = SplusUserApp.getGraph();

        // Create the window
        JFrame window = new JFrame("Graph Buttons Example");

        // Create buttons which query Spotfire S+ to
        // produce graphs

        buttonActionListener = new ActionListener(){
            public void actionPerformed(ActionEvent e){
                String cmd = e.getActionCommand() + "\n";
                try {
                    SplusUserApp.eval(cmd, false, false,
                        false, false, false);
                }
            }
        }
    }
}
```

```
        catch (Exception ex) {
            System.out.println(ex);
        }
    }
};

JPanel buttonPanel = new JPanel();
buttonPanel.setLayout(new BorderLayout(buttonPanel,
    BorderLayout.Y_AXIS));
buttonPanel.add(makeButton("plot(sin(1:10))"));
buttonPanel.add(makeButton("plot(rnorm(100))"));
buttonPanel.add(makeButton(
    "print(example.normal.qq())"));
buttonPanel.add(makeButton("show colors, linetypes",
    "{image(matrix(data=1:100,nrow=10,ncol=10));" +
    "for (a in 1:50) { ang <- a*0.06; " +
    "lines(c(5,5+10*cos(ang))," +
    "c(5,5+10*sin(ang)), col=a );for (a in 1:50) { " +
    "ang <- -a*0.06; lines(c(5,5+10*cos(ang))," +
    "c(5,5+10*sin(ang)), lty=a )}"));
buttonPanel.add(makeButton(
    "for (x in 1:10) plot(1:x)"));

// Create an Exit button

JButton exitButton = new JButton("Exit");
exitButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae){
        System.exit(0);
    }
});

// Also exit if the window is closed by the user

window.addWindowListener(new WindowAdapter(){
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
});

// Add the elements to the content pane
```

```
Container contentPane = window.getContentPane();
contentPane.setLayout(new BorderLayout());
contentPane.add(splusGraph, BorderLayout.CENTER);
contentPane.add(buttonPanel, BorderLayout.EAST);
contentPane.add(exitButton, BorderLayout.SOUTH);

// Show the window
window.setSize(600, 400);
window.show();
}

// Utility methods to create a button
private static JButton makeButton(String splusCode){
    return makeButton(splusCode, splusCode);
}

private static JButton makeButton(String buttonName,
    String splusCode){
    JButton b = new JButton(buttonName);
    b.setActionCommand(splusCode);
    b.addActionListener(buttonActionListener);
    return b;
}
}
```

Sending a JDBC ResultSet to Spotfire S+

Java provides the JDBC standard for accessing data from a database. In JDBC, a `ResultSet` represents data returned from doing a particular query to a particular database. The `ResultSetUtilities` class in the `com.insightful.splus.util` package provides support for creating a `S-PLUS data.frame` from a `ResultSet`.

This class is used by registering a `ResultSet` under a particular name. The S-PLUS function `javaGetResultSet("name")` function can then create a `data.frame` containing the values in the `ResultSet`.

The `ResultSetExample` shows how to register and access a `ResultSet`. In this example, we use a `LocalResultSet` object. This is an inner class defined to make the example self-contained and is not presented here.

```
import com.insightful.splus.*;
import com.insightful.splus.util.ResultSetUtilities;
```

```
import java.sql.*;
import java.io.*;
import java.math.BigDecimal;

public class ResultSetExample {
    public ResultSetExample() {

        LocalResultSet theSet = new LocalResultSet();

        // Register the ResultSet with ResultSetUtilities.
        ResultSetUtilities.register("mySetId", theSet);

        // Save ResultSet data in Spotfire S+ as "test.df".

        String expr = "assign(\"test.df\",
            javaGetResultSet(\"mySetId\", client = T),
            where = 1)\n";

        SplusDataResult result = SplusUserApp.eval(expr);

        // Query Spotfire S+ to print test.df to show
        // it's there
        System.out.println("Printing test.df in
            Spotfire S+.");

        expr = "if (exists(\"test.df\")) print(test.df) else
            print(\"test.df does not exist\")\n";

        result = SplusUserApp.eval(expr, true, false, false,
            false, false);
        System.out.println(result.getOutput());
    }

    public static void main(String [] args) {
        new ResultSetExample();
        System.exit(0);
    }
}
```

Sending Multiple Columns to Java

The `SplusDataResult` mechanism provides access to a single vector of values. Multiple calls to `eval()` can be used to retrieve multiple columns of data from a `data.frame`.

Java has no direct equivalent to a `data.frame`. However, a collection such as a `Vector` or `ArrayList` is available for storing multiple arrays of data. These arrays can then be used in other Java routines, such as writing to a database via `JDBC`.

The `DataFrameArrayListExample` shows how to create an `ArrayList` containing the columns of a `data.frame`.

Transferring a Graph to the Client

All of the file operations performed by Spotfire S+ are done on the server file system. Often it is desirable to have a file saved on the client file system. The `FileUtilities` class in `com.insightful.splus.util` provides a `transferBytes()` function that is useful for passing a file over file streams. Typically one file stream will be created in the client JVM, and the other will be obtained from the `SplusSession` using `getFileInputStream()` or `getFileOutputStream()`.

The `TransferGraphExample` shows how to create a graph on the server and transfer it to the client.

A Simple Application

The examples above use specific methods for the given application. The `ImageCalculatorExample` combines various techniques presented above to create an application that lets the user type in S-PLUS expressions that are executed to create text and graphics output.

INTERFACING WITH C AND FORTRAN CODE

5

Overview	99
When to Use the C and Fortran Interfaces	100
When Should You Consider the C or Fortran Interface?	100
Reasons for Avoiding C or Fortran	100
Using C and Fortran Code with Spotfire S+ for Windows	102
Calling Simple C Code from Spotfire S+	102
Calling Simple Fortran Code from Spotfire S+	105
Calling C Routines from Spotfire S+ for Windows	109
Calling Fortran Routines from Spotfire S+	111
Writing C and Fortran Routines Suitable for Use with Spotfire S+ for Windows	114
Exporting Symbols	114
Modifying Header Files	115
Building a Chapter with WatcomC/Fortran	117
Dynamically Linking Your Code	118
Common Concerns in Writing C and Fortran Code for Use with Spotfire S+ for Windows	119
Changes in S.h	119
Handling IEEE Special Values	121
I/O in C Functions	124
I/O in Fortran Subroutines	124
Reporting Errors and Warnings	125
Calling Fortran From C	130
Calling C From Fortran	132
Calling Functions in the S-PLUS Engine DLL	132

Using C Functions Built into Spotfire S+ for Windows	134
Allocating Memory	134
Generating Random Numbers	136
Calling Spotfire S+ Functions from C Code (Windows)	138
The .Call Interface (Windows)	146
Requirements	146
Returning Variable-Length Output Vectors	147
S Object Macros	148
Debugging Loaded Code (Windows)	151
Debugging C Code	151
A Simple Example: Filtering Data (Unix)	155
Calling C or Fortran Routines From Spotfire S+ for Unix	157
Writing C and Fortran Routines Suitable for Use in Spotfire S+ for Unix	161
Compiling and Dynamically Linking your Code (Unix)	162
Common Concerns in Writing C and Fortran Code for Use with Spotfire S+ for Unix	166
Using C Functions Built into Spotfire S+ for Unix	177
Calling S-PLUS Functions From C Code (Unix)	180
The .Call Interface (Unix)	187
Debugging Loaded Code (Unix)	192
A Note on StatLib (Windows and Unix)	195

OVERVIEW

A powerful feature of Spotfire S+ is the ability to extend its functionality, enabling you to interface with compiled languages, including C, Fortran, and C++. Interfaces to these languages allow you to combine the speed and efficiency of compiled code with the robust, flexible programming environment of Spotfire S+. Your compiled routines are loaded into Spotfire S+ via dynamic loading, that is, your compiled code, in the form of a dynamic link library (DLL) on Windows[®] or a shared library on Unix, is loaded while Spotfire S+ is running.

After you load the compiled routines, the `.C`, `.Call`, and `.Fortran` functions are used to call compiled routines directly from Spotfire S+.

This chapter describes how to do the following tasks:

- Decide when and where to use compiled code.
- Write C, C++, and Fortran routines suitable for use in Spotfire S+.
- Create a loadable object (DLL or shared library) as part of an S-PLUS chapter.
- Load the object in a Spotfire S+ session.
- Troubleshoot problems you may encounter with dynamic loading.
- Debug your compiled code.

Each of these tasks can become quite complicated, so we begin with an overview of when to use them and provide a simple example that shows the basic flavor of writing, compiling, and using compiled code.

Note

Spotfire S+ for Windows is compiled with Microsoft Visual C++ 6.0 and Compaq (formerly Digital) Visual Fortran 6.0. TIBCO Software Inc. provides several useful enhancements that make compiling C, C++, and Fortran code quite simple in the Visual Studio environment, so our examples use that environment for simplicity. However, any C, C++, or Fortran compiler capable of creating a fully relocatable DLL can be used to compile code for use with Spotfire S+.

WHEN TO USE THE C AND FORTRAN INTERFACES

The key to effective use of compiled code is knowing when and where not to use such code. The following subsections provide some criteria for deciding whether compiled code is the right choice for your situation, and outline the basic procedure for using compiled code in Spotfire S+.

When Should You Consider the C or Fortran Interface?

Compiled C or Fortran code runs faster than interpreted Spotfire S+ code, but is neither as flexible nor as resilient as equivalent Spotfire S+ code. Mismatching data types and overrunning arrays are just two types of errors that can occur in compiled code but do not occur in Spotfire S+ code. The best time to use compiled code is when you have such code already written and tested. Another good time to use compiled code is when you cannot use S-PLUS vectorized functions to solve your problem without explicit loops or recursion. Recursion in S tends to be very memory intensive; simulations that work for small cases may fail as the number of iterations rises. If the iterated computation is trivial, you can realize huge performance gains by moving that portion of the calculation to compiled code.

Reasons for Avoiding C or Fortran

Except via the `.Call` interface, compiled code deals only with data types fixed when the code is compiled. The C and Fortran interfaces expect only the most basic data types, which correspond in Spotfire S+ to storage modes, which underlie the S-PLUS class structure and determine how data is actually stored. In general, there is a mode corresponding to all the basic classes, such as `logical`, `character`, `integer`, `single`, `numeric`, and `complex`. If your code does something numerical, it may be fine to convert all the inputs to double precision (class `numeric`) and return double precision results.

If your code rearranges data, however, you probably do not want to change the modes of the data, so Spotfire S+ code would be better than compiled code. The C and Fortran interfaces ignore the class of data sets, so they are not object oriented. To work on more general types of S-PLUS data objects, you can still use C code, but via the `.Call` interface, discussed later in this chapter. Even with `.Call`, working with objects other than those of the simple vector types is difficult.

It is usually harder to develop and debug compiled code than S-PLUS functions. With compiled code, you must make sure not only that the compiled code works, but also that the S-PLUS function that calls it works and is compatible with the compiled code.

Compiled code is usually not as portable as Spotfire S+ code. Other users who would like to use your code may not have the appropriate compilers or the compilers on other machines may not be compatible with one another. Your code may also depend upon certain libraries that others may not have.

A good strategy is to do as much as possible in Spotfire S+ code, including error checking, data rearrangements, selections and conversions, storage allocation, and input/output, and use compiled code to do only the numerical or character string calculations required. When developing new functions in Spotfire S+, you should probably write the entire function in Spotfire S+ code first. Then, if the pure Spotfire S+ version is too slow or memory intensive (and you expect it to be used a lot), look for bottlenecks and rewrite those parts in C.

USING C AND FORTRAN CODE WITH SPOTFIRE S+ FOR WINDOWS

Spotfire S+ for Windows is compiled with Microsoft Visual C++ 6.0 and Compaq (formerly Digital) Visual Fortran 6.0. TIBCO Software Inc. provides several useful enhancements that make compiling C, C++, and Fortran code quite simple in the Visual Studio environment. Our examples will use that environment for simplicity. However, any C, C++, or Fortran compiler capable of creating a fully relocatable DLL can be used to compile code for use with Spotfire S+. Later in the chapter, there is an example using the Watcom compiler.

Calling Simple C Code from Spotfire S+

The basic steps for interfacing C code with Spotfire S+ using the Visual Studio/C++ Environment on Windows are as follows:

1. Obtain C source code (write it, get someone else to write it for you, download it from the Internet, and so on).
2. Create Visual Studio project for the source code.
3. Build the Visual Studio project to create dynamic link library (DLL) that can be loaded into Spotfire S+.
4. Load the DLL into Spotfire S+.
5. Write a Spotfire S+ function that calls your C code via the `.Call` function.
6. Run your S-PLUS function.

The steps for calling Fortran code are essentially the same, but you use the Visual Fortran compiler within Visual Studio.

To illustrate the steps we show how to create a function to apply a first-order linear recursive filter to a vector of data. A pure Spotfire S+ version of the function is seen in the following example:

```
Ar <- function(x, phi)
{
  n <- length(x)
  if (n>1)
    for (i in 2:n)
```

```
x[i] <- phi * x[i - 1] + x[i]
x
}
```

Looping is one area where Spotfire S+ tends to be significantly slower than compiled code, so this is a good candidate for implementation in C. (Note that the S-PLUS filter function computes a recursive filter efficiently, but for the sake of the example we ignore that).

A C function for the filter is

```
void arsim(double *x, long *n, double *phi)
{
    long i;
    for (i=1; i<*n; i++)
        x[i] = *phi * x[i-1] + x[i] ;
}
```

This code is purely C language code; there are no dependencies on C libraries, Spotfire S+, or the Windows API. Such code should be portable to most operating systems and most Windows compilers. It is quite simple to create a DLL from this code using Visual C++ 6.0[®]:

1. Start Visual C++ 6.0, and from the **File** menu, select **New**.
2. From the **New** dialog, select the **Projects** tab.
3. In the **Project Workspace** dialog, specify **ar** as the name for the project and for **Project Type** choose **S-PLUS Chapter DLL (.C and .Call)** (not **MFCAppWizard (dll)**). Press **OK** to create the project.
4. In the dialog that pops up, enter the full path to the file **Sqpe.dll** which is typically **C:\Program Files\TIBCO\spplus82\cmd\Sqpe.dll** then click the **Finished** button. (The **C:\Program Files\TIBCO\spplus82** part of the path is the location where you installed Spotfire S+). A **New Project** information dialog pops up. Note the **Project** directory, ending with **ar**, listed at the bottom. Click **OK** to close it.
5. In the left pane of Visual C++, click the **File View** tab (at the bottom) and then expand the **ar** files folder and then the **Source Files** folder.
6. Double click the **ar.cxx** file icon to open up the sample C++ code in the right panel.

7. Select all the **code ar.cxx** and delete it. Replace it with the code for the `arsim` function written above.
8. Double-click the **ar.def** file to open up the module definitions file in the right panel.
9. Replace the comments under the section labeled **EXPORTS** with **arsim** and press return.
10. Save the **ar.def** file by selecting **Save** from the **File** menu in Visual C++.
11. From the **Build** menu, choose the **Rebuild All** selection. This will compile the `arsim` function and create a dynamic link library (DLL) called **S.dll** in the project directory.

The next step is load the **S.dll** into Spotfire S+. There are several ways to load the DLL. After starting Spotfire S+ you can do one of the following:

- Explicitly load the **S.dll** with the `dyn.open` function:

```
dyn.open("__path__to__ar__/ar/S.dll")
```

or declare the **ar** project directory a Spotfire S+ chapter and attach it, as follows:

```
dyn.attach("__path__to__ar__/ar/S.dll")
```
- Open the dialog from **File ► Chapters ► Attach/Create Chapter** and then use the **Browse** button to navigate to your **ar** project directory. Click the **OK** button. This will create a **.Data** subdirectory in **ar**, attach the directory, and load the **S.dll** into Spotfire S+.

You now need an S-PLUS function that calls the `arsim` C function. Here is the basic version:

```
arC <- function(x, phi)
{
  .C("arsim",
    as.numeric(x),
    length(x),
    as.numeric(phi))[[1]]
}
```

We are passing in three arguments to the `arsim` C function and we use `as.numeric` to coerce the arguments to the correct type that the C code is expecting (the `length` function returns an integer).

Define the function in a Spotfire S+ **Script** window by pressing F10.

Try running `arC` at the Spotfire S+ command line:

```
> arC(1:8, .25)

[1] 1.000000 2.250000 3.562500 4.890625 6.222656
7.555664 8.888916 10.222229
```

You lose some flexibility in the function by writing it in C. Our `arC` function converts all input data to double precision, so it will not work correctly for complex data sets or objects with special arithmetic methods. The pure Spotfire S+ version works for all these cases. If complex data is important for your application, you could write C code for the complex case and have the Spotfire S+ code decide which C function to call. Similarly, to make `arC` work for data in classes with special arithmetic methods, you could have it call the C code only after coercing the data to class `numeric`, so that it could not invoke special arithmetic methods. This might be too conservative, however, as there could be many classes of data without arithmetic methods which could use the fast C code.

Another approach would be to make `arC` a generic function for which the default method calls the C code for numeric data. For classes with special arithmetic methods, pure Spotfire S+ code could be dispatched. Those classes of data without special arithmetic methods could include a class method for `arC` that would coerce the data to class `numeric` and invoke the default method on the now numeric data, thus using the fast compiled code, then post-process the result if needed (perhaps just restoring the class). Using the object-oriented approach is more work to set up, but gives you the chance to combine the speed of compiled code with the flexibility of Spotfire S+ code.

Note

In order to recompile an existing **S.dll**, detach the Spotfire S+ chapter containing the **S.dll**, recompile, then reattach after the DLL is compiled. Or if you use `dyn.open()`, call `dyn.close()` prior to compiling the code.

Calling Simple Fortran Code from Spotfire S+

The basic steps for interfacing Fortran code with Spotfire S+ for Windows are essentially the same as the steps for interfacing with C code with Spotfire S+, but you use the Visual Fortran compiler within Visual Studio. For the sake of consistency, we will translate the C example in the previous section to its Fortran equivalent.

The basics steps for Fortran are the same as the steps for C code:

1. Obtain Fortran source code (write it, get someone else to write it for you, download it from the Internet, etc.).
2. Create a Visual Fortran project for the source code.
3. Build the Visual Fortran project to create dynamic link library (DLL) that can be loaded into Spotfire S+.
4. Load the DLL into Spotfire S+.
5. Write an S-PLUS function that calls your Fortran code via the .C function.
6. Run your S-PLUS function.

We will the same first-order linear recursive filter as we used for above for C for implementation in Fortran. An example of Fortran implementation for the filter is the following:

```
subroutine arsim(x, n, phi)
  c inputs:
  double precision x(1), phi
  integer n
  integer i
  do 10 i = 2, n
    x(i) = phi * x(i-1) + x(i)
  10 continue
  return
end
```

To create a DLL from this code using Visual Fortran 6:

1. Start Visual Fortran 6.0, and from the **File** menu, select **New**.
2. From the **New** dialog, select the **Projects** tab.
3. In the **Project Workspace** dialog, specify **ar** as the name for the project and for **Project Type** choose **S-PLUS Chapter DLL (.Fortran)** (not **MFCAppWizard (dll)**). Press **OK** to create the project.
4. In the dialog that pops up, enter the full path to the file **Sqpe.dll** which is typically
C:\Program Files\TIBCO\splus82\cmd\Sqpe.dll

then click the Finished button. (The **C:\Program Files\TIBCO\splus82** part of the path is the location where you installed Spotfire S+). A **New Project** information dialog pops up. Note the **Project** directory, ending with **ar**, listed at the bottom. Click **OK** to close it.

5. In the left pane of Visual Fortran, click the **File View** tab (at the bottom) and then expand the **ar** files folder.
6. Double-click the **ar.f** file icon in the **Resource** files folder to open up the sample Fortran code in the right panel.
7. Select all the code `ar.f` and delete it. Replace it with the code for the `arsim` routine written above.
8. From the **Project** menu choose **Settings**.
9. Select the **Pre-link step** tab (you may have to use the arrow keys at the right-hand side of the dialog to navigate to the appropriate tab).
10. Under **Pre-link** command(s), click inside the outlined box.
11. Type the full path to **spexport.exe** in your **\$HOME\cmd** directory, specify **ar.f** as the output file with the `-o` flag, and specify the object files for which you want symbols exported.
An example (the following should be typed on a single line):

```
C:\Program Files\TIBCO\splus82\cmd\spexport.exe  
-o ar.def *.obj
```
12. From the **Build** menu, choose the **Build S.dll** selection. This will compile the `arsim` function and create a dynamic link library (DLL) called **S.dll** in the project directory.

The next step is load the **S.dll** into Spotfire S+, and there are several ways to load it. After starting Spotfire S+, you can do the following:

- Explicitly load the **S.dll** with the `dyn.open` function:

```
dyn.open("__path__to__ar__/ar/S.dll")
```
- or declare the `ar` project directory a Spotfire S+ chapter and attach it:

Open the dialog from **File ► Chapters ► Attach/Create Chapter** and then use the **Browse** button to navigate to your **ar** project directory. Click the **OK** button, which creates a **.Data** subdirectory in **ar**, attach the directory, and load the **S.dll** into Spotfire S+.

You now need an S-PLUS function that calls the `arsim` Fortran function. Here is the basic version:

```
arFor <- function(x, phi) {  
  .Fortran("arsim",  
    as.numeric(x),  
    length(x),  
    as.numeric(phi))[[1]]  
}
```

We are passing in three arguments to the `arsim` Fortran function and we use `as.numeric` to coerce the arguments to the correct type that the Fortran code is expecting (the `lengthb` function returns an integer).

Trying out the `arFor`:

```
> arFor(1:8, .25)  
[1] 1.000000 2.250000 3.562500 4.890625 6.222656  
7.555664 8.888916 10.222229
```

The rest of this chapter includes more details on using C and Fortran code with Spotfire S+.

CALLING C ROUTINES FROM SPOTFIRE S+ FOR WINDOWS

To call a C function, use the S-PLUS function `.C`, giving it the name of the function (as a character string) and one S-PLUS argument for each C argument. For example, a typical “vectorized” calculation, such as `sin`, requires you to pass an S-PLUS data object `x` and its length `n` to the C function performing the calculation:

```
.C("my_sin_vec", x = as.double(x),  
  n = as.integer(length(x)))
```

we will define the C routine `my_sin_vec` in the section *Writing C and Fortran Routines Suitable for Use with Spotfire S+ for Windows* on page 114.

To return results to Spotfire S+, modify the data pointed to by the arguments. The value of the `.C` function is a list with each component matching one argument to the C function. If you name these arguments, as we did in the preceding example, the return list has named components. Your S-PLUS function can use the returned list for further computations or to construct its own return value, which generally omits those arguments which are not altered by the C code. Thus, if we wanted to just use the returned value of `x`, we could call `.C` as follows:

```
.C("my_sin_vec", x = as.double(x),  
  n = as.integer(length(x)))$x.
```

All arguments to C routines called via `.C` must be pointers. All such routines should be void functions; if the routine does return a value, it could cause Spotfire S+ to crash. Spotfire S+ has many classes that are not immediately representable in C. To simplify the interface between Spotfire S+ and C, the types of data that Spotfire S+ can pass to C code are restricted to the following S-PLUS classes: `single`, `integer`, `double`, `complex`, `logical`, `character`, `raw`, and `list`.

Table 5.1 shows the correspondence between Spotfire S+ modes and C types.

Table 5.1: Correspondence between S-PLUS classes and C types.

Spotfire S+ Storage Mode	Corresponding C Type
"logical"	long*
"integer"	long*'
"single"	float*'
"double"	double*
"complex"	s_complex*
"character"	char**
"raw"	char*
"list"	s_object**

Warning

Do *not* declare integer data as C ints, particularly if you want your code to be portable among machines that Spotfire S+ supports. While there is currently no difference on Windows, there is a distinction on other platforms.

The include file **S.h** described later in this chapter contains the typedef for the type `s_complex` that defines it as the struct composed of two doubles, `re` and `im`.

Calling C++

To call a C++ function, you also use the `.C` function (or, alternatively, the `.Call` or `Connect C++` interface, mentioned later in this chapter). Using `.C` is not a direct C++ interface, and hence Spotfire S+ will have no understanding of C++ name mangling. Thus, to call a C++ function, you must declare it inside an `extern "C"` braced expression. For example, here is some simple code to compute squares:

```
#include "S.h"
extern "C" {
```

```
void squareC(double* pdX, double* pdY, long* p1Len)
{
    S_EVALUATOR
    //Validate the input arguments
    if((pdX == NULL) || (pdY == NULL) || p1Len == NULL)
        PROBLEM "Invalid input" ERROR;
    //Perform element-by-element operation
    //to square each element of input
    for(long n=0; n< *p1Len; n++)
        pdY[n] = pdX[n] * pdX[n];
    return;
}
}
```

In the above code, the macro `S_EVALUATOR` is required because we are using the macros `PROBLEM` and `ERROR`; all three macros are discussed later in the chapter.

We can call this with `.C` using the simple Spotfire S+ code shown below:

```
square <- function(x){
  len = length(x)
  y = .C("squareC",
        as.double(x),
        y = double(len),
        len)$y
  y
}
```

Calling Fortran Routines from Spotfire S+

To call a Fortran subroutine, use the S-PLUS function `.Fortran`, giving it the name of the subroutine (as a character string) and one S-PLUS argument for each Fortran argument. For example, a typical “vectorized” calculation, such as sine, requires you to pass an S-PLUS data object `x` and its length `n` to the Fortran subroutine performing the calculation:

```
.Fortran("my_sin_vec", x = as.double(x),
n = as.integer(length(x)))
```

Note

You can call only Fortran *subroutines* from Spotfire S+; you cannot call Fortran functions.

To return results to Spotfire S+, modify the data pointed to by the arguments. The value of the `.Fortran` function is a list with each component matching one argument to the Fortran subroutine. If you name these arguments, as we did in the preceding example, the return list has named components. Your S-PLUS function can use the returned list for further computations or to construct its own return value, which generally omits those arguments which are not altered by the Fortran code. Thus, if we wanted to return just the object `x`, we could call `.Fortran` as follows:

```
.Fortran("my_sin_vec", x = as.double(x), n =
as.integer(length(x)))$x
```

Spotfire S+ has many data classes that are not representable immediately in Fortran. To simplify the interface between Spotfire S+ and Fortran, the types of data that Spotfire S+ can pass to Fortran code are restricted to the following Spotfire S+ storage modes: `single`, `integer`, `double`, `complex`, `logical`, and `character`. The following table shows the correspondence between Spotfire S+ modes and Fortran types.

Table 5.2: *Correspondence between S-PLUS classes and C types.*

Spotfire S+ Storage Mode	Corresponding Fortran Type
"logical"	LOGICAL
"integer"	INTEGER
"single"	REAL

Table 5.2: Correspondence between S-PLUS classes and C types.

Spotfire S+ Storage Mode	Corresponding Fortran Type
"double"	DOUBLE PRECISION
"complex"	DOUBLE COMPLEX
"character"	CHARACTER(*)

Warnings

Spotfire S+ will not pass arrays of character strings to Fortran routines; only the first element.

The Fortran type DOUBLE COMPLEX (or COMPLEX*16) is a complex number made of double precision parts; it may not be available with all Fortran compilers. It is available in the Compaq (formerly Digital) Visual Fortran and Watcom Fortran compilers.

When passing character data to Fortran routines, the compiled code should be expecting two arguments for each character argument passed; one for the data itself and another integer argument giving the number of characters in the previous argument. If your compiler cannot generate code to do this, do not pass character data to Fortran routines.

WRITING C AND FORTRAN ROUTINES SUITABLE FOR USE WITH SPOTFIRE S+ FOR WINDOWS

If you have a routine for which some of the arguments are not pointers, or which returns a value, you must write a wrapper routine which passes all data via pointer arguments, does not return a value, and calls the routine of interest. For example, we might have a sin function routine written in C and declared as follows:

```
double sin(double x)
```

You cannot call this via the .C interface, because it both takes a double-precision argument by value and returns a value. You must write an Spotfire S+-compatible wrapper for `sin()` as follows, and then load both procedures:

```
extern double sin() ;
void my_sin (double *x)
{
    *x = sin(*x) ;
}
```

Since `sin()` does not take a vector argument, you probably want to use the wrapper function to provide a vectorized form of it:

```
#include <S.h>
#include <math.h> /* to declare extern double sin() */
void my_sin_vec(double *x,long *n)
{
    long i ;
    for (i=0 ; i < *n ; i++)
        x[i] = sin(x[i]) ;
}
```

Exporting Symbols

For C and C++ code, there are two ways to ensure your symbols are exported correctly: either via header files or through the module definition file. Your code will generally be considered cleaner if your header files are correctly coded; use the module definition file for compiling code without header files (often, simple C routines that are passed around just as .c files).

For Fortran code, which does not have header files, you must use the module definition file to ensure your symbols are exported correctly. We discuss both ways to ensure symbols are exported correctly.

Note

When building a chapter with Watcom C or Fortran using CHAPTER or createChapter, all globally accessible symbols are automatically exported.

Modifying Header Files

In general, C and C++ functions are declared in header files. If a project includes header files that declare, appropriately exported, all the routines it intends to call, the built application automatically exports all the symbols it needs.

If you change or modify a function's definition, you also need to update its declaration in the header file. For example, when you create a new Spotfire S+ Chapter DLL, both a source file and its associated header file are created for you. If you modify the template function itself, particularly if you modify the template's parameter list, you need to also modify the associated header file.

Specifically, consider our ar example. When we originally create the project, it includes a source file **ar.cxx** containing the function arC function as follows:

```
////////////////////////////////////  
// ar.cxx: Implementation of ar2C and ar2Call  
////////////////////////////////////  
#include "S.h"  
#include "sconnect.h"  
#include "ar.h"  
////////////////////////////////////  
// arC() - SPLUS-interface via .C()  
//  
// See Chapter 16 of the "Spotfire S+ Programmer's Guide"  
// for details on how the interface works.  
// See ar.ssc for implementation of the S function  
// that calls ar2C()  
////////////////////////////////////  
  
void arC(double* pdX, double* pdY, long* plLen)
```

```
{
  S_EVALUATOR
  // TODO: Replace the example codes below with your own
  // code.
  //Validate the input arguments
  if((pdX == NULL) || (pdY == NULL) || p1Len == NULL)
    PROBLEM "Invalid input" ERROR;
  //Perform element by element operation
  for(long n=0; n< *p1Len; n++)
    pdY[n] = pdX[n] * pdX[n]; //
  //The output = the input squared
  return;
}
```

When we pull out the definition of `arC` and replace it with the definition of `arsim`, we need to modify the header file `ar.h` to remove the reference to `arC` and replace it with the reference to `arsim`. That is, we need to change the line in `ar.c` reading:

```
AR_DLL_API(void) arC(double*, double*, long* );
```

to:

```
AR_DLL_API(void) arsim(double*, long*, double*);
```

Updating the header file's declarations guarantees that the compiler and linker will export the appropriate symbols.

Using a Module Definition File

When you create a Visual Studio project for your C, C++, or Fortran code, a module definition file is created automatically as part of the process. However, the created file is typically just a stub, with no real information about exported symbols.

To create a useful module definition file, use the program `spexport.exe` included with Spotfire S+ in the `cmd` subdirectory of your `SHOME` directory. The simplest way to do this is to make it a standard part of your build process by including it as a pre-link step in your project settings, as follows:

1. From the **Project** menu in Visual Studio, choose **Settings**.
2. Select the **Pre-link step** tab (you might have to use the arrow keys at the right-hand side of the dialog to navigate to the appropriate tab).

3. Under **Pre-link command(s)**, click inside the outlined box.
4. Type the full path to **spexport.exe**, specify an output file using the `-o` flag, and specify the object files for which you want symbols exported.

For example (the following should be typed on a single line):

```
c:\Program Files\TIBCO\spplus82\cmd\spexport.exe -o ar.def *.obj
```

You can also create your own module definition file by hand. To get a list of symbols from a DLL, use the `DUMPBIN` utility described in the section Listing Symbols Using `DUMPBIN` on page 132.

To create a module definition file for use with the Watcom compiler, use the **spexport.exe** program as shown in Step 4 of the above example, but add the `-w` flag. Watcom Version 11 is assumed; if you have version 10.x, specify `-w10`.

Building a Chapter with WatcomC/ Fortran

If you are using Watcom C or Fortran, you can build a new chapter for Spotfire S+ very easily as follows:

1. Ensure that the directories containing your compiler, linker, and make utility are included in your `PATH` environment variable. Ensure that your `WATCOM` environment variable is set to the directory containing your Watcom compiler.
2. Ensure that the **Splus.ini** file located in the **cmd** directory under your Spotfire S+ installation directory refers to the correct files. The contents of the file should read as follows:

```
[chapter]
rules = wrules.mak
make = wmake.exe
```

3. From a Commands Prompt or DOS window, call the `CHAPTER` utility as follows:

```
spplus CHAPTER -d c:\myproj -m
```

or, from within Spotfire S+, call the `createChapter` function:

```
> createChapter("c:\myproj", T)
```

Dynamically Linking Your Code

Whenever you attach a Spotfire S+ chapter containing a shared object **S.dll**, including whenever you start up Spotfire S+ in such a chapter, the shared object is opened and the code it contains is loaded into Spotfire S+.

You can open shared objects without attaching a chapter by using the `dyn.open` function. For example, if your colleague Fred has C code you want to access from your own set of S-PLUS functions, you might open Fred's **S.dll** shared object as follows (assuming his files are mapped to your **H:** drive):

```
> dyn.open("H:/mysplus/S.dll")
```

You can close previously opened shared objects using `dyn.close`:

```
> dyn.close("H:/mysplus/S.dll")
```

If you are actively developing code, you may want to load, test, rebuild, unload, and reload your code repeatedly during a given Spotfire S+ session. To do this, you could use the `dyn.open` and `dyn.close` functions described above, but you may find, especially if you initially loaded your code automatically on startup, that `dyn.close` does not completely remove the DLL from your session. A safer and surer way to ensure that the old DLL (and all its symbols) are unloaded before the new DLL is loaded is to call `synchronize` after rebuilding the DLL. For example, if you are developing your code in your current working chapter, you could unload and reload the DLL with the following call:

```
> synchronize(1)
```

COMMON CONCERNS IN WRITING C AND FORTRAN CODE FOR USE WITH SPOTFIRE S+ FOR WINDOWS

While the actual calls to `.C` and `.Fortran` are straightforward, you may encounter problems loading new compiled code into Spotfire S+ and we will discuss some common problems. We also describe some C procedures and macros which you may use to write more portable code, to generate random numbers from C code, to call S-PLUS functions from your C code, to report errors, to allocate memory, and to call Fortran procedures from C code.

In order to have access in C to most functions and macros described below, you will have to include the header file **S.h** in your source files:

```
#include <S.h>
```

and make sure that you specify the `%SHOME%\include` directory in your compiler directives. That directory is specified automatically when you create your projects using Visual C++. If you will be using any Windows API calls in your code, so that you need to include **windows.h**, include **windows.h** first, then **S.h** and any other include files you need.

Changes in S.h

The file **S.h** has changed significantly since S-PLUS 2000; if you have existing code that includes **S.h**, you may have to modify your calls to the internal Spotfire S+ routines. In particular, most of the calls now require the use of the macro `S_EVALUATOR` and an additional argument, `S_evaluator`. For examples of using the newer macro and the new argument, see the following:

- section Using C Functions Built into Spotfire S+ for Windows on page 134

or

- section Using C Functions Built into Spotfire S+ for Unix on page 177

In addition, some variables have been renamed and some routines which previously had declarations in **S.h** have had their declarations moved elsewhere. In general, these changes affect only variables and

routines which were previously undocumented. A new variable, `S_COMPATIBILITY`, allows you to compile code that uses some of the redefined variables. If you define `S_COMPATIBILITY` (before including **S.h**) as follows:

```
#define S_COMPATIBILITY 1
```

you obtain the old definitions of the following variables:

- TRUE, FALSE, and MAYBE (now S_TRUE, S_FALSE, and S_MAYBE);
- complex (now s_complex);
- NULL_MODE (now S_MODE_NULL);
- LGL, INT, REAL, DOUBLE, CHAR, LIST, COMPLEX, RAW;
- ANY, STRUCTURE, MAX_ATOMIC, atomic_type (now S_MODE_LGL, and so on).

Defining S_COMPATIBILITY as 10 instead of 1 adds the following old definitions:

vector, boolean, void_fun, fun_ptr (now s_object, s_boolean, s_void_fun, and s_fun_ptr respectively)

We recommend that you migrate any code that uses the old variable names to use the new names, because of potential conflicts with other applications, particularly under the Windows operating systems.

Handling IEEE Special Values

Spotfire S+ handles IEEE special values such as NaN, Inf or -Inf, for all supported numeric classes (integer, single or double).

- NaN represents the number you obtain when you divide 0 by 0.
- Inf represents the number you obtain when you divide 1 by 0.
- -Inf represents the number you obtain when you divide -1 by 0.

In addition, Spotfire S+ supports NA, which represents a missing value: that is, a value to use when none is available. S-PLUS functions attempt to handle properly computations when missing values are present in the data. Both NaN and NA are displayed as NA, but the data values are properly kept as different values.

The .C and .Fortran functions have two arguments, the NAOK and the specialsock argument, that you can use to specify whether your code can handle missing values or IEEE special values (Inf and NaN), respectively. Their default value is FALSE: if any argument to .C or .Fortran contains an NA (or Inf or NaN), you get an error message and

your code is not called. To specify these arguments, you must use their complete names, and you cannot use these names for the arguments passed to your C or Fortran code.

Warning

The `NAOK` and `special sok` arguments refer to all of the arguments to your compiled code. You can allow NAs or IEEE special values in all of the arguments or none of them. Because typically you do not want NAs for certain arguments, such as the length of a data set, you must specially check those arguments if you use `NAOK=T` (or `special sok=T`).

Dealing with IEEE special values is easily done in C as long as you use the macros described below. It is possible, yet undocumented here, to do the same in Fortran, but refer to your Fortran compiler documentation for details.

It is often simplest to remove NAs from your data in the Spotfire S+ code, but is sometimes better done in C. If you allow NAs, you should deal with them using the C macros `is_na()` and `na_set()` described below. The arguments to `.C` and `.Fortran` cannot contain any NAs unless the special argument `NAOK` is `T`. The following macros test for and set NAs in your C code:

```
is_na(x,mode)
na_set(x,mode)
```

The argument `x` must be a pointer to a numeric type and the argument `mode` must be one of the symbolic constants `S_MODE_LGL` (S-PLUS class "logical"), `S_MODE_INT` (S-PLUS class "integer"), `S_MODE_REAL` (S-PLUS class "single"), `S_MODE_DOUBLE`, or `S_MODE_COMPLEX`, corresponding to the type `x` points to: `long`, `long`, `float`, `double`, or `complex`, respectively. For example, the following C code sums a vector of double precision numbers, setting the sum to NA if any addends are NA:

```
#include <S.h>
void my_sum(double *x, long *n, double *sum) {
    long i;
    *sum = 0 ;
    for (i = 0 ; i < *n ; i++)
        if (is_na(&x[i], S_MODE_DOUBLE)) {
            na_set(sum, S_MODE_DOUBLE);
        }
}
```

```
        break;
    }
    else
        *sum += x[i];
    }
}
```

Use the following S-PLUS function to call this routine:

```
> my.sum <- function(x) .C("my_sum", as.double(x),
    as.integer(length(x)), double(1), NAOK = T)[[3]]
```

Call this from Spotfire S+ as follows:

```
> my.sum(c(1,NA,2))
```

```
[1] NA
```

```
> my.sum(1:4)
```

```
[1] 10
```

If you omit the argument NAOK=T in the call to .C, you get the following message:

```
> my.sum <- function(x)
    .C("my_sum", as.double(x), as.integer(length(x)),
    double(1))[3]]
> my.sum(c(1,NA,2))
```

```
Problem in .C("my_sum",: subroutine my_sum: 1 missing
value(s) in argument 1
Use traceback() to see the call stack
```

Warning

Both `is_na()` and `na_set()` have arguments that might be evaluated several times; therefore, do not use expressions with side effects in them, such as `na_set(&x[i++], S_MODE_DOUBLE)`. Otherwise, the side effects may occur several times. The call `is_na(x,mode)` returns 0 if *x is not an NA and nonzero otherwise (the nonzero value is not necessarily 1). The return value tells what sort of value *x is: `Is_NA` meaning a true NA and `Is_NaN` meaning an IEEE not-a-number. To assign a NaN to a value, use the alternative macro `na_set3(x,mode, type)`, where `type` is either `Is_NA` or `Is_NaN`. The macro `na_set(x,mode)` is defined as `na_set3(x,mode,Is_NA)`.

You can use the macros `is_inf(x,mode)` and `inf_set(x,mode,sign)` to deal with IEEE infinities. If you allow IEEE special values, your code should be aware that `x != x` is TRUE if `x` is a NaN. On machines supporting IEEE arithmetic (including most common workstations), `1/0` is **Inf** and `0/0` is **NaN** without any warnings given. You must set the `.C` argument **specialsock** to **T** if you want to let Spotfire S+ pass **NaNs** or **Infs** to your C code. The call `is_inf(x,mode)` returns 0 if `*x` is not infinite and ± 1 if `*x` is $\pm \infty$, respectively. The call `set_inf(x,mode,sign)` sets `*x` to an infinity of the given mode and sign, where the sign is specified by the integer +1 for positive infinities and -1 for negative infinities. Similarly, the call `is_nan(x, mode)` returns 0 if `*x` is not a NaN and 1 if it is.

I/O in C Functions

File input and output is fully supported in C code called from Spotfire S+, but input and output directed to the standard streams `STDIN`, `STDOUT`, and `STDERR` requires special handling. This special handling is provided by the header file **newredef.h**, which is included when you include **S.h**. For example, if you use the `printf()` function to add debugging statements to your code, you must include **S.h**, which includes **newredef.h**, to ensure that your messages appear in a Spotfire S+ GUI window rather than simply disappear. The **newredef.h** file does not support `scanf()`; to read user input from the Spotfire S+ GUI, use `fgets()` to read a line, and then use `sscanf()` to interpret the line.

I/O in Fortran Subroutines

Fortran users cannot use Fortran `WRITE` or `PRINT` statements because they conflict with the I/O in Spotfire S+. Therefore, Spotfire S+ provides the following three subroutines as analogs of the S-PLUS `cat` function:

- `DBLEPR` Prints a double precision variable.
- `REALPR` Prints a real variable.
- `INTPR` Prints an integer variable

As an example of how to use them, here is a short Fortran subroutine for computing the net resistance of 3 resistors connected in parallel:

```
SUBROUTINE RESIS1(R1, R2, R3, RC)
C   COMPUTE RESISTANCES
RC = 1.0/(1.0/R1 + 1.0/R2 + 1.0/R3)
CALL REALPR('First Resistance', -1, R1,1)
```

```
RETURN  
END
```

The second argument to REALPR specifies the number of characters in the first argument; the -1 can be used if your Fortran compiler inserts null bytes at the end of character strings. The fourth argument is the number of values to be printed.

Here is a S-PLUS function that calls RESIS1:

```
> parallel<-function(r1,r2,r3) {  
  .Fortran("resis1",as.single(r1),as.single(r2),  
  as.single(r3),as.single(0))[[4]]  
}
```

Running parallel produces the following:

```
> parallel(25,35,75)  
  
First    Resistance  
[1]      25  
[1]    12.2093
```

Reporting Errors and Warnings

Spotfire S+ provides two functions, stop and warning, for detecting and reporting error and warning conditions. In most cases, you should try to detect errors in your Spotfire S+ code, before calling your compiled code. However, Spotfire S+ does provide several tools to aid error reporting in your compiled code.

C Functions

The include file **S.h** defines macros that make it easy for your C code to generate error and warning messages. The PROBLEM and ERROR macros together work like the S-PLUS function stop:

```
PROBLEM "format string", arg1, ..., argn  
ERROR
```

The PROBLEM and WARN macros together work like the warning function:

```
PROBLEM "format string", arg1, ..., argn  
WARN
```

The odd syntax in these macros arises because they are wrappers for the C library function `printf()`; the `PROBLEM` macro contains the opening parenthesis and the `ERROR` and `WARN` macros both start with the closing parenthesis. The format string and the other arguments must be arguments suitable for the `printf()` family of functions. For example, the following C code fragment:

```
#include <S.h>
double x ;
S_EVALUATOR
...
if (x <= 0)
    PROBLEM "x should be positive, it is %g", x
    ERROR ;
```

is equivalent to the Spotfire S+ code:

```
> if (x<=0) stop(paste("x should be positive, it is", x))
```

Both print the message and exit all of the currently active S-PLUS functions calls. Spotfire S+ then prompts you to try again. Similarly, the C code:

```
#include <S.h>
double x ;
S_EVALUATOR
...
if (x <= 0)
    PROBLEM "x should be positive, it is %g", x
    WARN;
```

is equivalent to the Spotfire S+ code:

```
> if (x<=0) warning(paste("x should be positive, it is",
    x))
```

Warning

The messages are stored in a fixed length buffer before printing, so your message must not overflow this buffer. The buffer length is given by `ERROR_BUF_LENGTH` in **S.h** and is currently 4096 bytes. If your message exceeds this length, Spotfire S+ is likely to crash.

**Fortran
Subroutines**

Many of the I/O statements encountered in a typical Fortran routine arise in error handling-when the routine encounters a problem, it writes a message.

A previous section proposed using DBLEPR, REALPR, and INTPR for any necessary printing. An alternative approach in Spotfire S+ is to use the Fortran routines XERROR and XERRWV for error reporting, in place of explicit WRITE statements. For example, consider again the Fortran routine RESIS1, which computes the net resistance of 3 resistors connected in parallel. A check for division by 0 is appropriate, using XERROR:

```
      SUBROUTINE RESIS1(R1, R2, R3, RC)
      C COMPUTE RESISTANCES
      IF (R1 .EQ. 0 .OR. R2 .EQ. 0 .OR. R3 .EQ. 0) THEN
      CALL XERROR( 'Error : division by 0'
      +LEN('Error : division by 0'),99,2)
      RETURN
      END IF
      RC = 1.0/(1.0/R1 + 1.0/R2 + 1.0/R3)
      CALL REALPR('First Resistance', -1, R1,1)
      RETURN
      END
```

XERROR takes four arguments: a character string message, an integer giving the length of the string in message, an error number (which must be unique within the routine), and an error level. If message is a quoted string, the length-of-message argument can be given as LEN(message).

The XERRWV routine acts like XERROR but also allows you to print two integer values, two real values, or both.

The first four arguments to XERRWV, like the first four arguments to XERROR, are the message, the message length, the error ID, and the error level. The fifth and eighth arguments are integers in the range 0-2 that indicate, respectively, the number of integer values to be reported and the number of real (single precision) values to be reported. The sixth and seventh arguments hold the integer values to be reported, the ninth and tenth arguments hold the real values to be reported.

In the following call to XERRWV, the fifth argument is 1, to indicate that one integer value is to be reported. The sixth argument says that n is the integer to be reported:

```
XERRWV(MSG,LMSG,1,1,1,n,0,0,0.0,0.0)
```

The following Fortran subroutine, test.f, shows a practical application of XERRWV:

```
subroutine test(x, n, ierr)
  real*8 x(1)
  integer n, ierr, LMSG
  character*100 MSG
  ierr = 0
  if (n.lt.3) then
    MSG = 'Integer (I1) should be greater than 2'
    LMSG = len('Integer (I1) should be greater than 2')
    CALL XERRWV(MSG,LMSG,1,1,1,n,0,0,0.0,0.0)
    ierr = 1
  return
endif
do 10 i = 2, n
10  x(1) = x(1) + x(i)
return
end
```

```
> .Fortran("test", as.double(1:2), length(1:2),
  as.integer(1))
```

```
[[1]]:
[1] 1 2
[[2]]:
[1] 2
```

Warning messages:

```
1: Integer (I1) should be greater than 2 in:
.Fortran("test", ....
2: in message above, i1=2 in:
.Fortran("test", ....
```

The error message is duplicated because our Spotfire S+ code interprets the error status from the Fortran. The messages issued by XERROR and XERRWV are stored in an internal message table. Spotfire

S+ provides several functions you can use to manipulate this message table within functions that call Fortran routines using `XERROR` and `XERRWV`:

- `xerror.summary` prints out the current state of the internal message summary table, listing the initial segment of the message, the error number, the severity level, and the repetition count for each message.
- `xerror.clear` clears the message table. This function takes an optional argument, `doprint`. If `doprint=T`, the message table is printed before it is cleared.
- `xerror.maxpr` limits the number of times any one message is queued or printed. The default is 10.

For example, we can write a S-PLUS test function to take advantage of these functions as follows:

```
test <- function(x)
{
  xerror.clear()
  val <- .Fortran("test",
    as.double(x),
    length(x),
    ierr = integer(1))
  if(options()$warn == 0)
    xerror.summary()
  val[[1]][1]
}
```

Calling it as before (after setting the option `warn` to 0) yields the following result:

```
> test(1:2)

  error message summary
message start                nerr level count
Integer (I1) should be greater than 2    1    1    1
other errors not individually tabulated = 0
[1] 1
Warning messages:
1: Integer (I1) should be greater than 2 in:
.Fortran("test", ....
```

```
2: in message above, i1 = 2 in:
.Fortran("test", ....
```

See the `xerror` help file for more information on the S-PLUS functions used with `XERROR`, and the `XERROR` help file for more information on `XERROR` and `XERRWV`.

Calling Fortran From C

Spotfire S+ contains a few C preprocessor macros to help smooth over differences between machines in how to call C code from Fortran and vice versa. The following macros are needed to allow distinctions between the declaration, definition, and invocation of a Fortran common block or Fortran subroutine (coded in either C or Fortran):

Table 5.3: Fortran macros to call from C.

Macro Name	Description
F77_NAME	Declaration of a Fortran subroutine.
F77_SUB	Definition of a Fortran subroutine.
F77_CALL	Invocation of a Fortran subroutine.
F77_COMDECL	Declaration of a Fortran common block.
F77_COM	Usage of a Fortran common block.

As an example of the proper use of the F77 macros, consider the following example C code fragment:

```
...
/* declaration of a common block defined in Fortran */
extern long F77_COMDECL(Forblock)[100];
...
/* declaration of a subroutine defined in Fortran */
void F77_NAME(Forfun)(double *, long *, double *);
...
/* declaration of a function defined in C, callable by
* Fortran */
double F77_NAME(Cfun)(double *, long *);
```

```
...
/* usage of the above common block */
for (i = 0; i < 100; i++) F77_COM(Forblock)[i] = 0;
...
/* invocation of the above functions */
F77_CALL(Forfun)(s1, n1, result);
if (F77_CALL(Cfun)(s2, n2) < 0.0)
...
/* definition of the above 'callable by Fortran' function
*/
double F77_SUB(Cfun)(double *weights, long
*number_of_weights);
...
```

If you are loading code originally written for a specific UNIX compiler (including some submissions to StatLib), you may find that that code does not compile correctly in Windows because not all of these macros are used. Usually, such code does not use the `F77_CALL` macro to invoke the functions (using `F77_SUB` instead), does not use the `F77_COMDECL` macro to declare the Fortran common block (using `F77_COM` instead), and leaves out the `F77_NAME` macro altogether. If you attempt to load such code without substituting `F77_CALL` for `F77_SUB` at the appropriate places, you get compilation errors such as the following:

```
xxx.c(54): Error! E1063: Missing operand
xxx.c(54): Warning! W111: Meaningless use of an expression
xxx.c(54): Error! E1009: Expecting ';' but found 'fortran'
```

Similarly, if you attempt to statically load code without substituting `F77_COMDECL` for `F77_COM` where appropriate, you get a link error such as the following:

```
file xxx.obj(xxx.c): undefined symbol Forblock
```

Finally, if you attempt to statically load code without using `F77_NAME` to declare the subroutine, you get a link error of the following form:

```
file xxx.obj(xxx.c): undefined symbol Cfun
```

Fortran passes all arguments by reference, so a C routine calling Fortran must pass the address of all the arguments.

Warning

Fortran character arguments are passed in many ways, depending on the Fortran compiler. It is impossible to cover up the differences with C preprocessor macros. Thus, to be portable, avoid using character and logical arguments to Fortran routines which you would like to call from C.

Calling C From Fortran

You cannot portably call C from Fortran without running the Fortran through a macro processor. You need a powerful macro processor like `m4` (and even `m4` cannot do all that is needed) and then your code does not look like Fortran any more. We can give some guidelines:

- Try not to do it.
- To be portable, do not use logical or character arguments (this applies to C-to-Fortran calls as well) because C and Fortran often represent them differently.

Calling Functions in the S-PLUS Engine DLL

If your DLL calls internal S-PLUS functions, you will need an import library from the S-PLUS engine, **Sqpe.dll**, to resolve those calls. When you install Spotfire S+, you install import libraries, all named **Sqpe.lib** created with Microsoft Visual C++ Version 6 and Watcom 10.5. If you are using one of these compilers, you are all set. If you are not using one of those compilers, the import libraries might not work with your compiler.

Listing Symbols in Your DLL

When you load a DLL with `dyn.open` or by attaching the chapter that contains it, all its exported symbols are immediately accessible via the functions `.C`, `.Fortran`, and `.Call`. If Spotfire S+ complains that a symbol is not in its load table, most likely the symbol is not properly exported (for instance, because it includes C++ name mangling). To help solve such problems, many compilers offer utilities to help you list symbols exported from a DLL.

Listing Symbols Using DUMPBIN

If you have Visual C++, you can use the `DUMPBIN` utility to view a list of exported symbols in your DLL. You run the `DUMPBIN` utility from a command prompt, as follows:

```
DUMPBIN /exports [/out:filename] dllname
```

The `/exports` switch tells DUMPBIN to report all exported symbols; the optional `/out` switch allows you to specify a file name for DUMPBIN's output. (This is very useful if your DLL exports a lot of symbols.)

For example, to view the symbols exported from the **S.dll** in our examples directory, we can use DUMPBIN as follows:

```
E:\splus6\library\examples\>dumpbin /exports S.dll
Microsoft (R) COFF Binary File Dumper Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights
reserved.
```

```
Dump of file S.dll
```

```
File Type: DLL
```

```
Section contains the following exports for S.dll
```

```
    0 characteristics
3B385498 time date stamp Tue Jun 26 02:23:36 2001
    0.00 version
    1 ordinal base
    3 number of functions
    3 number of names 0 3
number of functions
    3 number of names
ordinal    hint    RVA      name
    1        0    00001000  quantum
    2        1    000010E0  randu
    3        2    00001130  zero_find
```

```
Summary
```

```
1000 .data
1000 .rdata
1000 .reloc
5000 .text
```

As expected, there are only three exported symbols in this DLL. If we are obtaining a DLL compiled by someone else, DUMPBIN `/exports` might be an essential part of using the DLL.

USING C FUNCTIONS BUILT INTO SPOTFIRE S+ FOR WINDOWS

In the previous section, we introduced a number of routines built into Spotfire S+ with the purpose of avoiding certain difficulties in compiling code and generating useful output. This section describes some more generally useful routines that can help you allocate memory as Spotfire S+ does or generate random numbers.

For a more general interface to Spotfire S+ routines and objects, see Chapter 3, CONNECT/C++.

Allocating Memory

Spotfire S+ includes two families of C routines for storage allocation and reallocation. You can use either of these families, or use the standard library functions `malloc()`, `calloc()`, `realloc()`, and `free()`.

However, be very careful to use only one family for any particular allocation; mixing calls using the same pointer variable can be disastrous. The first Spotfire S+ family consists of the two routines `S_alloc()` and `S_realloc()`, which may be used instead of the standard `malloc()` and `realloc()`. The storage they allocate lasts until the current evaluation frame goes away (at the end of the function calling `.C`). If space cannot be allocated, `S_alloc()` and `S_realloc()` perform their own error handling; they will not return a NULL pointer. You cannot explicitly free storage allocated by `S_alloc()` and `S_realloc()`, but you are guaranteed that the storage is freed by the end of the current evaluation frame. (There is no `S_free()` function, and using `free()` to release storage allocated by `S_alloc()` might cause Spotfire S+ to crash.) `S_alloc()` and `S_realloc()` are declared a bit differently from `malloc()` and `realloc()`. (However, `S_alloc` has many similarities to `calloc()`. For example, it zeroes storage and has two sizing arguments). `S_alloc()` is declared as follows in **S.h**:

```
char * S_alloc(long n, int size);
```

Similarly, `S_realloc()` is declared as follows in **S.h**:

```
char * S_realloc(char *p, long new, long old, int size,  
s_evaluator *S_evaluator);
```

`S_alloc()` allocates (and fills with 0s) enough space for an array of `n` items, each taking up `size` bytes. For example, the following call allocates enough space for ten doubles:

```
S_EVALUATOR
...
S_alloc(10, sizeof(double), S_evaluator)
```

The macro `S_EVALUATOR` is required to properly declare the `S_evaluator` argument. `S_realloc()` takes a pointer, `p`, to space allocated by `S_alloc()` along with its original length, `old`, and `size`, `size`, and returns a pointer to space enough for new items of the same size. For example, the following expands the memory block size pointed to by `p` from 10 doubles to 11 doubles, zeroing the 11th double location:

```
S_EVALUATOR
...
S_realloc(p,11,10, sizeof(double), S_evaluator)
```

The contents of the original vector are copied into the beginning of the new one and the trailing new entries are filled with zeros. You must ensure that `old` and `size` were the arguments given in the call to `S_alloc()` (or a previous call to `S_realloc()`) that returned the pointer `p`. The new length should be longer than the old. As a special case, if `p` is a `NULL` pointer (in which case `old` must be 0), then `S_realloc()` acts just like `S_alloc()`.

The second Spotfire S+ family of allocation routines consists of the three macros `Calloc()`, `Realloc()`, and `Free()`. (Note the capitalization.)

`Calloc()` and `Realloc()` are simple wrappers for `calloc()` and `realloc()` that do their own error handling if space can not be allocated (they will not return if the corresponding wrapped function returns a `NULL` pointer). `Free()` is a simple wrapper for `free()` that

sets its argument to NULL. As with `calloc()`, `realloc()`, and `free()`, memory remains allocated until freed. This may be before or after the end of the current frame.

Warning

If you use `malloc()` or `realloc()` directly, you must free the allocated space with `free()`. Similarly, when using `Calloc()` or `Realloc()`, you must free the allocated space with `Free()`. Otherwise, memory will build up, possibly causing Spotfire S+ to run out of memory unnecessarily. However, be aware that because S processing may be interrupted at any time (for example, when the user presses the interrupt key, or if further computations encounter an error and dump), it is sometimes difficult to guarantee that the memory allocated with `malloc()` or `realloc()` (or `Calloc()` or `Realloc()`) is freed.

Note

If, in a call to `S_alloc()`, `S_realloc()`, `Calloc()` or `Realloc()`, the requested memory allocation cannot be obtained, those routines call `ERROR`. See the section Reporting Errors and Warnings on page 125 for more information on the `ERROR` macro.

Generating Random Numbers

Spotfire S+ includes user-callable C routines for generating standard uniform and normal pseudo-random numbers. It also includes procedures to get and set the permanent copy of the random number generator's seed value. The following routines each return one pseudo-random number:

```
double unif_rand(S_evaluator *S_evaluator);  
double norm_rand(S_evaluator *S_evaluator);
```

Before calling either function, you must get the permanent copy of the random seed from disk into Spotfire S+ (which converts it to a convenient internal format) by calling `seed_in((long *)NULL, S_evaluator *S_evaluator)`. You can specify a particular seed using `setseed(long *seed, S_evaluator *S_evaluator)`, which is equivalent to the S-PLUS function `set.seed`. When you are finished generating random numbers, you must push the permanent copy of the random seed out to disk by calling `seed_out((long *)NULL, S_evaluator *S_evaluator)`. If you do not call `seed_in()` before the

random number generators, they fail with an error message. If you do not call `seed_out()` after a series of calls to `unif_rand()` or `norm_rand()`, the next call to `seed_in()` retrieves the same seed as the last call and you get the same sequence of random numbers again.

The seed manipulation routines take some time so we recommend calling `seed_in()` once, then calling `unif_rand()` or `norm_rand()` as many times as you need to, then calling `seed_out()` before returning from your C function. A simple C function to calculate a vector of standard normals is implemented as follows:

```
#include <S.h>
my_norm(double *x, long *n) {
  S_EVALUATOR
  long i;
  seed_in( (long *) NULL, S_evaluator);
  for (i=0 ; i<*n ; i++)
    x[ i ] = S_DOUBLEVAL(norm_rand(S_evaluator));
  seed_out( (long *) NULL, S_evaluator);
}
```

To call it from Spotfire S+, define the function `my.norm` as follows:

```
> my.norm <- function(n)
  .C("my_norm", double(n), as.integer(n))[[1]]
```

Of course, it is simpler and safer to use the S-PLUS function `rnorm` to generate a fixed number of normal variates to pass into an analysis function. We recommend that you generate the random variates in C code only when you cannot tell how many random variates you need, as when using a rejection method of generating nonuniform random numbers.

Warning

Because of possible differences in the way Microsoft Visual C++ and other compilers (particularly Watcom C/C++) handle return values from floating point functions, the example above uses the `S_DOUBLEVAL` macro (defined when **S.h** is included). The `S_DOUBLEVAL` or `S_FLOATVAL` macros, defined in **compiler.h**, may be needed when calling floating point functions internal to Spotfire S+ from DLLs compiled with other non-Microsoft compilers; see the section *Calling Functions in the S-PLUS Engine DLL* on page 132.

CALLING SPOTFIRE S+ FUNCTIONS FROM C CODE (WINDOWS)

To this point, we have shown how to call C and Fortran routines from S-PLUS functions. You can also call S-PLUS functions from C code, using the supplied C routine `call_S()`. The `call_S()` routine is useful as an interface to numerical routines which operate on C or Fortran functions, but it is not a general purpose way to call S-PLUS functions. The C routine calling `call_S()` must be loaded into Spotfire S+, the arguments to the function must be simple, and the nature of the output must be known ahead of time. Because of these restrictions, `call_S()` cannot be used to call S-PLUS functions from an independent C application, as you might call functions from a subroutine library.

For a more general interface to Spotfire S+ routines and objects, see Chapter 3, `CONNECT/C++`.

The C function `call_S()` calls a S-PLUS function from C, but `call_S()` must be called by C code called from Spotfire S+ via `.C`. The `call_S()` function has the following calling sequence:

```
call_S(void *func, long nargs, void **arguments,  
char **modes, long *lengths, char **names,  
long nres, void **results);
```

where:

- `func` is a pointer to a list containing one S-PLUS function. This should have been passed via an argument in a `.C` call, as follows:

```
.C("my_c_code", list(myfun))
```

This calls C code starting with the following lines:

```
my_c_code(void **Sfunc) {  
    ...  
    call_S(*Sfunc, ...);  
    ...  
}
```

The S-PLUS function must return an atomic vector or list of atomic vectors.

- `nargs` is the number of arguments to give to the S-PLUS function `func`.
- `arguments` is an array of `nargs` pointers to the data being passed to `func`. These can point to any atomic type of data, but must be cast to type `void*` when put into arguments.
- `modes` is an array of `nargs` character strings giving the S-PLUS names, for example, "double" or "integer", of the modes of the arguments given to `func`.
- `lengths` is an array of `nargs` longs, giving the lengths of the arguments.
- `names` is an array of `nargs` strings, giving the names to be used for the arguments in the call to `func`. If you do not want to call any arguments by name, `names` may be `(char **)NULL`; if you do not want to call the `i`th argument by name, `names[i]` may be `(char *)NULL`.
- `nres` is the maximum number of components expected in the list returned by `func` (if `func` is expected to return an atomic vector, then `nres` should be 1).
- `results` is filled in by `call_S()`; it contains generic pointers to the components of the list returned by `func` (or a pointer to the value returned by `func` if the value were atomic).

Your C code calling `call_S()` should cast the generic pointers to pointers to some concrete type, like `float` or `int`, before using them. If `func` returns a list with fewer components than `nres`, the extra elements of `results` are filled with `NULLs`. Notice that `call_S()` does not report the lengths or modes of the data pointed to by `results`; you must know this a priori.

To illustrate the use of `call_S()`, we construct (in Fortran) a general purpose differential equation solver, `heun()`, to solve systems of differential equations specified by a S-PLUS function. Other common applications involve function optimization, numerical integration, and root finding.

The `heun()` routine does all its computations in single precision and expects to be given a subroutine of the following form:

```
f(t, y, dydt)
```

where the scalar t and vector y are given, and the vector $dydt$, the derivative, is returned. Because the $f()$ subroutine calls the S-PLUS function, it must translate the function's argument list into one that $call_S()$ expects. Because not all the data needed by $call_S$ can be passed into $f()$ via an argument list of the required form, we must have it refer to global data items for things like the pointer to the S-PLUS function and the modes and lengths of its arguments. The following file of C code, `dfeq.c`, contains a C function $f()$ to feed to the solver `heun()`. It also contains a C function `dfeq()` which initializes data that $f()$ uses and then calls `heun()` (which repeatedly calls $f()$):

```
#include <S.h>
extern void F77_NAME(heun)();
/* pointer to Splus function to be filled in */
static void *Sdydt ;
/* descriptions of the functions's two arguments */
static char *modes[] = {"single", "single" };
static long lengths[] = {1, 0 };
    /* neqn = lengths[1] to be filled in */
static char *names[] = { "t", "y" };

/*
    t [input]: 1 long ; y [input]: neqn long ;
    yp [output]: neqn long
*/
static void f(float *t, float *y, float *yp) {
    void *in[2] ; /* for two inputs to Splus function,
        t and y */
    void *out[1] ; /* for one output vector of
        Splus function */

    int i;
    in[0] = (void *)t;
    in[1] = (void *)y;
    call_S(Sdydt, 2L,
        in, modes, lengths, names, /* 2 arguments */
        1L, out/* 1 result */);

    /* the return value out must be 1 long - i.e., Splus
```

```
function must return an atomic vector or a list of one
atomic vector. We can check that it is at least 1 long. */
if (!out[0])
  PROBLEM
  "Splus function returned a 0 long list"
  RECOVER(NULL_ENTRY);

/* Assume out[0] points to lengths[1] single precision
  numbers. We cannot check this assumption here. */
for(i=0;i<lengths[1];i++)
  yp[i] = ((float *)out[0])[i] ;
return ;
}

/* called via .C() by the Splus function dfreq(): */
void dfreq(void **Sdydtp, float *y, long *neqn,
  float *t_start, float *t_end, float *step,
  float *work) {
  /* Store pointer to Splus function and
  number of equations */
  Sdydt = *Sdydtp ;

/* call Fortran differential equation solver */
  F77_CALL(heun)(f, neqn, y, t_start, t_end, step, work);
}
```

Warning

In the C code, the value of the S-PLUS function was either atomic or was a list with at least one atomic component. To make sure there was no more than one component, you could look for two values in results and make sure that the second is a null pointer.

The following S-PLUS function, dfreq, does some of the consistency tests that our C code could not do (because call_S did not supply enough information about the output of the S-PLUS function). It also allocates the storage for the scratch vector. Then it repeatedly calls the C routine, dfreq(), to have it integrate to the next time point that we are interested in:

```
> dfreq <- function(func, y , t0 = 0, t1 = 1, nstep = 100,
  stepsize = (t1-t0)/nstep)
```

```

{
  if (length(func) != 3 ||
      any(names(func) != c("t","y", "")))
    stop("arguments of func must be called t and y")
  y <- as.single(y)
  t0 <- as.single(t0)
  neqn <- length(y)
  test.val <- func(t = t0, y = y)
  stop("y and func(t0,y) must be same length")
  if(storage.mode(test.val) != "single")
    stop("func must return single precision vector")
  val <- matrix(as.single(NA), nrow = nstep + 1,
               ncol = neqn)
  val[1, ] <- y
  time <- as.single(t0 + seq(0, nstep) * stepsize)
  for(i in 1:nstep) {
    val[i + 1, ] <- .C("dfeq", list(func), y=val[i, ],
                     neqn=as.integer(neqn),
                     t.start=as.single(time[i]),
                     t.end=as.single(time[i + 1]),
                     step=as.single(stepsize),
                     work=single(3 * neqn))$y
  }
  list(time=time, y=val)
}

```

The following subroutine is the Fortran code, `heun.f`, for Heun's method of numerically solving a differential equation. It is a first order Runge-Kutta method. Production quality differential equation solvers let you specify a desired local accuracy rather than step size, but the code that follows does not:

```

C   Heun's method for solving dy/dt=f(t,y),
C   using step size h :
C   k1 = h f(t,y)
C   k2 = h f(t+h,y+k1)
C   ynext = y + (k1+k2)/2

      subroutine heun(f, neqn, y, tstart, tend, step, work
      integer neqn
      real*4 f, y(neqn), tstart, tend, step, work(neqn,3)
C   work(1,1) is k1, work(1,2) is k2, work(1,3) is y+k1

```

```
integer i, nstep, istep
real*4 t
external f
nstep = max((tend - tstart) / step, 1.0)
step = (tend - tstart) / nstep
do 30 istep = 1, nstep
    t = tstart + (istep-1)*step
    call f(t, y, work(1,1))
    do 10 i = 1, neqn
        work(i,1) = step * work(i,1)
        work(i,3) = y(i) + work(i,1)
10    continue
    call f(t+step, work(1,3), work(1,2))
    do 20 i = 1, neqn
        work(i,2) = step * work(i,2)
        y(i) = y(i) + 0.5 * (work(i,1) + work(i,2))
20    continue
30    continue
    return
end
```

To try out this example of `call_S`, exercise it on a simple one dimensional problem as follows:

```
> graphsheet()
> a <- dfeq(function(t,y)t^2, t0=0, t1=10, y=1)
> plot(a$time,a$y)
> lines(a$time, a$time^3/3+1) # compare to
    #theoretical solution
```

You can increase `nstep` to see how decreasing the step size increases the accuracy of the solution. The local error should be proportional to the square of the step size and when you change the number of steps from 100 to 500 (over the same time span) the error does go down by a factor of about 25. An interesting three-dimensional example is the Lorenz equations, which have a strange attractor:

```
> chaos.func<-function(t, y) {
    as.single(c(10 * (y[2] - y[1]),
        - y[1] * y[3] + 28 * y[1] - y[2],
        y[1] * y[2] - 8/3 * y[3]))
}
```

```
> b <- dfreq(chaos.func, y=c(5,7,19), t0=1, t1=10,
             nstep=300)
> b.df <- data.frame(b$time,b$y)
> pairs(b.df)

 2 4 6 8 10
-20 0 10 20
```

The resulting plot is shown in Figure 5.1.

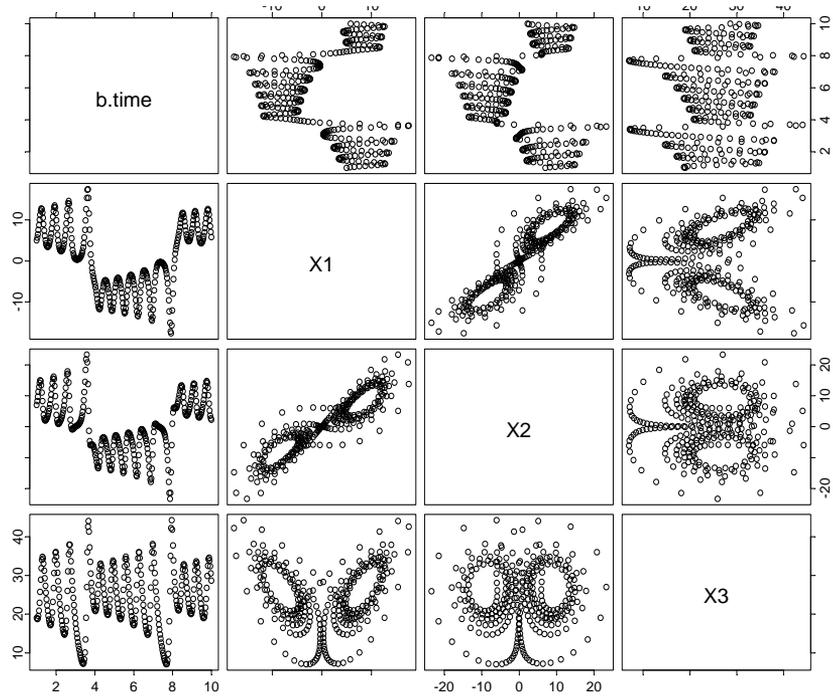


Figure 5.1: Viewing the Lorenz equations, as solved by dfreq.

Warnings

Because `call_S` does not describe the output of the S-PLUS function it calls, you must know about it ahead of time. You can test the function for a variety of values before calling `call_S` to check for gross errors, but you cannot ensure that the function will not return an unacceptable value for certain values of its arguments.

The `call_S` function expects that the output of the function given to it has no attributes. If it does have attributes, such as dimensions or names, they are stripped.

THE .CALL INTERFACE (WINDOWS)

The `.call` interface is a powerful interface that allows you to manipulate S-PLUS objects from C code. It is more efficient than the standard `.C` interface, but because it allows you to work directly with S-PLUS objects, without the usual Spotfire S+ protection mechanisms, you must be careful in programming with it to avoid memory faults and corrupted data.

The `.call` interface provides you with several capabilities the standard `.C` interface lacks, including the following:

- The ability to create variable-length output variables, as opposed to the pre-allocated objects the `.C` interface expects to write to.
- A simpler mechanism for evaluating S-PLUS expressions within C.
- The ability to establish direct correspondence between C pointers and S-PLUS objects.

The `.call` interface is also the point of departure for using CONNECT/C++, a powerful suite of C++ classes and methods to give C++ programmers access to S-PLUS objects and methods. See Chapter 3, CONNECT/C++, for more information.

Requirements

To use the `.call` interface, you must ensure your code meets the following requirements¹:

- The return value and all arguments have C type `"s_object *"`.
- The code must include the standard Spotfire S+ header file **S.h**.
- If the routine deals with S-PLUS objects, it must include a declaration of the evaluator using the macro `S_EVALUATOR`, appearing in the declaration part of the routine and not followed by a semicolon.

1. Chambers, J.M. (1998) Programming with Data. New York: Springer-Verlag. p. 429.

As with `.C`, the required arguments to `.Call` include the name of the C routine being called and one argument for each argument to the C routine.

Returning Variable-Length Output Vectors

Occasionally, we do not know how long the output vector of a procedure is until we have done quite a bit of processing of the data. For example, we might want to read all the data in a file and produce a summary of each line. Until we have counted the lines in the file, we do not know how much space to allocate for a summary vector. Generally, `.C` passes your C procedure a pointer to a data vector allocated by your S-PLUS function so you must know the length ahead of time. You could write two C procedures: one to examine the data to see how much output there is and one to create the output. Then you could call the first in one call to `.C`, allocate the correct amount of space, and call the second in another call to `.C`. The first could even allocate space for the output vector as it is processing the input and have the second simply copy that to the vector allocated by your S-PLUS function.

With the `.Call` interface, however, you can create the desired S-PLUS object directly from your C code.

Here is an example which takes a vector `x` of integers and returns a sequence of integers, of length `max(x)`:

```
#include "S.h"
s_object *makeseq(s_object *sobjX)
{
    S_EVALUATOR
    long i, n, xmax, *seq, *x ;
    s_object *sobjSeq ;

    /* Convert the s_objects into C data types: */
    sobjX = AS_INTEGER(sobjX) ;
    x = INTEGER_POINTER(sobjX) ;
    n = GET_LENGTH(sobjX) ;

    /* Compute max value: */
    xmax = x[0] ;
    if(n > 1) {
        for(i=1; i<n; i++) {
            if(xmax < x[i]) xmax = x[i] ;
        }
    }
}
```

```
    }  
  }  
  if(xmax < 0)  
    PROBLEM "The maximum value (%ld) is  
    negative.", xmax ERROR ;  
  
  /* Create a new s_object, set its length and get a C integer  
  pointer to it */  
  sobjSeq = NEW_INTEGER(0) ;  
  SET_LENGTH(sobjSeq, xmax) ;  
  seq = INTEGER_POINTER(sobjSeq) ;  
  
  for(i=0; i<xmax; i++) {  
    seq[i] = i + 1 ;  
  }  
  
  return(sobjSeq) ;  
}
```

Use the following Spotfire S+ code to call `makeseq()`:

```
"makeseq" <-  
function(x)  
{  
  x <- as.integer(x)  
  .Call("makeseq", x)  
}
```

S Object Macros

The `makeseq` example has several interesting features, but perhaps the most useful is its extensive use of S object macros. These macros are defined when you include **S.h**, and allow you to create, modify, and manipulate actual S-PLUS structures from within your C code. There are five basic macros, each of which is implemented particularly for the basic data types listed in Table 5.1. These macros are described in Table 5.4. To obtain the full name of the desired macro, just substitute the basic data type from Table 5.1 in ALLCAPS for the word type in the macro name given in Table 5.4. Thus, to create a new numeric S-PLUS object, use the macro `NEW_NUMERIC`.

Table 5.4: *S* object macros

Macro	Description
NEW_type(n)	Create a pointer to an S object of class <i>type</i> and length n.
AS_type(obj)	Coerce obj to an S object of class <i>type</i> .
IS_type(obj)	Test whether obj is an S object of class <i>type</i> .
type_POINTER(obj)	Create a pointer of type <i>type</i> to the data part of obj.
type_VALUE(obj)	Returns the value of obj, which should have length 1.

The `makeseq` code uses the `AS_INTEGER` macro to coerce the `sobjx` object to type `INTEGER`; the `NEW_INTEGER` macro to create the returned sequence object; and the `INTEGER_POINTER` macro to access the data within those objects.

The `makeseq` code also uses built-in macros for getting and setting basic information about the S objects: in addition to the `GET_LENGTH` and `SET_LENGTH` macros used in `makeseq`, there are also `GET_CLASS` and `SET_CLASS` macros to allow you to obtain class information about the various S objects passed into your code.

Evaluating S-PLUS Expressions from C

You can evaluate a S-PLUS expression from C using the macros `EVAL` and `EVAL_IN_FRAME`. Both take as their first argument a S-PLUS object representing the expression to be evaluated; `EVAL_IN_FRAME` takes a second argument, `n`, representing the S-PLUS frame in which the evaluation is to take place.

For example, consider the internal C code for the `lapply` function, which was first implemented by John Chambers in his book *Programming with Data*:

```
#include "S_engine.h"
/* See Green Book (Programing with Data by J.M. Chambers)
appendix A-2 */
```

```
s_object *
S_qapply(s_object *x, s_object *expr, s_object *name_obj,
         s_object *frame_obj)
{
    S_EVALUATOR
    long frame, n, i;
    char *name;
    s_object **els;
    x = AS_LIST(x) ;
    els = LIST_POINTER(x);
    n = LENGTH(x);
    frame = INTEGER_VALUE(frame_obj) ;
    name = CHARACTER_VALUE(name_obj) ;
    for(i=0;i<n;i++) {
        ASSIGN_IN_FRAME(name, els[i], frame) ;
        SET_ELEMENT(x, i, EVAL_IN_FRAME(expr,
                                         frame)) ;
    }
    return x;
}
```

This uses the more general macro `EVAL_IN_FRAME` to specify the specific frame in which to evaluate the specified expression. Note also the `SET_ELEMENT` macro; this must *always* be used to perform assignments into S-PLUS list-like objects from C.

DEBUGGING LOADED CODE (WINDOWS)

Frequently the code you are dynamically linking is known, tested, and reliable. But what if you are writing new code, perhaps as a more efficient engine for a routine developed in Spotfire S+? You may well need to debug both the C or Fortran code and the S-PLUS function that calls it. The first step in debugging C and Fortran routines for use in Spotfire S+ is to make sure that the C function or Fortran subroutine is of the proper form, so that all data transfer from Spotfire S+ to C or Fortran occurs through arguments. Both the input from Spotfire S+ and the expected output need to be arguments to the C or Fortran code. The next step is to ensure that the classes of all variables are consistent. This often requires that you add a call such as `as(variable, "single")` in the call to `.C` or `.Fortran`. If the Spotfire S+ code and the compiled code disagree on the number, classes, or lengths of the argument vectors, Spotfire S+'s internal data might be corrupted and it will probably crash. By using `.C` or `.Fortran` you are trading the speed of compiled code for the safety of Spotfire S+ code. In this case, you usually get an application error message before your Spotfire S+ session crashes. Once you have verified that your use of the interface is correct, and you have determined there is a problem in the C or Fortran code, you can use an analog of the `cat` statement to trace the evaluation of your routine.

Debugging C Code

If you are a C user, you can use C I/O routines, provided you include **S.h**. Thus, you can casually sprinkle `printf` statements through your C code just as you would use `cat` or `print` statements within a S-PLUS function. (If your code is causing Spotfire S+ to crash, call `fflush()` after each call to `printf()` to force the output to be printed immediately.)

Debugging C Code Using a Wrapper Function

If you cannot uncover the problem with generous use of `printf()`, the following function, `.Cdebug`, (a wrapper function for `.C`) can sometimes find cases where your compiled code writes off the end of an argument vector. It extends the length of every argument given to it and fills in the space with a flag value. Then it runs `.C` and checks that the flag values have not been changed. If any have been changed, it prints a description of the problem. Finally, it shortens the arguments down to their original size so its value is the same as the value of the corresponding `.C` call.

```
.Cdebug <- function(NAME, ..., NAOK = F, specialsok = F,
ADD = 500, ADD.VALUE = -666)
{
  args <- list(...)
  tail <- rep(as.integer(ADD.VALUE), ADD)
  for(i in seq(along = args))
    {
      tmp <- tail
      storage.mode(tmp) <- storage.mode(args[[i]])
      args[[i]] <- c(args[[i]], tmp)
    }
  args <- c(NAME = NAME, args, NAOK = NAOK,
           specialsok = specialsok)
  val <- do.call(".C", args)
  for(i in seq(along = val))
    {
      tmp <- tail
      storage.mode(tmp) <- storage.mode(args[[i]])
      taili <- val[[i]][seq(to = length(val[[i]]),
                           length = ADD)]
      if((s <- sum(taili != tmp)) > 0) {
        cat("Argument ", i, "(", names(val)[i],
           ") to ", NAME, " has ", s, " altered
           values after end of array\n ",
           sep = "")
      }
      length(val[[i]]) <- length(val[[i]]) - ADD
    }
  }
  val
}
```

For example, consider the following C procedure, oops():

```
oops(double *x, long* n)
{
  long i;
  for (i=0 ; i <= *n ; i++) /* should be <, not <= */
    x[i] = x[i] + 10 ;
}
```

Because of the misused `<=`, this function runs off the end of the array `x`. If you call `oops()` using `.C` as follows, you get an Application Error General Protection Fault that crashes your Spotfire S+ session:

```
> .C("oops", x=as.double(1:66), n=as.integer(66))
```

If you use `.Cdebug` instead, you get some information about the problem:

```
> .Cdebug("oops", x=as.double(1:66), n=as.integer(66))
```

```
Argument 1(x) to oops has 1 altered values after end of array
```

```
x:
```

```
[1] 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28  
[19] 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46  
[37] 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64  
[55] 65 66 67 68 69 70 71 72 73 74 75 76
```

```
n:
```

```
[1] 66
```

The `.Cdebug` function cannot tell when you run off the beginning of an argument vector or when you write anywhere else in memory. If inspecting your source code and using S-PLUS functions like `.Cdebug` is not enough to pinpoint a problem, try the following:

1. Write a short main program that calls your procedure.
2. Compile and link the main program and your procedure for debugging.

Source-Level Debugging

If your compiled routines are fairly complicated, you may want more help in debugging than can be provided by simple print statements. Microsoft Visual C++ and Compaq Visual Fortran come with sophisticated visual debuggers.

If you are using Microsoft Visual C++, you can easily do source-level debugging of your code. Simply follow the instructions for creating a DLL as outlined in the section Using C and Fortran Code with Spotfire S+ for Windows on page 102.

Before creating the DLL, you should ensure that the default project configuration (under **Build ► Set Active Configuration**) is set to Win32 Debug. You must also specify the executable to be used for your debug session. To do this, select **Project ► Settings** to display

the **Project Settings** dialog box, and then select the **Debug** tab. Under **Settings For**, select **Win32 Debug**, and for **Executable for debug session**, provide the full path to the Spotfire S+ executable (**SPLUS.EXE** in the XMA subdirectory of your Spotfire S+ installation). You should also set your **S_PROJ** directory to the current project chapter in **Program arguments** as follows:

```
S_PROJ=.
```

(The period says to use the current directory.) When you have started your debug session, remember that the DLL will have been created in the **Debug** subdirectory of your project directory.

A SIMPLE EXAMPLE: FILTERING DATA (UNIX)

In this section, we develop a function to apply a first order linear recursive filter to a vector of data. The S-PLUS function `filter` does what we want, but we'll ignore it for now in favor of the following pure S code:

```
Ar <- function(x, phi)
{
  n <- length(x)
  if (n>1)
    for (i in 2:n)
      x[i] <- phi * x[i - 1] + x[i]
  x
}
```

Looping is traditionally one area where Spotfire S+ tends to be significantly slower than compiled code, so we can rewrite the above code in C as follows, creating a file `Ar.c`:

```
void arsim(double *x, long *n, double *phi)
{
  long i;
  for (i=1; i<*n; i++)
    x[i] = *phi * x[i-1] + x[i] ;
}
```

This code is purely C language code; there are no dependencies on C libraries, or on Spotfire S+. Such code should be portable to most operating systems. It is quite simple to create a shared object from this code:

1. Create the file `Ar.c` shown above.
2. Run the Spotfire S+ **CHAPTER** utility with `Ar.c` as a parameter:

Splus CHAPTER Ar.c

The **CHAPTER** utility creates a makefile for use with the **make** utility.

3. Run the Spotfire S+ **make** utility:

Splus make

The result is a shared object file, `S.so`.

If you've done the above three steps from within Spotfire S+, you can open the file `S.so` simply by calling `dyn.open("S.so")`. If you performed them outside of Spotfire S+, simply start Spotfire S+ within the current Spotfire S+ chapter, and the `S.so` file will be opened automatically.

To run the filtering code, we can either call `.C` directly, or we can write a simple S function to do it for us. If we want to use our loaded call very often, it will save us time to define the function:

```
ar.compiled <-  
function(x, phi)  
{  
  .C("arsim",  
    as.double(x),  
    length(x),  
    as.double(phi))[[1]]  
}
```

Trying the code with a call to `ar.compiled` yields the following:

```
> ar.compiled(1:20, .75)  
[1] 1.000000 2.750000 5.062500 7.796875 10.847656  
[6] 14.135742 17.601807 21.201355 24.901016 28.675762  
[11] 32.506822 36.380116 40.285087 44.213815 48.160362  
[16] 52.120271 56.090203 60.067653 64.050739 68.038055
```

You lose some flexibility in the function by writing it in C. Our `ar.compiled` function converts all input data to double precision, so it won't work correctly for complex data sets nor objects with special arithmetic methods. The pure Spotfire S+ version works for all these cases. If complex data is important for your application, you could write C code for the complex case and have the Spotfire S+ code decide which C function to call. Similarly, to make `ar.compiled` work for data in classes with special arithmetic methods, you could have it call the C code only after coercing the data to class `"numeric"`, so that it could not invoke special arithmetic methods. This might be too conservative, however, as there could be many classes of data without arithmetic methods which could use the fast C code.

Another approach would be to make `ar.compiled` a generic function, for which the default method calls the C code for numeric data. For classes with special arithmetic methods, pure Spotfire S+ code could be dispatched. Those classes of data without special arithmetic methods could include a *class* method for `ar.compiled` that would coerce the data to class `"numeric"` and invoke the default method on the now numeric data, thus using the fast compiled code, then post-process the result if needed (perhaps just restoring the class). Using the object-oriented approach is more work to set up, but gives you the chance to combine the speed of compiled code with the flexibility of Spotfire S+ code.

CALLING C OR FORTRAN ROUTINES FROM SPOTFIRE S+ FOR UNIX

Calling C

To call a C function, use the S-PLUS function `.C()`, giving it the name of the C function (as a character string) and one S-PLUS argument for each C argument. For example, a typical “vectorized” calculation, such as sine, requires you to pass a S-PLUS data object `x` and its length `n` to the C function performing the calculation:

```
.C("my_sin_vec", x = as.double(x),
  n = as.integer(length(x)))
```

(We’ll define the C routine `my_sin_vec` in the section Writing C and Fortran Routines Suitable for Use in Spotfire S+ for Unix (page 161).)

To return results to Spotfire S+, modify the data pointed to by the arguments. The value of the `.C()` function is a list with each component matching one argument to the C function. If you name these arguments, as we did in the preceding example, the return list has named components. Your S-PLUS function can use the returned list for further computations or to construct its own return value, which generally omits those arguments which are not altered by the C code. Thus, if we wanted to just use the returned value of `x`, we could call `.C()` as follows:

```
.C("my_sin_vec", x = as.double(x),
  n = as.integer(length(x)))$x.
```

All arguments to C routines called via `.C()` must be pointers. All such routines should be `void` functions; if the routine does return a value, it could cause Spotfire S+ to crash. Spotfire S+ has many classes that are not immediately representable in C. To simplify the interface between Spotfire S+ and C, the types of data that Spotfire S+ can pass to C code are restricted to the following S-PLUS *classes*: "single", "integer", "numeric", "complex", "logical", and "character". Table 5.5 shows the correspondence between S-PLUS classes and C types.

Table 5.5: *Correspondence between S-PLUS classes and C types.*

S-PLUS classes	Corresponding C type
"logical"	long *
"integer"	long *

Table 5.5: Correspondence between S-PLUS classes and C types.

"single"	float *
"numeric"	double *
"complex"	s_complex *
"character"	char **
"raw"	char *
"list"	s_object **

Warning

Do *not* declare integer data as C ints, particularly if you want your code to be portable among machines that Spotfire S+ supports. While there is currently no difference on Windows, there is a distinction on other platforms.

The include file **S.h** described later in this chapter contains the typedef for the type `s_complex` that defines it as the struct composed of two doubles, `re` and `im`.

Calling C++

To call a C++ function, you also use the `.C` function (or, alternatively, the `.Call` function discussed later in this chapter). There is no direct C++ interface, and hence Spotfire S+ has no understanding of C++ name mangling. Thus, to call a C++ function, you must declare it inside an extern "C" braced expression. For example, here is some simple code to compute squares:

```
#include "S.h"

extern "C" {

void squareC(double* pdX, double* pdY, long* pLen)
{
    S_EVALUATOR
    //Validate the input arguments
    if((pdX == NULL) || (pdY == NULL) || pLen == NULL))
        PROBLEM "Invalid input" ERROR;
    //Perform element-by-element operation
    //to square each element of input
```

```

for(long n=0; n< *p1Len; n++)
    pdY[n] = pdX[n] * pdX[n];
return;
}
}

```

We can call this with `.C` using the simple Spotfire S+ code shown below:

```

square <- function(x)
{
    len = length(x)
    y = .C("squareC",
          as.double(x),
          y = double(len),
          len)$y
    y
}

```

Calling Fortran To call a Fortran subroutine, use the S-PLUS function `.Fortran()`, giving it the name of the subroutine (as a character string) and one S-PLUS argument for each Fortran argument. For example, a typical “vectorized” calculation, such as sine, requires you to pass a S-PLUS data object `x` and its length `n` to the Fortran subroutine performing the calculation:

```

.Fortran("my_sin_vec", x = as.double(x),
        n = as.integer(length(x)))

```

Note
You can call only Fortran <i>subroutines</i> from Spotfire S+; you cannot call Fortran <i>functions</i> .

To return results to Spotfire S+, modify the data pointed to by the arguments. The value of the `.Fortran()` function is a list with each component matching one argument to the Fortran subroutine. If you name these arguments, as we did in the preceding example, the return list has named components. Your S-PLUS function can use the returned list for further computations or to construct its own return value, which generally omits those arguments which are not altered by the Fortran code. Thus, if we wanted to return just the object `x`, we could call `.Fortran()` as follows:

```
.Fortran("my_sin_vec", x = as.double(x), n =  
as.integer(length(x)))$x
```

Spotfire S+ has many data classes that are not immediately representable in Fortran. To simplify the interface between Spotfire S+ and Fortran, the types of data that Spotfire S+ can pass to Fortran code are restricted to the following S-PLUS classes: "single", "integer", "numeric", "complex", "logical", and "character". The following table shows the correspondence between S-PLUS classes and Fortran types.

S-PLUS classes	Corresponding FORTRAN type
"logical"	LOGICAL
"integer"	INTEGER
"single"	REAL
"numeric"	DOUBLE PRECISION
"complex"	DOUBLE COMPLEX
"character"	CHARACTER(*)

Warnings

Spotfire S+ will not pass arrays of character strings to Fortran routines; only the first element.

The Fortran type DOUBLE COMPLEX (or COMPLEX*16) is a complex number made of double precision parts; it may not be available with all Fortran compilers.

WRITING C AND FORTRAN ROUTINES SUITABLE FOR USE IN SPOTFIRE S+ FOR UNIX

If you have a routine for which some of the arguments are not pointers, or which returns a value, you must write a *wrapper* routine which passes all data via pointer arguments, does not return a value, and calls the routine of interest. For example, we might have a sine function routine written in C and declared as follows:

```
double sin(double x)
```

You cannot call this via the .C interface, because it both takes a double-precision argument by value and returns a value. You must write a Spotfire S+-compatible wrapper for `sin()` as follows, and then load both procedures:

```
extern double sin() ;
void my_sin (double *x)
{
    *x = sin(*x) ;
}
```

Since `sin()` does not take a vector argument, you probably want to use the wrapper function to provide a vectorized form of it:

```
#include <S.h>
#include <math.h> /* to declare extern double sin() */
void my_sin_vec(double *x, long *n)
{
    long i ;
    for (i=0 ; i < *n ; i++)
        x[i] = sin(x[i]) ;
}
```

(To work along with the following section, you might want to save the above vectorized code in a file **mysin.c** in an existing Spotfire S+ chapter.)

COMPILING AND DYNAMICALLY LINKING YOUR CODE (UNIX)

S-PLUS 5.x and later abandon the old Spotfire S+ techniques of static and dynamic loading in favor of a dynamic linking process using shared objects or libraries (**.so** files). In practice, this new process is generally more convenient than the old, and involves only four steps:

1. Create your C or Fortran code, as in **mysin.c**, in a Spotfire S+ chapter.
2. Call the Spotfire S+ **CHAPTER** utility with your code files as parameters.

CHAPTER creates a makefile for use with the **make** utility. This makefile sets flags appropriate for code to be used with Spotfire S+.

3. Call the **make** utility as a Spotfire S+ utility.

The **make** utility compiles the code according to the rules specified in the makefile, and links the code into a shared object, by default named **S.so**.

4. Attach the chapter in a Spotfire S+ session (for example, by starting Spotfire S+ from the chapter). Spotfire S+ automatically opens the file **S.so**, if it exists, and loads all the symbols contained therein into the Spotfire S+ load table, so your C and Fortran routines are instantly available for your use.

You can, of course, perform at least steps 2 and 3 from within Spotfire S+. In that case, you don't need to stop your Spotfire S+ session and restart; instead, you can use the `dyn.open` function to open your newly compiled **S.so** file. (If you've just recompiled an existing **S.so**, you should use `dyn.close` to close the shared object before calling `dyn.open` to reopen it.)

You've seen several examples of writing C code for use with Spotfire S+; now let's take a closer look at steps 2–4.

Using the **CHAPTER** Utility with Source Code

You've probably used the **CHAPTER** utility often to create new Spotfire S+ work directories or projects. You may have also used it to add help files for your own S-PLUS functions to the system help. Using **CHAPTER** with source code is similar to using it with help files.

CHAPTER creates a makefile for all the source files (C, C++, Fortran, Ratfor [a structured form of Fortran], Spotfire S+, or SGML help) found in the specified chapter, with rules for turning your source code into a shared object that Spotfire S+ can use. The makefile includes compiler flags compatible with those used by the code already in Spotfire S+, and on various platforms may include flags that, for example, specify the way Fortran character strings are passed or specify which memory model is used.

For example, suppose you have the file **mysin.c** shown in the previous section. You can create a makefile including this file (or modify an existing makefile) by calling **CHAPTER** as follows from the directory containing **mysin.c**:

Splus CHAPTER mysin.c

If you've created **mysin.c** in a new directory, **CHAPTER** will both create the appropriate makefile and initialize the directory as a valid Spotfire S+ chapter. If you add **mysin.c** to an existing Spotfire S+ chapter, **CHAPTER** will leave the previously initialized database alone, and only create or modify the makefile. To use the resulting makefile, your system must have the appropriate compiler (C, C++, and/or Fortran) and libraries.

You can, if you need to, modify the makefile created by **CHAPTER**. Below is the makefile created by the above call to **CHAPTER**:

```
# makefile for local CHAPTER
SHELL=/bin/sh

SRC= mysin.c
OBJ= mysin.o
FUN=
HELPSGML=

# Use LOCAL_CFLAGS to add arguments for the C compiler
LOCAL_CFLAGS=
# Use LOCAL_CXXFLAGS to add arguments for the C++ compiler
LOCAL_CXXFLAGS=
# Use LOCAL_FFLAGS to add arguments for the FORTRAN
# compiler
LOCAL_FFLAGS=

# Use LOCAL_LIBS to add arguments or additional libraries
# to the linker
# LOCAL_LIBS="-lf2c"
LOCAL_LIBS=
```

```
include $(SHOME)/library/S_FLAGS

all: install.funs S.so install.help

install.funs: $(FUN)
    @if [ X$(FUN) != X ] ; then \
    cat $(FUN) | $(SHOME)/cmd/Splus ; \
    fi

install.help: $(HELPSGML)
    @if [ X$(HELPSGML) != X ] ; then \
    $(SHOME)/cmd/Splus HINSTALL ../Data $(HELPSGML) ; \
    $(SHOME)/cmd/Splus BUILD_JHELP ; \
    fi

S.so: $(OBJ)
    @if [ X$(OBJ) != X ]; then \
    $(SHOME)/cmd/Splus LIBRARY S.so $(OBJ) \
    $(LOCAL_LIBS) ; \
    fi

dump:
    @if test -d ../Data; then Splus dumpChapter $(SRC); \
    fi

boot:
    @if test -s all.Sdata; \
    then (BOOTING_S="TRUE" export BOOTING_S; echo \
    "terminate(should have been booting S)" | \
    $(SHOME)/cmd/Splus); \
    fi

clean:
    -rm $(OBJ)
```

This makefile includes three primary targets: `install.funs`, `S.so`, and `install.help`. For the purposes of this chapter, the most important is `S.so`, which causes your code to be compiled and linked into the shared object **S.so**. Do not attempt to modify `$(SHOME)/library/S_FLAGS`; this will probably make your code incompatible with Spotfire S+.

Compiling Your Code

With the makefile created by **CHAPTER**, compiling your code is simple: just run the **make** command as a Spotfire S+ utility as follows:

Splus make

The “Splus” in front of make allows Spotfire S+ to set its environment variables appropriately before calling the standard make utility; in particular it defines the **SHOME** environment variable used in the makefile.

The **make** utility executes the necessary commands to compile your code into the shared object **S.so**.

Dynamically Linking Your Code

Whenever you attach a Spotfire S+ chapter containing a shared object **S.so**, including whenever you start up Spotfire S+ in such a chapter, the shared object is opened and the code it contains is loaded into Spotfire S+.

You can open shared objects without attaching a chapter by using the `dyn.open` function. For example, if your colleague Fred has C code you want to access from your own set of S-PLUS functions, you might open his **S.so** shared object as follows:

```
> dyn.open("/users/fred/mysplus/S.so")
```

You can close previously opened shared objects using `dyn.close`:

```
> dyn.close("/users/fred/mysplus/S.so")
```

COMMON CONCERNS IN WRITING C AND FORTRAN CODE FOR USE WITH SPOTFIRE S+ FOR UNIX

While the actual calls to `.C()` and `.Fortran()` are straightforward, you may encounter problems loading new compiled code into Spotfire S+ and we will discuss some common problems. We also describe some C procedures and macros which you may use to write more portable code, to generate random numbers from C code, to call S-PLUS functions from your C code, to report errors, to allocate memory, and to call Fortran procedures from C code.

In order to have access in C to most functions and macros described below, you will have to include the header file **S.h** in your source files:

```
#include <S.h>
```

and make sure that you specify the `$$HOME/include` include directory in your compiler directives. That directory is specified automatically by the **makefile** created by the **CHAPTER** utility (as part of `$$HOME/library/S_FLAGS`).

The file **S.h** has changed significantly since S-PLUS 5.1; some variables have been renamed, some routines which previously had headers in **S.h** have had their headers moved elsewhere. In general, these changes affected only variables and routines which were undocumented. A new variable, `S_COMPATIBILITY`, allows you to compile code that uses some of the redefined variables. If you define `S_COMPATIBILITY` (before including **S.h**) as follows:

```
#define S_COMPATIBILITY 1
```

you obtain the old definitions of the following variables:

TRUE, FALSE, and MAYBE (now `S_TRUE`, `S_FALSE`, and `S_MAYBE`)
complex (now `s_complex`)
NULL_MODE (now `S_MODE_NULL`)
LGL, INT, REAL, DOUBLE, CHAR, LIST, COMPLEX, RAW
ANY, STRUCTURE, MAX_ATOMIC, `atomic_type` (now `S_MODE_LGL`, etc.)

Defining `S_COMPATIBILITY` as 10 instead of 1 adds the following old definitions:

vector, boolean, `void_fun`, `fun_ptr` (now `s_object`, `s_boolean`, `s_void_fun`, and `s_fun_ptr`, respectively)

We recommend that you migrate any code that uses the old variable names to use the new names, because of potential conflicts with other applications, particularly under the Windows operating systems.

If you have code that still won't compile after defining `S_COMPATIBILITY` because of routines with missing headers, you can try including the header file `S_engine.h` instead of `S.h`.

Warning:

The routines with headers in `S_engine.h` are used and needed by Spotfire S+, but they are NOT RECOMMENDED for use by programmers outside TIBCO Software Inc. Such use is not supported, and may fail with catastrophic results. All risks associated with such unsupported use are the programmer's responsibility.

Handling IEEE Special Values

Spotfire S+ handles IEEE special values such as NaN, Inf, or -Inf, for all supported numeric classes (integer, single or numeric). NaN represents the number you obtain when you divide 0 by 0. Inf represents the number you obtain when you divide 1 by 0. -Inf represents the number you obtain when you divide -1 by 0. In addition, Spotfire S+ supports NA, which represents a missing value, i.e., a value to use when none is available. Spotfire S+ functions attempt to properly handle computations when missing values are present in the data. Both NaN and NA are displayed as NA, but the data values are properly kept as different values.

The `.C()` and `.Fortran()` functions have two arguments, the `NAOK` and the `special sok` argument, that you can use to specify whether your code can handle missing values or IEEE special values (Inf and NaN), respectively. Their default value is FALSE: if any argument to `.C()` or `.Fortran()` contains an NA (or Inf or NaN), you get an error message and your code is not called. To specify these arguments, you must use their complete names, and you cannot use these names for the arguments passed to your C or Fortran code.

Warning

The `NAOK` and `special sok` arguments refer to all of the arguments to your compiled code—you can allow NA's or IEEE special values in all of the arguments or none of them. Since typically you don't want NA's for certain arguments, such as the length of a data set, you must specially check those arguments if you use `NAOK=T` (or `special sok=T`).

Dealing with IEEE special values is easily done in C as long as you use the macros described below. It is possible, yet undocumented here, to do the same in Fortran, but refer to your Fortran compiler documentation for details.

It is often simplest to remove NA's from your data in the Spotfire S+ code, but is sometimes better done in C. If you allow NA's, you should deal with them using the C macros `is_na()` and `na_set()` described below. The arguments to `.C()` and `.Fortran()` cannot contain any NA's unless the special argument `NAOK` is `T`. The following macros test for and set NA's in your C code:

```
is_na(x,mode)
na_set(x,mode)
```

The argument `x` must be a pointer to a numeric type and the argument `mode` must be one of the symbolic constants `S_MODE_LGL` (S-PLUS class "logical"), `S_MODE_INT` (S-PLUS class "integer"), `S_MODE_REAL` (S-PLUS class "single"), `S_MODE_DOUBLE`, or `S_MODE_COMPLEX`, corresponding to the type `x` points to: long, long, float, double, or `s_complex`, respectively. For example, the following C code sums a vector of double precision numbers, setting the sum to NA if any addends are NA:

```
#include <S.h>
void my_sum(double *x, long *n, double *sum) {
  long i;
  *sum = 0 ;
  for (i = 0 ; i < *n ; i++)
    if (is_na(&x[i], S_MODE_DOUBLE)) {
      na_set(sum, S_MODE_DOUBLE);
      break;
    }
  else
    *sum += x[i];
}
```

Use the following S-PLUS function to call this routine:

```
> my.sum <- function(x) .C("my_sum", as.double(x),
                           as.integer(length(x)),
                           double(1), NAOK = T)[[3]]
```

Call this from Spotfire S+ as follows:

```
> my.sum(c(1,NA,2))
[1] NA
> my.sum(1:4)
[1] 10
```

If you omit the argument `NAOK=T` in the call to `.C()`, you get the following message:

```
> my.sum2 <- function(x)
      .C("my_sum", as.double(x),
        as.integer(length(x)), double(1))[[3]]
> my.sum2(c(1,NA,2))
Problem in .C("my_sum",: subroutine my_sum: Missing values
in argument 1
Use traceback() to see the call stack
```

Warning

Both `is_na()` and `na_set()` have arguments that may be evaluated several times. Therefore don't use expressions with side effects in them, such as `na_set(&x[i++], S_MODE_DOUBLE)`. Otherwise, the side effects may occur several times. The call `is_na(x,mode)` returns 0 if `*x` is not an NA and non-zero otherwise—the non-zero value is not necessarily 1. The return value tells what sort of value `*x` is: `IS_NA` meaning a true NA and `IS_NaN` meaning an IEEE not-a-number. To assign a NaN to a value, use the alternative macro `na_set3(x,mode, type)`, where `type` is either `IS_NA` or `IS_NaN`. The macro `na_set(x,mode)` is defined as `na_set3(x,mode, IS_NA)`.

You can use the macros `is_inf(x,mode)` and `inf_set(x,mode,sign)` to deal with IEEE infinities. If you allow IEEE special values, your code should be aware that `x != x` is TRUE if `x` is a NaN. In any case you should be aware that on machines supporting IEEE arithmetic (that includes most common workstations), `1/0` is Inf and `0/0` is NaN without any warnings given. You must set the `.C()` argument `specialok` to T if you want to let Spotfire S+ pass NaN's or Inf's to your C code. The call `is_inf(x,mode)` returns 0 if `*x` is not infinite and ± 1 if `*x` is $\pm \infty$, respectively. The call `set_inf(x,mode,sign)` sets `*x` to an infinity of the given mode and sign, where the sign is specified by the integer +1 for positive infinities and -1 for negative infinities. Similarly, the call `is_nan(x,mode)` returns 0 if `*x` is not a NaN, and 1 if it is.

I/O in C Functions

File input and output is fully supported in C code called from Spotfire S+, but input and output directed to the standard streams `STDIN`, `STDOUT`, and `STDERR` require special handling. This special handling is provided by the header files `S_newio.h` and `newredef.h`, which are included automatically when you include `S.h`. This allows you, for example, to use the `printf()` function to add debugging statements to your code.

You can override the special handling by using the define `NO_NEWIO` in your code before including `S.h`. For example:

```
...  
#define NO_NEWIO  
#include <S.h>  
...
```

The special handling does not support `scanf()`; if you need to read user input from the GUI, use `fgets()` to read a line then use `sscanf()` to interpret the line.

I/O in Fortran Subroutines

Fortran users cannot use any Fortran `WRITE` or `PRINT` statements since they conflict with the I/O in Spotfire S+. Therefore, Spotfire S+ provides the following three subroutines as analogs of the S-PLUS `cat` function:

<code>DBLEPR</code>	Prints a double precision variable
<code>REALPR</code>	Prints a real variable
<code>INTPR</code>	Prints an integer variable

As an example of how to use them, here is a short Fortran subroutine for computing the net resistance of 3 resistors connected in parallel:

```
      SUBROUTINE RESIS1(R1, R2, R3, RC)  
C      COMPUTE RESISTANCES  
      RC = 1.0/(1.0/R1 + 1.0/R2 + 1.0/R3)  
      CALL REALPR('First Resistance', -1, R1,1)  
      RETURN  
      END
```

The second argument to `REALPR` specifies the number of characters in the first argument; the `-1` can be used if your Fortran compiler inserts null bytes at the end of character strings. The fourth argument is the number of values to be printed.

Here is a S-PLUS function that calls `RESIS1`:

```
> parallel.resistance<-function(r1,r2,r3) {  
  .Fortran("resis1",as.single(r1),as.single(r2),  
    as.single(r3),as.single(0))[[4]]  
}
```

Running `parallel` produces the following:

```
> parallel(25,35,75)  
First Resistance  
[1] 25  
[1] 12.2093
```

Reporting Errors and Warnings

Spotfire S+ provides two functions, `stop` and `warning`, for detecting and reporting error and warning conditions. In most cases, you should try to detect errors in your Spotfire S+ code, before calling your compiled code. However, Spotfire S+ does provide several tools to aid error reporting in your compiled code.

C Functions

The include file `S.h` defines macros that make it easy for your C code to generate error and warning messages. The `PROBLEM` and `RECOVER` macros together work like the `S-PLUS` function `stop`:

```
PROBLEM "format string", arg1, ..., argn
RECOVER(NULL_ENTRY)
```

The `PROBLEM` and `WARNING` macros together work like the `warning` function:

```
PROBLEM "format string", arg1, ..., argn
WARNING(NULL_ENTRY)
```

The odd syntax in these macros arises because they are wrappers for the C library function `sprintf()`; the `PROBLEM` macro contains the opening parenthesis and the `RECOVER` and `WARNING` macros both start with the closing parenthesis. The format string and the other arguments must be arguments suitable for the `printf()` family of functions. For example, the following C code fragment:

```
#include <S.h>
double x ;
...
if (x <= 0)
    PROBLEM "x should be positive, it is %g", x
    RECOVER(NULL_ENTRY) ;
```

is equivalent to the Spotfire S+ code:

```
if (x<=0) stop(paste("x should be positive, it is", x))
```

Both print the message and exit all of the currently active `S-PLUS` functions calls. Spotfire S+ then prompts you to try again. Similarly, the C code:

```
#include <S.h>
double x ;
...
if (x <= 0)
    PROBLEM "x should be positive, it is %g", x
    WARNING(NULL_ENTRY) ;
```

is equivalent to the Spotfire S+ code:

```
if (x<=0) warning(paste("x should be positive, it is", x))
```

Fortran Subroutines

Many of the I/O statements encountered in a typical Fortran routine arise in error handling—when the routine encounters a problem, it writes a message.

A previous section proposed using DBLEPR, REALPR, and INTPR for any necessary printing. An alternative approach in Spotfire S+ is to use the Fortran routines XERROR and XERRWV for error reporting, in place of explicit WRITE statements. For example, consider again the Fortran routine RESIS1, which computes the net resistance of 3 resistors connected in parallel. A check for division by 0 is appropriate, using XERROR:

```
      SUBROUTINE RESIS1(R1, R2, R3, RC)
C
C      COMPUTE RESISTANCES
C
      IF (R1 .EQ. 0 .OR. R2 .EQ. 0 .OR. R3 .EQ. 0) THEN
      CALL XERROR("Error : division by 0",
+      LEN("Error : division by 0"),99,2)
      RETURN
      END IF
      RC = 1.0/(1.0/R1 + 1.0/R2 + 1.0/R3)
      CALL REALPR("First Resistance", -1, R1,1)
      RETURN
      END
```

XERROR takes four arguments: a character string message, an integer giving the length of the string in message, an error number (which must be unique within the routine), and an error level. If message is a quoted string, the length-of-message argument can be given as LEN(message).

The XERRWV routine acts like XERROR but also allows you to print two integer values, two real values, or both.

The first four arguments to XERRWV, like the first four arguments to XERROR, are the message, the message length, the error ID, and the error level. The fifth and eighth arguments are integers in the range 0–2 that indicate, respectively, the number of integer values to be reported and the number of real (single precision) values to be reported. The sixth and seventh arguments hold the integer values to be reported, the ninth and tenth arguments hold the real values to be reported.

In the following call to XERRWV, the fifth argument is 1, to indicate that one integer value is to be reported. The sixth argument says that *n* is the integer to be reported:

```
      XERRWV(MSG,LMSG,1,1,1,n,0,0,0.0,0.0)
```

The following Fortran subroutine, **test.f**, shows a practical application of XERRWV:

```
subroutine test(x, n, ierr)
real*8 x(1)
integer n, ierr, LMSG
character*100 MSG
ierr = 0
if (n.lt.3) then
  MSG ="Integer (I1) should be greater than 2"
  LMSG = len("Integer (I1) should be greater than 2")
  CALL XERRWV(MSG,LMSG,1,1,1,n,0,0,0.0,0.0)
  ierr = 1
  return
endif
do 10 i = 2, n
10 x(1) = x(1) + x(i)
return
end
```

```
> .Fortran("test", as.double(1:2), length(1:2), integer(1))
[[1]]:
[1] 1 2

[[2]]:
[1] 2

[[3]]:
[1] 1
```

Warning messages:

```
1: Integer (I1) should be greater than 2 in:
   .Fortran("test", ....
2: in message above, i1=2 in:
   .Fortran("test", ....
```

The error message is duplicated because our Spotfire S+ code interprets the error status from the Fortran code. The messages issued by XERROR and XERRWV are stored in an internal message table. Spotfire S+ provides several functions for manipulating the message table within functions that call Fortran routines using XERROR and XERRWV:

<code>xerror.summary</code>	Prints out the current state of the internal message summary table. Lists the initial segment of the message, the error number, the severity level, and the repetition count for each message.
<code>xerror.clear</code>	Clears the message table. Takes an optional argument <code>doprint</code> : if <code>doprint=T</code> , the message table is printed before it is cleared.

`xerror.maxpr` Limits the number of times any one message is queued or printed. The default is 10.

For example, we can rewrite our S-PLUS test function to take advantage of these functions as follows:

```
test <- function(x)
{
  xerror.clear()
  val <- .Fortran("test",
                 as.double(x),
                 length(x),
                 integer(1))
  if(options()$warn == 0)
    xerror.summary()
  val[[1]][1]
}
```

Calling it as before (after setting the option `warn` to 0) yields the following result:

```
> test(1:2)
      error message summary
message start                nerr  level  count
Integer (I1) should be greater than 2      1      1      1
other errors not individually tabulated = 0

[1] 1
Warning messages:
1: Integer (I1) should be greater than 2 in:
  .Fortran("test", ....
2: in message above, i1 = 2 in:
  .Fortran("test", ....
```

See the `xerror` help file for more information on the S-PLUS functions used with `XERROR`, and the `XERROR` help file for more information on `XERROR` and `XERRWV`.

Calling Fortran From C

Spotfire S+ contains a few C preprocessor macros to help smooth over differences between machines in how to call C code from Fortran and vice versa. The following macros are needed to allow distinctions between the declaration, definition, and invocation of a Fortran common block or Fortran subroutine (coded in either C or Fortran):

`F77_NAME` declaration of a Fortran subroutine.
`F77_SUB` definition of a Fortran subroutine.

F77_CALL	invocation of a Fortran subroutine.
F77_COMDECL	declaration of a Fortran common block.
F77_COM	usage of a Fortran common block.

As an example of the proper use of the F77 macros, consider the following example C code fragment:

```
...
/* declaration of a common block defined in Fortran */
extern long F77_COMDECL(Forblock)[100];
...
/* declaration of a subroutine defined in Fortran */
void F77_NAME(Forfun)(double *, long *, double *);
...
/* declaration of a function defined in C, callable by
 * Fortran */
double F77_NAME(Cfun)(double *, long *);
...
/* usage of the above common block */
for (i = 0; i < 100; i++) F77_COM(Forblock)[i] = 0;
...
/* invocation of the above functions */
F77_CALL(Forfun)(s1, n1, result);
if (F77_CALL(Cfun)(s2, n2) < 0.0)
...
/* definition of the above 'callable by Fortran' function
 */
double F77_SUB(Cfun)(double *weights, long
*number_of_weights);
...

```

If you are loading code originally written for a specific UNIX compiler (including some submissions to StatLib), you may find that code does not compile correctly in Windows because not all of these macros are used. Usually, such code does not use the F77_CALL macro to invoke the functions (using F77_SUB instead), does not use the F77_COMDECL macro to declare the Fortran common block (using F77_COM instead), and leaves out the F77_NAME macro altogether. If you attempt to load such code without substituting F77_CALL for F77_SUB at the appropriate places, you get compilation errors such as the following:

```
xxx.c(54): Error! E1063: Missing operand
xxx.c(54): Warning! W111: Meaningless use of an expression
xxx.c(54): Error! E1009: Expecting ';' but found 'fortran'
```

Similarly, if you attempt to statically load code without substituting `F77_COMDECL` for `F77_COM` where appropriate, you get a link error such as the following:

```
file xxx.obj(xxx.c): undefined symbol Forblock
```

Finally, if you attempt to statically load code without using `F77_NAME` to declare the subroutine, you get a link error of the following form:

```
file xxx.obj(xxx.c): undefined symbol Cfun
```

Fortran passes all arguments by reference, so a C routine calling Fortran must pass the address of all the arguments.

Warning
Fortran character arguments are passed in many ways, depending on the Fortran compiler. It is impossible to cover up the differences with C preprocessor macros. Thus, to be portable, avoid using character and logical arguments to Fortran routines which you would like to call from C.

Calling C From Fortran

You cannot portably call C from Fortran without running the Fortran through a macro processor. You need a powerful macro processor like `m4` (even it cannot do all that is needed) and then your code doesn't look like Fortran any more.

We can give some guidelines:

- Try not to do it.
- To be portable, do not use logical or character arguments (this applies to C-to-Fortran calls as well) because C and Fortran often represent them differently.

USING C FUNCTIONS BUILT INTO SPOTFIRE S+ FOR UNIX

In the previous section, we introduced a number of routines built into Spotfire S+ with the purpose of avoiding certain difficulties in compiling code and generating useful output. This section describes some more generally useful routines that can help you allocate memory as Spotfire S+ does or generate random numbers.

Allocating Memory

Spotfire S+ includes two families of C routines for storage allocation and reallocation. You can use either of these families, or use the standard library functions `malloc()`, `calloc()`, `realloc()`, and `free()`. However, be very careful to use only *one* family for any particular allocation; mixing calls using the same pointer variable can be disastrous. The first Spotfire S+ family consists of the two routines `S_alloc()` and `S_realloc()`, which may be used instead of the standard `malloc()` and `realloc()`. The storage they allocate lasts until the current evaluation frame goes away (at the end of the function calling `.C()`) or until memory compaction in a Spotfire S+ loop reclaims it. If space cannot be allocated, `S_alloc()` and `S_realloc()` perform their own error handling; they will not return a NULL pointer. You cannot explicitly free storage allocated by `S_alloc()` and `S_realloc()`, but you are guaranteed that the storage is freed by the end of the current evaluation frame. (There is no `S_free()` function, and using `free()` to release storage allocated by `S_alloc()` will cause Spotfire S+ to crash.) `S_alloc()` and `S_realloc()` are declared a bit differently from `malloc()` and `realloc()` (although `S_alloc` has many similarities to `calloc()`—for example, it zeroes storage and has two arguments). `S_alloc()` is declared as follows in **S.h**:

```
void * S_alloc(long n, size_t size, s_evaluator
*S_evaluator);
```

Similarly, `S_realloc()` is declared as follows in **S.h**:

```
void * S_realloc(void *p, long New, long old, size_t size,
s_evaluator *S_evaluator);
```

`S_alloc()` allocates (and fills with 0's) enough space for an array of `n` items, each taking up `size` bytes. For example, the following call allocates enough space for ten doubles:

```
S_alloc(10, sizeof(double), S_evaluator)
```

`S_realloc()` takes a pointer, `p`, to space allocated by `S_alloc()` along with its original length, `old`, and `size`, `size`, and returns a pointer to space enough for `New` items of the same size. For example, the following expands the memory block size pointed to by `p` from 10 doubles to 11 doubles, zeroing the 11th double location:

```
S_realloc(p,11,10, sizeof(double), S_evaluator)
```

The contents of the original vector are copied into the beginning of the new one and the trailing new entries are filled with zeros. You must ensure that `old` and `size` were the arguments given in the call to `S_alloc()` (or a previous call to `S_realloc()`) that returned the pointer `p`. The new length should be longer than the old. As a special case, if `p` is a `NULL` pointer (in which case `old` must be `0L`), then `S_realloc()` acts just like `S_alloc()`.

The second Spotfire S+ family of allocation routines consists of the three macros `Calloc()`, `Realloc()`, and `Free()`; note the capitalization. `Calloc()` and `Realloc()` are simple wrappers for `calloc()` and `realloc()` that do their own error handling if space can not be allocated (they will not return if the corresponding wrapped function returns a `NULL` pointer). `Free()` is a simple wrapper for `free()` that sets its argument to `NULL`. As with `calloc()`, `realloc()`, and `free()`, memory remains allocated until freed—this may be before or after the end of the current frame.

Warning

If you use `malloc()` or `realloc()` directly, you must free the allocated space with `free()`. Similarly, when using `Calloc()` or `Realloc()`, you must free the allocated space with `Free()`. Otherwise, memory will build up, possibly causing Spotfire S+ to run out of memory unnecessarily. However, be aware that because S processing may be interrupted at any time (e.g., when the user hits the interrupt key or if further computations encounter an error and dump), it is sometimes difficult to guarantee that the memory allocated with `malloc()` or `realloc()` (or `Calloc()` or `Realloc()`) is freed.

Note

If, in a call to `S_alloc()`, `S_realloc()`, `Calloc()` or `Realloc()`, the requested memory allocation cannot be obtained, those routines call `RECOVER()`. See the section Reporting Errors and Warnings (page 171) for more information on the `RECOVER()` macro.

Generating Random Numbers

Spotfire S+ includes user-callable C routines for generating standard uniform and normal pseudo-random numbers. It also includes procedures to get and set the permanent copy of the random number generator's seed value. The following routines (which have no arguments) each return one pseudo-random number:

```
double unif_rand(void);
```

```
double norm_rand(void);
```

Before calling either function, you must get the permanent copy of the random seed from disk into Spotfire S+ (which converts it to a convenient internal format) by calling `seed_in((long *)NULL, S_evaluator)`. You can specify a particular seed using `setseed(long *seed)`, which is equivalent to the S-PLUS function `set.seed`. When you are finished generating random numbers, you must push the permanent copy of the random seed out to disk by calling `seed_out((long *)NULL, S_evaluator)`. If you do not call `seed_in()` before the random number generators, they fail with an error message. If you do not call `seed_out()` after a series of calls to `unif_rand()` or `norm_rand()`, the next call to `seed_in()` retrieves the same seed as the last call and you get the same sequence of random numbers again. The seed manipulation routines take some time so we recommend calling `seed_in()` once, then calling `unif_rand()` or `norm_rand()` as many times as you wish, then calling `seed_out()` before returning from your C function. A simple C function to calculate a vector of standard normals is implemented as follows:

```
#include <S.h>
void my_norm(double *x, long *n_p) {
    long i, n = *n_p ;
    seed_in( (long *) NULL, S_evaluator);
    for (i=0 ; i<n ; i++)
        x[ i ] = norm_rand(S_evaluator);
    seed_out( (long *) NULL, S_evaluator);
}
```

To call it from Spotfire S+, define the function `my.norm` as follows:

```
my.norm <- function(n)
    .C("my_norm", double(n), as.integer(n))[[1]]
```

Of course it is simpler and safer to use the S-PLUS function `rnorm` to generate a fixed number of normal variates to pass into an analysis function. We recommend that you generate the random variates in C code only when you cannot tell how many random variates you will need, as when using a rejection method of generating non-uniform random numbers.

CALLING S-PLUS FUNCTIONS FROM C CODE (UNIX)

To this point, we have shown how to call C and Fortran routines from S-PLUS functions. You can also call S-PLUS functions from C code, using the supplied C routine `call_S()`. The `call_S()` routine is useful as an interface to numerical routines which operate on C or Fortran functions, but it is not a general purpose way to call S-PLUS functions. The C routine calling `call_S()` must be loaded into Spotfire S+, the arguments to the function must be simple, and the nature of the output must be known ahead of time. Because of these restrictions, `call_S()` cannot be used to call S-PLUS functions from an independent C application, as you might call functions from a subroutine library.

The C function `call_S()` calls a S-PLUS function from C, but `call_S()` must be called by C code called from Spotfire S+ via `.C()`. The `call_S()` function has the following calling sequence:

```
call_S(void *func, long nargs, void **arguments,  
       char **modes, long *lengths, char **names,  
       long nres, void **results);
```

where:

`func` is a pointer to a list containing one S-PLUS function. This should have been passed via an argument in a `.C` call, as follows:

```
.C("my_c_code", list(myfun))
```

This calls C code starting with the following lines:

```
my_c_code(void **Sfunc) {  
    ...  
    call_S(*Sfunc, ...);  
    ...  
}
```

The S-PLUS function must return an atomic vector or list of atomic vectors.

`nargs` is the number of arguments to give to the S-PLUS function `func`.

`arguments` is an array of `nargs` pointers to the data being passed to `func`. These can point to any atomic type of data, but must be cast to type `void*` when put into `arguments`.

`modes` is an array of `nargs` character strings giving the Spotfire S+ names, e.g., "double" or "integer", of the modes of the arguments given to `func`.

<code>lengths</code>	is an array of <code>nargs</code> longs, giving the lengths of the arguments.
<code>names</code>	is an array of <code>nargs</code> strings, giving the names to be used for the arguments in the call to <code>func</code> . If you don't want to call any arguments by name, <code>names</code> may be <code>(char **)NULL</code> ; if you don't want to call the <code>i</code> th argument by name, <code>names[i]</code> may be <code>(char *)NULL</code> .
<code>nres</code>	is the maximum number of components expected in the list returned by <code>func</code> (if <code>func</code> is expected to return an atomic vector, then <code>nres</code> should be 1).
<code>results</code>	is filled in by <code>call_S()</code> ; it contains generic pointers to the components of the list returned by <code>func</code> (or a pointer to the value returned by <code>func</code> if the value were atomic).

Your C code calling `call_S()` should cast the generic pointers to pointers to some concrete type, like `float` or `int`, before using them. If `func` returns a list with fewer components than `nres`, the extra elements of `results` are filled with `NULL`'s. Notice that `call_S()` does not report the lengths or modes of the data pointed to by `results`; you must know this a priori.

To illustrate the use of `call_S()`, we construct (in Fortran) a general purpose differential equation solver, `heun()`, to solve systems of differential equations specified by a S-PLUS function. Other common applications involve function optimization, numerical integration, and root finding.

The `heun()` routine does all its computations in single precision and expects to be given a subroutine of the following form:

```
f(t, y, dydt)
```

where the scalar `t` and vector `y` are given and the vector `dydt`, the derivative, is returned. Because the `f()` subroutine calls the S-PLUS function, it must translate the function's argument list into one that `call_S()` expects. Since not all the data needed by `call_S` can be passed into `f()` via an argument list of the required form, we must have it refer to global data items for things like the pointer to the S-PLUS function and the modes and lengths of its arguments. The following file of C code, **dfeq.c**, contains a C function `f()` to feed to the solver `heun()`. It also contains a C function `dfeq()` which initializes data that `f()` uses and then calls `heun()` (which repeatedly calls `f()`):

```
#include <S.h>
extern void F77_NAME(heun());
/* pointer to Splus function to be filled in */
```

```

static void *Sdydt ;

/* descriptions of the functions's two arguments */
static char *modes[] = {"single", "single" };
static long lengths[] = {1, 0 };
    /* neqn = lengths[1] to be filled in */
static char *names[] = { "t", "y" };

/*
    t [input]: 1 long ; y [input]: neqn long ;
    yp [output]: neqn long
*/
static void f(float *t, float *y, float *yp) {
    char *in[2] ; /* for two inputs to Splus function,
                   t and y */
    char *out[1] ; /* for one output vector of
                   Splus function */

    int i;
    in[0] = (void *)t;
    in[1] = (void *)y;
    call_S(Sdydt, 2L,
          in, modes, lengths, names, /* 2 arguments */
          1L, out/* 1 result */);

    /* the return value out must be 1 long - i.e., Splus
    function must return an atomic vector or a list of one
    atomic vector. We can check that it is at least 1 long. */

    if (!out[0])
        PROBLEM
            "Splus function returned a 0 long list"
        RECOVER(NULL_ENTRY);

    /* Assume out[0] points to lengths[1] single precision
    numbers. We cannot check this assumption here. */

    for(i=0;i<lengths[1];i++)
        yp[i] = ((float *)out[0])[i] ;
    return ;
}

/* called via .C() by the Splus function dfreq(): */
void dfreq(void **Sdydtp, float *y, long *neqn,
           float *t_start, float *t_end, float *step,
           float *work) {
    /* Store pointer to Splus function and
    number of equations */
    Sdydt = *Sdydtp ;
    lengths[1] = *neqn ;

```

```
/* call Fortran differential equation solver */
  F77_CALL(heun)(f, neqn, y, t_start, t_end, step, work);
}
```

Warning

In the C code, the value of the S-PLUS function was either atomic or was a list with at least one atomic component. To make sure there was no more than one component, you could look for 2 values in results and make sure that the second is a null pointer.

The following S-PLUS function, `dfeq`, does some of the consistency tests that our C code could not do (because `call_S` did not supply enough information about the output of the S-PLUS function). It also allocates the storage for the scratch vector. Then it repeatedly calls the C routine, `dfeq()`, to have it integrate to the next time point that we are interested in:

```
> dfeq <- function(func, y , t0 = 0, t1 = 1, nstep = 100,
  stepsize = (t1-t0)/nstep)
{
  if (length(func) != 3 ||
      any(names(func) != c("t","y", "")))
    stop("arguments of func must be called t and y")
  y <- as.single(y)
  t0 <- as.single(t0)
  neqn <- length(y)
  test.val <- func(t = t0, y = y)
  if(neqn != length(test.val))
    stop("y and func(t0,y) must be same length")
  if(storage.mode(test.val) != "single")
    stop("func must return single precision vector")
  val <- matrix(as.single(NA), nrow = nstep + 1, ncol =
neqn)
  val[1, ] <- y
  time <- as.single(t0 + seq(0, nstep) * stepsize)
  for(i in 1:nstep) {
    val[i + 1, ] <- .C("dfeq", list(func), y=val[i, ],
                      neqn=as.integer(neqn),
                      t.start=as.single(time[i]),
                      t.end=as.single(time[i + 1]),
                      step=as.single(stepsize),
                      work=single(3 * neqn))$y
  }
  list(time=time, y=val)
}
```

The following subroutine is the Fortran code, `heun.f`, for Heun's method of numerically solving a differential equation. It is a first order Runge-Kutta method. Production quality differential equation solvers let you specify a desired local accuracy rather than step size, but the code that follows does not:

```

C      Heun's method for solving dy/dt=f(t,y),
C      using step size h :
C      k1 = h f(t,y)
C      k2 = h f(t+h,y+k1)
C      ynext = y + (k1+k2)/2

      subroutine heun(f, neqn, y, tstart, tend, step, work)
      integer neqn
      real*4 f, y(neqn), tstart, tend, step, work(neqn,3)
C work(1,1) is k1, work(1,2) is k2, work(1,3) is y+k1
      integer i, nstep, istep
      real*4 t
      external f
      nstep = max((tend - tstart) / step, 1.0)
      step = (tend - tstart) / nstep
      do 30 istep = 1, nstep
         t = tstart + (istep-1)*step
         call f(t, y, work(1,1))
         do 10 i = 1, neqn
            work(i,1) = step * work(i,1)
            work(i,3) = y(i) + work(i,1)
10        continue
         call f(t+step, work(1,3), work(1,2))
         do 20 i = 1, neqn
            work(i,2) = step * work(i,2)
            y(i) = y(i) + 0.5 * (work(i,1) + work(i,2))
20        continue
30      continue
      return
      end

```

To try out this example of `call_S`, exercise it on a simple one-dimensional problem as follows:

```

> a <- dfdq(function(t,y)t^2, t0=0, t1=10, y=1)
> plot(a$time,a$y)
> lines(a$time, a$time^3/3+1) # compare to
                             #theoretical solution

```

You can increase `nstep` to see how decreasing the step size increases the accuracy of the solution. The local error should be proportional to the square of the step size and when you change the number of steps from 100 to 500 (over the same time span) the error does go down by a factor of about 25. An interesting three-dimensional example is the Lorenz equations, which have a strange attractor:

```
> chaos.func<-function(t, y) {
  as.single(c(10 * (y[2] - y[1]),
             - y[1] * y[3] + 28 * y[1] - y[2],
             y[1] * y[2] - 8/3 * y[3]))
}
> b <- dfeq(chaos.func, y=c(5,7,19), t0=1, t1=10,
           nstep=300)
> b.df <- data.frame(b$time,b$y)
> pairs(b.df)
```

The resulting plot is shown in Figure 5.2.

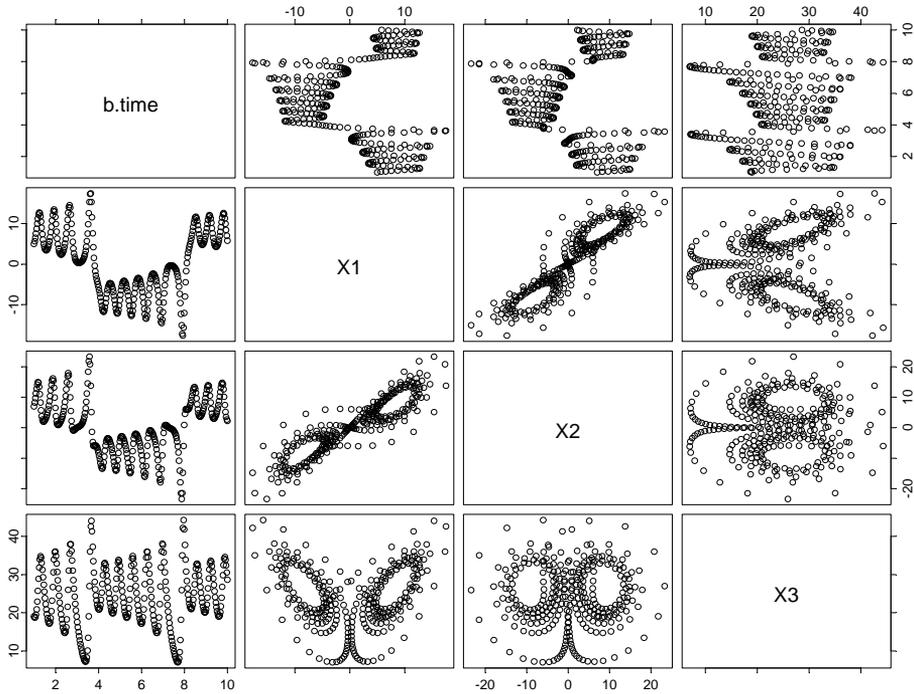


Figure 5.2: Viewing the Lorenz equations, as solved by dfeq.

Warnings

Since `call_S` doesn't describe the output of the S-PLUS function it calls, you must "know" about it ahead of time. You can test the function for a variety of values before calling `call_S` to check for gross errors, but you cannot ensure that the function won't return an unacceptable value for certain values of its arguments.

The `call_S` function expects that the output of the function given to it has no attributes. If it does have attributes, such as dimensions or names, they are stripped.

THE .CALL INTERFACE (UNIX)

The .Call interface is a powerful, yet dangerous, interface that allows you to manipulate S-PLUS objects from C code. It is more efficient than the standard .C interface, but because it allows you to work directly with S-PLUS objects, without the usual Spotfire S+ protection mechanisms, it also allows you to create a variety of bugs, including memory faults and corrupted data.

The .Call interface provides you with several capabilities the standard .C interface lacks, including the following

- the ability to create variable-length output variables, as opposed to the preallocated objects the .C interface expects to write to.
- a simpler mechanism for evaluating S-PLUS expressions within C.
- the ability to establish direct correspondence between C pointers and S-PLUS objects.

Requirements

To use the .Call interface, you must ensure your code meets the following requirements¹:

1. The return value and all arguments have C type "s_object *".
2. The code must include the standard Spotfire S+ header file **S.h**.
3. If the routine deals with S-PLUS objects, it must include a declaration of the evaluator using the macro S_EVALUATOR, appearing in the declaration part of the routine and *not* followed by a semicolon.

As with .C, the required arguments to .Call include the name of the C routine being called and one argument for each argument to the C routine.

1. Chambers, J.M. (1998) *Programming with Data*. New York: Springer-Verlag. p. 429.

Returning Variable-Length Output Vectors

Occasionally, we do not know how long the output vector of a procedure is until we have done quite a bit of processing of the data. For example, we might want to read all the data in a file and produce a summary of each line. Until we have counted the lines in the file, we don't know how much space to allocate for a summary vector. Generally, .C passes your C procedure a pointer to a data vector allocated by your S-PLUS function so you must know the length ahead of time. You could write two C procedures: one to examine the data to see how much output there is and one to create the output. Then you could call the first in one call to .C, allocate the correct amount of space, and call the second in another call to .C. The first could even allocate space for the output vector as it is processing the input and have the second simply copy that to the vector allocated by your S-PLUS function.

With the .Call interface, however, you can create the desired S-PLUS object directly from your C code.

Here is an example which takes a vector x of integers and returns a sequence of integers, of length $\max(x)$:

```
#include "S.h"
s_object *makeseq(s_object *sobjX)
{
    S_EVALUATOR
    long i, n, xmax, *seq, *x ;
    s_object *sobjSeq ;

    /* Convert the s_objects into C data types: */
    sobjX = AS_INTEGER(sobjX) ;
    x = INTEGER_POINTER(sobjX) ;
    n = GET_LENGTH(sobjX) ;

    /* Compute max value: */
    xmax = x[0] ;
    if(n > 1) {
        for(i=1; i<n; i++) {
            if(xmax < x[i]) xmax = x[i] ;
        }
    }

    if(xmax < 0)
        PROBLEM "The maximum value (%ld) is
negative.", xmax ERROR ;
}
```

```

/* Create a new s_object, set its length and get a C integer
pointer to it */
    sobjSeq = NEW_INTEGER(0) ;
    SET_LENGTH(sobjSeq, xmax) ;
    seq = INTEGER_POINTER(sobjSeq) ;

    for(i=0; i<xmax; i++) {
        seq[i] = i + 1 ;
    }

    return(sobjSeq) ;
}

```

Use the following Spotfire S+ code to call `makeseq()`:

```

"makeseq" <-
function(x)
{
    x <- as.integer(x)
    .Call("makeseq", x)
}

```

S Object Macros

The `makeseq` example has several interesting features, but perhaps the most useful is its extensive use of S object macros. These macros are defined when you include `S.h`, and allow you to create, modify, and manipulate actual S-PLUS structures from within your C code. There are five basic macros, each of which is implemented particularly for the basic data types listed in Table 5.5. These macros are described in Table 5.6. To obtain the full name

Table 5.6: *S object macros.*

Macro	Description
<code>NEW_type(n)</code>	Create a pointer to an S object of class <i>type</i> and length <i>n</i> .
<code>AS_type(obj)</code>	Coerce <i>obj</i> to an S object of class <i>type</i> .
<code>IS_type(obj)</code>	Test whether <i>obj</i> is an S object of class <i>type</i> .

Table 5.6: *S* object macros.

Macro	Description
<code>type_POINTER(obj)</code>	Create a pointer of type <i>type</i> to the data part of <code>obj</code> .
<code>type_VALUE(obj)</code>	Returns the value of <code>obj</code> , which should have length 1.

of the desired macro, just substitute the basic data type from Table 5.5 in ALLCAPS for the word *type* in the macro name given in Table 5.6. Thus, to create a new numeric S-PLUS object, use the macro `NEW_NUMERIC`.

The `makeseq` code uses the `AS_INTEGER` macro to coerce the `objX` object to type `INTEGER`; the `NEW_INTEGER` macro to create the returned sequence object; and the `INTEGER_POINTER` macro to access the data within those objects.

The `makeseq` code also uses built-in macros for getting and setting basic information about the *S* objects: in addition to the `GET_LENGTH` and `SET_LENGTH` macros used in `makeseq`, there are also `GET_CLASS` and `SET_CLASS` macros to allow you to obtain class information about the various *S* objects passed into your code.

Evaluating S-PLUS Expressions from C

You can evaluate a S-PLUS expression from C using the macros `EVAL` and `EVAL_IN_FRAME`. Both take as their first argument a S-PLUS object representing the expression to be evaluated; `EVAL_IN_FRAME` takes a second argument, `n`, representing the Spotfire S+ frame in which the evaluation is to take place.

For example, consider the internal C code for the `lapply` function, which was first implemented by John Chambers in his book *Programming with Data*:

```
#include "S_engine.h"
/* See Green Book (Programing with Data by J.M. Chambers)
appendix A-2 */

s_object *
S_qapply(s_object *x, s_object *expr, s_object *name_obj,
         s_object *frame_obj)
{
    S_EVALUATOR
```

```
long frame, n, i;
char *name;
s_object **els;
x = AS_LIST(x) ;
els = LIST_POINTER(x);
n = LENGTH(x);
frame = INTEGER_VALUE(frame_obj) ;
name = CHARACTER_VALUE(name_obj) ;
for(i=0;i<n;i++) {
    ASSIGN_IN_FRAME(name, els[i], frame) ;
    SET_ELEMENT(x, i, EVAL_IN_FRAME(expr,
        frame)) ;
}
return x;
}
```

This uses the more general macro `EVAL_IN_FRAME` to specify the specific frame in which to evaluate the specified expression. Note also the `SET_ELEMENT` macro; this must *always* be used to perform assignments into S-PLUS list-like objects from C.

DEBUGGING LOADED CODE (UNIX)

Frequently the code you are dynamically linking is known, tested, and reliable. But what if you are writing new code, perhaps as a more efficient engine for a routine developed in Spotfire S+? You may well need to debug both the C or Fortran code and the S-PLUS function that calls it. The first step in debugging C and Fortran routines for use in Spotfire S+ is to make sure that the C function or Fortran subroutine is of the proper form, so that all data transfer from Spotfire S+ to C or Fortran occurs through arguments. Both the input from Spotfire S+ and the expected output need to be arguments to the C or Fortran code. The next step is to ensure that the classes of all variables are consistent. This often requires that you add a call such as `as.single(variable)` in the call to `.C` or `.Fortran`. If the Spotfire S+ code and the compiled code disagree on the number, classes, or lengths of the argument vectors, Spotfire S+'s internal data may be corrupted and it will probably crash—by using `.C` or `.Fortran` you are trading the speed of compiled code for the safety of Spotfire S+ code. In this case, you usually get an application error message before your Spotfire S+ session crashes. Once you've verified that your use of the interface is correct, and you've determined there's a problem in the C or Fortran code, you can use an analog of the `cat` statement to trace the evaluation of your routine.

Debugging C Code

If you are a C user, you can use C I/O routines, provided you include `S.h`. Thus, you can casually sprinkle `printf` statements through your C code just as you would use `cat` or `print` statements within a S-PLUS function. (If your code is causing Spotfire S+ to crash, call `fflush()` after each call to `printf()` to force the output to be printed immediately.)

Debugging C Code Using a Wrapper Function

If you cannot uncover the problem with generous use of `printf()`, the following function, `.Cdebug`, (a wrapper function for `.C`) can sometimes find cases where your compiled code writes off the end of an argument vector. It extends the length of every argument given to it and fills in the space with a flag value. Then it runs `.C` and checks that the flag values have not been changed. If any have been changed, it prints a description of the problem. Finally, it shortens the arguments down to their original size so its value is the same as the value of the corresponding `.C` call.

```
.Cdebug <- function(NAME, ..., NAOK = F, specialsok = F,
  ADD = 500, ADD.VALUE = -666)
{
  args <- list(...)
```

```

tail <- rep(as.integer(ADD.VALUE), ADD)
for(i in seq(along = args))
{
  tmp <- tail
  storage.mode(tmp) <- storage.mode(args[[i]])
  args[[i]] <- c(args[[i]], tmp)
}
args <- c(NAME = NAME, args, NAOK = NAOK,
         specialsok = specialsok)
val <- do.call(".C", args)
for(i in seq(along = val))
{
  tmp <- tail
  storage.mode(tmp) <- storage.mode(args[[i]])
  taili <- val[[i]][seq(to = length(val[[i]]),
                      length = ADD)]
  if((s <- sum(taili != tmp)) > 0) {
    cat("Argument ", i, "(", names(val)[i],
        ") to ", NAME, " has ", s, " altered
        values after end of array\n ",
        sep = "")
  }
  length(val[[i]]) <- length(val[[i]]) - ADD
}
val
}

```

For example, consider the following C procedure, `oops()`:

```

oops(double *x, long* n)
{
  long i;
  for (i=0 ; i <= *n ; i++) /* should be <, not <= */
    x[i] = x[i] + 10 ;
}

```

Because of the misused `<=`, this function runs off the end of the array `x`. If you call `oops()` using `.C` as follows, you crash your Spotfire S+ session:

```
> .C("oops", x=as.double(1:66), n=as.integer(66))
```

If you use `.Cdebug` instead, you get some information about the problem:

```

> .Cdebug("oops", x=as.double(1:66), n=as.integer(66))
Argument 1(x) to oops has 1 altered values after end of
array
x:

```

```
[1] 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28  
[19] 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46  
[37] 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64  
[55] 65 66 67 68 69 70 71 72 73 74 75 76  
n:  
[1] 66
```

The `.Cdebug` function cannot tell when you run off the beginning of an argument vector or when you write anywhere else in memory. If inspecting your source code and using S-PLUS functions like `.Cdebug` is not enough to pinpoint a problem, try the following:

1. Write a short main program that calls your procedure.
2. Compile and link the main program and your procedure for debugging.

A NOTE ON STATLIB (WINDOWS AND UNIX)

StatLib is a system for distributing statistical software, data sets, and information by electronic mail, FTP, and the World Wide Web. It contains a wealth of user-contributed S-PLUS functions, many of which rely upon C and Fortran code that is also provided. Much of this code has been precompiled for use with Spotfire S+ for Windows.

- To access StatLib by FTP, open a connection to: **lib.stat.cmu.edu**. Login as **anonymous** and send your e-mail address as your password. The FAQ (frequently asked questions) is in **/S/FAQ**, or in HTML format at **<http://www.stat.math.ethz.ch/S-FAQ>**.
- To access StatLib with a web browser, visit **<http://lib.stat.cmu.edu/>**.
- To access StatLib by e-mail, send the message: **send index** from S to **statlib@lib.stat.cmu.edu**. You can then request any item in StatLib with the request **send item from S** where item is the name of the item.

If you find a module you want, check to see if it is pure S code or if it requires C or Fortran code. If it does require C or Fortran code, see if there is a precompiled Windows version-look in the **/DOS/S** directories. The precompiled versions generally require you to do nothing more than install the code.

Introduction	198
Using Spotfire S+ as an Automation Server	199
A Simple Example	199
Exposing Objects to Client Applications	205
Exploring Properties and Methods	207
Programming With Object Methods	209
Programming With Object Properties	219
Passing Data to Functions	220
Automating Embedded Spotfire S+ Graph Sheets	223
Using Spotfire S+ as an Automation Client	224
A Simple Example	224
High-Level Automation Functions	230
Reference Counting Issues	232
Automation Examples	235
Server Examples	235
Client Examples	239

INTRODUCTION

Automation, formerly known as OLE automation, makes it possible for one application, known as the automation client, to directly access the objects and functionality of another application, the automation server. The server application *exposes* its functionality through a type library of objects, properties, and methods, which can then be manipulated programmatically by a client application. Automation thus provides a handy way for programs and applications to share their functionality.

In this chapter, we explore the procedures for using Spotfire S+ as both an automation server and an automation client. We begin by showing you how to expose S-PLUS objects and functions and how to use them as building blocks in the program code of client applications. Later in the chapter, we examine the functions provided in the S-PLUS programming language for accessing and manipulating the automation objects exposed by server applications.

Note

This chapter is dedicated to Microsoft Windows[®] users running the Spotfire S+ GUI. You must call all automation client functions from the Spotfire S+ GUI. If you call the automation functions from the Console program, they generate errors.

USING SPOTFIRE S+ AS AN AUTOMATION SERVER

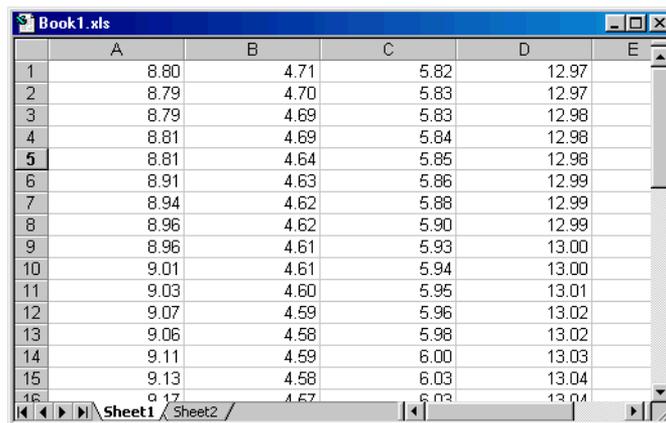
Programs and applications supporting automation client features can access all the functionality of Spotfire S+ by referring to the S-PLUS type library or the HTML-based object help system. The type library is a disk file containing information about S-PLUS objects and functions, as well as help and syntax information. The object help system is a set of HTML files with an **index.htm** showing the object hierarchy of objects exposed by Spotfire S+ via automation and how to use them in a Visual Basic script.

Before explaining in detail how to program with S-PLUS automation objects, let's first take a look at a simple example.

A Simple Example

To demonstrate how to use Spotfire S+ as an automation server, we present a simple example using automation to pass data from an Excel worksheet to Spotfire S+, which then performs a covariance estimation on the data and returns the resulting covariance matrix to Excel.

Consider the sample data shown in Figure 6.1. **Sheet1** of the Excel workbook **Book1.xls** contains data in 4 columns of 39 rows (not all rows are shown in Figure 6.1).



	A	B	C	D	E
1	8.80	4.71	5.82	12.97	
2	8.79	4.70	5.83	12.97	
3	8.79	4.69	5.83	12.98	
4	8.81	4.69	5.84	12.98	
5	8.81	4.64	5.85	12.98	
6	8.91	4.63	5.86	12.99	
7	8.94	4.62	5.88	12.99	
8	8.96	4.62	5.90	12.99	
9	8.96	4.61	5.93	13.00	
10	9.01	4.61	5.94	13.00	
11	9.03	4.60	5.95	13.01	
12	9.07	4.59	5.96	13.02	
13	9.06	4.58	5.98	13.02	
14	9.11	4.59	6.00	13.03	
15	9.13	4.58	6.03	13.04	
16	9.17	4.57	6.03	13.04	

Figure 6.1: Sample data in *Book1.xls*.

Note

The sample data in **Book1.xls** are taken from the **freeny.x** matrix included with Spotfire S+. You can recreate this example by exporting the data into a new Excel worksheet and following the steps outlined below.

By writing a program in Visual Basic for Applications (Office 97), we can automate a conversation between Excel and Spotfire S+ to perform our task. The complete code for one such program is shown in Figure 6.2.

```

Sub RunAutomationCOV()
    Dim pDataValues As Variant
    ConvertSheetRangeToArray pDataValues, Sheets("Sheet1").Range("A1:D39")
    Dim automationCOV As Object
    Set automationCOV = CreateObject("S-PLUS.automationCOV")
    automationCOV.x = pDataValues
    automationCOV.Run
    Dim pCovarianceDataFrame As Object
    Set pCovarianceDataFrame = GetDataFrame("CovDF")
    ConvertArrayToSheetRange "A", "1", "Sheet2", _
        pCovarianceDataFrame.DataAsArray
End Sub

Sub ConvertSheetRangeToArray(pArray As Variant, _
    ByRef rangeToConvert As Range)
    pArray = rangeToConvert.Value
End Sub

Function GetDataFrame(sDataFrameName As String) As Object
    Dim pApp As Object
    Set pApp = CreateObject("S-PLUS.Application")
    Set GetDataFrame = pApp.GetObject("DataFrame", sDataFrameName)
End Function

Sub ConvertArrayToSheetRange(sStartCol As String, sStartRow As String, _
    sSheetName As String, pArray As Variant)
    UpperBound1 = UBound(pArray, 1)
    UpperBound2 = UBound(pArray, 2)
    sRangeToFill = Trim$(sStartCol) + Trim$(sStartRow) + ":" + _
        Chr(Asc(UCase$(sStartCol)) + (Trim(Str(UpperBound2)) - 1)) + _
        Trim(Str(UpperBound1))
    Sheets(sSheetName).Range(sRangeToFill).Value = pArray
End Sub

```

Figure 6.2: Complete code for our VBA program.

Hint

The example shown in Figure 6.2 can be found in **samples/oleauto/vba/excel/Book1.xls** in the Spotfire S+ program folder.

Before we examine the VBA code in detail, let's first define a new S-PLUS function and register it for use as an automation object.

1. Open a new **Script** window in Spotfire S+ and enter the code shown in Figure 6.3.

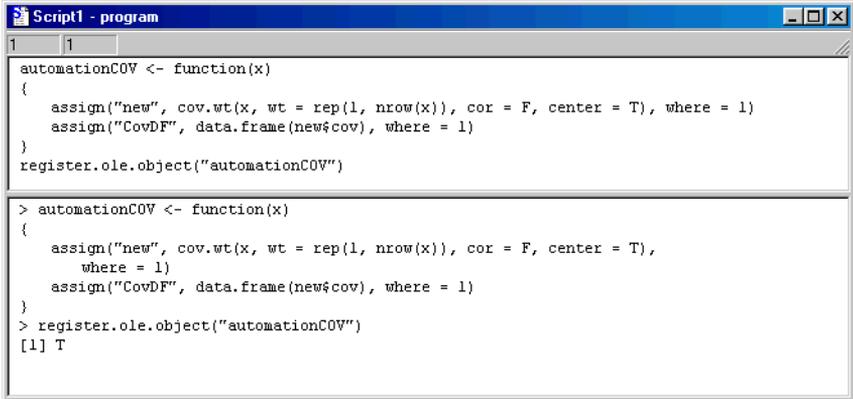
A screenshot of a Spotfire S+ Script window titled "Script1 - program". The window has a standard Windows-style title bar with minimize, maximize, and close buttons. Below the title bar is a toolbar with a "Run" button (a right-pointing triangle) and a "Script" button. The main area of the window is a code editor with a light gray background. The code is as follows:

```
1 1
automationCOV <- function(x)
{
  assign("new", cov.wt(x, wt = rep(1, nrow(x)), cor = F, center = T), where = 1)
  assign("CovDF", data.frame(new$cov), where = 1)
}
register.ole.object("automationCOV")
```

Figure 6.3: *Defining and registering a new S-PLUS function.*

Our new function, `automationCOV`, calls the built-in S-PLUS function `cov.wt` to perform a weighted covariance estimation on the data received from Excel and extracts the `cov` component of the resulting list for return to Excel. After defining the new function, we use the `register.ole.object` command to make it available to Excel.

2. Click the **Run** button  on the **Script** window toolbar. As shown in Figure 6.4, `automationCOV` is now defined and registered as an automation object.



```

Script1 - program
1 1
automationCOV <- function(x)
{
  assign("new", cov.wt(x, wt = rep(1, nrow(x)), cor = F, center = T), where = 1)
  assign("CovDF", data.frame(new$cov), where = 1)
}
register.ole.object("automationCOV")

> automationCOV <- function(x)
{
  assign("new", cov.wt(x, wt = rep(1, nrow(x)), cor = F, center = T),
  where = 1)
  assign("CovDF", data.frame(new$cov), where = 1)
}
> register.ole.object("automationCOV")
[1] T

```

Figure 6.4: *Running the script.*

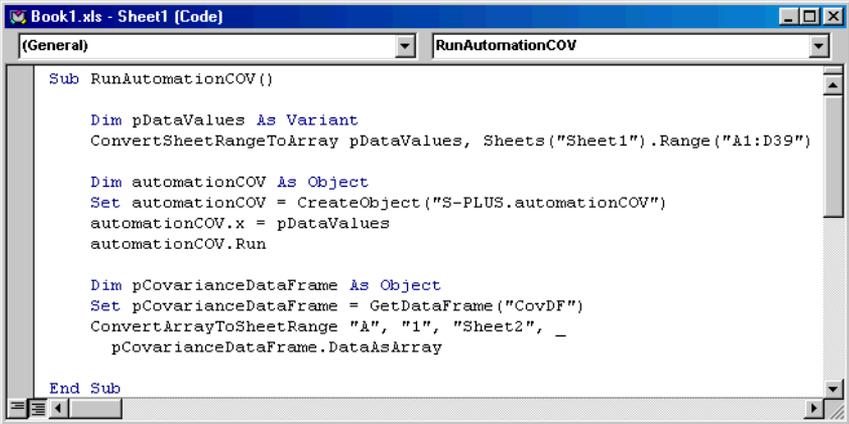
3. Close the **Script** window. At the prompt to save the script in a file, click **No**.

Note

If you prefer, you can define and register `automationCOV` directly from the **Commands** window.

Now that we have defined and registered our S-PLUS function, the next step is to write the module in Visual Basic.

4. With **Book1.xls** open in Excel, choose **Tools ► Macro ► Visual Basic Editor** from the main menu.
5. If the **Project Explorer** window is not open, open it by choosing **View ► Project Explorer**.
6. Double-click **Sheet1** under the **Book1.xls** project to open the code window for **Sheet1**.
7. Enter the code for the first procedure in the module, `RunAutomationCOV`, as shown in Figure 6.5.



```

Book1.xls - Sheet1 [Code]
(General) RunAutomationCOV

Sub RunAutomationCOV()

    Dim pDataValues As Variant
    ConvertSheetRangeToArray pDataValues, Sheets("Sheet1").Range("A1:D39")

    Dim automationCOV As Object
    Set automationCOV = CreateObject("S-PLUS.automationCOV")
    automationCOV.x = pDataValues
    automationCOV.Run

    Dim pCovarianceDataFrame As Object
    Set pCovarianceDataFrame = GetDataFrame("CovDF")
    ConvertArrayToSheetRange "A", "1", "Sheet2", _
        pCovarianceDataFrame.DataAsArray

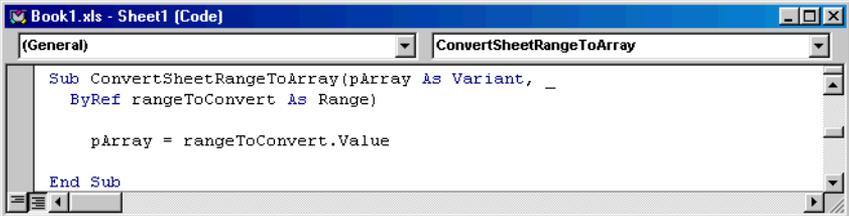
End Sub

```

Figure 6.5: *The RunAutomationCOV procedure.*

RunAutomationCOV represents the central task we want to automate. In the first section of code, we declare a variable, `pDataValues`, in which to store the data on **Sheet1**. A call to the next procedure we will write, `ConvertSheetRangeToArray`, converts the range data into an array.

8. Enter the code for `ConvertSheetRangeToArray`, as shown in Figure 6.6.



```

Book1.xls - Sheet1 [Code]
(General) ConvertSheetRangeToArray

Sub ConvertSheetRangeToArray(pArray As Variant, _
    ByRef rangeToConvert As Range)

    pArray = rangeToConvert.Value

End Sub

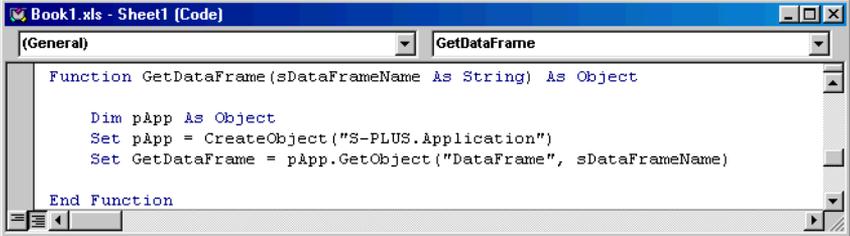
```

Figure 6.6: *The ConvertSheetRangeToArray procedure.*

In the next section of code in `RunAutomationCOV` (see Figure 6.5), we declare a variable to capture our `automationCOV` function, pass the Excel data as a parameter to the function, and then run the function.

In the final section of code in RunAutomationCOV (see Figure 6.5), we declare a variable, pCovarianceDataFrame, in which to store our results and call the GetDataFrame function, the next procedure we will write, to return the results to Excel.

9. Enter the code for GetDataFrame, as shown in Figure 6.7.



```

Function GetDataFrame (sDataFrameName As String) As Object

    Dim pApp As Object
    Set pApp = CreateObject("S-PLUS.Application")
    Set GetDataFrame = pApp.GetObject("DataFrame", sDataFrameName)

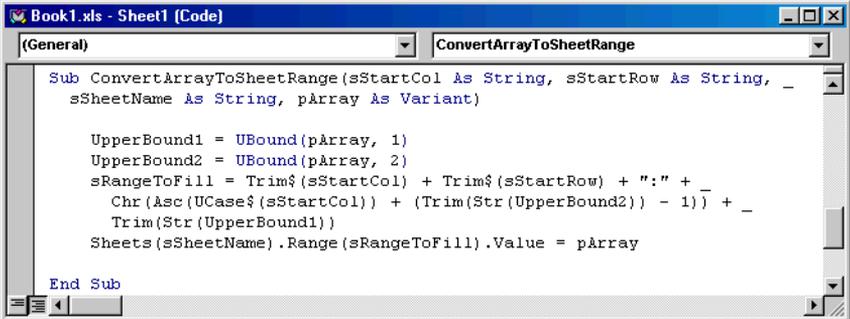
End Function

```

Figure 6.7: The GetDataFrame function.

The last procedure we will write, ConvertArrayToSheetRange, is called in the last line of RunAutomationCOV and returns the covariance matrix to **Sheet2** in **Book1.xls**.

10. Enter the code for ConvertArrayToSheetRange, as shown in Figure 6.8.



```

Sub ConvertArrayToSheetRange (sStartCol As String, sStartRow As String, _
    sSheetName As String, pArray As Variant)

    UpperBound1 = UBound(pArray, 1)
    UpperBound2 = UBound(pArray, 2)
    sRangeToFill = Trim$(sStartCol) + Trim$(sStartRow) + ":" + _
        Chr(Asc(UCase$(sStartCol)) + (Trim(Str(UpperBound2)) - 1)) + _
        Trim(Str(UpperBound1))
    Sheets(sSheetName).Range(sRangeToFill).Value = pArray

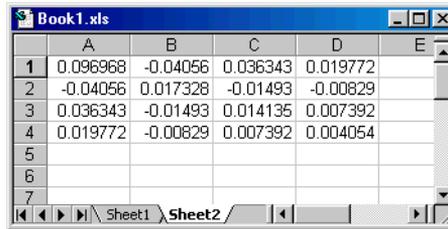
End Sub

```

Figure 6.8: The ConvertArrayToSheetRange procedure.

With all the coding complete, it's time to run the module.

11. Click the **Run Sub/User Form** button  on the **Visual Basic** toolbar. The results are shown in Figure 6.9.



	A	B	C	D	E
1	0.096968	-0.04056	0.036343	0.019772	
2	-0.04056	0.017328	-0.01493	-0.00829	
3	0.036343	-0.01493	0.014135	0.007392	
4	0.019772	-0.00829	0.007392	0.004054	
5					
6					
7					

Figure 6.9: *The covariance matrix returned to Excel.*

Exposing Objects to Client Applications

When you start Spotfire S+ for the first time, the single automation object S-PLUS.Application is exposed for use by automation client programs. By default, no other objects are exposed.

There are a number of ways in which S-PLUS automation objects can be exposed to, or hidden from, client applications. Table 6.1 lists the S-PLUS functions that you can use at any time to register or unregister automation objects.

Table 6.1: *S-PLUS functions for exposing and hiding automation objects.*

Function	Description
register.all.ole.objects	This function registers all S-PLUS objects with the system registry and builds or rebuilds the type library file. Returns T for success or F for failure.
unregister.all.ole.objects	This function unregisters all S-PLUS objects and removes the type library file. Returns T for success or F for failure.
register.ole.object	This function registers one or more S-PLUS objects with the system registry and builds or rebuilds the type library file. Returns T for success or F for failure.

Table 6.1: *S-PLUS functions for exposing and hiding automation objects. (Continued)*

Function	Description
<code>unregister.ole.object</code>	This function unregisters one or more S-PLUS objects and rebuilds the type library file. Returns <code>T</code> for success or <code>F</code> for failure.

With the exception of functions, all the built-in S-PLUS objects can be exposed simultaneously with a call to:

```
register.all.ole.objects()
```

Due to their large number, function objects are not exposed at one time because it would be too time-consuming. Instead, to expose any of the built-in functions, or any of those that you have defined, call:

```
register.ole.object(names)
```

where *names* is a character vector of the function names you want to expose. You can also use this function to register one or more particular S-PLUS objects.

To unregister all your S-PLUS objects, making them no longer available to automation clients, call:

```
unregister.all.ole.objects()
```

To unregister one or more particular S-PLUS objects, call:

```
unregister.ole.object(names)
```

with the desired *names* argument.

Caution

Unregistering your S-PLUS objects means that no automation client will be able to access those objects, which could potentially cause a client program to fail.

When you expose S-PLUS objects for use in automation, several entries are added to your Windows system registry. Automation client programs use these entries to identify what objects can be automated

and which application to use to automate them. When you hide your S-PLUS objects, these registry entries are removed so that client programs can no longer find them.

Note

Among these registry entries, the ProgID (program identifier) entry or human-readable name of the object (for example, S-PLUS.Application or S-PLUS.GraphSheet) is what you refer to in your client program script. This ProgID entry is mapped to a universally unique number entry called a UUID (universally unique identifier) in the Windows system registry. Under this UUID entry is stored the pathname on your system to the Spotfire S+ program, which is used by your client program to create and automate the object.

**Exploring
Properties and
Methods**

To start a conversation with Spotfire S+ from a client application, you must first create or get a S-PLUS object. Once an instance of the object has been created, it can be manipulated through its properties and methods.

The S-PLUS type library (installed by default to **cmd\Sp6obj.tlb** in the Spotfire S+ program folder) is useful for knowing what methods and properties are available for particular S-PLUS objects when programming in an automation client such as Visual Basic. However, the type library does not reveal the object hierarchy or how objects are related in Spotfire S+, nor does it provide much information on how to use the properties of S-PLUS objects.

Note

Although Spotfire S+ has an automation type library, it does not support “early binding” in an automation client such as Visual Basic. The types listed in the type library file are listed for informational purposes only. When you declare a S-PLUS variable in a client, you must declare it as the generic “object” type. Spotfire S+ supports only the “IDispatch” interface and “late binding” for all objects that are automatable.

For an easier way of seeing how to program S-PLUS objects, use the HTML-based object help system. In the **help\AutomationObjects** folder of the Spotfire S+ program folder, you will find a complete set of HTML files documenting the S-PLUS object hierarchy as distributed with the program, including an **index.htm** file displaying the entire S-PLUS object hierarchy. These files provide detailed

programming information, listing, for each automation object, not only its properties and methods, but also its possible containment paths, possible container objects, and possible child objects.

You can update the object help system at any time to reflect the complete object hierarchy for all objects currently registered, including any S-PLUS functions you write and expose using `register.ole.object`. If you choose, you can also create a new set of HTML files in a different folder on your system.

Table 6.2 lists the S-PLUS functions you can use to refresh or remove the type library or to refresh the object help system.

Table 6.2: *S-PLUS functions for documenting automation objects.*

Function	Description
<code>rebuild.type.library</code>	This function removes and then rebuilds the type library file with all currently registered S-PLUS objects. Returns <code>T</code> for success or <code>F</code> for failure.
<code>destroy.type.library</code>	This function removes the type library file from disk. Note that executing this command does not unregister any objects but simply removes the type library. Returns <code>T</code> for success or <code>F</code> for failure.
<code>rebuild.html.library(html.path, method.language.type = "basic")</code>	Argument <code>html.path</code> specifies the path, including drive letter, where you want the set of HTML files to be saved to disk. Optional argument <code>method.language.type</code> specifies the language used to write out the example syntax for methods in the HTML files; the default value for this argument is <code>basic</code> , but <code>c</code> or <code>c++</code> can also be used. This function creates the <code>html.path</code> specified, if it does not already exist, and writes an index.htm file that shows the complete object hierarchy for the automatable S-PLUS object system. Returns <code>T</code> for success or <code>F</code> for failure.

Note

The function `rebuild.html.library` uses only the currently registered objects to form the hierarchy and list of objects. Therefore, be sure to run `register.all.ole.objects` prior to calling this function to ensure that all objects appear in the help files.

Programming With Object Methods

S-PLUS automation objects are owned by Spotfire S+ but can be created and manipulated remotely through their properties and methods. For example, to start Spotfire S+ from a client application, simply call the `CreateObject` method on the S-PLUS application object. In automation terminology, Spotfire S+ is said to be *instantiated*.

Since, by default, only the `S-PLUS.Application` object is exposed, how then do you create, for example, an `S-PLUS.GraphSheet` object in your client program? If you try to do so directly, you will get an error in your client program indicating that the `S-PLUS.GraphSheet` object cannot be found. The following example in Visual Basic illustrates the point.

```
Dim pApplication As Object
Set pApplication = CreateObject("S-PLUS.Application")
'The above CreateObject succeeds because the Application
'object is exposed by S-PLUS.

Dim pGraphSheet As Object
Set pGraphSheet = CreateObject("S-PLUS.GraphSheet")
'The above CreateObject fails because the GraphSheet
'object has not yet been exposed by S-PLUS.
```

There are two ways in which you can create unexposed objects in an automation client program:

1. Call the S-PLUS function `register.all.ole.objects` to simultaneously expose all the built-in objects (and any function objects you have previously registered with `register.ole.object`). Once all your objects are registered, you can simply create an object directly in the client program.

2. Follow the object hierarchy shown in the **index.htm** file of the object help system to see how to create one object from another parent object until you get to the object you desire. This approach can be used, for example, to automate functions that you have not yet exposed using `register.ole.object`.

As an example of the second approach, consider again our question of how to create an `S-PLUS.GraphSheet` object in a client program. The object hierarchy shows that a `GraphSheet` object is a child of the `Application` object. To create a `GraphSheet` object, you must first create the `Application` object and then use the `CreateObject` method of the `Application` object to create the `GraphSheet`, as shown in the following Visual Basic script.

```
Dim pApplication As Object
Set pApplication = CreateObject("S-PLUS.Application")
    'The above CreateObject succeeds because the Application
    'object is exposed by S-PLUS.

Dim pGraphSheet As Object
Set pGraphSheet = pApplication.CreateObject("S-PLUS.GraphSheet")
    'The above CreateObject succeeds because S-PLUS
    'automatically exposes the GraphSheet object when you
    'create it as a child of the Application object.
```

Because a `GraphSheet` object is recognized as a child of the `Application` object, Spotfire S+ automatically exposes the `GraphSheet` object once you create it for the first time by calling the `CreateObject` method of the `Application` object. Once you create a child object in this way, you can create the child object type directly, as in the following Visual Basic script.

```
Dim pGraphSheet As Object
Set pGraphSheet = CreateObject("S-PLUS.GraphSheet")
    'The above CreateObject now succeeds because you
    'previously created a GraphSheet object as a child
    'of the Application object.
```

The following example in Visual Basic shows you how to create a GraphSheet object and then add an arrow to it using the CreateObject method.

```
Function CreateArrowInSPlus () As Integer
    Dim mySplus As Object
    Dim myGS As Object
    Dim myArrow As Object

    'Instantiate the Application object
    Set mySplus = CreateObject("S-PLUS.Application")

    'Instantiate the GraphSheet object
    Set myGS = mySplus.CreateObject("GraphSheet")

    'Add an arrow to this GraphSheet object
    Set myArrow = myGS.CreateObject("S-PLUS.Arrow")
End Function
```

Notice the form in which CreateObject is used in its third occurrence. Here, CreateObject is called as a method of the GraphSheet object and so creates the arrow as a child object of the GraphSheet container. Had we instead used

```
CreateObject("S-PLUS.Arrow")
```

a new GraphSheet object would have been created with the arrow added to that one.

Another method common to most S-PLUS automation objects is the GetObject function. You can use GetObject to get a reference to an object that already exists in Spotfire S+. In the next section, we list the common methods available for most automation objects.

Common Object Methods

Except for function objects, all S-PLUS automation objects have a set of common methods, listed in Table 6.3. Once an object has been created using CreateObject or GetObject, the other methods can be called. Consult the HTML files discussed on page 207 for detailed information concerning parameters for these methods.

Table 6.3: *Common object methods.*

Method	Description
BeginTransaction	Starts remembering property set calls so that all changes can be applied at once when CommitTransaction is called.
CancelTransaction	Cancels remembering property set calls made after the last call to BeginTransaction.
ClassName	Returns a string representing this object's class name.
CommitTransaction	Commits all property changes made since the last call to BeginTransaction.
Containees	Returns an array of objects contained by this object. Returns an array of containee objects of the class name specified or an empty array if none are found.
Container	Returns the object that is the container of this object.
CreateObject	Creates an object or child object of a particular type.
GetMethodArgumentNames	Returns a string array of argument names that can be used with the specified method for this object.
GetMethodArgumentTypes	Returns a string array of argument data types that must be passed as parameters to the specified method for this object. Data types returned depend on the language type specified.
GetMethodHelpString	Returns a string containing a description of the method specified for this object.
GetObject	Gets an object or child object of the type specified, identified by an object path name.
GetObjectPicture	Returns an array of byte values in a variant representing the Windows metafile format picture of this object. If unsuccessful, returns an empty variant.

Table 6.3: Common object methods. (Continued)

Method	Description
GetObjectRectangle	Returns the rectangular coordinates (client or screen, depending on the input parameter) in a variant that contains this object. If unsuccessful, returns an empty variant.
GetPropertyAllowedValues	Returns a string array of allowable values or allowable range of values for the specified property for this object.
GetPropertyInformation	Returns a string array of information about the specified property for this object.
GetSelectedObjects	Returns an array of currently selected objects that are contained in this object.
GetSelectedText	Returns a string containing the currently selected text in this object. If no selected text is found, returns an empty string.
Methods	Returns a comma-delimited string listing all allowable methods for this object.
MethodsList	Returns a string array of method names that can be called on this object.
Objects	Called with <code>Containses</code> parameter, returns a comma-delimited string listing all allowable child objects for this object. Called with <code>Containers</code> parameter, returns a list of objects that could contain this object.
ObjectsList	Depending on the parameter specified, returns a string array of class names for objects that can be valid children or parents of this object.
PathName	Returns a string representing this object's path name in Spotfire S+.
Properties	Returns a comma-delimited string listing all the properties for this object.

Table 6.3: Common object methods. (Continued)

Method	Description
PropertiesList	Returns a string array of property names that can be used with this object to set or get values.
RemoveObject	Removes a child object from this container object.
SelectObject	Selects this object in all views, returning TRUE if successful or FALSE if not.
ShowDialog	Displays a modal property dialog for this object that allows you to change any or all of its properties, pausing the client program until OK or Cancel is pressed in the dialog.
ShowDialogInParent	Displays a modal property dialog for this object in the client program, pausing the program while the dialog is displayed. Returns TRUE if successful or FALSE if not.
ShowDialogInParentModeless	Displays a modeless property dialog for the object in the client program, which continues executing while the dialog is displayed. Returns TRUE if successful or FALSE if not.

**Additional
Methods for the
Application
Object**

The Spotfire S+ “application” object is used to instantiate a Spotfire S+ session within a client application. All the common object methods can be applied to the application object. In addition, it has a number of specific methods, listed in Table 6.4. For detailed information concerning parameters for these methods, consult the HTML files discussed on page 207.

Table 6.4: Methods for the application object.

Method	Description
ChooseGraphAndPlotType	Displays the graph gallery dialog similar to that displayed by the CreatePlotsGallery function. The dialog allows the selection of axis and plot types and returns the axis and plot type strings selected when the dialog is accepted.

Table 6.4: *Methods for the application object. (Continued)*

Method	Description
ChooseGraphAndPlotTypeModeless	Displays the graph gallery dialog similar to that displayed by the ChooseGraphAndPlotType function except that it is modeless and allows the client program to continue running while the dialog is displayed. Returns a dialog handle that can be used in the functions GetHwndForModelessDialog and CloseModelessDialog.
CloseModelessDialog	Closes and destroys the dialog specified by the dialog handle. Returns FALSE on failure.
ExecuteString	Executes a string representing any valid S-PLUS syntax.
ExecuteStringResult	Returns a string representing the output from executing the string passed in. The format of the return string depends on the setting of the second parameter. If TRUE, the older S-PLUS 3.3 output formatting is applied. If FALSE, the new format is used.
GetHwndForModelessDialog	Gets the window handle (identifier) for a dialog handle as returned by the ChooseGraphAndPlotTypeModeless function.
GetOptionValue	Gets the current setting for an option (as in the Options ► General Settings dialog).
GetSAPIObject	Returns a binary SAPI object into a variant byte array given the name of the object in Spotfire S+. If no object is found, returns an empty variant.
GetSelectedGraphAndPlotType	Returns the selected graph and plot type as strings from the dialog handle specified. (An empty variant is returned for no selection.) Use the function ChooseGraphAndPlotTypeModeless to get the dialog handle to use in this function.
SetOptionValue	Sets an option value in the program (as in the Options ► General Settings dialog).

Table 6.4: *Methods for the application object. (Continued)*

Method	Description
SetSAPIObject	Sets a binary SAPI object created in the client program into Spotfire S+, making it available to other Spotfire S+ operations. Returns TRUE if successful or FALSE if not.

Additional Methods for Graph Objects

In addition to the common object methods listed in Table 6.3, Table 6.5 lists a number of methods available specifically for creating graphs and plots. Consult the HTML files discussed on page 207 for detailed information concerning parameters for these methods.

Table 6.5: *Methods for graph objects.*

Method	Description
ChooseGraphAndPlotType	Displays the graph gallery dialog similar to that displayed by the <code>CreatePlotsGallery</code> function. The dialog allows the selection of axis and plot types and returns the axis and plot type strings selected when the dialog is accepted.
CreateConditionedPlots	Returns TRUE if successful or FALSE if not.
CreateConditionedPlotsGallery	Returns TRUE if successful or FALSE if not.
CreateConditionedPlotsSeparateData	Returns TRUE if successful or FALSE if not.
CreateConditionedPlotsSeparateDataGallery	Returns TRUE if successful or FALSE if not.
CreatePlots	Returns TRUE if successful or FALSE if not.
CreatePlotsGallery	Returns TRUE if successful or FALSE if not.

Table 6.5: *Methods for graph objects. (Continued)*

Method	Description
ExecuteStringResult	Takes in a string representing any valid S-PLUS syntax and a boolean parameter indicating how the result should be formatted. Returns a string representing the result of executing the syntax passed in. (You can use %GSNAME% in the syntax string to get the GraphSheet object name substituted in the command.)

Methods for Function Objects Function objects differ from other S-PLUS objects in that they do not have all the same methods as other automation objects. The methods available for functions are listed Table 6.6. For detailed information concerning parameters for these methods, consult the HTML files discussed on page 207.

Table 6.6: *Methods for function objects.*

Method	Description
ClassName	Returns a string representing this object's class name.
GetMethodArgumentNames	Returns a string array of argument names that can be used with a specified method for this object.
GetMethodArgumentTypes	Returns a string array of argument data types that must be passed as parameters to the specified method for this object. Data types returned depend on the language type specified.
GetMethodHelpString	Returns a string containing a description of the method specified for this object.
GetParameterClasses	Returns an array of strings representing the class names of the return value followed by each of the parameters of this function.
GetPropertyAllowedValues	Returns a string array of allowable values or range of values for the specified property for this object.

Table 6.6: *Methods for function objects. (Continued)*

Method	Description
GetPropertyInformation	Returns a string array of information about the specified property for this object.
Methods	Returns a comma-delimited string listing all allowable methods for this function.
MethodsList	Returns a string array of method names that can be called on this object.
PathName	Returns a string representing this object's path name in Spotfire S+.
Properties	Returns a comma-delimited string listing all allowable arguments (parameters) for this function.
PropertiesList	Returns a string array of property names that can be used with this object to set or get values.
Run	Runs this function using the arguments (properties) most recently set.
SetParameterClasses	Specifies the class of function parameters. Returns TRUE if successful or FALSE if not.
ShowDialog	Displays a dialog for this function that allows you to change any or all of the function's arguments, pausing the client program until OK or Cancel is pressed in the dialog.
ShowDialogInParent	Displays a modal property dialog for this object in the client program, pausing the program while the dialog is displayed. Returns TRUE if successful or FALSE if not.
ShowDialogInParentModeless	Displays a modeless property dialog for this object in the client program, which continues executing while the dialog is displayed. Returns TRUE if successful for FALSE if not.

Programming With Object Properties

You can set and get the properties of an automation object to modify its appearance or behavior. For example, a property of the application object called `Visible` controls whether the Spotfire S+ main window will be visible in the client application.

When setting a series of properties for an object, you can use the `BeginTransaction` and `CommitTransaction` methods in a block to apply the changes all at once. The following example in Visual Basic illustrates how to use `BeginTransaction` and `CommitTransaction` to set color properties for an arrow on a `GraphSheet` object.

```
Sub ChangeArrowPropertiesInSPlus()  
    Dim myGS As Object  
    Dim myArrow As Object  
  
    Set myGS = CreateObject("S-PLUS.GraphSheet")  
    Set myArrow = myGS.CreateObject("S-PLUS.Arrow")  
  
    'Set properties for the Arrow object  
    myArrow.BeginTransaction  
    myArrow.LineColor = "Green"  
    myArrow.HeadColor = "Green"  
    myArrow.CommitTransaction  
  
    sLineColor = myArrow.LineColor  
End Sub
```

Because an object updates itself whenever a property is set, using a `BeginTransaction/CommitTransaction` block can save you time and speed up your client program.

Unlike other S-PLUS objects, function objects only update when the `Run` method is called. Therefore, the `BeginTransaction` and `CommitTransaction` (and `CancelTransaction`) methods are not supported, or even needed, for function objects.

As an example, suppose the following function has been defined and registered in a Spotfire S+ script as follows:

```
myFunction <- function(a,b) {return(a)}  
register.ole.object("myFunction")
```

The following Visual Basic example illustrates how to set a series of properties, or parameters, for the function object defined above.

```
Sub RunSPlusFunction()  
    Dim mySFunction As Object  
    Set mySFunction = CreateObject("S-PLUS.myFunction")  
  
    'Set properties for the function object  
    mySFunction.a = "1"  
    mySFunction.b = "2"  
    mySFunction.ReturnValue = "myVariable"  
  
    mySFunction.Run  
End Sub
```

Passing Data to Functions

The parameters, or arguments, of a function (and the function's return value) are properties of the function object and can be passed by value or by reference. When the data already exist in Spotfire S+, passing by reference is faster because the data do not have to be copied into the client before they can be used. However, when the data to be passed are from a variable defined in the client, the data should be passed by value. Note that the return value must not be passed by reference.

By default, all parameter data are passed by value as a data frame. This default behavior could cause errors if the function expects a data type other than a data frame. You can control the data types used in a function object in one of two ways:

- By calling the `SetParameterClasses` method of the function with a comma-delimited string specifying the data types (or class names) for each of the parameters and the return value of the function.
- By setting the `ArgumentClassList` property of the `FunctionInfo` object with a comma-delimited string specifying the data types (or class names) for each of the parameters and the return value of the function.

For any parameter you want to pass by reference instead of by value, place an ampersand character (&) at the beginning of its class name in the string.

We can use the following Spotfire S+ script to define and register a function called MyFunction:

```
MyFunction <- function(a) {return(as.data.frame(a))}  
register.ole.object("MyFunction")
```

and then use SetParameterClasses to adjust how the data from Visual Basic are interpreted by MyFunction.

```
Dim pArray(1 to 3) as double  
pArray(1) = 1.0  
pArray(2) = 2.0  
pArray(3) = 3.0  
  
Dim pMyFunction as Object  
Set pMyFunction = CreateObject("S-PLUS.MyFunction")  
  
if ( pMyFunction.SetParameterClasses("data.frame,vector") = TRUE ) then  
  pMyFunction.a = pArray  
  pMyFunction.Run  
  
  Dim pReturnArray as Variant  
  pReturnArray = pMyFunction.ReturnValue  
end if
```

The following example shows how a vector in Spotfire S+ can be passed by reference to MyFunction in Spotfire S+, instead of passing data from variables in Visual Basic.

```
Dim pApp as Object
Set pApp = CreateObject("S-PLUS.Application")

Dim pMyVectorInSPLUS as Object
Set pMyVectorInSPLUS = pApp.GetObject("vector", "MyVector")

Dim pMyFunction as Object
Set pMyFunction = pApp.CreateObject("MyFunction")

if ( pMyFunction.SetParameterClasses("data.frame,&vector") = TRUE ) then
  Set pMyFunction.a = pMyVectorInSPLUS
  pMyFunction.Run

  Dim pReturnArray as Variant
  pReturnArray = pMyFunction.ReturnValue
end if
```

Notice how the vector object `MyVector` is obtained from Spotfire S+ using `GetObject` and assigned directly to `pMyFunction.a` to avoid having to get the data from `MyVector` into a variant and then assign that variant data to `pMyFunction.a`. This is possible when you specify the `&` before a class name in `SetParameterClasses`.

As an alternative to using `SetParameterClasses` in the client, you can define the parameter classes using the `ArgumentClassList` property when you define the `FunctionInfo` object to represent the function in Spotfire S+. This approach has the advantage of simplifying the automation client program code but does require some additional steps in Spotfire S+ when defining the function.

Consider the following Spotfire S+ script to define the function `MyFunction` and a `FunctionInfo` object for this function:

```
MyFunction <- function(a)
{
  return(a)
}

guiCreate(
  "FunctionInfo", Function = "MyFunction",
  ArgumentClassList = "vector, vector" )
```

The script sets `ArgumentClassList` to the string "vector, vector" indicating that data passed into and out of `MyFunction` via automation will be done using S-PLUS vectors. When this approach is used, the corresponding client code becomes simpler because we no longer need to set the parameter classes for the function before it is used.

```
Dim pArray(1 to 3) as double
pArray(1) = 1.0
pArray(2) = 2.0
pArray(3) = 3.0

Dim pMyFunction as Object
Set pMyFunction = CreateObject("S-PLUS.MyFunction")
pMyFunction.a = pArray
pMyFunction.Run

Dim pReturnArray as Variant
pReturnArray = pMyFunction.ReturnValue
```

Automating Embedded Spotfire S+ Graph Sheets

With Spotfire S+ automation support, it is easy to embed **Graph Sheets** in any automation client, such as Visual Basic, Excel, Word, and others. You can create, modify, and save an embedded **Graph Sheet** with plotted data without ever leaving your client program.

The **vbembed** example that ships with Spotfire S+ demonstrates how to embed a Spotfire S+ **Graph Sheet**, add objects to it, modify these objects by displaying their property dialogs in the client program, delete objects from it, and save a document containing the embedded **Graph Sheet**. The **vcembed** example is a Visual C++/MFC application that shows how to embed and automate a Spotfire S+ **Graph Sheet** in a C++ automation client. The **PlotData.xls** example illustrates how to embed a Spotfire S+ **Graph Sheet**, add a plot to it, send data from an Excel worksheet to be graphed in the plot, and modify the plot's properties using property dialogs. See Table 6.9 on page 235 for help in locating these examples.

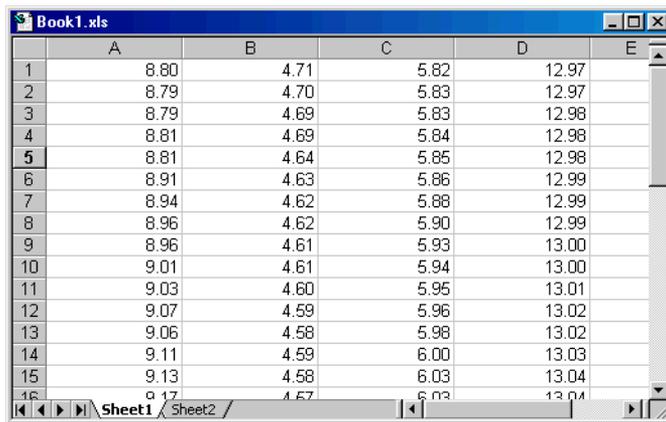
USING SPOTFIRE S+ AS AN AUTOMATION CLIENT

In addition to being used as an automation server, Spotfire S+ can also function as an automation client. A program in the S-PLUS programming language can create and manipulate the automation objects exposed by other applications through their type libraries. Spotfire S+ provides a number of functions that allow you to create objects, set and get properties, call methods, and manage reference counting. Before discussing these functions in detail, let's take a look at a simple example.

A Simple Example

To demonstrate how to use Spotfire S+ as an automation client, we revisit the simple example presented on page 199, this time reversing the roles of Spotfire S+ and Excel. In this scenario, S-PLUS functions as the client application, retrieving data from an Excel worksheet, performing a covariance estimation on the data, and returning the resulting covariance matrix to Excel.

Consider again the sample data of Figure 6.1, reproduced in Figure 6.10. **Sheet1** of the Excel workbook **Book1.xls** contains data in 4 columns of 39 rows (not all rows are shown in Figure 6.10).



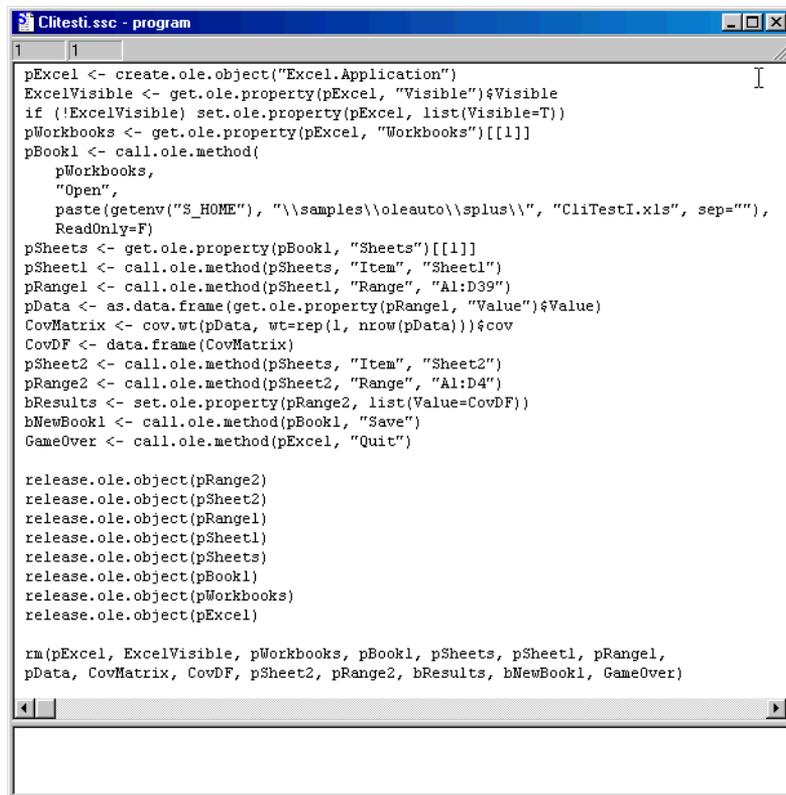
	A	B	C	D	E
1	8.80	4.71	5.82	12.97	
2	8.79	4.70	5.83	12.97	
3	8.79	4.69	5.83	12.98	
4	8.81	4.69	5.84	12.98	
5	8.81	4.64	5.85	12.98	
6	8.91	4.63	5.86	12.99	
7	8.94	4.62	5.88	12.99	
8	8.96	4.62	5.90	12.99	
9	8.96	4.61	5.93	13.00	
10	9.01	4.61	5.94	13.00	
11	9.03	4.60	5.95	13.01	
12	9.07	4.59	5.96	13.02	
13	9.06	4.58	5.98	13.02	
14	9.11	4.59	6.00	13.03	
15	9.13	4.58	6.03	13.04	
16	9.17	4.57	6.03	13.04	

Figure 6.10: Sample data in **Book1.xls**.

Note

The sample data in **Book1.xls** are taken from the `freeny.x` matrix included with Spotfire S+. You can recreate this example by exporting the data into a new Excel worksheet and following the steps outlined below.

By writing a script in the S-PLUS programming language, we can automate a conversation between Spotfire S+ and Excel to perform our task. The complete code of one such script is shown in Figure 6.11.



```

pExcel <- create.ole.object("Excel.Application")
ExcelVisible <- get.ole.property(pExcel, "Visible")$Visible
if (!ExcelVisible) set.ole.property(pExcel, list(Visible=T))
pWorkbooks <- get.ole.property(pExcel, "Workbooks")[[1]]
pBook1 <- call.ole.method(
  pWorkbooks,
  "Open",
  paste(getenv("S_HOME"), "\\samples\\oleauto\\splus\\", "CliTestI.xls", sep=""),
  ReadOnly=F)
pSheets <- get.ole.property(pBook1, "Sheets")[[1]]
pSheet1 <- call.ole.method(pSheets, "Item", "Sheet1")
pRange1 <- call.ole.method(pSheet1, "Range", "A1:D39")
pData <- as.data.frame(get.ole.property(pRange1, "Value")$Value)
CovMatrix <- cov.wt(pData, wt=rep(1, nrow(pData)))$cov
CovDF <- data.frame(CovMatrix)
pSheet2 <- call.ole.method(pSheets, "Item", "Sheet2")
pRange2 <- call.ole.method(pSheet2, "Range", "A1:D4")
bResults <- set.ole.property(pRange2, list(Value=CovDF))
bNewBook1 <- call.ole.method(pBook1, "Save")
GameOver <- call.ole.method(pExcel, "Quit")

release.ole.object(pRange2)
release.ole.object(pSheet2)
release.ole.object(pRange1)
release.ole.object(pSheet1)
release.ole.object(pSheets)
release.ole.object(pBook1)
release.ole.object(pWorkbooks)
release.ole.object(pExcel)

rm(pExcel, ExcelVisible, pWorkbooks, pBook1, pSheets, pSheet1, pRange1,
  pData, CovMatrix, CovDF, pSheet2, pRange2, bResults, bNewBook1, GameOver)

```

Figure 6.11: Complete code for our Spotfire S+ script.

Hint

The example shown in Figure 6.11 can be found in **samples/oleauto/splus/Clitesti.ssc** in the Spotfire S+ program folder.

1. Open a new **Script** window in Spotfire S+ and enter the first block of code, as shown in Figure 6.12.

```

1
pExcel <- create.ole.object("Excel.Application")
ExcelVisible <- get.ole.property(pExcel, "Visible")$Visible
if (!ExcelVisible) set.ole.property(pExcel, list(Visible=T))
pWorkbooks <- get.ole.property(pExcel, "Workbooks")[[1]]
pBook1 <- call.ole.method(
  pWorkbooks,
  "Open",
  paste(getenv("S_HOME"), "\\samples\\oleauto\\splus\\", "CliTest1.xls", sep=""),
  ReadOnly=F)
pSheets <- get.ole.property(pBook1, "Sheets")[[1]]
pSheet1 <- call.ole.method(pSheets, "Item", "Sheet1")
pRange1 <- call.ole.method(pSheet1, "Range", "A1:D39")
pData <- as.data.frame(get.ole.property(pRange1, "Value")$Value)
CovMatrix <- cov.wt(pData, wt=rep(1, nrow(pData)))$cov
CovDF <- data.frame(CovMatrix)
pSheet2 <- call.ole.method(pSheets, "Item", "Sheet2")
pRange2 <- call.ole.method(pSheet2, "Range", "A1:D4")
bResults <- set.ole.property(pRange2, list(Value=CovDF))
bNewBook1 <- call.ole.method(pBook1, "Save")
GameOver <- call.ole.method(pExcel, "Quit")

```

Figure 6.12: *The core code for our Spotfire S+ script.*

The code in Figure 6.12 represents the central task we want to automate. Let's examine each line in detail.

We start a conversation with Excel from Spotfire S+ by creating an instance of Excel using the S-PLUS function `create.ole.object`:

```
pExcel <- create.ole.object("Excel.Application")
```

To see what's happening in Excel as the script runs, we can set the `Visible` property of the Excel application object to `True`. To do so, we first capture the value of the `Visible` property using the `get.ole.property` function:

```
ExcelVisible <- get.ole.property(pExcel,"Visible")$Visible
```

Note

The `get.ole.property` function returns a list of properties. Use the `$` or `[[[]]` operator to extract the value of an individual component of the list.

We then test the value of the `Visible` property and set it to `True` using the `set.ole.property` function:

```
if (!ExcelVisible) set.ole.property(pExcel,list(Visible=T))
```

Note

The `set.ole.property` function expects a list of properties to set.

To open the **Book1.xls** workbook, we first get the value of the `Workbooks` property of the Excel application object:

```
pWorkbooks <- get.ole.property(pExcel, "Workbooks")[[1]]
```

and then call the `Open` method on the `pWorkbooks` object using the `S-PLUS` function `call.ole.method`:

```
pBook1 <- call.ole.method(pWorkbooks, "Open",  
paste(getenv("SHOME"), "\\samples\\oleauto\\splus\\",  
"CliTestI.xls", sep=""), ReadOnly=F)
```

Note

When using `call.ole.method` to call a method on an automation object, consult the type library of the server application for a list of arguments relevant to the method you are calling.

When using `call.ole.method` to call a method on an automation object, you can specify the parameters as `null` if you do not want to specify a parameter in the method you are calling.

In this example, we are automating a conversation with Excel; therefore, we must follow the Excel object model and navigate through Excel's object hierarchy in order to access a range of cells:

```
pSheets <- get.ole.property(pBook1, "Sheets")[[1]]
pSheet1 <- call.ole.method(pSheets, "Item", "Sheet1")
pRange1 <- call.ole.method(pSheet1, "Range", "A1:D39")
```

Having arrived at the level of actual cell contents, we can now capture our data with the following statement:

```
pData <- as.data.frame(get.ole.property(pRange1,
"Value")$Value)
```

In the next two statements, we use standard S-PLUS functions to perform the covariance estimation and convert the resulting matrix into a data frame:

```
CovMatrix <- cov.wt(pData, wt=rep(1, nrow(pData)))$cov
CovDF <- data.frame(CovMatrix)
```

With the results now stored in a S-PLUS variable, we again navigate through the Excel object hierarchy to the target range of cells on **Sheet2**:

```
pSheet2 <- call.ole.method(pSheets, "Item", "Sheet2")
pRange2 <- call.ole.method(pSheet2, "Range", "A1:D4")
```

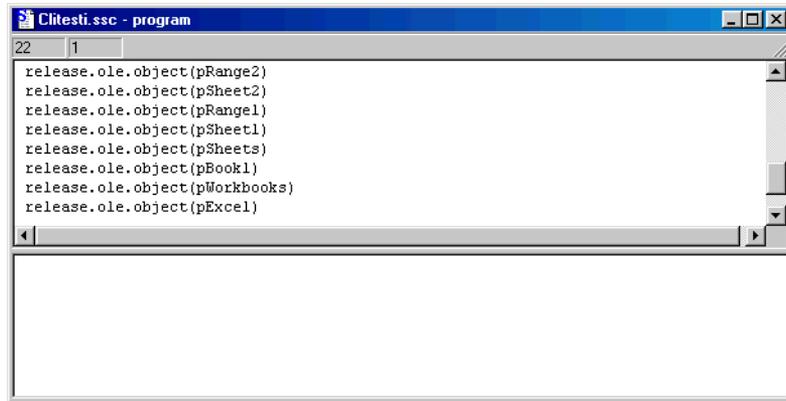
and place the results in the target range:

```
bResults <- set.ole.property(pRange2, list(Value=CovDF))
```

Finally, the last two statements save the workbook and close the Excel application:

```
bNewBook1 <- call.ole.method(pBook1, "Save")
GameOver <- call.ole.method(pExcel, "Quit")
```

2. Now add the second block of code to the script, as shown in Figure 6.13.



```

22 1
release.ole.object(pRange2)
release.ole.object(pSheet2)
release.ole.object(pRange1)
release.ole.object(pSheet1)
release.ole.object(pSheets)
release.ole.object(pBook1)
release.ole.object(pWorkbooks)
release.ole.object(pExcel)

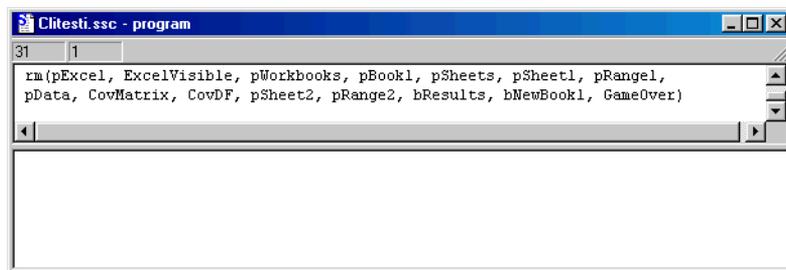
```

Figure 6.13: Code for releasing all OLE objects.

As we will see in Reference Counting Issues on page 232, objects that are created directly or indirectly during program execution must be released at program end to allow the server application to close. This is accomplished in our second block of code.

After releasing all the OLE objects, the last thing to do is to clean up our working data by deleting all the data objects created during execution of the script.

3. Add the last block of code to the script, as shown in Figure 6.14.



```

31 1
rm(pExcel, ExcelVisible, pWorkbooks, pBook1, pSheets, pSheet1, pRange1,
pData, CovMatrix, CovDF, pSheet2, pRange2, bResults, bNewBook1, GameOver)

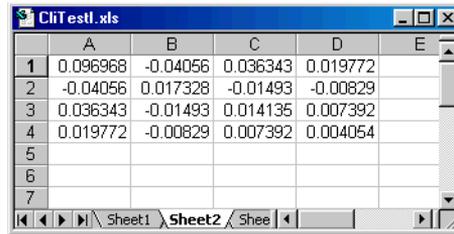
```

Figure 6.14: Code for removing all data objects.

With all the coding complete, it's time to run the script.

4. Click the **Run** button  on the **Script** window toolbar.

- After the script finishes running, start Excel, open the **CliTest1.xls** workbook (saved in your default folder), and click the tab for **Sheet2**. The results are shown in Figure 6.15.



	A	B	C	D	E
1	0.096968	-0.04056	0.036343	0.019772	
2	-0.04056	0.017328	-0.01493	-0.00829	
3	0.036343	-0.01493	0.014135	0.007392	
4	0.019772	-0.00829	0.007392	0.004054	
5					
6					
7					

Figure 6.15: *The covariance matrix sent to Excel.*

High-Level Automation Functions

As demonstrated in the last section, you can use Spotfire S+ as an automation client by writing a program in the S-PLUS programming language to create and manipulate the automation objects of other applications.

Spotfire S+ provides the functions listed in Table 6.7 for creating objects, setting and getting properties, and calling methods. The automation objects of the server application are passed as arguments to these functions. In addition, Spotfire S+ provides several other functions for managing reference counting; see page 232 for details.

For each of the functions listed in Table 6.7, two optional character vector attributes called “error” and “warning” may be returned with each return value. These attributes can be used to stop the program and display warnings, errors, etc.

Table 6.7: *High-level automation functions.*

Function	Description
<code>create.ole.object(<i>name</i>)</code>	<p>The <i>name</i> argument is the name of the instance, a character vector of length 1.</p> <p>This function returns a character vector of class <code>OLEClient</code>, representing the particular instance. The first string in the vector is the instance ID of the object, stored in Spotfire S+. The other strings are various pieces of information about the instance (such as server name). On error, <code>NULL</code> is returned.</p>
<code>set.ole.property(<i>instance</i>, <i>properties</i>)</code>	<p>The <i>instance</i> argument is an object previously created within the same session. The <i>properties</i> argument is a list of elements, each element being a property name set to a desired value, which must be an atomic type of length 1.</p> <p>This function returns a vector of logicals, with <code>T</code> for each property successfully set and <code>F</code> otherwise.</p>
<code>get.ole.property(<i>instance</i>, <i>property.names</i>)</code>	<p>The <i>instance</i> argument is an object previously created within the same session. The <i>property.names</i> argument is a character vector of property names appropriate to the instance, as specified in the type library of the server application.</p> <p>This function returns a list of the values of the properties specified in the <i>property.names</i> argument. <code>NULL</code> is returned for a property that cannot be fetched.</p>

Table 6.7: *High-level automation functions. (Continued)*

Function	Description
<code>call.ole.method(instance, method.name, ...)</code>	<p>The <i>instance</i> argument is an object previously created within the same session. The <i>method.name</i> argument is the name of the method to be invoked on the instance object. “...” represents the arguments to be passed as arguments to the method. You can pass null arguments to the automation server object if you do not want to specify its parameters. Otherwise, only supported types may be passed. That is, at this point, atomic types (character, single, integer, numeric vectors) of length 1. Methods for particular objects and arguments to specific methods can be found in the type library of the server application.</p> <p>This function returns a S-PLUS object containing the result of calling the method on the particular instance. On error, NULL is returned.</p>

Reference Counting Issues

When you create an object using `create.ole.object`, or when an object is created indirectly from the return of a property using `get.ole.property` or `call.ole.method`, the object is given a reference count of one. This means that your Spotfire S+ program has a lock on the object and the server application that created the object for your program cannot close until you release the references to it by reducing the reference count to zero.

If you assign the variable that represents one of these objects to another object, you should increment the object’s reference count by one to indicate that an additional variable now has a lock on the object. If you remove a variable or a variable that represents an automation object goes out of scope, you should decrement the reference count of the object the variable being destroyed represents.

To help you manage reference counting on automation objects, Spotfire S+ provides the four functions listed in Table 6.8. For each of these functions, two optional character vector attributes called “error”

and “warning” may be returned with each return value. These attributes can be used to stop the program and display warnings, errors, etc.

Table 6.8: *Functions for managing reference counting.*

Function	Description
<code>ole.reference.count(instance)</code>	The <i>instance</i> argument is an object previously created within the same session. This function returns the value of the reference count for the instance. On error, -1 is returned.
<code>add.ole.reference(instance)</code>	The <i>instance</i> argument is an object previously created within the same session. This function increments the reference count for the instance by 1 and returns the new value. On error, -1 is returned.
<code>release.ole.object(instance)</code>	The <i>instance</i> argument is an object previously created within the same session. This function decrements the reference count for the instance by 1 and returns the new value. On error, -1 is returned.
<code>is.ole.object.valid(instance)</code>	The <i>instance</i> argument is an object previously created within the same session. This function returns T if the instance is valid or F otherwise.

Although reference counting must be handled manually, Spotfire S+ guards against the two major types of reference counting bugs: resource leaks and freezing due to using an invalid object handle.

If you release all reference counts on an object and then attempt to set or get a property or call a method on this object, Spotfire S+ gives you an error that the object is no longer valid. You can check the validity of any automation object by using the S-PLUS function `is.ole.object.valid`.

If you fail to release all references to objects you create during a Spotfire S+ session, the server application owning the objects will remain loaded and running. However, exiting Spotfire S+ automatically releases all objects, reduces all reference counts to zero, and closes all server applications.

AUTOMATION EXAMPLES

Spotfire S+ ships with a number of examples that illustrate how to use Spotfire S+ as both an automation server and an automation client.

Server Examples The examples listed in Table 6.9 can be found in the **samples/oleauto** folder in the Spotfire S+ program folder.

Table 6.9: *Automation server examples.*

Name	Client Application	Description
vb/ChoosePt	Visual Basic 6	Project showing how to use the <code>ChooseGraphAndPlotType</code> automation method to display the Insert Graph dialog and allow a user select the graph and plot type.
vb/CreatePt	Visual Basic 6	Project showing how to use the following automation methods: <ul style="list-style-type: none"> • <code>CreatePlots</code> • <code>CreateConditionedPlots</code> • <code>CreateConditionedPlotsSeparateData</code>
vb/Dialogs	Visual Basic 6	Project showing how to use the following automation methods: <ul style="list-style-type: none"> • <code>ShowDialog</code> • <code>ShowDialogInParent</code> • <code>ShowDialogInParentModeless</code>
vb/gspages	Visual Basic 6	Project showing how to embed a Graph Sheet with multiple pages, set up a tab control that has tabs for each page, and support changing active pages in the embedded Graph Sheet by clicking on tabs in the tab control.

Table 6.9: Automation server examples. (Continued)

Name	Client Application	Description
vb/mixedddf	Visual Basic 6	Project showing how to create a two-dimensional array of mixed data (that is, columns of different data types) and send it to a <code>data.frame</code> object in Spotfire S+ and also how to retrieve the data from the <code>data.frame</code> object into Visual Basic for display.
vb/objects	Visual Basic 6	Project showing how to use the following automation methods: <ul style="list-style-type: none"> • <code>ObjectContainees</code> • <code>ObjectContainer</code> • <code>ClassName</code> • <code>PathName</code>
vb/vbclient	Visual Basic 6	Project showing how to create a Graph Sheet , add an arrow to it, change the properties of the arrow, show a dialog for the arrow, execute S-PLUS commands, modify option values, get an object, and send and receive data.
vb/vbembed	Visual Basic 6	Project showing how to embed a Spotfire S+ Graph Sheet , modify it, save it, and delete objects in it, and how to display an object dialog.
vb/vbrunfns	Visual Basic 6	Project showing how to register a S-PLUS function, pass binary data to the function, and receive the result back into Visual Basic.
vba/excel/auto_VBA.xls	Visual Basic for Applications with Excel 97	Example showing how to send and receive data and convert Excel ranges to arrays.

Table 6.9: Automation server examples. (Continued)

Name	Client Application	Description
vb/mixedddf	Visual Basic 6	Project showing how to create a two-dimensional array of mixed data (that is, columns of different data types) and send it to a <code>data.frame</code> object in Spotfire S+ and also how to retrieve the data from the <code>data.frame</code> object into Visual Basic for display.
vb/objects	Visual Basic 6	Project showing how to use the following automation methods: <ul style="list-style-type: none"> • <code>ObjectContainees</code> • <code>ObjectContainer</code> • <code>ClassName</code> • <code>PathName</code>
vb/vbclient	Visual Basic 6	Project showing how to create a Graph Sheet , add an arrow to it, change the properties of the arrow, show a dialog for the arrow, execute S-PLUS commands, modify option values, get an object, and send and receive data.
vb/vbembed	Visual Basic 6	Project showing how to embed a Spotfire S+ Graph Sheet , modify it, save it, and delete objects in it, and how to display an object dialog.
vb/vbrunfns	Visual Basic 6	Project showing how to register a S-PLUS function, pass binary data to the function, and receive the result back into Visual Basic.
vba/excel/auto_VBA.xls	Visual Basic for Applications with Excel 97	Example showing how to send and receive data and convert Excel ranges to arrays.

Table 6.9: Automation server examples. (Continued)

Name	Client Application	Description
vba/excel/Book1.xls	Visual Basic for Applications with Excel 97	Example showing how to pass data from an Excel worksheet to Spotfire S+, which then performs a covariance estimation on the data and returns the resulting covariance matrix to Excel.
vba/excel/PlotData.xls	Visual Basic for Applications with Excel 97	Example showing how to embed a Graph Sheet and add and modify a plot in it.
vba/excel/XferToDF.xls	Visual Basic for Applications with Excel 97	Example showing how to transfer Excel ranges to S-PLUS data frames and back to Excel ranges.
visualc/autocInt	Visual C++ 6.0	Non-MFC based C++ project showing how to use automation to access the Spotfire S+ command line.
visualc/vcembed	Visual C++ 6.0	Project showing how to create and manipulate an embedded Graph Sheet in an MFC application. This example uses several MFC classes that make interacting with automation objects from Spotfire S+ much easier in MFC code. See the readme.txt in the vcembed directory for more information.

Client Examples

The examples listed in Table 6.10 can be found in the **samples/oleauto/splus** folder in the Spotfire S+ program folder.

Table 6.10: *Automation client examples.*

Name	Server Application	Description
Clitesta.ssc	Excel 97	Script showing how to use Spotfire S+ commands to start Excel and call method of Excel to convert inches to points and return the result in Spotfire S+.
Clitestb.ssc	Excel 97	Script showing how to use Spotfire S+ commands to start Excel, set a property, and get a property.
Clitestic.ssc	Excel 97	Script showing how to use Spotfire S+ commands to set a range of data in an Excel worksheet with data from a S-PLUS vector and then how to get the data back from Excel into another vector.
Clitestd.ssc	Excel 97	Script showing how to get a property value from Excel.
Cliteste.ssc	Excel 97	Script showing how to send a vector from Spotfire S+ to Excel and transpose it to a row in Excel.
Clitestf.ssc	Excel 97	Script showing how to send a vector from Spotfire S+ to Excel and transpose it to a row in Excel using a different set of steps than in Cliteste.ssc .
Clitestg.ssc	Excel 97	Script showing how to send the data from a data frame in Spotfire S+ into a new worksheet and range in Excel and then how to get the range data from Excel back into a new data frame in Spotfire S+.

Table 6.10: Automation client examples. (Continued)

Name	Server Application	Description
clitesth.ssc	Visual Basic 6	Example combining a Visual Basic automation server project called GetArray with a Spotfire S+ automation client script that calls it to retrieve data into a data frame (used in conjunction with the VB automation server in the Getarray folder). The automation server exposes an object type called <code>MyArrayObject</code> having a method called <code>GetArray</code> that gets a randomly generated, two-dimensional array of mixed data (that is, columns having different data types). The Spotfire S+ script uses this method to get the array and then set it into a newly created data frame.
Clitesti.ssc	Excel 97	Script showing how to retrieve data from an Excel worksheet, perform a covariance estimation on the data, and return the resulting covariance matrix to Excel.

CALLING SPOTFIRE S+ USING DDE

7

Introduction	242
Working With DDE	243
Starting a DDE Conversation	244
Executing Spotfire S+ Commands	246
Sending Data to Spotfire S+	246
Getting Data From Spotfire S+	248
Enabling and Disabling Response to DDE Requests	250

INTRODUCTION

Communications between programs can take place in a number of ways. For example, data can be passed between programs by having one program write an ASCII file to disk and having another program read that file. It is also possible for programs to exchange data by using the Windows[®] clipboard. In each of these methods, the process is sequential and normally requires human intervention to coordinate the action.

Dynamic Data Exchange (DDE), on the other hand, is a mechanism supported by Microsoft Windows that permits two different programs running under Windows to communicate in real time without outside intervention. This communication can take the form of two programs passing data back and forth or it can take the form of one program requesting another program to take specific action. It can also take place under program control (without human intervention) and as often as required.

In this chapter, we explain how to communicate with Spotfire S+ using DDE and provide some example code.

Note
This chapter is dedicated to Windows users.

WORKING WITH DDE

DDE uses the metaphor of a conversation. In a DDE conversation, one program initiates the conversation and another program responds. The program initiating the conversation is called the *destination* program, or *client*, while the responding program is called the *source* program, or *server*.

Spotfire S+ can function as a DDE server, although not as a DDE client (except via the Windows clipboard). Any application that supports DDE client functions, such as Visual Basic, Visual C++, Excel, Lotus 1-2-3, and PowerBuilder, can initiate a DDE conversation with Spotfire S+. Spotfire S+ supports the basic DDE functions Connect, Execute, Poke, Request, Advise, Unadvise, and Terminate.

Note

Spotfire S+ can function as a DDE server or a DDE client via the clipboard by using **Copy Paste Link**:

- You can copy data from a S-PLUS data object, such as a data frame or vector, into the clipboard and then paste the data into another OLE- or DDE-supporting program as a DDE link to the data in Spotfire S+. This connection is a hot link between a block of cells in the S-PLUS data object and a block in the document of the other program. If DDE server support is enabled in Spotfire S+, whenever you copy data from a data object to the clipboard, DDE link information is transferred at the same time. Then when you paste into another program, the **Paste Special** or **Paste Link** option will be enabled if that client program supports DDE linking.
- You can copy data from a server program that supports DDE linking into the clipboard and then choose **Paste Link** from the **Edit** menu when a S-PLUS data object, such as a data frame or vector, is in focus. (Note that string data will always be represented in S-PLUS as character, not factor, data when you **Paste Link** from a DDE server application.) This will paste the link into the data object that currently has the focus, starting the pasted block at the current cell location. This tells Spotfire S+ to request a DDE link to the data specified in the server program's document. Then whenever the data change in the server document, the changes are automatically updated in the S-PLUS data object where you pasted the linked data.

Spotfire S+ supports DDE conversations in the Microsoft CF_TEXT format, which is simply an array of text characters terminated by a null character and with each line ending with a carriage return-linefeed (CR-LF) pair. Binary transfers are not supported.

Starting a DDE Conversation

The three steps of a DDE conversation are as follows:

1. Initiate the conversation.
2. Issue one or more DDE commands.
3. Terminate the conversation.

A DDE client application opens a conversation with a particular server. Each server has a *server name* to which it will respond; the server name for Spotfire S+ is Spotfire S+.

Because an application can have multiple DDE conversations open at the same time, it is necessary to have an identifier, called the *channel number*, for each conversation. A channel number is established when the DDE conversation is initiated.

A DDE conversation must also have a topic. Table 7.1 lists the DDE conversation topics supported by Spotfire S+.

Table 7.1: Supported topics for DDE conversations.

Topic	Description	Example in Visual Basic for Applications
System [System]	The System topic is used to request special information from Spotfire S+. This information includes the names of data objects (such as data frames and vectors) that can be used in conversations, allowable conversation topic names, and other information.	<pre> Channel = Application.DDEInitiate(_ "Spotfire S+", "System") Info = Application.DDERequest(_ Channel, "Topics") Application.DDETerminate Channel Info = { "System" "[DataSheet]", "[Execute]", "SCommand" } </pre>

Table 7.1: Supported topics for DDE conversations. (Continued)

Topic	Description	Example in Visual Basic for Applications
[DataSheet]	The [DataSheet] topic is an identifier used to specify that the conversation is about a block of data from a data object. This topic is optional; you can simply specify the name of the data object as the conversation topic.	<pre>Channel = Application.DDEInitiate(_ "Spotfire S+", "[DataSheet]exsurf") Info = Application.DDERequest(_ Channel, "r1c1:r3c2") Application.DDETerminate Channel Info = { -2.00, -2.00; -1.87, -2.00; -1.74, -2.00 }</pre>
Any data object name (such as the name of a data frame or vector)	Same as the [DataSheet] topic above.	<pre>Channel = Application.DDEInitiate(_ "Spotfire S+", "exsurf") Info = Application.DDERequest(_ Channel, "r1c1:r3c2") Application.DDETerminate Channel Info = { -2.00, -2.00; -1.87, -2.00; -1.74, -2.00 }</pre>
SCommand [Execute]	SCommand or [Execute] identifies that the conversation contains strings of valid Spotfire S+ commands or expressions to be executed by Spotfire S+.	<pre>Channel = Application.DDEInitiate(_ "Spotfire S+", "SCommand") Info = Application.DDERequest(_ Channel, "objects(0)") Application.DDETerminate Channel Info = { ".Copyright .Options .Program version" }</pre>

The following example illustrates a simple DDERequest conversation in which Visual Basic for Applications (VBA), as used by Microsoft Word or Excel, is the client and Spotfire S+ is the server.

```
exec_channel = Application.DDEInitiate( "Spotfire S+", "SCommand" )
ReturnResult = Application.DDERequest( exec_channel, "summary(corn.rain)" )
Application.DDETerminate exec_channel
```

Note

A sample Visual Basic DDE client program called **vbclient**, as well as example Excel spreadsheets with VBA and macro scripts demonstrating connection to Spotfire S+ via DDE, can be found in the **samples/dde** folder of the Spotfire S+ program folder.

The first statement initiates a conversation with Spotfire S+ using the topic `SCommand`. (You can use either `SCommand` or `[Execute]` for executing a Spotfire S+ command and requesting the result via `DDERequest`.) This statement returns the channel number for the conversation and assigns it to `exec_channel`, which is then used in the `DDERequest` statement. The `DDERequest` statement executes the command `summary(corn.rain)` in Spotfire S+ and returns the result of the execution in an array called `ReturnResult`. Finally, the conversation is ended by calling `DDETerminate`.

Executing Spotfire S+ Commands

The `DDEExecute` command executes a command in Spotfire S+ but does not return the output from the execution. You can use either `SCommand` or `[Execute]` as the topic of a `DDEExecute` conversation.

Consider the following example of `DDEExecute`, written in VBA.

```
exec_channel = Application.DDEInitiate( "Spotfire S+", "SCommand" )  
Application.DDEExecute exec_channel, "summary(corn.rain)"  
Application.DDETerminate exec_channel
```

First, a conversation is initiated with the `DDEInitiate` command. Next, `DDEExecute` is used to execute the Spotfire S+ command `summary(corn.rain)`. The output from this command will go either to the **Commands** window, if open, or to a **Report** window in Spotfire S+. No output is returned to the calling program. Finally, the conversation is ended with a call to `DDETerminate`.

The exact format of the `DDEExecute` command will vary from program to program; that is, Excel has its own format, as does Lotus 1-2-3 and Visual Basic. The essential requirements, however, are the same—a channel number and a command string.

A client program can issue multiple DDE commands, including `DDEExecute`, `DDEPoke`, and `DDERequest`, to Spotfire S+ as long as the DDE conversation with the topic `SCommand` or `[Execute]` is open.

Sending Data to Spotfire S+

When used in a conversation with the topic `SCommand` or `[Execute]`, the `DDEPoke` command behaves in the same way as the `DDEExecute` command; that is, it executes the commands you poke but does not return any results.

Consider the following VBA script in Excel using DDEPoke.

```
Channel = Application.DDEInitiate("Spotfire S+", "SCommand")
Application.DDEPoke Channel, "", Sheets("Sheet1").Range("A1")
Application.DDETerminate Channel
```

First, DDEInitiate is used to open an SCommand conversation with Spotfire S+. Next, DDEPoke is used to send the string located in the cell at **A1** in **Sheet1** of the current workbook from Excel to Spotfire S+. Spotfire S+ then executes this string, sending any output from the command either to the **Commands** window, if open, or to a **Report** window in Spotfire S+. Finally, the conversation is terminated using the DDETerminate command.

Like DDEExecute, DDEPoke must have a channel number specifying the conversation. The second parameter, called the *item text*, is ignored by Spotfire S+ when poking commands using an SCommand or [Execute] conversation. The item text parameter can be set to any value; VBA requires that it be set to some value. The third parameter is a cell reference telling Excel where to find the string representing the commands to send to Spotfire S+. Whatever is in that cell will be executed.

You can also use DDEPoke to send data from a DDE client program into an existing S-PLUS data object, such as a data frame or vector, when used in a conversation with the topic [DataSheet] or the name of the data object in which you want to change the data.

Consider the following example of DDEPoke in a VBA script in Excel.

```
Channel = Application.DDEInitiate("Spotfire S+", "exsurf")
Application.DDEPoke Channel, "r1c1:r3c2", Sheets("Sheet1").Range("A1:B3")
Application.DDETerminate Channel
```

First, a conversation is initiated with the topic name set to the name of an existing S-PLUS data object, in this case, the data frame exsurf. Next, the data in **Sheet1** in the range **A1** through **B3** of the current workbook in Excel are sent to the cells r1c1:r3c2 in exsurf, that is, to the cells starting at row 1, column 1 and extending to row 3, column 2. The statement Sheets("Sheet1").Range("A1:B3") is the Excel syntax for specifying the data for Excel to send to Spotfire S+; the

r1c1:r3c2 string specifies the item string of the DDEPoke command and tells Excel where to put the data in the `exsurf` data frame. Finally, the conversation is ended with a call to `DDETerminate`.

Getting Data From Spotfire S+

`DDEExecute` and `DDEPoke` let you send commands and data to Spotfire S+; with `DDERequest`, you can ask Spotfire S+ to send data back to the calling program.

When used in a conversation with the topic `SCommand` or `[Execute]`, the `DDERequest` command behaves in the same way as the `DDEExecute` command except that any output from the execution is sent back to the calling application; that is, it executes the commands specified and returns the result of the execution to the variable assigned to the `DDERequest` call.

Consider the following VBA script in Excel using `DDERequest` to execute commands in Spotfire S+ and return the results.

```
Channel = Application.DDEInitiate("Spotfire S+", "SCommand")
ReturnData = Application.DDERequest(Channel, "summary(corn.rain)")
Application.DDETerminate Channel

stringRangeToFillSpec = "A1" + ":" + "A" + Trim(Str(UBound(ReturnData)))
Sheets("Sheet1").Range(stringRangeToFillSpec).Value = _
    Application.Transpose(ReturnData)
```

First, `DDEInitiate` is used to open an `SCommand` conversation with Spotfire S+. Next, `DDERequest` is used to send the string `summary(corn.rain)` to Spotfire S+ for execution. The output from this command is passed back to Excel to the array `ReturnData`. Finally, the conversation is terminated using the `DDETerminate` command.

The string variable `stringRangeToFillSpec` is created using the upper bound of the `ReturnData` array (the call to `UBound(ReturnData)`) to specify the range of cells in **Sheet1** of the current workbook to receive the returned data. The `ReturnData` array is then transposed so that each element of the array is in a row of column **A** in **Sheet1**. **Sheet1**, column **A** now contains the summary data for the data frame `corn.rain`.

Note

The format of the text in an array returned by `DDERequest` in an `SCommand` or `[Execute]` conversation can be controlled via an option in Spotfire S+. By default, output is sent back using a method of most S-PLUS objects called `print`, which provides the best general purpose formatting for returned results. However, if you depend on the format as returned in earlier versions of S-PLUS, you can set this option to use the older formatting style. To do so, choose **Options ► General Settings** from the main menu. In the **DDE Server Support** group, select the **Old Format for DDE Request** check box.

Like `DDEExecute` and `DDEPoke`, `DDERequest` must have a channel number specifying the conversation. The item `text` parameter specifies the commands to execute in Spotfire S+ when using an `SCommand` or `[Execute]` conversation.

You can also use `DDERequest` to transfer data from an existing S-PLUS data object, such as a data frame or vector, into a DDE client program when used in a conversation with the topic `[DataSheet]` or the name of the data object from which you want to transfer the data.

Consider the following example of `DDERequest` in a VBA script in Excel.

```
Channel = Application.DDEInitiate("Spotfire S+", "exsurf")
ReturnData = Application.DDERequest(Channel, "r1c1:r3c2")
Application.DDETerminate Channel

sStartCell = "A1"
sSheetName = "Sheet1"

sRangeToFill = sStartCell + ":" + _
  Chr(Asc("A")) + (Trim(Str(UBound(ReturnData, 2))) - 1) + _
  Trim(Str(UBound(ReturnData, 1)))
Sheets(sSheetName).Range(sRangeToFill).Value = ReturnData
```

First, a conversation is initiated with the topic name set to the name of an existing S-PLUS data object, in this case, the data frame `exsurf`. Next, the data in the cells `r1c1:r3c2` in `exsurf`, that is, in the cells starting at row 1, column 1 and extending to row 3, column 2, are sent to Excel into the array `ReturnData`. The `r1c1:r3c2` string specifies the

item string of the DDERequest command and tells Spotfire S+ which cells from the data frame `exsurf` to send back to Excel. Finally, the conversation is ended with a call to `DDETerminate`.

Two strings are assigned values specifying the starting cell (**A1**) and the sheet (**Sheet1**) where the subsequent commands are to copy the data in the `ReturnData` array. Using the upper bounds of the `ReturnData` array (calls to `UBound`), the array of data is copied into the cells of the desired sheet.

Enabling and Disabling Response to DDE Requests

At any time during a Spotfire S+ session, you can suspend Spotfire S+'s response to messages sent from DDE client applications.

To temporarily suspend all links to any S-PLUS data objects, do the following:

1. From the main menu, choose **Options ► General Settings** to open the **General Settings** dialog with the **General** page in focus, as shown in Figure 7.1.

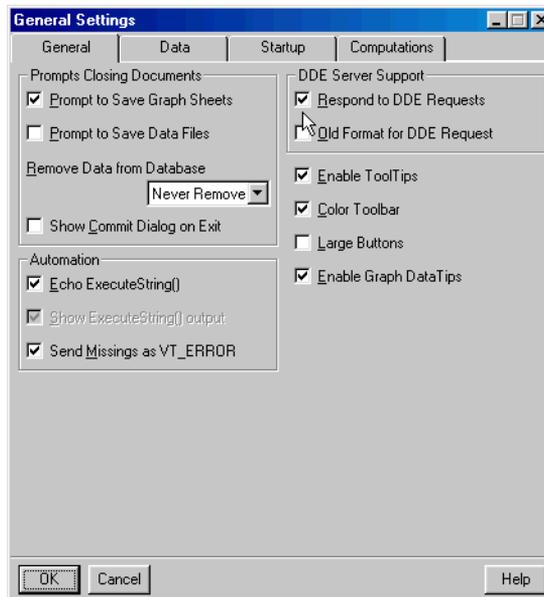


Figure 7.1: The **General** page of the **General Settings** dialog.

2. In the **DDE Server Support** group, deselect the **Respond to DDE Requests** check box.

EXTENDING THE USER INTERFACE

8

Overview	253
Motivation	253
Approaches	253
Architecture	254
Menus	255
Creating Menu Items	255
Menu Item Properties	256
Modifying Menu Items	260
Displaying Menus	262
Saving and Opening Menus	263
Toolbars and Palettes	264
Creating Toolbars	264
Toolbar Object Properties	265
Modifying Toolbars	267
Creating Toolbar Buttons	268
ToolbarButton Object Properties	269
Modifying Toolbar Buttons	271
Displaying Toolbars	273
Saving and Opening Toolbars	274
Dialogs	276
Creating Dialogs	278
Creating Property Objects	279
Property Object Properties	280
Modifying Property Objects	282
Creating FunctionInfo Objects	282
FunctionInfo Object Properties	284
Modifying FunctionInfo Objects	285
Displaying Dialogs	286
Example: The Contingency Table Dialog	287
Dialog Controls	290
Control Types	290

Copying Properties	303
ActiveX Controls in Spotfire S+ dialogs	307
Callback Functions	327
Interdialog Communication	329
Example: Callback Functions	329
Class Information	333
Creating ClassInfo Objects	333
ClassInfo Object Properties	334
Modifying ClassInfo Objects	336
Example: Customizing the Context Menu	336
Style Guidelines	341
Basic Issues	341
Basic Dialogs	342
Modeling Dialogs	351
Modeling Dialog Functions	357
Class Information	364
Dialog Help	366

OVERVIEW

In Spotfire S+, it is easy to create customized dialogs and invoke them with toolbar buttons and menu items. Similarly, menus and toolbars can be created and modified by the user. This chapter describes in detail how to create and modify the dialogs, menus, and toolbars which make up the interface.

Note
This chapter is dedicated to Windows® users.

Motivation

Users may be interested in customizing and extending the user interface to varying degrees. Possible needs include:

- Removing menu items and toolbars to simplify the interface.
- Changing menu item and toolbar layout.
- Creating new menu items or toolbars to launch scripts for repetitive analyses.
- Developing menus and dialogs for new statistical functions, either for personal use or for distribution to colleagues.

The tools for creating menus, toolbars, and dialogs in Spotfire S+ are quite flexible and powerful. In fact, all of the built-in statistical dialogs in Spotfire S+ are constructed using the tools described in this chapter. Thus, users have the ability to create interfaces every bit as sophisticated as those used for built-in functionality.

Approaches

This chapter discusses both point-and-click and command-based approaches for modifying the user interface. The point-and-click approach may be preferable for individual interface customizations, such as adding and removing toolbars. Programmers interested in sharing their interface extensions with others will probably prefer using commands, as this enables easier tracking and distribution of interface modifications.

Architecture The Spotfire S+ graphical user interface is constructed from *interface objects*. Menu items, toolbars, toolbar buttons, and dialog controls are all objects. The user modifies the interface by creating, modifying, and removing these objects.

The two components of extending the interface are:

- Creating user interface objects.
- Creating functions to perform actions in response to the interface.

The user interface objects discussed here are:

- MenuItem objects used to construct menus.
- Toolbar objects defining toolbars.
- ToolbarButton objects defining the buttons on toolbars.
- Property objects defining dialog controls, groups, and pages.
- FunctionInfo objects connecting dialog information to functions.
- ClassInfo objects describing right-click and context menu actions for objects in the Object Explorer.

The two types of functions used with the user interface are:

- *Callback functions* which modify dialog properties based on user actions within a dialog.
- Functions executed when **OK** or **Apply** is pressed in a dialog. While any standard S-PLUS function may be called from a dialog, we provide style guidelines describing the conventions used in the built-in statistical dialogs.

The following sections discuss these topics from both command and user interface perspectives.

MENUS

Menus are represented as a hierarchy of MenuItem objects. Each object has a type of Menu, MenuItem, or Separator:

- Menu creates a submenu.
- MenuItem causes an action to occur when selected.
- Separator displays a horizontal bar in the menu, visually separating two group of menu items.

Different menus may be created by modifying MenuItem objects. By default, the main menu in Spotfire S+ is SPlusMenuBar. A MenuItem may be added to or deleted from this menu to modify the interface. Alternately, users may create whole new menus which may be saved, opened, and used as the default menu.

A MenuItem is also used to construct context menus. These are the menus displayed when a user right-clicks on an object in the Object Explorer. Context menus are discussed in detail in the section Context Menu (page 365).

Creating Menu Items

A MenuItem may be created using commands or from within the Object Explorer.

Using Commands

To create a menu item, use the guiCreate function with classname="MenuItem". The name of the object will specify the location of the menu item in the menu hierarchy. Specify Type="Menu" for a menu item which will be the "parent" for another menu item, Type="MenuItem" for a menu item which performs an action upon select, or Type="Separator" for a separator bar.

The following commands will create a new top-level menu item with two child menu items launching dialogs for the sqrt and lme functions:

```
guiCreate(classname="MenuItem",
  Name="$$SPlusMenuBar$MyStats",
  Type="Menu", MenuItemText="&My Stats", Index="11",
  StatusBarText="My Statistical Routines")
```

```
guiCreate(classname="MenuItem",
  Name="$$SPlusMenuBar$MyStats$Sqrt",
```

```
Type="MenuItem", MenuItemText("&Square Root...",  
Action="Function", Command="sqrt")
```

```
guiCreate(classname="MenuItem",  
Name="$$SPlusMenuBar$MyStats$Lme",  
Type="MenuItem", MenuItemText="Linear &Mixed Effects...",  
Action="Function", Command="lme")
```

See the section Menu Item Properties (page 256) for details regarding property names and values.

Using the Object Explorer

To create a menu item, first open the Object Explorer and filter by **MenuItem** to see the hierarchy of menu items. Navigate to the menu item above where the new menu item should appear. Right-click on this menu item, and select **Insert MenuItem** from the context menu. The **MenuItem Object** dialog shown in Figure 8.1 appears.

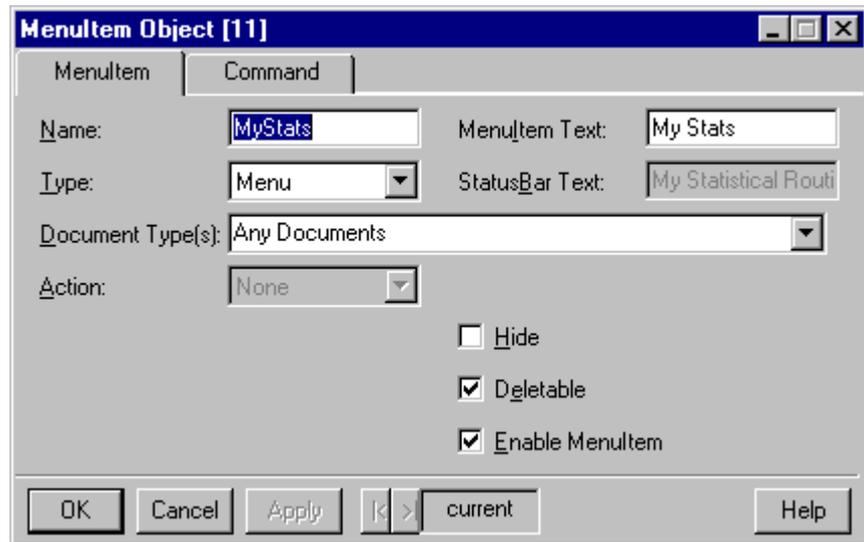


Figure 8.1: The *MenuItem* page of the *MenuItem Object* property dialog.

Menu Item Properties

The properties of a MenuItem object determine characteristics such as the prompt for the menu item and the action performed when the menu item is selected. These properties may be specified and modified using the property dialog for the MenuItem object, or programmatically via the commands `guiCreate` and `guiModify`. See the `guiCreate("MenuItem")` help file in the Language Reference help for syntax details.

The following properties are specified on the first tab of the **MenuItem Object** property dialog, shown in Figure 8.1:

Name The name of the MenuItem object. The name determines the menu to which the menu item belongs, and the position within the menu hierarchy.

Type The type of MenuItem object.

- **Menu** creates a submenu.
- **MenuItem** causes an action to occur when selected.
- **Separator** displays a horizontal bar in the menu, visually separating two group of menu items.

Document Type Depending on the type of document window type—**Graph Sheet, Commands** window, etc.—which has the focus, the item may or may not be visible. **Document Type** specifies the document types for which the item should be visible in the menu system. Selecting **All Documents** causes the item to be always visible. Selecting **No Documents** ensures that the item will be visible when no document window has the focus; for example, when no document window is open.

Action The following options apply to MenuItem objects of type **MenuItem**:

- **None.** No action is performed when the item is selected. This is useful when designing a menu system. It is not necessary to specify commands to execute when the type is set to **None**.
- **BuiltIn.** One of the actions associated with the default menus or toolbars is performed when the item is selected. These are listed on the Command page in the **Built-In Operation** dropdown box. This option allows you to use any of the “intrinsic” menu actions in a customized dialog, such as **Window ► Cascade**.
- **Function.** Under this option, a S-PLUS function, either built-in or user-created, is executed when the item is selected.
- **Open.** The file specified on the **Command** page is opened when this item is selected. The file will be opened by the application associated to it by the operating system.

- **Print.** The file specified on the **Command** page is printed when this item is selected. The file will be printed from the application currently associated to it by the operating system.
- **Run.** The file specified on the **Command** page is opened and run as a script by Spotfire S+ when this item is selected.

MenuItem Text The text which will represent the item in the menu system. This does not apply to **Separator** items. Include an ampersand (&) to specify the keyboard accelerator value.

StatusBar Text The text which will appear in the status bar when the item has the focus in the menu.

Hide Logical value indicating whether to make the item invisible. When the item is hidden, its icon in the Object Explorer appears grayed out. This can also be specified through the context menu.

Deletable Logical values indicating whether to allow the item to be deleted.

The rest of the properties are found on the **Command** page, as shown in Figure 8.2.

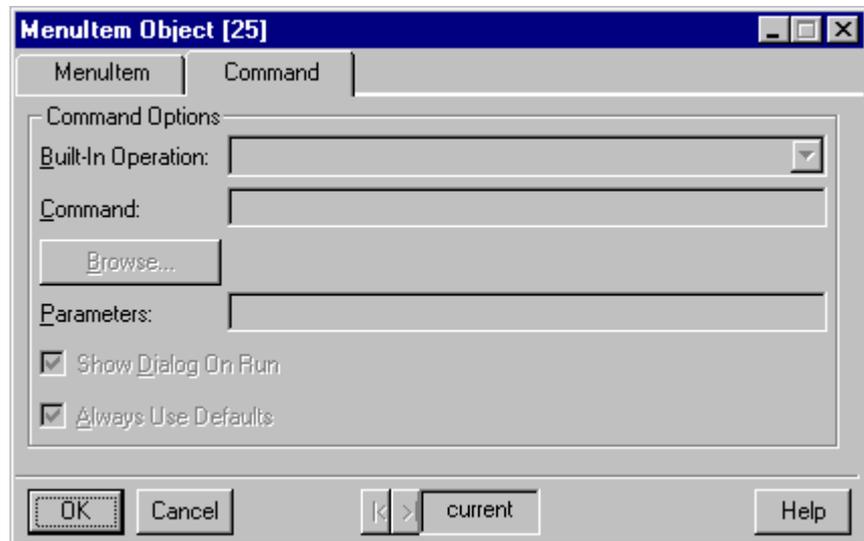


Figure 8.2: The *MenuItem Object* property dialog for a MenuItem object, Command page.

Built-In Operation desired when **Action** is set to **BuiltIn**. Built-in actions are actions performed by the interface such as launching the **Open File** dialog.

Command The name of a S-PLUS function, or a path and filename. This field is enabled when **Action** is set to **Function**, **Open**, **Print**, or **Run** on the **MenuItem** page. Use the Object Explorer to identify the folder.

Parameters This is relevant when **Action** is set to **Function**. This property specifies the arguments for the function which will execute when the item is selected. The easiest way to specify these arguments is to work through the **Customize** dialog available through the context menu for the **MenuItem** in the Object Explorer. For details on doing this, see the section Using the Context Menu (page 261) below.

Show Dialog On Run This is relevant when **Action** is set to **Function** on the Command page. It is a logical value which indicates whether the dialog associated to the function should open when the item is selected. This can also be specified through the context menu.

Always Use Defaults This is relevant when **Action** is set to **Function** on the Command page. Logical value indicating whether to force the use of the default values when the function executes. This can also be specified through the context menu.

Spotfire S+ makes a distinction between the default argument values for a function as defined in the function's dialog (via the `FunctionInfo` object) and as defined by the function itself. **Always Use Defaults** refers to the "dialog" defaults. Table 8.1 summarizes how **Show Dialog On Run** and **Always Use Defaults** work together. In it, "function" refers to the S-PLUS function associated to the menu item, and "dialog" refers to the dialog associated to that

function.

Table 8.1: Summary of *Show Dialog On Run* and *Always Use Defaults*.

Show Dialog On Run	Always Use Defaults	Action when the menu item is selected.
checked	checked	The dialog always opens in its default state when the menu item is selected. Changes are accepted, but do not persist as dialog defaults.
checked	unchecked	The dialog always opens when the menu item is selected. Changes are accepted and persist as dialog defaults.
unchecked	checked	The dialog does not appear and the function executes using the current dialog defaults.
unchecked	unchecked	The dialog will appear once; either when the menu item is selected or when Customize is selected from the menu item's context menu in the Object Explorer. After that, the dialog does not appear and the function executes using the current dialog defaults.

Modifying Menu Items

MenuItem objects can be modified using either programming commands, their property dialogs, or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If you are simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

Using Commands

The `guiModify` command is used to modify an existing menu item. Specify the name of the MenuItem to modify, and the properties to modify with their new values.

The following command will add status bar text for the “Square Root” dialog created in the section Creating Menu Items (page 255).

```
guiModify(classname="MenuItem",
    Name="$SPlusMenuBar$MyStats$Sqrt",
    StatusBarText="Calculate a square root.")
```

Using the Property Dialog

MenuItem objects can be modified through the same property dialogs which are used to create them. To modify a MenuItem object, open the Object Explorer to a page with filtering set to **MenuItem**. Right-click on the MenuItem object's icon in the right pane and choose **MenuItem** from the context menu. See the previous sections for details on using the property dialog.

Using the Context Menu

MenuItem objects can be modified with their context menus which are accessible through the Object Explorer. The following choices appear after right-clicking on a MenuItem object in the Object Explorer.

Insert MenuItem Select this to create a new MenuItem object.

Customize This appears when **Action** is set to **Function**. Select this to open the dialog associated to the function. Any changes to the dialog persist as dialog defaults.

Show Dialog On Run This appears when **Action** is set to **Function**. Check this to cause the dialog associated to the function to open when the item is selected. See Table 8.1 for details.

Always Use Defaults This appears when **Action** is set to **Function**. Check this to force the use of the default values when the function executes. See Table 8.1 for details. Spotfire S+ makes a distinction between the default argument values for a function as defined in the function's dialog (via the `FunctionInfo` object) and as defined by the function itself. **Always Use Defaults** refers to the "dialog" defaults.

Hide Select this to hide the menu item. It will not appear in the menu system and the MenuItem object icon will appear grayed out.

Delete Select this to delete the MenuItem object. The menu item will no longer be available.

Save Select this to save the MenuItem object (and any other MenuItem it contains in the menu hierarchy) to a file.

Save As Similar to **Save**, but this allows you to save a copy of the MenuItem object to a different filename.

MenuItem Select this to access the **MenuItem** page of the MenuItem object's property dialog.

Command Select this to access the **Command** page of the MenuItem object's property dialog.

Show Menu In Spotfire S+ Select this to cause the menu to be displayed in the main Spotfire S+ menu bar. This choice is available only for MenuItem objects having **Type Menu**.

Restore Default Menus Select this to restore the default Spotfire S+ menus in the main menu bar. For example, this will undo the effect of selecting **Show Menu In Spotfire S+**. This choice is available only for MenuItem objects having type **Menu**.

Save MenuItem Object as default Select this to make the MenuItem object the default. When a new MenuItem object is created, its property dialog will initially resemble that of the default object, except in the **Name** field.

Help Select this to open a help page describing MenuItem objects.

Manipulating Menu Items in the Object Explorer

Menu items are easily copied, moved, and deleted through the Object Explorer.

Moving Menu Items

To move a menu item into a different menu, locate the menu item icon in the Object Explorer. Select the icon, hold down the ALT key, and drag it onto the menu where it will be added.

To move the menu item within its current menu, hold down the SHIFT key and drag the menu item icon to the desired location.

Copying Menu Items

To copy a menu item into a different menu, hold down the CTRL key and drag its icon onto the menu to which it is to be added.

To copy a menu item within its current menu, hold down the SHIFT and CTRL keys and drag the menu item icon to the desired location.

Deleting Menu Items

To delete a menu item, right-click on the menu item in the Object Explorer and select **Delete** from the context menu.

Displaying Menus

If the user modifies the default menu, which by default is named **SPlusMenuBar**, the modifications will be displayed upon changing the window in focus. If the user creates a new menu, the menu must be explicitly displayed in Spotfire S+. This may be done programmatically or in the Object Explorer.

Using Commands The function `guiDisplayMenu` will display the specified menu as the main menu in Spotfire S+. As a simple example, we can set the context-menu for `lm` to be the main menu bar, and then restore the menus to the default of `SPlusMenuBar`:

```
guiDisplayMenu("lm")
guiDisplayMenu("SPlusMenuBar")
```

Using the Object Explorer After creating a menu system, right-click on the `MenuItem` object in the Object Explorer that you want used as the main menu. Select **Show Menu In Spotfire S+** from the context menu to display the menu system.

To restore the default Spotfire S+ menus, select **Restore Default Menus** in the context menu for that same `MenuItem` object. Alternatively, select **Show Menu In Spotfire S+** in the context menu for the `MenuItem` object which represents the default Spotfire S+ menus.

Saving and Opening Menus Menus may be saved as external files. These files may be opened at a later time to recreate the menu in Spotfire S+.

Using Commands The `guiSave` command is used to save a menu as an external file:

```
guiSave(classname="MenuItem", Name="SPlusMenuBar",
        FileName="MyMenu.smn")
```

The `guiOpen` command is used to open a menu file:

```
guiOpen(classname="MenuItem", FileName="MyMenu.smn")
```

Using the Object Explorer To save a menu to an external file, right-click on the `MenuItem` object in the Object Explorer and select **Save As** in the context menu. Enter a filename in the **Save As** dialog and click **OK**. The extension **.smn** is added to the filename.

To open a menu which has been saved in an external file, right-click on the default `MenuItem` object and select **Open** from the context menu. In the **Open** dialog, navigate to the desired file, select it, and click **OK**. The new menu is visible in the Object Explorer. Its name is the name of the external file, without the extension **.smn**.

TOOLBARS AND PALETTES

In Spotfire S+, toolbars and palettes represent the same type of object. When a toolbar is dragged into the client area below the other toolbars, it is displayed there as a palette. When a palette is dragged to the non-client area, close to a toolbar or menu bar, it “docks” there as a toolbar.

Toolbars are represented in the Object Explorer as `Toolbar` objects. These contain `ToolbarButton` objects which represent their buttons.

While it is not hard to create or modify toolbars through the user interface (as shown in this section), it is sometimes easier to work with toolbars programmatically using the `guiCreate` and `guiModify` commands. For details on the syntax, see the `guiCreate(“Toolbar”)` and `guiCreate(“ToolbarButton”)` help files in the Language Reference help.

Creating Toolbars

Toolbars may be created using commands or from within the Object Explorer.

Using Commands

To create a menu item, use the `guiCreate` function with `classname=“Toolbar”`.

The following command will create a new toolbar:

```
guiCreate(classname="Toolbar", Name="My Toolbar")
```

This will add a small empty toolbar which by default will be docked below the active document toolbar. Until we add buttons, the toolbar is not particularly interesting or useful.

Using the Object Explorer

To create a `Toolbar` object, first open the Object Explorer and filter by **Toolbar** to see the toolbars and toolbar buttons. To create a new toolbar, right-click on the default object icon (labeled **Toolbar**) in the left pane of the Object Explorer. Select **New Toolbar** from the context menu. (Alternatively, right-click in the Spotfire S+ application window, outside of any open document window, and choose **New Toolbar** from the context menu.) The **New Toolbar** dialog appears,

as shown in Figure 8.3.

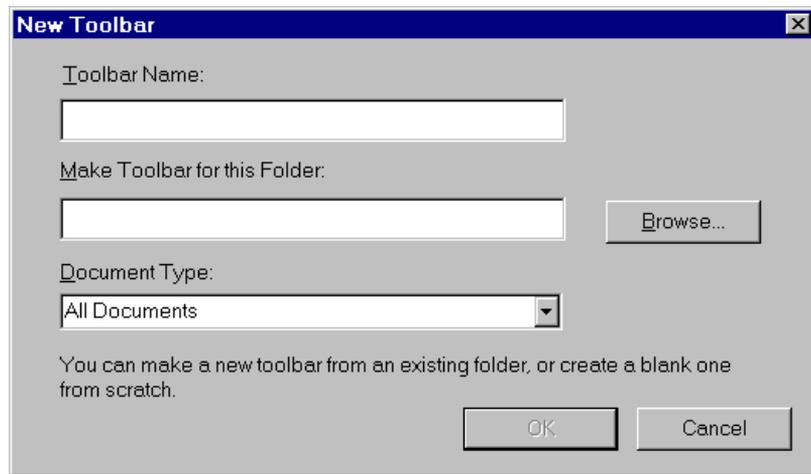


Figure 8.3: *New Toolbar dialog.*

To modify the default settings that appear in this property dialog in the future, right-click on the default object icon, choose Properties from the context menu, fill out the dialog with the desired defaults, and click **OK**.

Toolbar Name Enter a name for the new toolbar.

Make Toolbar for this Folder Enter a path for a folder (directory). The new toolbar will contain a toolbar button for each file in the indicated folder. Use the **Browse** button, if desired, to identify the folder. If no folder is specified, the toolbar will contain a single button with the `ToolbarButton` defaults.

Document Type Select the document types which will, when in focus, allow the toolbar to be visible.

Click **OK** and a new `Toolbar` object appears in the Object Explorer.

Toolbar Object Properties

The properties of a `Toolbar` object determine characteristics such as the name and location of the toolbar. These properties may be specified and modified using the property dialog for the `Toolbar` object, or programmatically via the commands `guiCreate` and `guiModify`. See the `guiCreate("Toolbar")` help file in the Language Reference help for syntax details.

The following properties are specified in the Toolbar property dialog, shown in Figure 8.4.

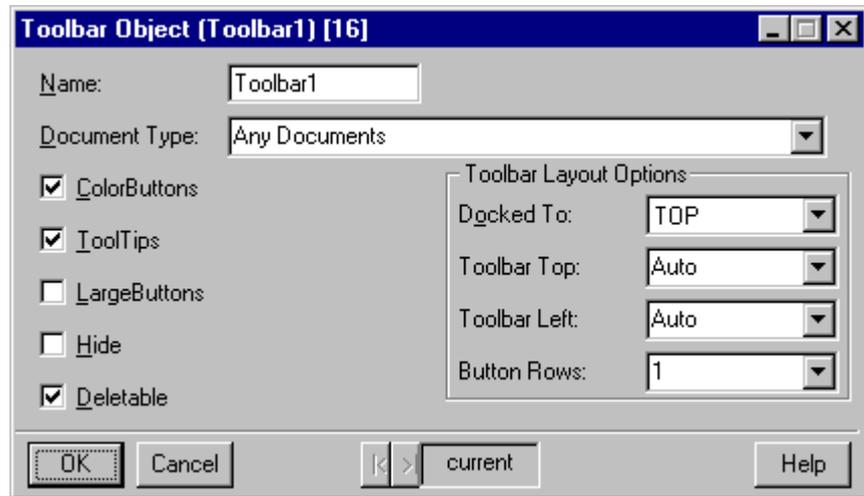


Figure 8.4: *Toolbar Object properties dialog.*

Document Type Depending on the type of document window—Graph Sheet, Commands window, etc.—which has the focus, a toolbar may or may not be visible. Specify the document types for which the toolbar should be visible. Selecting **All Documents** causes the toolbar to be always visible. Selecting **No Documents** ensures that the toolbar will be visible when no document window has the focus; for example, when no window is open.

ColorButtons Logical value indicating whether to display button images in color.

ToolTips Logical value indicating whether to enable tool tips for the toolbar.

LargeButtons Logical value indicating whether to display large-sized buttons.

Hide Logical value indicating whether to hide the toolbar.

Deletable Logical value indicating whether to allow permanent deletion of the toolbar.

Docked To The side of the Spotfire S+ window to which the toolbar will be docked, or **None** to float the toolbar as a palette.

Toolbar Top The top coordinate of the toolbar in pixels.

Toolbar Left The left coordinate of the toolbar in pixels.

Button Rows The number of rows of buttons in the toolbar.

Modifying Toolbars

Toolbar objects can be modified using either programming commands, their property dialogs, or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If you are simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

Using Commands

The `guiModify` command is used to modify an existing Toolbar. Specify the name of the Toolbar to modify and the properties to modify with their new values.

The position of the Toolbar on the screen is specified by the pixel location of the Top and Left corners of the Toolbar. The following command will automatically set these values so that the Toolbar is placed in the upper left corner of the screen:

```
guiModify(classname="Toolbar", Name="My Toolbar",  
          Top="Auto", Left="Auto")
```

Using the Property Dialog

Toolbar objects can be modified through the same property dialogs which are used to create them. To modify a Toolbar object, open the Object Explorer to a page with filtering set to **Toolbar**. Right-click on the Toolbar object's icon in the right pane and choose **Properties** from the context menu. See the previous sections for details on using the property dialog.

Using the Context Menu

Toolbar objects can be modified with their context menus which are accessible through the Object Explorer. The following choices appear after right-clicking on a Toolbar object in the Object Explorer.

New Toolbar Select this to open a new toolbar.

New Button Select this to add a new button to the toolbar.

Hide Select this to hide the toolbar.

Delete Select this to delete the toolbar.

Open Select this to open a toolbar that has been saved in an external file.

Save Select this to save a toolbar to its external file, when one exists.

Save As Select this to save a toolbar to an external file.

Unload Select this to unload a toolbar from memory. The toolbar is no longer available for display. To reload a built-in toolbar, restart Spotfire S+. To reload a toolbar that has been saved to an external file, open that file.

Restore Default Toolbar Select this to restore a built-in toolbar to its default state after it has been modified.

Properties Select this to display the property dialog for the Toolbar object.

Buttons Select this to display a dialog used for displaying or hiding different buttons on the toolbar.

Refresh Icons Select this to refresh the icon images on the toolbar buttons after they may have been modified.

Save Toolbar Object as default Save a modified version of a toolbar as the default for that toolbar.

Help Select this to display a help page on toolbars.

Creating Toolbar Buttons

A Toolbar generally contains multiple toolbar buttons, each of which performs an action when pressed. Toolbar buttons may be created using commands or from within the Object Explorer.

Using Commands To create a `ToolbarButton`, use the `guiCreate` function with `classname="ToolbarButton"`. The name of the button determines the toolbar upon which it is placed.

The following command creates a toolbar button which launches the Linear Regression dialog:

```
guiCreate( "ToolbarButton", Name = "My Toolbar$Linreg",  
          Action="Function", Command="menuLm" )
```

Creating sophisticated dialogs such as the Linear Regression dialog is discussed later in the section Dialogs (page 276) and the section Style Guidelines (page 341).

Toolbar buttons can also be used to call built-in Windows interface commands. The following command will create a toolbar button which launches the standard file **Open** dialog:

```
guiCreate( "ToolBarButton", Name = "My Toolbar$Open",
  Action="BuiltIn",
  BuiltInOperation="$$SPlusMenuBar$No_Documents$File$Open")
```

Using the Object Explorer

To add a button to an existing toolbar, right-click on the corresponding **ToolBar** object in the Object Explorer and select **New Button** from the context menu. The **ToolBarButton** property dialog appears, as in Figure 8.5.

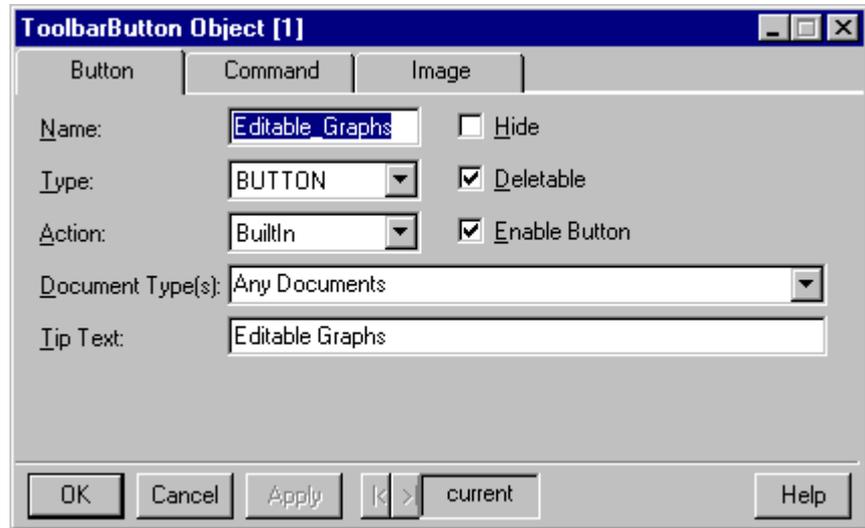


Figure 8.5: *ToolBarButton* property dialog.

ToolBarButton Object Properties

The properties of a **ToolBarButton** object determine characteristics such as the button image for the menu item and the action performed when the button is selected. These properties may be specified and modified using the property dialog for the **ToolBarButton** object, or programmatically via the commands `guiCreate` and `guiModify`. See the `guiCreate("ToolBarButton")` help file in the Language Reference help for syntax details.

The following properties are specified on the **Button** page of the **ToolBarButton** property dialog, shown in Figure 8.5:

Name The name of the button.

Type Select **BUTTON** to create a button, or select **SEPARATOR** to create a gap between buttons in the toolbar.

Action This applies to `ToolBarButton` objects of type **BUTTON**.

- **None.** No action is performed when the button is clicked.
- **BuiltIn.** One of the actions associated with the default menus or toolbars is performed when the item is selected. These are listed on the Command page in the **Built-In Operation** dropdown box. This option allows you to use in a customized toolbar any of the "intrinsic" menu or toolbar actions, such as **Window ► Cascade**.
- **Function.** An S-PLUS function is executed when the button is clicked. Optionally, the dialog for the function can be made to appear.
- **Open.** The file specified on the Command page is opened when the button is clicked. The file will be opened by the application associated to it by the operating system.
- **Print.** The file specified on the Command page is printed when the button is clicked. The file will be printed by the application currently associated to it by the operating system.
- **Run.** The file specified on the Command page is opened and run as a script by Spotfire S+ when the button is clicked.
- **Expression.** Enter a valid S-PLUS expression in the Command tab, and this expression is executed when the button is pressed.

Tip Text The tool tip text for the button.

Hide Logical value indicating whether to make the button invisible. When the item is hidden, its icon in the Object Explorer appears grayed out. This can also be specified through the context menu.

Deletable Logical value indicating whether to allow the item to be deleted.

The next set of properties are found on the **Command** page of the **ToolBarButton** property dialog.

Built-In Operation Type of action to perform when the button is selected.

Command The name of an S-PLUS function, or path and filename. This field is enabled when **Action** is set to **Function**, **Open**, **Print**, or **Run** on the button page. Use the **Browse** button to identify the folder.

Parameters This is relevant when **Action** is set to **Function**. This property specifies the arguments for the function which will execute when the item is selected. The easiest way to specify these arguments is to work through the **Customize** dialog available through the context menu for the **ToolBarButton** in the Object Explorer. For details on doing this, see the section Using the Context Menu (page 272) below.

Show Dialog On Run This is relevant when **Action** is set to **Function**. Logical value indicating whether to display the dialog associated with the specified function when the button is selected.

Always Use Defaults This is relevant when **Action** is set to **Function**. Logical value indicating whether to force the use of the default values when the function executes.

Spotfire S+ makes a distinction between the default argument values for a function as defined in the function's dialog (via the `FunctionInfo` object) and as defined by the function itself. **Always Use Defaults** refers to the *dialog* defaults. Table 8.1 above summarizes how **Show Dialog On Run** and **Always Use Defaults** work together.

The next set of properties are found on the **Image** page of the **ToolBarButton** property dialog.

Image FileName The path and filename of a bitmap file whose image will be displayed on the toolbar button. Use the **Browse** button, if desired, to identify the file.

To modify a `ToolBarButton` object, use either the **ToolBarButton** property dialog described above or the context menu, described below.

Modifying ToolBar Buttons

`ToolBarButton` objects can be modified using either programming commands, their property dialogs or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If you are simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

Using Commands

The `guiModify` command is used to modify an existing toolbar button. Specify the name of the `ToolBarButton` to modify, and the properties to modify with their new values.

The following command will specify a new value for the tooltip text, which is the text displayed when the mouse is hovered over the button:

```
guiModify( "ToolbarButton", Name = "My Toolbar$Open",  
          TipText="Open Document")
```

Using the Property Dialog

ToolbarButton objects can be modified through the same property dialogs which are used to create them. To modify a ToolbarButton object, open the Object Explorer to a page with filtering set to **Toolbar**. Right-click on the ToolbarButton object's icon in the right pane and choose **Button** from the context menu. See the previous sections for details on using the property dialog.

Using the Context Menu

ToolbarButton objects can be modified with their context menus which are accessible through the Object Explorer. The following choices appear after right-clicking on a ToolbarButton object in the Object Explorer.

Insert Button Select this to insert a new toolbar button next to the current one.

Customize This appears when **Action** is set to **Function**. Select this to open the dialog associated to the function. Any changes to the dialog persist as dialog defaults.

Hide Select this to hide the toolbar button.

Delete Select this to delete the toolbar button.

Edit Image Select this to open the bitmap file, using the operating systems default bitmap editor, which contains the icon image of the toolbar button.

Button. Select this to open the **Button** page of the property dialog for the toolbar button.

Command Select this to open the **Command** page of the property dialog for the toolbar button.

Image Select this to open the Image page of the property dialog for the toolbar button.

Save ToolbarButton Object as default Select this to save a copy of the ToolbarButton object as the default ToolbarButton object.

Help Select this to open a help page on toolbar buttons.

Manipulating Toolbars in the Object Explorer

Toolbar buttons are easily copied, moved, and deleted through the Object Explorer.

Displaying Toolbars

Using Commands The `Hide` property of a toolbar determines whether or not it is displayed. To display a toolbar, set this property to false:

```
guiModify(classname="Toolbar", Name="My Toolbar", Hide=F)
```

To hide the toolbar, set this property to true:

```
guiModify(classname="Toolbar", Name="My Toolbar", Hide=T)
```

Using the Toolbars Dialog

To hide (or unhide) a toolbar, right-click on the `Toolbar` object and select **Hide** (or **Unhide**) from the context menu. To selectively hide or display toolbars, right-click outside of any open windows or toolbars and select **Toolbars** from the context menu. A dialog like that shown in Figure 8.6 appears. Use the checkboxes to specify which toolbars will be visible.

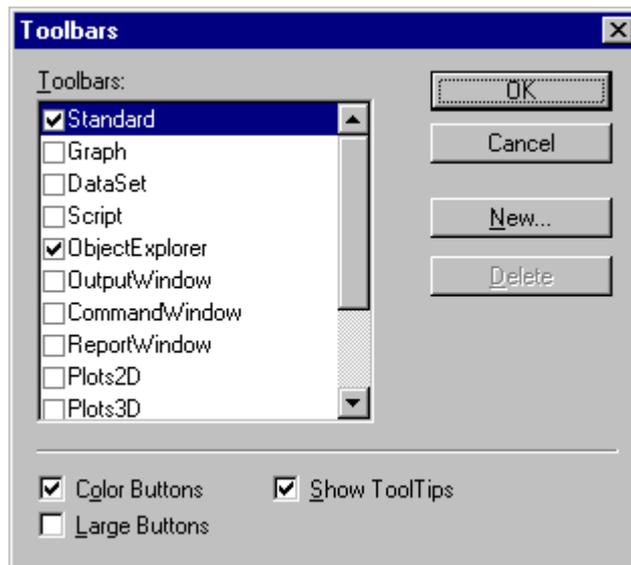


Figure 8.6: *The `Toolbar` dialog.*

To hide (or unhide) a toolbar button, right-click on the `ToolbarButton` object and select **Hide** (or **Unhide**) from the context menu. To selectively hide or display the buttons in a toolbar, right-click the

Toolbar object and select **Buttons** from the context menu. A dialog similar to that shown in Figure 8.7 appears. Use the checkboxes to specify which buttons will be visible in the toolbar.

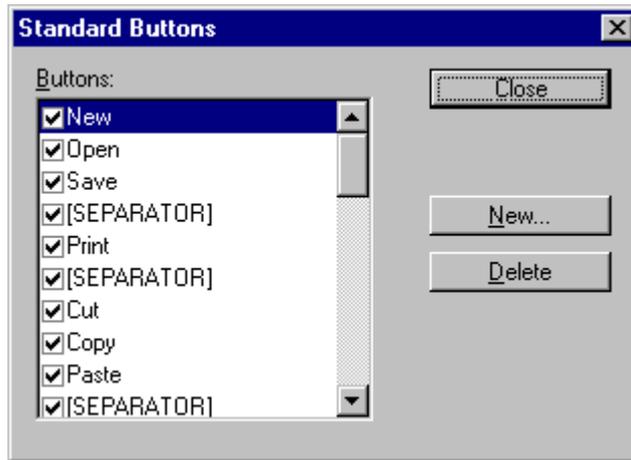


Figure 8.7: The **Buttons** dialog.

Saving and Opening Toolbars

A toolbar and the related toolbar buttons may be saved to an external file. This file may be opened at a later time to restore the toolbar and the toolbar buttons.

Using Commands

The `guiSave` command is used to save a toolbar as an external file:

```
guiSave(classname="Toolbar", Name="My Toolbar",  
        FileName="MyToolbar.stb")
```

The `guiOpen` command is used to open a toolbar file:

```
guiOpen(classname="Toolbar", FileName="MyToolbar.stb")
```

Note

Do not try to open a toolbar file while the toolbar it represents is loaded into Spotfire S+; this results in an error message. You can see which toolbars are currently loaded by right-clicking in the Spotfire S+ window outside of any open document windows. To unload a toolbar, go to the Object Explorer, right-click on the toolbar item, and choose **Unload**.

Using the Object Explorer

To save a toolbar to an external file, right-click on the **Toolbar** object in the Object Explorer and select **Save As** in the context menu. Enter

a filename in the **Save As** dialog and click **OK**. The extension **.STB** is added to the filename.

To open a toolbar which has been saved in an external file, right-click on the default **Toolbar** object and select **Open** from the context menu. In the **Open** dialog, navigate to the desired file, select it, and click **OK**. The new toolbar is visible in the Object Explorer. Its name is the name of the external file, without the extension **.STB**.

DIALOGS

Almost all of the dialogs in Spotfire S+ have either a corresponding graphical user interface object or a corresponding function.

The dialog for a GUI object such as a `BoxPlot` displays the properties of the object, and allows the modification of these properties. When **Apply** or **OK** is pressed, the object is then modified to have the newly specified properties. While these dialogs are created using the same infrastructure as is discussed here, they are not generally modified by the user.

The dialog for a function allows the user to specify the arguments to the function. The function is then called with these arguments when **Apply** or **OK** is pressed. In Spotfire S+, users may write their own functions and create customized dialogs corresponding to the functions. This section discusses the creation of such dialogs.

Think of a function dialog as the visual version of some S-PLUS function. For every function dialog there is one S-PLUS function, and for every S-PLUS function there is a dialog. The dialog controls in the dialog correspond to arguments in the function, and vice versa. In addition, all function dialogs are displayed with **OK**, **Cancel**, **Apply** (modeless) buttons that do not have any corresponding arguments in the functions. When the **OK** or **Apply** button is pressed, the function is executed with argument values taken from the current values of the dialog controls.

A dialog typically consists of one to five tabbed pages, each containing groups of controls. The characteristics of the controls in the dialog are defined by **Property** objects. Properties may be of type **Page**, **Group**, or **Normal**. A **Page** will contain **Groups** which in turn contain **Normal** properties. The primary information regarding **Pages** and **Groups** is their name, prompt, and what other properties they contain. Normal properties have far more characteristics describing features such as the type of control to use, default values, option lists, and whether to quote the field's value when providing it in the function call. Together the **Property** objects determine the look of the dialog and its controls.

Filter by **Property** in the Object Explorer (Figure 8.8) to see objects of this type.

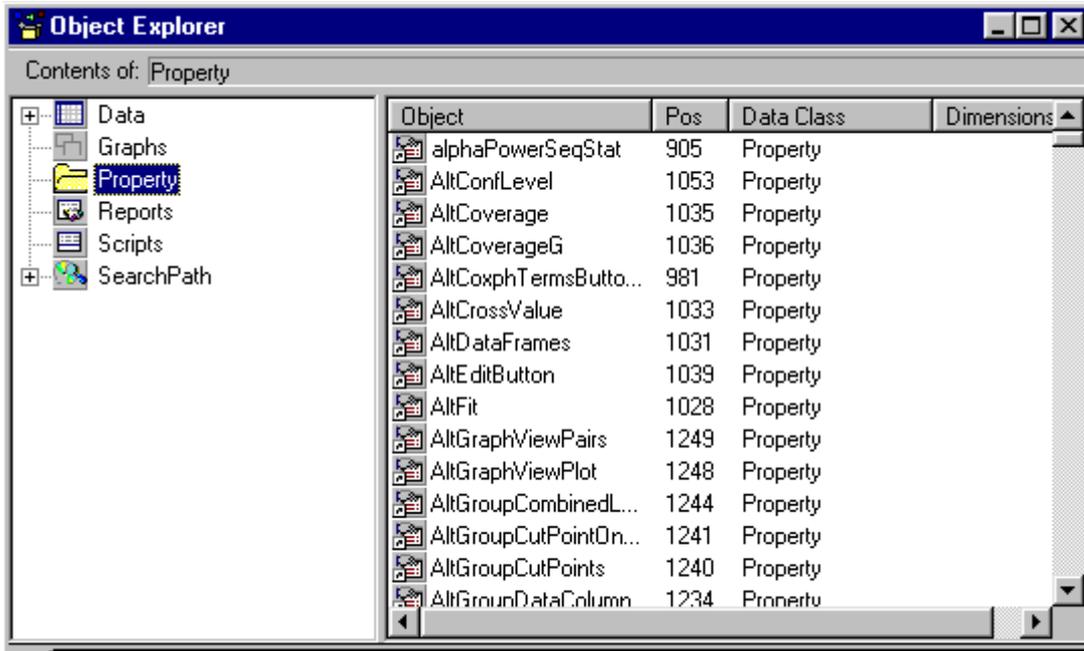


Figure 8.8: *The Object Explorer showing all Property objects*

While the Property objects define the controls in a dialog, they do not contain information on which Property objects relate to each of the arguments in the function. This information is contained in a FunctionInfo object. Each function for which a dialog is constructed needs a FunctionInfo object describing what Property objects to use when constructing the dialog for the function, as well as other related information. If a function does not have a FunctionInfo object and its dialog is requested by a MenuItem or ToolbarButton, a simple default dialog will be launched in which an edit field is present for each argument to the function.

Filter by **FunctionInfo** in the Object Explorer (Figure 8.9) to see objects of this type.

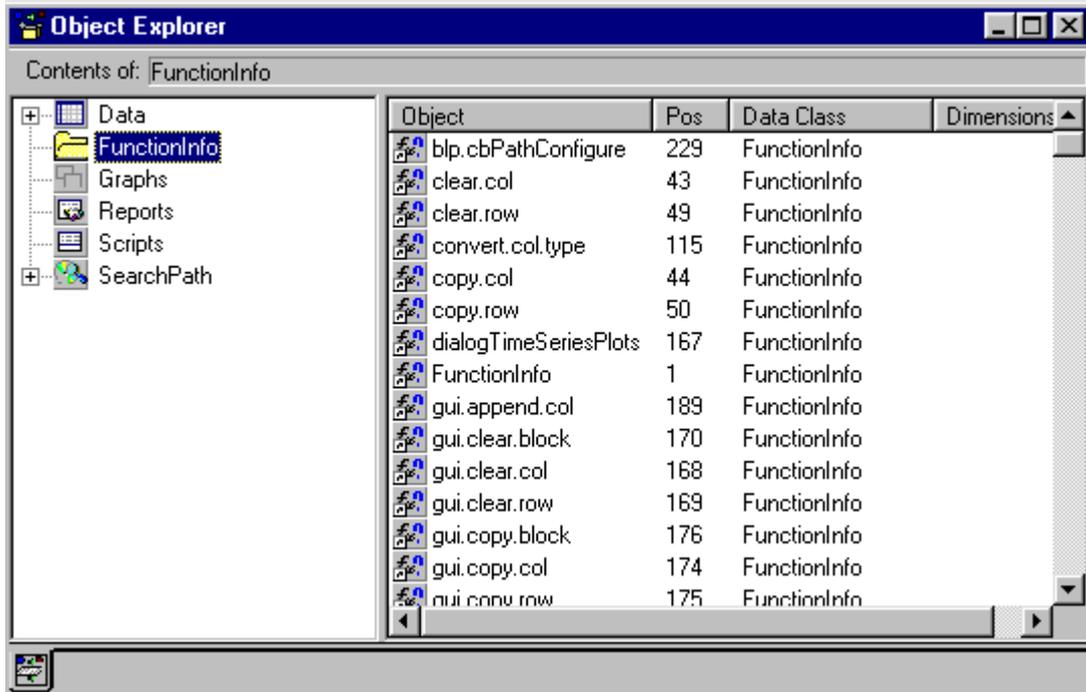


Figure 8.9: The Object Explorer showing all *FunctionInfo* objects.

While it is not hard to create or modify Property and FunctionInfo objects through the user interface, as shown in this section, it is usually preferable to work with them programmatically using the `guiCreate` and `guiModify` commands. For details on the syntax, see the `guiCreate("Property")` and `guiCreate("FunctionInfo")` help files in the Language Reference help.

Creating Dialogs

To create a dialog in Spotfire S+, follow these steps:

1. Identify the S-PLUS function which will be called by the dialog. This can be either a built-in or a user-created function.
2. Create the "Property" objects, such as pages, group boxes, list boxes, and check boxes, which will populate the dialog.

3. Create a `FunctionInfo` object having the same name as the function in step 1. The `FunctionInfo` object holds the layout information of the dialog, associates the values of the `Property` objects in the dialog with values for the arguments of the S-PLUS function, and causes the S-PLUS function to execute.

Creating Property Objects

Property objects may be created using commands or from within the Object Explorer.

Using Commands

To create a `Property` object, use `guiCreate` with `classname="Property"`. The following command creates a list box:

```
guiCreate(classname="Property", Name="MyListProperty",
          Type="Normal", DialogControl="List Box",
          DialogPrompt="Method", OptionList=c("MVE", "MLE",
          "Robust"), DefaultValue="MLE")
```

Using the Object Explorer

To create a `Property` object, open the Object Explorer to a page with filtering set to **Property**. Right-click on any property in the right pane and choose **Create Property** from the context menu. The property dialog shown in Figure 8.10 appears.

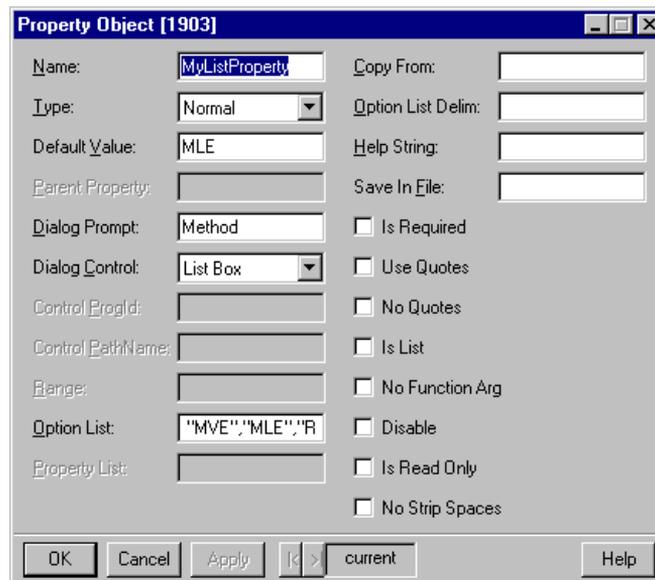


Figure 8.10: *The property dialog for a **Property** object.*

Property Object Properties

The properties of a Property object determine characteristics such as the prompt text, control type, and default value. These properties may be specified and modified using the property dialog for the Property object, or programmatically via the commands `guiCreate` and `guiModify`. See the `guiCreate("Property")` help file in the Language Reference help for syntax details.

The following properties are specified in the Property object property dialog, shown in Figure 8.10.

Name The name of the Property object. To create a Property object, a name must be specified.

Type The type of property. `Group` or `WideGroup` creates a group box. `Page` creates a tabbed page. `Normal` creates any other type of Property object.

Default Value The default value for the Property object. This will be displayed when the dialog opens.

Parent Property The name of a parent property, if any. This is used by certain internal Property objects.

Dialog Prompt The text for the label which will appear next to the control in the dialog.

Dialog Control The type of control to use. Examples are `Button`, `Check Box`, `List Box`, and `Combo Box`. Control types are described in the section `Dialog Controls` (page 290).

Range The range of acceptable values for the function argument associated with this property. For instance, if the values must be between 1 and 10, enter `1:10`.

Option List A comma-separated list. The elements of the list are used, for example, as the labels of `Radio Buttons` or as the choices in the dropdown box of a `String List Box`. A property may have either a range or an option list, but not both. Ranges are appropriate for continuous variables. Option lists are appropriate where there is a finite list of allowable values.

Property List A comma-separated list of the Property objects included in the Group box or on the Page. This applies to Property objects having Type Page or Group.

Tip...
A Property object may only be called once by a given <code>FunctionInfo</code> object.

Copy From The name of a Property object to be used as a template. The current Property object will differ from the template only where specified in the property dialog. See the section Dialog Controls (page 290) for lists of internal and standard Property objects that can be used in dialogs via **Copy From**.

Option List Delim A character used as the delimiter for Option List, such as comma, colon or semi-colon. Comma is the default.

Help String The text of the tool tip for this Property object.

Save in File The name of the file in which to save the Property definition.

Is Required Logical value indicating whether to require the Property object to have a value when **OK** or **Apply** is clicked in the dialog.

Use Quotes Logical value indicating whether to force quotes to be placed around the value of the Property object when the value is passed to the S-PLUS function.

No Quotes Logical value indicating whether to prohibit quotes from being placed around the value of the Property object when the value is passed to the S-PLUS function. This option is ignored when Is List (described below) is not checked.

Is List Logical value indicating whether to cause a multiple selection in a drop-down list to be passed as an S-PLUS list object to the S-PLUS function.

No Function Arg Logical value indicating whether to not pass the value of this Property object as an argument to the S-PLUS function. The Property object must still be referenced by the `FunctionInfo` object.

Disable Logical value indicating whether to cause the Property object to be disabled when the dialog starts up.

Is Read Only Logical value indicating whether the corresponding control is for read only.

No Strip Spaces Logical value indicating whether to include or remove spaces between elements in the Option List.

Modifying Property Objects

Property objects can be modified using either programming commands, their property dialogs, or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If you are simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

Using Commands

The `guiModify` command is used to modify an existing Property object. Specify the Name of the Property object to modify, and the properties to modify with their new values.

```
guiModify(classname="Property", Name="MyListProperty",  
          DefaultValue="Robust")
```

Using the Property Dialog

Property objects may be modified through the Property object property dialog.

To modify a Property object, open the Object Explorer to a page with filtering set to **Property**. Right click on the Property object's icon in the right pane and choose **Properties** from the context menu. Refer to the previous sections for details on using the property dialog.

Using the Context Menu

Property objects can be modified with their context menus. The context menu for an object is launched by right-clicking on the object in the Object Explorer. The context menu provides options such as creating, copying, and pasting the object, as well as a way to launch the property dialog.

Creating FunctionInfo Objects

FunctionInfo objects may be created using commands or from within the Object Explorer.

Using Commands

To create a FunctionInfo object, use the `guiCreate` command with `classname="FunctionInfo"`.

As a simple example, we will create a function `my.sqrt` which calculates and prints the square root of a number. We will create a dialog for this function and add a menu item to the **Data** menu which launches the dialog. We will create a property `MySqrtInput` specifying the input value, and since we don't want to store the result, we will use the standard property `SPropInvisibleReturnObject` for the result.

```
my.sqrt <- function(x){
  y <- sqrt(x)
  cat("\nThe square root of ",x," is ",y, ".\n",sep="")

  invisible(y)
}

guiCreate(classname="Property", Name="MySqrtInput",
  DialogControl="String", UseQuotes=F,
  DialogPrompt="Input Value")
guiCreate(classname="FunctionInfo", Name="my.sqrt",
  DialogHeader="Calculate Square Root",
  PropertyList="SPropInvisibleReturnObject, MySqrtInput",
  ArgumentList="#0=SPropInvisibleReturnObject,
#1=MySqrtInput")
guiCreate(classname="MenuItem",
  Name="$$SPlusMenuBar$Data$MySqrt",
  Type="MenuItem",MenuItemText="Square Root...",
  Action="Function", Command="my.sqrt")
```

Using the Object Explorer

Open the Object Explorer to a page with filtering set to **FunctionInfo**. Right-click on any `FunctionInfo` object in the right pane and choose **Create FunctionInfo** from the context menu. The property dialog shown in Figure 8.11 appears.

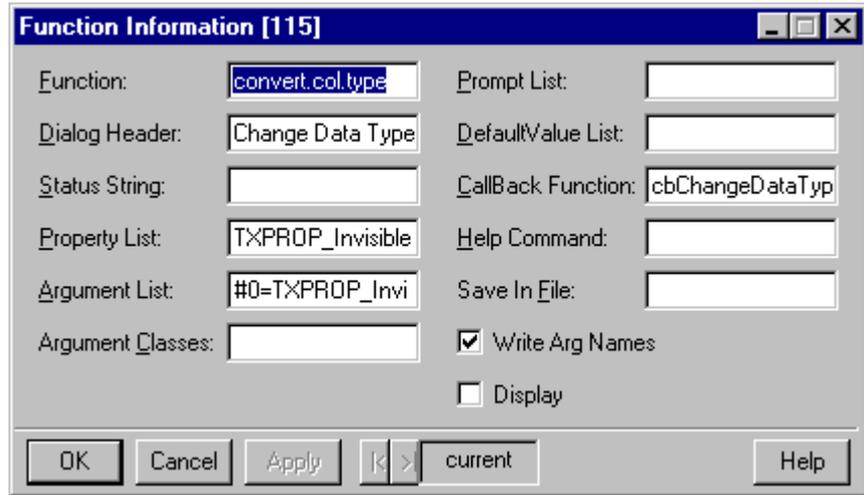


Figure 8.11: The property dialog for a `FunctionInfo` object.

FunctionInfo Object Properties

The properties of a `FunctionInfo` object describe how the properties in a dialog correspond to the related function. These properties may be specified and modified using the property dialog for the `FunctionInfo` object, or programmatically via the commands `guiCreate` and `guiModify`. See the `guiCreate("FunctionInfo")` help file in the Language Reference help for syntax details.

The following properties are specified in the `FunctionInfo` object property dialog, shown in Figure 8.11.

Function The name of the S-PLUS function which will execute when **OK** or **Apply** is clicked in the dialog. This is also the name of the `FunctionInfo` object.

Dialog Header The text that will appear at the top of the dialog.

Status String The string displayed when you move the mouse over the property in the dialog.

Property List A comma-separated list of `Property` objects to be displayed in the dialog. A given `Property` object can only occur once in this list. If pages or group boxes are specified, it is not necessary to

specify the Property objects that they comprise. Property objects in the list will be displayed in two columns, moving in order from top to bottom, first in the left-hand column and next in the right-hand column.

Argument List A comma-separated list in the form #0 = *PropName1*, #1 = *PropName2*, Here *PropName1*, *PropName2*, ..., are names of Property objects, not including page and group objects, and #1, ..., refer in order to the arguments of the function indicated in **Function Name**. The argument names may be used in place of #1, #2, The first item, #0, refers to the returned value of the function. Use **Argument List** if the order of the Property objects in the dialog differs from the order of the corresponding arguments of the S-PLUS function.

Argument Classes A comma-separated list of classes that are used in the dialog.

Prompt List A comma-separated list of labels for the Property objects in the dialog. These will override the default labels. The syntax for this list is the same as that for **Argument List**.

Default Value List A comma-separated list of default values for the Property objects. These will override the default values of the Property objects. The syntax for this list is the same as that for **Argument List**.

Callback Function The name of a function which will be executed on exit of any Property object in the dialog. **Callback Functions** are described in detail in the section **Callback Functions** (page 327).

Help Command The command to be executed when the Help button is pushed. This is a Spotfire S+ expression such as "help(my.function)".

Save in File The function information can be written to a file, which can be edited in the Command line or in the GUI.

Write Argument Names Logical value indicating whether to have argument names written when the function call is made.

Display Logical value indicating whether to cause information about the **FunctionInfo** object to be written in a message window (or in the output pane of a script window when the dialog is launched by a script). This debugging tool is turned off after **OK** or **Apply** is clicked in the dialog.

Modifying FunctionInfo Objects

FunctionInfo objects can be modified using either programming commands, their property dialogs, or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If

you are simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

Using Commands The `guiModify` command is used to modify an existing `FunctionInfo` object. Specify the Name of the `FunctionInfo` object to modify, and the properties to modify with their new values.

```
guiModify(classname="FunctionInfo", Name="my.sqrt",  
          DialogHeader="Compute Square Root")
```

Using the Property Dialog

`FunctionInfo` objects may be modified through the `FunctionInfo` object property dialog.

To modify a `FunctionInfo` object, open the Object Explorer to a page with filtering set to **FunctionInfo**. Right click on the `FunctionInfo` object's icon in the right pane and choose Properties from the context menu. Refer to the previous sections for details on using the dialog.

Using the Context Menu

`FunctionInfo` objects can be modified with their context menus. The context menu for an object is launched by right-clicking on the object in the Object Explorer. The context menu provides options such as creating, copying, and pasting the object, as well as a way to launch the property dialog.

Displaying Dialogs

There are several ways to display a dialog in Spotfire S+.

- Locate the associated function in the Object Explorer and double-click on its icon. If a function is not associated with a `FunctionInfo` object, then double-clicking on its icon will cause a default dialog to be displayed.
- Click on a toolbar button which is linked to the associated function.
- Select a menu item which is linked to the associated function.
- Use the function `guiDisplayDialog` in a **Script** or **Commands** window:

```
guiDisplayDialog("Function",Name="menuLm")
```

- Write the name of the function in a **Script** window, double-click on the name to select it, right-click to get a menu, and choose **Show Dialog**.

Example: The Contingency Table Dialog

This example looks into the structure behind the **Contingency Table** dialog. The Contingency Table dialog in Spotfire S+ (Figure 8.12) is found under **Statistics ► Data Summaries ► Crosstabulations**.

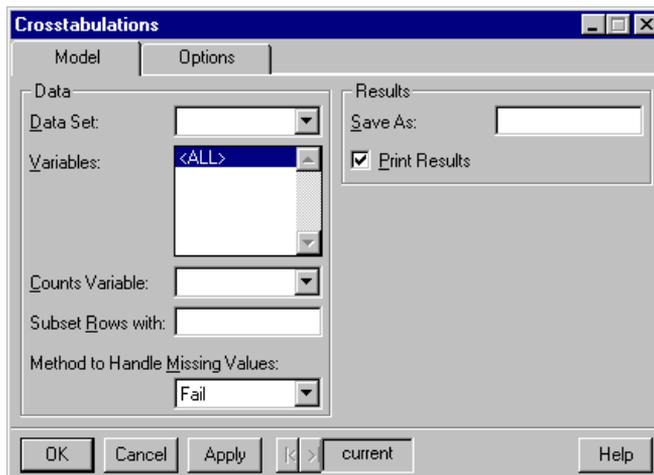


Figure 8.12: *The Contingency Table dialog.*

It has two tabbed pages named **Model** and **Options**. On the **Model** page are two group boxes, named **Data** and **Results**.

The FunctionInfo object for this dialog is called menuCrosstabs; its property dialog is shown in Figure 8.13 and is described below.

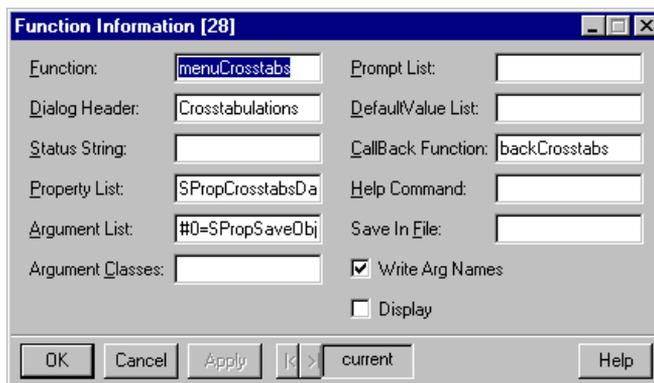


Figure 8.13: *The property dialog for the FunctionInfo object menuCrosstabs.*

Function Notice that this value is also menuCrosstabs; the S-PLUS function associated with this dialog has the same name as the FunctionInfo object. To look at the code behind the function

menuCrosstabs, type menuCrosstabs, or page(menuCrosstabs) at the prompt in the Commands window.

Dialog Header This is the header which appears at the top of the Contingency Table dialog. Try changing this and opening the dialog. The dialog will reflect the change. This change persists when Spotfire S+ is exited and restarted.

Status String This is currently empty. Try entering text here (do not forget to click **Apply** or **OK**) and opening the dialog.

Property List This shows only the Property objects for the two tabbed pages: SPropCrosstabsDataPage and SPropCrosstabsOptionsPage. To more easily see these values, right-click in the edit field and select **Zoom**. The zoom box shown in Figure 8.14 appears.

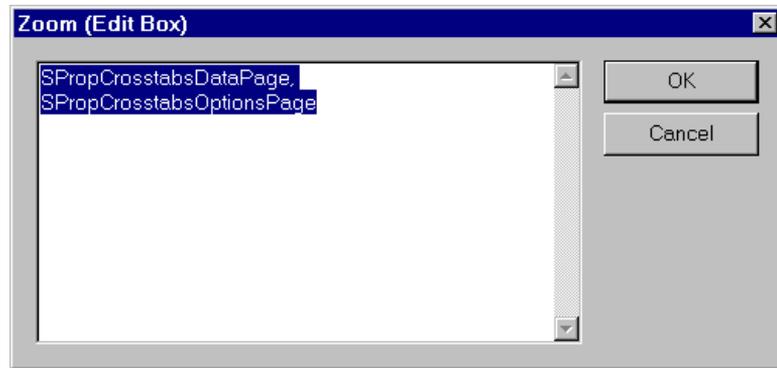


Figure 8.14: *The Zoom box shows the Property List.*

Using the Object Explorer, open the property dialog for the first of these. This is shown in Figure 8.15.

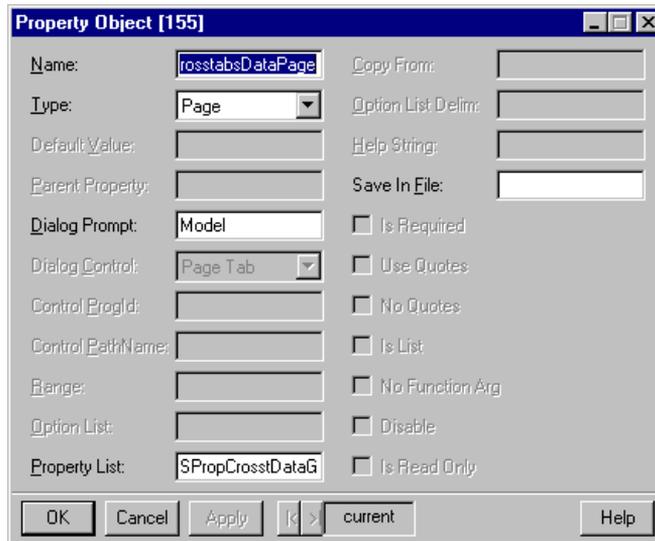


Figure 8.15: *The Property dialog for the SPropCrosstabsDataPage Property object.*

Argument List Use zoom, if desired, to view the assignments of Property object values to arguments of the function menuCrosstabs. Notice in Figure 8.13 that the return value is set to **SPropSaveObj**. This has been done consistently throughout the user interface.

Prompt List Since this is empty, fields in the dialog will have their default prompts (labels) as specified in their corresponding property objects.

Default Value List Since this is empty, fields in the dialog will have the default values as specified in their corresponding property objects.

Call Back Function The S-PLUS function backCrosstabs is executed each time a control in the dialog is exited. To look at the code behind the function, type

```
> backCrosstabs
```

at the prompt in the Commands window. Callback functions are discussed in the section Callback Functions (page 327).

Write Arg Names This is currently empty.

Display This is not checked, so debugging messages will not be shown when the dialog is displayed.

DIALOG CONTROLS

Control Types Spotfire S+ has a variety of dialog controls that can be used to represent the properties of an object (such as a user-defined function) in a dialog, which are described in Table 8.2. Note that the control type (first column) and the `DialogControl` (in the **Example** column) in Table 8.2 must be exactly the same when you use them in the `guiCreate` function. For more information on dialog controls, see the `guiCreate("Property")` help file in the Language Reference help.

Table 8.2: *Dialog control types.*

Control Type	Description	Example
Invisible	A control which does not appear on the dialog.	<pre>guiCreate("Property", Name = "ReturnValue", Type = "Normal", DialogControl = "Invisible")</pre>
Button	<p>A push-button control.</p> <p>The “DialogPrompt” subcommand is used to set the text inside the button.</p>	<pre>guiCreate("Property", Name = "myButton", Type = "Normal", DialogPrompt = "&MyButton", DialogControl = "Button")</pre>
Check Box	<p>A two-state check box control where one state is checked and the other is unchecked.</p> <p>The “DefaultValue” subcommand is used to set the state of the check box. If set to “0” or “F”, the box will be unchecked. If “1” or “T”, the box will be checked.</p>	<pre>guiCreate("Property", Name = "myCheckBox", Type = "Normal", DefaultValue = "T", DialogPrompt = "&My CheckBox", DialogControl = "Check Box")</pre>
Static Text	<p>A text field that is not editable usually used before other controls to title them.</p> <p>The “DialogPrompt” subcommand is used to specify the text of this static text field.</p>	<pre>guiCreate("Property", Name = "myStaticText", Type = "Normal", DialogPrompt = "MyStaticText", DialogControl = "Static Text")</pre>
String	<p>An editable field used to enter text.</p> <p>If the subcommand “UseQuotes” is set to TRUE, the string returned to the user function from this dialog has quotes around it. If not specified, no quotes are added.</p>	<pre>guiCreate("Property", Name = "myString", Type = "Normal", DialogControl = "String", DialogPrompt = "&My String", UseQuotes=T)</pre>

Table 8.2: *Dialog control types.*

Control Type	Description	Example
Hidden String	Same as “String” except that this control is hidden.	<pre>guiCreate("Property", Name = "myHiddenString", Type = "Normal", DialogControl = "Hidden String", DialogPrompt = "&My Hidden String", UseQuotes=T)</pre>
Wide String	Same as “String” except that this control takes up two dialog columns.	<pre>guiCreate("Property", Name = "myWideString", Type = "Normal", DialogControl = "Wide String", DialogPrompt = "&My String", UseQuotes=T)</pre>
Multi-line String	Same as “String” except that this control can accept strings with multiple lines.	<pre>guiCreate("Property", Name = "myMulti-line String", Type = "Normal", DialogControl = "Multi-line String", DialogPrompt = "&My Multi-line String", UseQuotes=T)</pre>
Wide Multi-line String	Same as “Multi-line String” except that this control takes up two dialog columns.	<pre>guiCreate("Property", Name = "myWide Multi-line String", Type = "Normal", DialogControl = "Wide Multi-line String", DialogPrompt = "&My Wide Multi-line String", UseQuotes=T)</pre>

Table 8.2: *Dialog control types.*

Control Type	Description	Example
List Box	A drop-list of strings. Only one string can be selected at a time. The selected string is not editable. The “DefaultValue” is used to specify the string from the list that is selected by default. The list of strings is specified as a comma-delimited list in “OptionList”. An optional subcommand “OptionListDelimiter” can be used to specify the delimiter.	<pre>guiCreate("Property", Name = "myListBox", Type = "Normal", DefaultValue = "Opt2", DialogPrompt = "A ListBox", DialogControl = "List Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>
Wide List Box	Same as “List Box” except that this control takes up two dialog columns.	<pre>guiCreate("Property", Name = "myWideListBox", Type = "Normal", DefaultValue = "Opt2", DialogPrompt = "Wide ListBox", DialogControl = "Wide List Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>
Sorted Wide List Box	Same as “Wide List Box” except that this control sorts columns.	<pre>guiCreate("Property", Name = "mySortedWideListBox", Type = "Normal", DefaultValue = "Opt2", DialogPrompt = "SortedWide ListBox", DialogControl = "Sorted Wide List Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>

Table 8.2: *Dialog control types.*

Control Type	Description	Example
Multi-select List Box	Similar to the “List Box” control except that multiple selections can be made from the string list.	<pre>guiCreate("Property", Name = "myMultiSelListBox", Type = "Normal", DefaultValue = "Opt2", DialogPrompt = "MultiSel ListBox", DialogControl = "Multi-select List Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>
Wide Multi-select List Box	Similar to the “Multi-select List Box” control except this control takes up two dialog columns.	<pre>guiCreate("Property", Name = "myWideMultiSelListBox", Type = "Normal", DefaultValue = "Opt2", DialogPrompt = "Wide MultiSel ListBox", DialogControl = "Wide Multi-select List Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>
Combo Box	Similar to a “List Box” control except that the selected string is editable. This allows the user to enter a string which is not part of the drop-list. Only one string can be selected at a time.	<pre>guiCreate("Property", Name = "myComboBox", Type = "Normal", DefaultValue = "Opt3", DialogPrompt = "A ComboBox", DialogControl = "Combo Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>
Wide Combo Box	Same as “Combo Box” except that this control takes up two dialog columns.	<pre>guiCreate("Property", Name = "myWideComboBox", Type = "Normal", DefaultValue = "Opt3", DialogPrompt = "Wide Combo Box", DialogControl = "Wide Combo Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>

Table 8.2: *Dialog control types.*

Control Type	Description	Example
Sorted Wide Combo Box	Same as “Wide Combo Box” except that this control sorts columns.	<pre>guiCreate("Property", Name = "mySortedWideComboBox", Type = "Normal", DefaultValue = "Opt2", DialogPrompt = "Sorted Wide Combo Box", DialogControl = "Sorted Wide Combo Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>
Multi-select Combo Box	Similar to the “Combo Box” control except that multiple selections can be made from the drop-list of strings.	<pre>guiCreate("Property", Name = "myMultiSelCombo", Type = "Normal", DefaultValue = "Opt3", DialogPrompt = "MultiSel Combo", DialogControl = "Multi-select Combo Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>
Wide Multi-select Combo Box	Similar to the “Multi-select Combo Box” control except this control takes up two dialog columns.	<pre>guiCreate("Property", Name = "myWideMultiSelCombo", Type = "Normal", DefaultValue = "Opt3", DialogPrompt = "Wide MultiSel Combo", DialogControl = "Wide Multi-select Combo Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>

Table 8.2: *Dialog control types.*

Control Type	Description	Example
Sorted Multi-select Combo Box	Similar to the “Multi-select Combo Box” control except this control sorts columns .	<pre>guiCreate("Property", Name = "mySortedMultiSelComboBox", Type = "Normal", DefaultValue = "Opt3", DialogPrompt = "Sorted Multi-Select Combo Box", DialogControl = "Sorted Multi-select Combo Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>
Sorted Wide Multi-select Combo Box	Similar to the “Wide Multi-select Combo Box” control except this control sorts columns.	<pre>guiCreate("Property", Name = "mySortedWideMultiSelComboBox", Type = "Normal", DefaultValue = "Opt3", DialogPrompt = "Sorted Wide MultiSel Combo Box", DialogControl = "Sorted Wide Multi-select Combo Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>
Float	Similar to a String control. This control accepts floating point numbers.	<pre>guiCreate("Property", Name = "myFloat", Type = "Normal", DefaultValue = "2.54", DialogPrompt = "A Float", DialogControl = "Float")</pre>
Float Auto	Similar to a ComboBox except that there is only one string “Auto” in the drop-list. You can enter a floating point number or select “Auto” from the drop list.	<pre>guiCreate("Property", Name = "myFloatAuto", Type = "Normal", DefaultValue = "2.54", DialogPrompt = "A FloatAuto", DialogControl = "Float Auto")</pre>

Table 8.2: Dialog control types.

Control Type	Description	Example
Float Range	Similar to the Float control except that a range of values can be specified using the “Range” subcommand. If the value entered is outside of the range, then an error will be displayed and the dialog will remain open.	<pre>guiCreate("Property", Name = "myFloatRange", Type = "Normal", DefaultValue = "2.54", DialogPrompt = "A Float Range", DialogControl = "Float Range", Range = "1.00:3.00")</pre>
Integer	Similar to the Float control. This control accepts integer whole numbers.	<pre>guiCreate("Property", Name = "myInteger", Type = "Normal", DefaultValue = "2", DialogPrompt = "An Int", DialogControl = "Integer")</pre>
Wide Integer	Same as “Integer” except this dialog control takes up two columns.	<pre>guiCreate("Property", Name = "myInteger", Type = "Normal", DefaultValue = "2", DialogPrompt = "An Int", DialogControl = "Wide Integer")</pre>
Integer Auto	Similar to a “Float Auto” control except this control accepts integers.	<pre>guiCreate("Property", Name = "myIntAuto", Type = "Normal", DefaultValue = "2", DialogPrompt = "An IntAuto", DialogControl = "Integer Auto")</pre>
Integer Range	Similar to a “Float Range” control except this control accepts integers.	<pre>guiCreate("Property", Name = "myIntRange", Type = "Normal", DefaultValue = "2", DialogPrompt = "An IntRange", DialogControl = "Integer Range", Range = "1:3")</pre>

Table 8.2: Dialog control types.

Control Type	Description	Example
Color List	A drop-list that allows selection of one element from a list of strings representing colors (i.e. Red, Green, etc.) Each list element has drawn a color sample next to it if the string represents a valid color name. Use the “OptionList” sub-command to specify colors in the list.	<pre>guiCreate("Property", Name = "myColorList", Type = "Normal", DefaultValue = "Red", DialogPrompt = "A ColorList", DialogControl = "Color List", OptionList = "Blue, Red, Green")</pre>
New Line	<p>Inserts an empty area the height of a single control between controls. Useful for inserting space in the second column of controls in a dialog so that a wide control can be used in the first column without overlapping controls in the second column. The “DialogPrompt” sub-command is not used.</p> <p>As an example, suppose you have six controls (not counting the invisible Return Value). The first is a String, the second is a Wide String, and all others are non-Wide controls. If you want to lay out the controls so that no overlap occurs in the second column from the second Wide String in the first column, you could insert a New Line control in the PropertyList sub-command for the FunctionInfo object.</p>	<pre>guiCreate("Property", Name = "xLINEFEED", Type = "Normal", DialogPrompt = "A New Line", DialogControl = "New Line") guiCreate("FunctionInfo", Function = "TestFn", DialogHeader = "Test", PropertyList = c("ReturnValue", "aString1", "aWideString", "aString2", "aString3", "xLINEFEED", "aString4", "aString5"), . . .</pre>
Page Tab	Adds a page of controls and groups of controls to a dialog. You must have at least one group of controls on a page before the page will display.	<pre>guiCreate("Property", Name = "myPageOne", Type = "Page", DialogPrompt = "Page 1", PropertyList = c("myGroup1", "myString"))</pre>

Table 8.2: *Dialog control types.*

Control Type	Description	Example
String List Box	A list box of strings that allows only single selections. This control differs from the List Box and Combo Box controls in that the list of strings is always visible. The “OptionList” subcommand is used to fill the list.	<pre>guiCreate("Property", Name = "myStringList", Type = "Normal", DefaultValue = "Opt3", DialogPrompt = "String List", DialogControl = "String List Box", OptionList = "Opt1, Opt2, Opt3")</pre>
Radio Buttons	A group of radio buttons which allow only one button to be selected. The buttons are exclusive which means that if one button is selected and another is clicked on, the original button is deselected and the button clicked on is selected. The “OptionList” subcommand is used to specify the names of the buttons in the group. This name is returned when a button is selected, as in the other list controls.	<pre>guiCreate("Property", Name = "myRadioButtons", Type = "Normal", DefaultValue = "Opt3", DialogPrompt = "Radio Buttons", DialogControl = "Radio Buttons", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>
Wide Radio Buttons	Same as “Radio Buttons” except this dialog control takes up two columns.	<pre>guiCreate("Property", Name = "myWideRadioButtons", Type = "Normal", DefaultValue = "Opt3", DialogPrompt = "Wide Radio Buttons", DialogControl = "Wide Radio Buttons", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", ")</pre>

Table 8.2: *Dialog control types.*

Control Type	Description	Example
Integer Spinner	An edit field with two buttons attached to increase or decrease the value in the edit field by some fixed increment. Use the “Range” subcommand to specify the start and end of the range of numbers allowed in the edit field and to specify the small and large increments. The small increment is used when you single-click once on the spinner arrows. The large increment is used when you click and hold on the spinner arrows.	<pre>guiCreate("Property", Name = "myIntSpinner", Type = "Normal", DefaultValue = "2", DialogPrompt = "Int Spinner", DialogControl = "Integer Spinner", Range = "-40:40,1,5")</pre>
Float Spinner	Similar to the Integer Spinner control except this control accepts floating point numbers.	<pre>guiCreate("Property", Name = "myFloatSpinner", Type = "Normal", DefaultValue = "2.5", DialogPrompt = "Float Spinner", DialogControl = "Float Spinner", Range = "-40.5:40.5,0.1,1.0")</pre>
Integer Slider	A visual slider control that allows adjustment of a numeric value by dragging a lever from one side of the control to the other. The left and right arrow keys can be used to move the slider by the small increment and the page up and page down keys can be used to move by the large increment. Use the “Range” subcommand to specify the start and end of the range of numbers allowed and to specify the small and large increments.	<pre>guiCreate("Property", Name = "myIntSlider", Type = "Normal", DefaultValue = "2", DialogPrompt = "Int Slider", DialogControl = "Integer Slider", Range = "-10:10,1,2")</pre>

Table 8.2: *Dialog control types.*

Control Type	Description	Example
Float Slider	Similar to the Integer Slider control except this control accepts floating point numbers and increments less than 1.	<pre>guiCreate("Property", Name = "myFloatSlider", Type = "Normal", DefaultValue = "2.1", DialogPrompt = "Float Slider", DialogControl = "Float Slider", Range = "-5:5,0.1,1")</pre>
OCX String	<p>Adds any specially written ActiveX control to the dialog. Use the "ControlProgId" subcommand to specify the ProgID of the ActiveX control you want to add, and use the "ControlServerPathName" subcommand to specify the pathname of the ActiveX control server program.</p> <p>See the ActiveX Controls in Spotfire S+ dialogs on page 307 for more information about ActiveX controls in Spotfire S+ dialogs.</p>	<pre>guiCreate("Property", Name = "myOCXControl", Type = "Normal", DefaultValue = "2", DialogPrompt = "My OCX", DialogControl = "OCX String", ControlProgId = "MyOCXServer.MyCtrl.1", ControlServerPathName = "c:\\myocx\\myocx.ocx")</pre>
Picture	<p>A small rectangle taking up one dialog column which can contain a Windows metafile picture (either Aldus placable or enhanced).</p> <p>The picture to draw in this control is specified as a string in the "DefaultValue" subcommand containing either the pathname to the WMF file on disk, or a pathname to a Windows 32-bit DLL followed by the resource name of the metafile picture in this DLL.</p>	<pre>guiCreate("Property", Name = "myPicture", DialogControl = "Picture", DialogPrompt = "My Picture", DefaultValue = "c:\\pics\\mypict.wmf");</pre>

Table 8.2: *Dialog control types.*

Control Type	Description	Example
Wide Picture	Same as “Picture” except this dialog control takes up two columns.	<pre>guiCreate("Property", Name = "myWidePicture", DialogControl = "Wide Picture", DialogPrompt = "My Wide Picture", DefaultValue = "c:\\pics\\mypict.wmf");</pre>
Wide Picture List Box	Same as “Wide Picture” except this dialog control is a drop-list of strings. Only one string can be selected at a time.	<pre>guiCreate("Property", Name = "myWidePicture", DialogControl = "Wide Picture", DialogPrompt = "My Wide Picture", DefaultValue = "c:\\pics\\mypict.wmf");</pre>

Picture Controls For both the Picture and the Picture List Box controls, you can specify either a pathname to a Windows metafile on disk or a pathname to a Windows 32-bit DLL and the resource name of the metafile in this DLL to use. The syntax for each of these is specified below:

Table 8.3: *Picture control pathname syntax.*

Pathname to Windows metafile	DLL Pathname and resource name of metafile
<p>"<i>[pathname]</i>" Example: "c:/spluswin/home/Meta1.WMF"</p>	<p>";<i>[pathname to DLL],[metafile resource name]</i> Example: ";c:/mydll/mydll.dll, MyMetaFile"</p> <p>Please note that the leading semicolon is required in this case and the comma is required between the DLL pathname and the name of the metafile resource.</p>

Several example Spotfire S+ scripts are available on disk which demonstrate how to use these new controls for your own dialogs. See the files **PictCtl1.ssc** and **PictCtl2.ssc** in the **Samples\Documents** directory within the directory where Spotfire S+ is installed.

Copying Properties

When creating a new dialog, it is often desirable to have controls similar to those used in previously existing dialogs. To use a Property already present in another dialog, simply refer to this Property when creating the `FunctionInfo` object and perhaps in the group or page containing the Property. Any of the properties used in the statistical dialogs are directly available for use by dialog developers.

Additionally, the dialog developer may wish to have a property which is a modified version of an existing property. One way to do so is to refer to the Property directly, and to overload specific aspects of the Property (such as the `DialogPrompt` or `DefaultValue`) in the `FunctionInfo` object.

Another way to create a new Property based on another Property is to specify the Property to **CopyFrom** when creating the new Property. The new Property will then be based on the **CopyFrom** Property, with any desired differences specified by the other properties of the object.

In this section we mention standard properties commonly used in Spotfire S+ dialogs, as well as internal properties useful for filling default values and option lists based on current selections.

Standard Properties

Any Property used in a built-in statistics dialog is available for reuse. To find the name of a particular Property, start by looking at the Property List in the `FunctionInfo` object for the dialog of interest. This will typically list Page or Group properties used in the dialog in order of their appearance in the dialogs (from top left to lower right). For a single-page dialog, locate the name of the Group object containing the Property of interest, and then examine the Property List for that Group object to locate the name of the Property of interest. For multi-page dialogs, find the name of the Property by looking at the `FunctionInfo` object for the Page name, then the Page object for the Group name, then the Group object for the desired Property name.

Once you know the name of the Property object, you may include it directly in a dialog by placing it in the Property List for the dialog or one of its groups or pages. Alternatively, you may create a new Property using **CopyFrom** to base the new Property on the existing Property.

For easy reference, Table 8.4 lists some of the properties used in the **Linear Regression** dialog which are reused in many of the other statistical dialogs. For the names of additional properties, examine the `FunctionInfo` object for `menuLm` and the related Property objects. Note that the naming convention used by TIBCO Software Inc. is

generally to start property names with `SProp`. When creating new properties, users may wish to use some other prefix to avoid name conflicts.

Table 8.4: *Linear Regression dialog properties.*

Dialog Prompt	Property Name
Data group	
Data Frame	<code>SPropDataFrameList</code>
Weights	<code>SPropWeights</code>
Subset Rows with	<code>SPropSubset</code>
Omit Rows with Missing Values	<code>SPropOmitMissing</code>
Formula group	
Formula	<code>SPropPFFormula</code>
Create Formula	<code>SPropPFButton</code>
Save Model Object group	
Save As	<code>SPropReturnObject</code>
Printed Results group	
Short Output	<code>SPropPrintShort</code>
Long Output	<code>SPropPrintLong</code>
Saved Results group	
Save In	<code>SPropSaveResultsObject</code>
Fitted Values	<code>SPropSaveFit</code>
Residuals	<code>SPropSaveResid</code>

Table 8.4: *Linear Regression dialog properties.*

Dialog Prompt	Property Name
Predict page	
New Data	SPropPredictNewdata
Save In	SPropSavePredictObject
Predictions	SPropPredictSavePred
Confidence Intervals	SPropPredictSaveCI
Standard Errors	SPropPredictSaveStdErr
Confidence Level	SPropConfLevel

Some other widely used properties and their associated purpose are listed below.

SPropInvisibleReturnObject

This Property object has an invisible control which does not appear in the dialog. It is used as the return value argument for dialogs whose results are never assigned.

SPropCurrentObject

This Property object is an invisible control whose default value is the name of the currently selected object. It is used by method dialogs launched from context menus, as discussed in the section Method Dialogs (page 366).

SPropFSpace1, ..., SPropFSpace8

These Property objects have a newline control. They are used to place spaces between groups to adjust the dialog layout.

Internal Properties

Internal properties are specifically designed to fill the default values and option lists based on the currently selected objects. For example, internal properties can be used to create a list box containing the names of the variables in the currently selected data frame.

If the dialog needs to fill these values in a more sophisticated way, this may be accomplished using callback functions. See the section Method Dialogs (page 366) for details.

Here are several internal property objects that can be used in dialogs either alone or by means of **CopyFrom**.

TXPROP_DataFrames

This Property object displays a dropdown box listing all data frames filtered to be displayed in any browser.

TXPROP_DataFrameColumns

This Property object displays a dropdown box listing all columns in the data frame selected in TXPROP_DataFrames. If no selection in TXPROP_DataFrames has been made, default values are supplied.

TXPROP_DataFrameColumnsND

This Property object displays a dropdown box of all columns in the data frame selected in TXPROP_DataFrames. If no selection in TXPROP_DataFrames has been made, default values are not supplied.

TXPROP_SplusFormula

This Property object causes a Spotfire S+ formula to be written into an edit field when columns in a data sheet view are selected. The response variable is the first column selected, and the predictor variables are the other columns.

TXPROP_WideSplusFormula

This Property object differs from TXPROP_SplusFormula only in that the formula is displayed in an edit field which spans two columns of the dialog, instead of one column.

ACTIVEX Controls in Spotfire S+ dialogs

Spotfire S+ supports the use of ActiveX controls in dialogs for user-defined functions created in the S-PLUS programming language. This feature allows greater flexibility when designing a dialog to represent a function and its parameters. Any ActiveX control can be added to the property list for a dialog, however, most ActiveX controls will not automatically communicate changed data back to the Spotfire S+ dialog nor will most tell Spotfire S+ how much space to give the control in the dialog. To fully support Spotfire S+ dialog

layout and data communication to and from Spotfire S+ dialogs, a few special ActiveX methods, properties, and events need to be implemented in the control by the control designer.

Examples of ActiveX controls which implement support for Spotfire S+ dialog containment are provided on disk in the **samples/oleauto/visualc/vcembed** directory beneath the program directory. These examples are C++ projects in Microsoft Visual C++ 4.1 using MFC (Microsoft Foundation Classes). Any MFC ActiveX project can be modified to support Spotfire S+ dialogs easily, and this will be discussed later in this section. The **samples/oleauto/visualc/vcembed** directory includes example scripts which use Spotfire S+ to test these ActiveX controls.

Adding an ActiveX control to a dialog

To use an ActiveX control for a creating a property in a dialog, specify a “DialogControl” of type “OCX String” and specify the program id (or PROGID) of the control using the “ControlProgId” subcommand. Below is an example Spotfire S+ script which creates a property that uses an ActiveX control:

```
guiCreate("Property",
    name = "OCXStringField",
    DialogControl = "OCX String",
    ControlProgId = "TXTESTCONTROL1.TxTestControl1Ctrl.1",
    ControlServerPathName = "c:/myocx/myocx.ocx",
    DialogPrompt = "&OCX String");
```

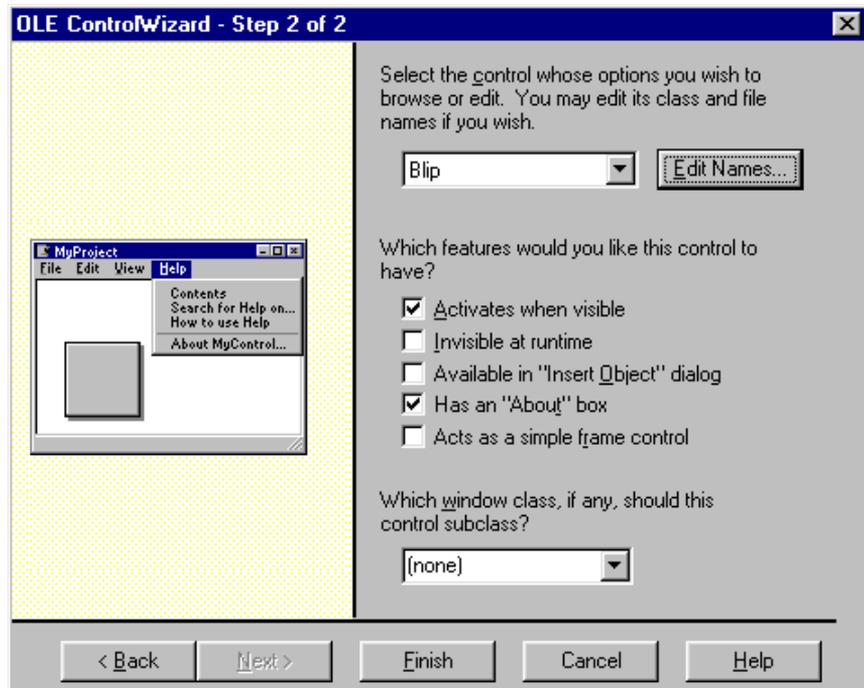
If you are editing or creating a property using the Object Explorer, the Property object dialog for the property you are editing allows you to set the dialog control type to “OCX String” from the “Dialog Control” drop-down list. When this is done, the “Control ProgId” and “ControlServerPathName” fields become enabled, allowing you to enter the PROGID of the ActiveX control and its location on disk, respectively. The “ControlServerPathName” value is used to autoregister the control, if necessary, before using the control.

If you are editing or creating a property using the Object Explorer, the Property object dialog for the property you are editing allows you to set the dialog control type to “OCX String” from the “Dialog Control” drop-down list. When this is done, the “Control ProgId” field becomes enabled allowing to you enter the PROGID of the ActiveX control.

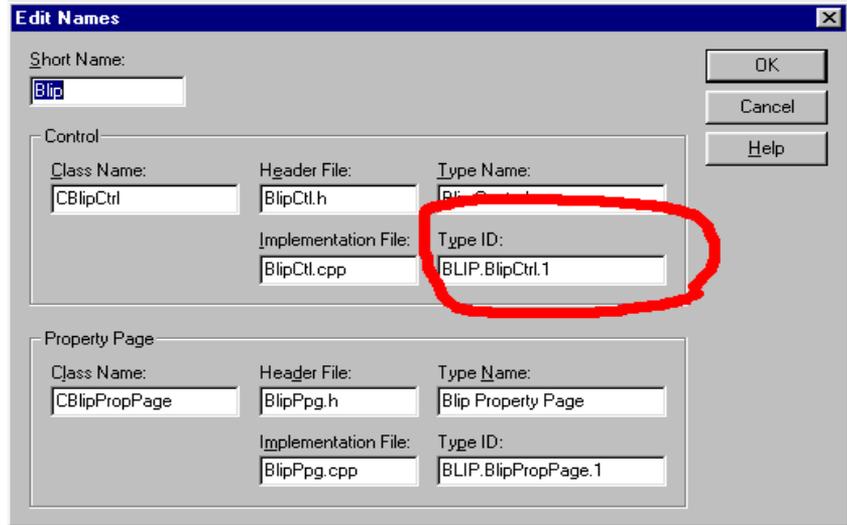
**Where can the
PROGID for the
control be found?**

When you add an ActiveX control to a Spotfire S+ dialog, you need to specify its PROGID, as mentioned above. The PROGID is a string which uniquely identifies this control on your system. If you create controls using the ControlWizard in Developer Studio as part of Microsoft Visual C++ 4.0 or higher, a default value for the PROGID is created by the ControlWizard during control creation that is based on the name of the project you use. For example, if your ControlWizard project name is "MyOCX", the PROGID that is generated is "MYOCX.MyOCXCtrl.1". The pattern takes the form [Project name].[Control class name without the leading 'C'].1. You can also find the PROGID used in an MFC ControlWizard project in the implementation CPP file of the control class. Search for the IMPLEMENT_OLECREATE_EX() macro in this file. The second parameter in this macro is the PROGID string you are looking for.

If you are using the OLE ControlWizard as part of Microsoft Visual C++ 4.0 or higher to develop your control, you can change the PROGID string for your control before it gets created by editing the names used for the control project. During the ControlWizard steps, you will see a dialog with the button "Edit Names" on it:



Click on this button and you will get another dialog allowing you to change the names used for classes in this project. Every control project in MFC has a class for the control and a class for the property sheet for the control. In the control class section of this dialog you will see the “Type ID” field. This is the PROGID for the control:



Registering an ActiveX control

It is important to register an ActiveX control with the operating system at least once before using it so that whenever the PROGID of the control is referred to (such as in the “ControlProgId” subcommand above), the operating system can properly locate the control on your system and run it. Registering an ActiveX control is usually done automatically during the creation of the control, such as in Microsoft Visual C++ 4.0 or higher. If the subcommand “ControlServerPathName” is specified in a Spotfire S+ script using the control, then this value will be used to register the control automatically. A control can also be registered manually by using a utility called **RegSvr32.exe**. This utility is included with development systems that support creating ActiveX controls, such as Microsoft Visual C++ 4.0 or higher. For your convenience, a copy of **RegSvr32.exe** is located in the **samples/oleauto/visualc/vcembed** directory, along with two useful batch files, **RegOCX.bat** and **UnRegOCX.bat**, which will register and unregister a control. You can modify these batch files for use with controls you design.

You typically do not ever need to unregister an ActiveX control, unless you wish to remove the control permanently from your system and no longer need to use it with any other container programs such as Spotfire S+. If this is the case, you can use **RegSvr32.exe** with the '/u' command line switch (as in **UnRegOCX.bat**) to unregister the control.

Why only “OCX String”?

In Spotfire S+, several different types of properties exist. There are string, single-select lists, multi-select lists, numeric, and others. This means that a property in a dialog communicates data depending on the type of property selected. A string property communicates string data to and from the dialog. A single-select list property communicates a number representing the selection from the list, a multi-select list communicates a string of selections made from the list with delimiters separating the selections. For ActiveX controls, only string communication has been provided in this version. This means that the control should pass a string representing the “value” or state of the control back to Spotfire S+. In turn, if Spotfire S+ needs to change the state of the control, it will communicate a string back to the control. Using a string permits the most general type of communication between Spotfire S+ and the ActiveX control, because so many different types of data can be represented with a string, even for example lists. In future versions, other Spotfire S+ property types may be added for ActiveX controls.

Common error conditions when using ActiveX controls in Spotfire S+

The most common problem when using an ActiveX control in a Spotfire S+ dialog is that the control does not appear; instead, a string edit field shows up when the dialog is created. This is usually caused when the ActiveX control is not registered with the operating system. After a control is first created and before it is ever used, it must be registered with the operating system. This usually occurs automatically in the development system used to make the control, such as Microsoft Visual C++. However, you can also manually register the control by using a utility called **RegSvr32.exe**, located in the **samples/oleauto/visualc/vcembed** directory. This utility is included with development systems that support creating ActiveX controls, such as Microsoft Visual C++ 4.0 or higher. You can modify these batch files for use with controls you design. More information is found in the section Registering an ActiveX control on page 310.

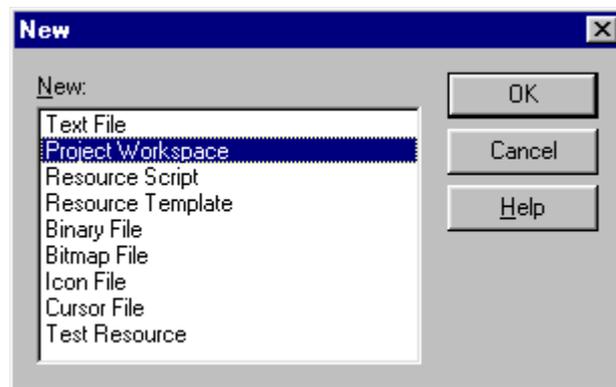
Designing ActiveX controls that support Spotfire S+

As mentioned earlier, examples of ActiveX controls which implement support for Spotfire S+ are provided on disk in the `samples\oleauto\visualc\vcembed\MyOCX` directory beneath the program directory. One of the examples in this directory is called MyOCX, and it is a C++ project in Microsoft Visual C++ 4.1 using MFC. There is also an example Spotfire S+ script in MyOCX which shows how to use this ActiveX control in a Spotfire S+ dialog. This example will be used here to show how to implement ActiveX controls for Spotfire S+. If you would rather skip this section and simply study the changes in the source files for MyOCX, all changes are marked in the source files with the step number (as listed below) that the change corresponds to. Just search for the string “Spotfire S+ Dialog change (STEP)” in all the files of the MyOCX project to find these modifications.

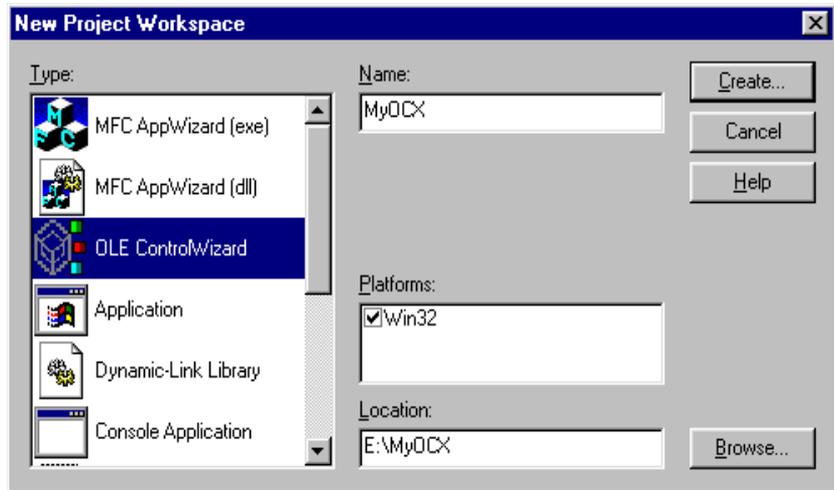
Version 4.0 or higher of Microsoft Visual C++ is used to demonstrate ActiveX control creation. Higher versions can also be used to create controls for Spotfire S+ but the dialogs and screens shown may be different.

I. Create the basic control

The first step to designing an ActiveX control in MFC should be to use the OLE ControlWizard that is part of the Developer Studio. Select **New** from the **File** menu in Developer Studio and then choose **Project Workspace** to start a new project.

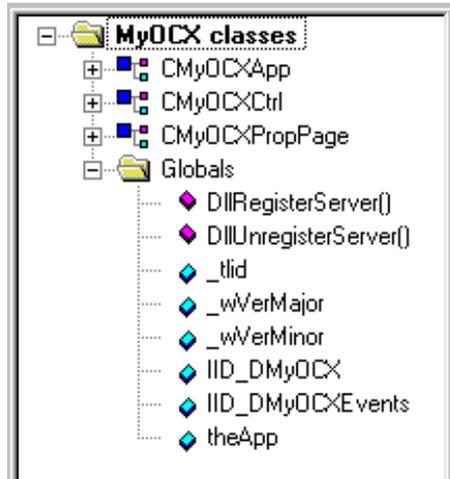


From the workspace dialog that appears, select **OLE ControlWizard** from the list of workspace types available. Enter a name for the project and specify the location, then click the **Create...** button.



After accepting this dialog, you will see a series of dialogs associated with the OLE Control Wizard, asking questions about how you want to implement your control. For now, you can simply accept the defaults by clicking **Next** on each dialog. When you reach the last dialog, click the **Finish** button. You will see a confirmation dialog showing you the choices you selected and names of classes that are about to be created. Click the **OK** button to accept and generate the project files.

In the **ClassView** page of the Project Workspace window in Visual C++, you will see the classes that the OLE ControlWizard created for your ActiveX control:



2. Add the S-PLUS support classes

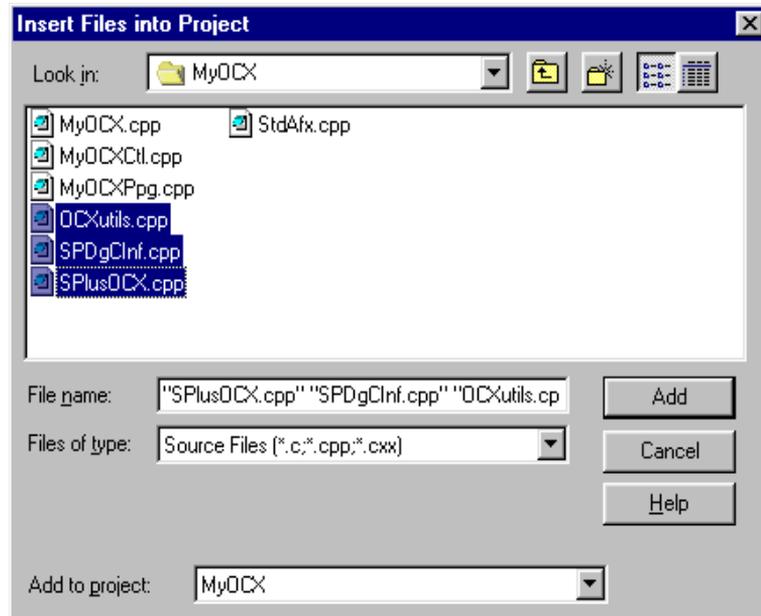
To start adding support for Spotfire S+ dialogs to your ActiveX control, copy the following files from the **samples/oleauto/visualc/vcembed/support** control example directory into the new ActiveX control project directory you just created:

```
OCXUtils.cpp  
OCXUtils.h  
SPDgCInf.cpp  
SPDgCInf.h  
SPlusOCX.cpp  
SPlusOCX.h  
SPlusOCX.idl
```

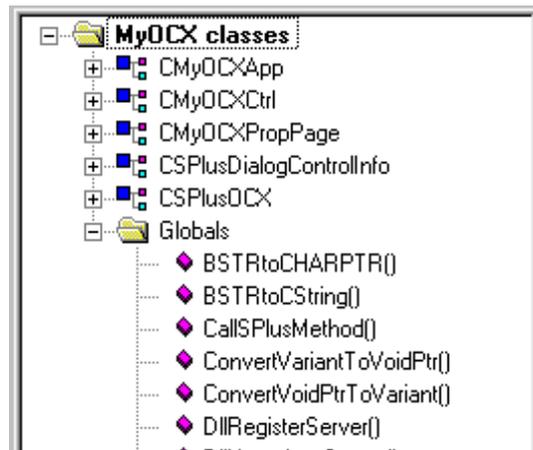
You also need to add these classes to your project before they will be compiled and linked to your control. To do this, select **Files into Project...** from the **Insert** menu in Visual C++. You will then see a standard file open dialog. Use this dialog to select the following files:

```
OCXUtils.cpp  
SPDgCInf.cpp  
SPlusOCX.cpp
```

To select all these files at once, hold down the CTRL key while using the mouse to click on the filenames in the list.



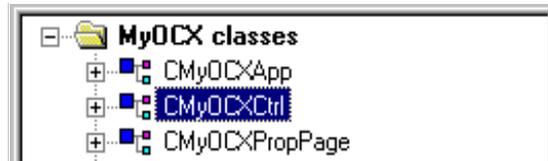
When these files are selected, click the **Add** button and the classes will appear as entries in your Project Workspace window.



3. Modify class inheritance

Next, we need to modify the inheritance of the class representing your ActiveX control so that it inherits from CSPlusOCX instead of from COleControl. CSPlusOCX is a parent class from which all ActiveX controls for which you desire support for Spotfire S+ dialogs can inherit. CSPlusOCX inherits directly from COleControl and its complete source code can be found in the **SPlusOCX.cpp** and **SPlusOCX.h** files.

To do this, first double-click on the class representing your ActiveX control in the **ClassView** page of the Project Workspace window to open the header for this class into your editor. In this example that is the CMyOCXCtrl class. Go to the top of this file in the editor.



Add the following line before the class declaration line for CMyOCXCtrl at the top of this header file:

```
#include "SPlusOCX.h"
```

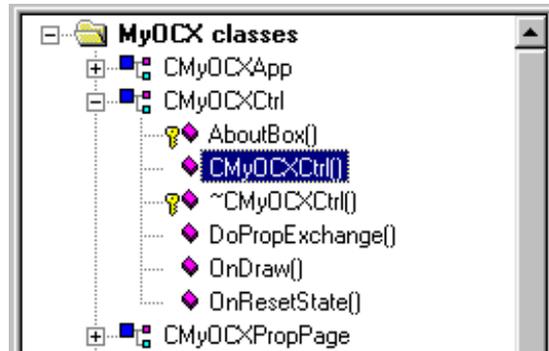
Modify the class declaration line

```
class CMyOCXCtrl : public COleControl
```

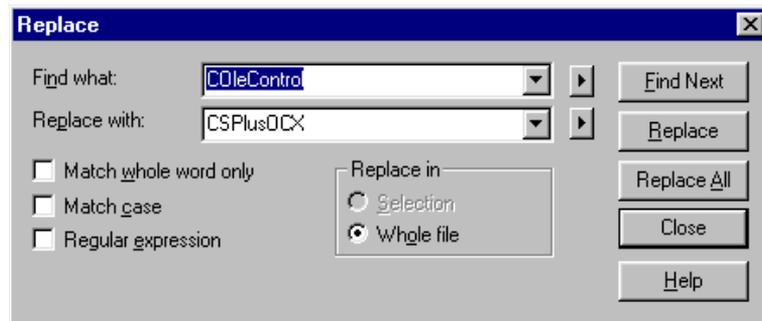
to read

```
class CMyOCXCtrl : public CSPlusOCX
```

Next, expand the class listing for CMyOCXCtrl so that all the methods are shown. To do this, click on the '+' next to CMyOCXCtrl in the **ClassView** page of the Project Workspace window.

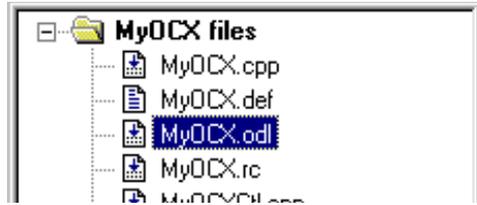


Then double-click on the constructor `CMyOCXCtrl()` to open the implementation CPP file for this class in your editor. Go to the top of this file. Using the find and replace function of the Developer Studio, replace all occurrences of `ColeControl` base class with the new base class name `CSPPlusOCX` in this file:



4. Modify your control's type library definition file

Switch to the `FssileView` page in the Project Workspace window and find the type library definition file (.ODL) for your ActiveX control. In this example it is **MyOCX.odl**. Double-click on this entry in the list to open this file into your editor. Go to the top of this file.



Find the “properties” definition section for the dispatch interface `_DMyOCX` in this file. It should look like:

```
dispinterface _DMyOCX
{
    properties:
        // NOTE - ClassWizard will maintain property information
        here.
        // Use extreme caution when editing this section.
        //{AFX_ODL_PROP(CMyOCXCtrl)
        //}}AFX_ODL_PROP
```

Add the following lines at the end of this section:

```
#define SPLUSOCX_PROPERTIES
#include "SPlusOCX.idl"
#undef SPLUSOCX_PROPERTIES
```

The section should now appear as follows:

```
dispinterface _DMyOCX
{
    properties:
        // NOTE - ClassWizard will maintain property information
        here.
        // Use extreme caution when editing this section.
        //{AFX_ODL_PROP(CMyOCXCtrl)
        //}}AFX_ODL_PROP
```

```
#define SPLUSOCX_PROPERTIES
#include "SPlusOCX.idl"
#undef SPLUSOCX_PROPERTIES
```

```

    methods:
    // NOTE - ClassWizard will maintain method information
    here.
    //    Use extreme caution when editing this section.
    //{{AFX_ODL_METHOD(CMyOCXCtrl)
    //}}AFX_ODL_METHOD

    [id(DISPID_ABOUTBOX)] void AboutBox();
};

```

Now, add the following lines at the end of the “methods” section just below the “properties” section you just modified:

```

#define SPLUSOCX_METHODS
#include "SPlusOCX.idl"
#undef SPLUSOCX_METHODS

```

This whole section should now appear as follows:

```

dispinterface _DMyOCX
{
    properties:
    // NOTE - ClassWizard will maintain property information
    here.
    //    Use extreme caution when editing this section.
    //{{AFX_ODL_PROP(CMyOCXCtrl)
    //}}AFX_ODL_PROP

#define SPLUSOCX_PROPERTIES
#include "SPlusOCX.idl"
#undef SPLUSOCX_PROPERTIES

    methods:
    // NOTE - ClassWizard will maintain method information
    here.
    //    Use extreme caution when editing this section.
    //{{AFX_ODL_METHOD(CMyOCXCtrl)
    //}}AFX_ODL_METHOD

    [id(DISPID_ABOUTBOX)] void AboutBox();

#define SPLUSOCX_METHODS
#include "SPlusOCX.idl"
#undef SPLUSOCX_METHODS

```

```
};
```

Next, locate the event dispatch interface sections. In this example, it appears as:

```
dispinterface _DMyOCXEvents
{
    properties:
        // Event interface has no properties

    methods:
        // NOTE - ClassWizard will maintain event information
        here.
        // Use extreme caution when editing this section.
        //{{AFX_ODL_EVENT(CMyOCXCtrl)
        //}}AFX_ODL_EVENT
};
```

Add the following lines in the “events” section:

```
#define SPLUSOCX_EVENTS
#include "SPlusOCX.idl"
#undef SPLUSOCX_EVENTS
```

The section should now appear as:

```
dispinterface _DMyOCXEvents
{
    properties:
        // Event interface has no properties

    methods:
        // NOTE - ClassWizard will maintain event information
        here.
        // Use extreme caution when editing this section.
        //{{AFX_ODL_EVENT(CMyOCXCtrl)
        //}}AFX_ODL_EVENT

#define SPLUSOCX_EVENTS
#include "SPlusOCX.idl"
#undef SPLUSOCX_EVENTS

};
```

Do not modify any other parts of this file at this time.

5. Build the control

Now is a good time to build this project. To do this, click on the Build toolbar button or select **Build MyOCX.OCX** from the Build menu in the Developer Studio. If you receive any errors, go back through the above steps to make sure you have completed them correctly. You may receive warnings:

```
OCXutils.cpp(125) : warning C4237: nonstandard extension
used : 'bool' keyword is reserved for future use
OCXutils.cpp(216) : warning C4237: nonstandard extension
used : 'bool' keyword is reserved for future use
```

These warnings are normal and can be ignored.

Several overrides of CSPlusOCX virtual methods still remain to be added to your ActiveX control class, but compiling and linking now gives you a chance to review the changes made and ensure that everything builds properly at this stage.

6. Add overrides of virtual methods to your control class

To support Spotfire S+ dialog layout and setting the initial value of the control from a Spotfire S+ property value, you need to override and implement several methods in your control class. To do this, edit the header for your control class. In this example, edit the **MyOCXCtl.h** file. In the declaration of the CMyOCXCtrl class, add the following method declarations in the “public” section:

```
virtual long GetSPlusDialogVerticalSize( void );
virtual long GetSPlusDialogHorizontalSize( void );
virtual BOOL SPlusOnInitializeControl(const VARIANT FAR&
    vInitialValue);
```

Next, open the implementation file for your control class. In this example, edit the file **MyOCXCtrl.cpp**. Add the following methods to the class:

```
long CMyOCXCtrl::GetSPlusDialogVerticalSize()
{
    return 3; // takes up 3 lines in dialog
}

long CMyOCXCtrl::GetSPlusDialogHorizontalSize()
{
    return 1; // takes up 1 column in dialog
}

BOOL CMyOCXCtrl::SPlusOnInitializeControl(const VARIANT
    FAR& vInitialValue)
{
    CString sInitialValue; sInitialValue.Empty();
    if ( GetStringFromVariant(
        sInitialValue,
        vInitialValue,
        "InitialValue" ) )
    {
        // Set properties here
    }

    return TRUE;
}
```

These three methods should be implemented in the control class of any ActiveX control supporting Spotfire S+ dialogs fully. The first two methods support dialog layout, while the third supports setting values for the control from Spotfire S+.

The value returned by `GetSPlusDialogVerticalSize()` should be a long number representing the number of lines the control takes up in a Spotfire S+ dialog. A line is the size of a String edit field property in a Spotfire S+ dialog. The value returned by `GetSPlusDialogHorizontalSize()` should be either 1 or 2. Returning 1 means that this control takes up only one column in a Spotfire S+ dialog. Returning 2 means the control takes up two columns. A column in a Spotfire S+ dialog is the width of a single String property

field. There are at most two columns in a Spotfire S+ dialog. In the example above, the MyOCX control takes up three lines and only one column in a Spotfire S+ dialog.

SPlusOnInitializeControl() is called when the control is first enabled in the Spotfire S+ dialog and every time the property that this control corresponds to in Spotfire S+ is changed. It receives a variant representing the initial value or current value (if any) for the control. This method should return TRUE to indicate successful completion and FALSE to indicate failure. Included in the file **OCXUtils.h** (copied previously into your control project directory) are numerous helper functions such as the one used here GetStringFromVariant() which will convert the incoming variant into a string if possible. You can then use this string to set one or more properties in your control.

To use the SPlusOnInitializeControl() in this example ActiveX control, first add a member string to the control class. Edit the **MyOCXCtrl.h** file and add a CString member variable called m_sValue to the CMyOCXCtrl class:

```
private:
    CString m_sValue;
```

Next, initialize this value in the constructor for CMyOCXCtrl by modifying the constructor definition in **MyOCXCtrl.cpp**:

```
CMyOCXCtrl::CMyOCXCtrl()
{
    InitializeIIDs(&IID_DMyOCX, &IID_DMyOCXEvents);
    // TODO: Initialize your control's instance data here.

    m_sValue.Empty();
}
```

Then, add lines to the definition of the override of SPlusOnInitializeControl() in your control class to set this member variable and refresh the control by modifying **MyOCXCtrl.cpp**:

```
BOOL CMyOCXCtrl::SPlusOnInitializeControl
(const VARIANT FAR& vInitialValue)
{
    CString sInitialValue; sInitialValue.Empty();
    if ( GetStringFromVariant(
```

```
        sInitialValue,  
        vInitialValue,  
        "InitialValue" ) )  
    {  
        // Set properties here  
  
        m_sValue = sInitialValue;  
        Refresh();  
  
    }  
  
    return TRUE;  
}
```

Finally, so we can see the effects of `SPlusOnInitializeControl()`, add a line to the `OnDraw` method of `CMyOCXCtrl` by editing the definition of this method in **MyOCXCtrl.h**:

```
void CMyOCXCtrl::OnDraw(  
    CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)  
{  
    // TODO: Replace the following code with your  
    // own drawing code.  
    pdc->FillRect(rcBounds,  
        CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)));  
    pdc->Ellipse(rcBounds);  
  
    // Display latest value  
    pdc->DrawText(  
        m_sValue, (LPRECT)&rcBounds, DT_CENTER | DT_VCENTER );  
}
```

Rebuild the project now to test these changes.

7. Test your new control in Spotfire S+

To try out your new control in Spotfire S+ you'll need to create a Spotfire S+ script which creates properties and displays a dialog. Open Spotfire S+ and open the script file from **samples/oleauto/visualc/ocx/MyOCX** called **MyOCX.ssc**. Notice that the script begins by creating three properties, one for the return value from a function and the other two for the parameters of a function. The property for `MyOCX` uses the type `OCX String` and the `PROGID` for the control we just created:

```

guiCreate("Property",
  name = "MyOCX",
  DialogControl = "OCX String",
  ControlProgId = "MYOCX.MyOCXCtrl.1",
  DialogPrompt = "My &OCX");

```

Run the script **MyOCX.ssc** and you will see a dialog containing an edit field and the MyOCX control you just created. When the dialog appears, the ActiveX control contains the text “Hello” because this is set as the initial value in the Spotfire S+ script callback function:

```

callbackMyOCXExample <- function(df)
{
  if(IsInitDialogMessage(df)) # Am I called to initialize
                              # the properties?
  {
    # Set the initial value of the MyOCX property
    df <- cbSetCurrValue(df,"MyOCX", "\"Hello\"")
  }
  ...
}

```

When you enter a string (use quotes around any string you enter in these dialog fields) in the edit field, the ActiveX control updates to show that string. When you click the **OK** or **Apply** buttons in the dialog, you will see the values of both properties printed in a report window.

Summary of steps to support Spotfire S+ dialogs in ActiveX controls

To summarize the above steps, the list below shows you the tasks necessary to adapt your MFC ActiveX control project to support Spotfire S+ dialogs:

1. Add Spotfire S+ dialog support files to your project:

```

OCXUtils.cpp
OCXUtils.h
SPDgCInf.cpp
SPDgCInf.h
SPlusOCX.cpp
SPlusOCX.h
SPlusOCX.idl

```

2. Change the inheritance of your control class from base class COleControl to CSPlusOCX.

3. Modify your control's ODL (type library definition file) to include SPlusOCX.idl sections.
4. Add virtual overrides of key CSPlusOCX methods to your control class:

```
virtual long GetSPlusDialogVerticalSize( void );  
virtual long GetSPlusDialogHorizontalSize( void );  
virtual BOOL SPlusOnInitializeControl(const VARIANT  
    FAR& vInitialValue);
```

Examples of ACTIVE X controls included with Spotfire S+

Examples of ActiveX controls which implement support for Spotfire S+ dialog containment are provided on disk in the **samples\oleauto\visualc\vcembed** directory beneath the program directory. These examples are C++ projects in Microsoft Visual C++ 4.1 using MFC (Microsoft Foundation Classes) and are intended for developers.

samples\oleauto\visualc\vcembed

myocx Microsoft Visual C++ 4.1 MFC project demonstrating how to write ActiveX controls that fully support Spotfire S+ dialogs.

ocx1 Microsoft Visual C++ 4.1 MFC project demonstrating how to write ActiveX controls that fully support Spotfire S+ dialogs.

support Microsoft Visual C++ 4.1 MFC headers and source files necessary for making ActiveX controls that fully support Spotfire S+ dialogs.

CALLBACK FUNCTIONS

In Spotfire S+, virtually any GUI object has an associated dialog. For example, a line plot is an object whose properties can be modified via its associated dialog. Similarly, an S-PLUS function can have an associated dialog. The properties of a function object are mapped to the function arguments, which can then be modified through its associated dialog. The function dialog can have an attached *callback function*.

A callback function provides a mechanism for modifying and updating properties (controls) of a live dialog. It is a tool for developing complex dialogs whose properties are dynamically changing based on the logic written in the callback function. The dialog subsystem executes the callback function while its associated dialog is up and running, in the following instances:

- Once, just before the dialog is displayed.
- When a dialog property (control) value is updated or modified by another mechanism, such as by the user.
- A button is clicked.

The user associates a callback function with a dialog by specifying its name in the corresponding function info object. The callback function takes a single data frame as its argument. This data frame argument has the dialog property names as row names. The elements in the data frame define the present state of the dialog. The Spotfire S+ programmer can access and modify these elements directly, however, there is a set of utility functions that simplify this task. Table 8.5 lists the utility functions that can be used inside a callback function to modify a dialog state. To get more complete information on these functions see the Language Reference help.

Table 8.5: Utility functions for use inside a callback function.

cbIsInitDialogMessage()	Returns TRUE if the callback function is called before the dialog window is displayed on the screen.
cbIsUpdateMessage()	Returns TRUE if the callback function is called when the user updates a property.
cbIsOkMessage()	Returns TRUE if the callback function is called when the OK button is clicked.
cbIsCancelMessage()	Returns TRUE if the callback function is called when the Cancel button is clicked.
cbIsApplyMessage()	Returns TRUE if the callback function is called when the Apply button is clicked.
cbGetActiveProp()	Gets the current active property in a dialog.
cbGetCurrValue()	Gets the current value of a property.
cbSetCurrValue()	Sets the current value of a property.
cbGetEnableFlag()	Gets the current state of the enable/disable flag of a property.
cbSetEnableFlag()	Sets the state of the enable/disable flag of a property.
cbGetOptionList()	Gets the list of items from list based properties, such as List-Box, ComboBox, Multi-selected ComboBox, and so on.
cbSetOptionList()	Sets the list of items from list based properties, such as ListBox, ComboBox, Multi-selected ComboBox, and so on.
cbGetPrompt()	Gets the Prompt string of a property.
cbSetPrompt()	Sets the Prompt string of a property.
cbGetDialogId()	Returns the unique ID of the dialog instance.

Since the usage of these functions facilitate readability and portability of the S-PLUS callback functions, we recommend that you use them instead of direct access to the data frame object.

Callback functions are the most flexible way to modify dialog properties such as default values and option lists. However, for specific cases it may be more straightforward to use a property based on an internal property, as described in the section Copying Properties (page 303). In particular, this is the easiest way to fill a field with the name of the currently selected data frame or a list of the selected data frame's variables.

Interdialog Communication

In some circumstances it may be useful to launch a second dialog when a dialog button is pushed. For example, the Formula dialog is available as a child dialog launched by the Linear Regression dialog. Information may then be communicated between dialogs using interdialog communication.

The child dialog is launched using `guiDisplayDialog`. Communication between the dialogs is performed by the functions `cbGetDialogId` and `guiModifyDialog`. The script file **samples/dialogs/dlgcomm.ssc** in the Spotfire S+ directory contains an example of such communication.

Example: Callback Functions

The example script below creates and displays a function dialog that uses a callback function to perform initialization, communication and updating properties within an active dialog. It is a complete script file (called **propcomm.ssc**) that can be opened into a script window and run.

```
#-----
# propcomm.ssc: creates and displays a function dialog.
#           It shows how to use a dialog callback function to perform
#           initialization, communication and updating properties within an
#           active dialog.
#-----

#-----
# Step 1: define the function to be executed when OK or Apply button is pushed
#-----

propcomm<- function(arg1, arg2) { print("Ok or Apply button in simple1 dialog
is pushed!") }
```

Chapter 8 Extending the User Interface

```
#-----  
# Step 2: create individual properties that we want to use for arguments in the  
function  
#-----  
  
guiCreate("Property", Name= "propcommInvisible", DialogControl= "Invisible");  
guiCreate("Property", Name= "propcommListBox", DialogControl= "List Box",  
    DialogPrompt= "&Grade",DefaultValue= "3",  
    OptionList= c("4", "3", "2", "1"))  
guiCreate("Property", Name= "propcommCheckBox", DialogControl= "Check Box",  
    DialogPrompt= "&Numerical Grade");  
  
#-----  
# Step 3: create the function info object  
#-----  
  
guiCreate("FunctionInfo", Function = "propcomm", PropertyList =  
    c("propcommInvisible", "propcommListBox", "propcommCheckBox"),  
    CallbackFunction = "propcommCallBack", Display ="T" )  
  
#-----  
# Step 4: define a callback function to be called by an instance of the dialog.  
# This callback mechanism is used to initialize, communicate and update  
properties in an active dialog.  
#-----  
  
propcommCallBack <- function(df)  
{  
    if(IsInitDialogMessage(df)) # Am I called to initialize the properties?  
    {  
        # override option list of a property  
        df <- cbSetOptionList(df, "propcommListBox", "excellent, good, fair,  
            poor, fail")  
  
        # override default value of a property  
        df <- cbSetCurrValue(df,"propcommListBox", "fair")  
        df <- cbSetOptionList(df, "propcommCheckBox", "F")  
    }  
    else if( cbIsOkMessage(df)) # Am I called when the Ok button is pushed?  
    {  
        display.messagebox("OK!")  
    }  
}
```

```

}
else if( cbIsCancelMessage(df)) # Am I called when the Cancel button is
    pushed?
{
    display.messagebox("Cancel!")
}
else if( cbIsApplyMessage(df)) # Am I called when the Apply button is
    pushed?
{
    display.messagebox("Apply!")
}
else # Am I called when a property value is updated?
{
    if (cbGetActiveProp(df) == "propcommCheckBox") # the check box was
        clicked?
    {
        # change the option list
        if(cbGetCurrValue(df, "propcommCheckBox") == "T")
        {
            df <- cbSetOptionList(df, "propcommListBox", "4.0, 3.0, 2.0, 1.0,
                0.0")
            df <- cbSetCurrValue(df,"propcommListBox", "4.0")

        }
        else
        {
            df <- cbSetOptionList(df, "propcommListBox", "excellent, good, fair,
                poor, fail")
            df <- cbSetCurrValue(df,"propcommListBox", "good")
        }
    }
}

df
}

#-----
# Step 5: display the dialog
#-----

guiDisplayDialog("Function", Name="propcomm");

```

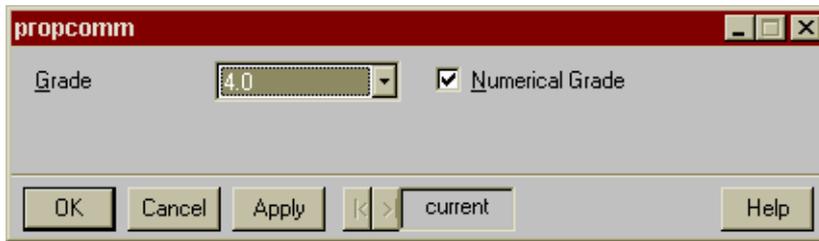


Figure 8.16: *Selecting the **Numerical Grade** checkbox will illustrate the callback function working.*

CLASS INFORMATION

Overview

A `ClassInfo` object allows information to be specified about both user-defined and interface objects. It is similar to the `FunctionInfo` object, which allows information to be specified for functions (primarily for the purpose of defining function dialogs).

There are three main uses of the `ClassInfo` object:

1. Defining a context menu (right-click menu) for objects.
2. Defining the double-click action for objects. That is, you can use it to specify what will happen when the user double-clicks or right-clicks on an object in the Object Explorer.
3. It allows the dialog header and dialog prompts for interface objects to be overridden.

Creating ClassInfo Objects

`ClassInfo` objects may be created using commands or from within the Object Explorer.

Using Commands

To create a `ClassInfo` object, use `guiCreate` with `classname="ClassInfo"`.

The `lmsreg` robust regression function returns a model of class "lms". The following commands will create a `ClassInfo` object indicating that the `print` function should be used as the double-click action, and define a context menu for lms objects:

```
guiCreate(classname="ClassInfo", Name="lms",
ContextMenu="lms",
DialogHeader="Least Median Squares Regression",
DoubleClickAction="print")
```

```
guiCreate(classname="MenuItem", Name="lms", Type="Menu",
DocumentType="lms")
```

```
guiCreate(classname="MenuItem", Name="lms$summary",
Type="MenuItem", DocumentType="lms", Action="Function",
Command="summary", MenuItemText="Summary",
ShowDialogOnRun=F)
```

```
guiCreate(classname="MenuItem", Name="lms$plot",  
Type="MenuItem", DocumentType="lms", Action="Function",  
Command="plot", MenuItemText="Plot",  
ShowDialogOnRun=F)
```

Using the Object Explorer

Open the Object Explorer and create a folder with filtering set to “ClassInfo”. Right-click on a ClassInfo object in the right pane, and choose **Create ClassInfo** from the context menu. The property dialog shown in Figure 8.17 appears.

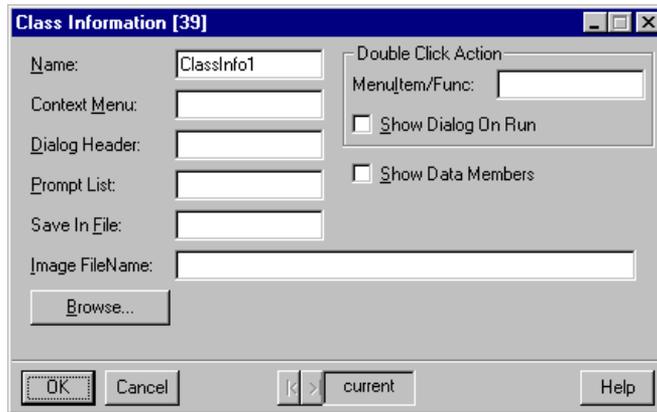


Figure 8.17: The property dialog for a ClassInfo object.

ClassInfo Object Properties

The properties of a ClassInfo object determine characteristics such as the double-click action and context menu for the class of interest. These properties may be specified and modified using the property dialog for the MenuItem object, or programmatically via the commands `guiCreate` and `guiModify`. See the `guiCreate("ClassInfo")` help file in the Language Reference help for syntax details.

The following properties are specified in the ClassInfo property dialog, shown in Figure 8.17:

The subcommand names of the properties are:

Name The name of the associated class. For instance, to specify information for the “lm” class, use this as the name. This also becomes the name of this instance of the ClassInfo object.

ContextMenu The name of the MenuItem object that defines the context menu (right-click menu) for this object in the browser. This is

the name of a MenuItem of type “Menu”, which must have been defined in the standard way for menus.

DoubleClickAction The name of a MenuItem of type “MenuItem” (that is, it is a single item instead of an entire menu) or a function. This specifies the action that will happen when the user double-clicks on the object in the browser. It allows a function to be called when the user double-clicks.

Show Dialog On Run Logical value indicating whether the dialog for the MenuItem or function will be displayed before execution.

DialogHeader Text specifying the dialog header for the associated object. This is only useful for interface objects.

PromptList Allows dialog prompts to be specified (and overridden). The syntax is the same as it is for the corresponding property of FunctionInfo objects: #0=”&My Prompt:”, #2=”Another &Prompt:”, PropertySubcommandName=”L&ast Prompt:”. That is, it is a list of assignments, in which the left-hand side denotes the property whose prompt is going to be overridden, and the right-hand side denotes the new prompt. There are two ways of denoting the property: by position, starting with 0, with the number preceded by a #; and by property subcommand name. (In the example above, “#0” denotes the 0th property of the object; “PropertySubcommandName“ is the subcommand name of the property to change.)

To find out the names of the properties of an object, you can use the following script:

```
guiGetPropertyNames(“classname”)
```

Note that all objects have two properties that may or may not be displayed on the dialog: TXPROP_ObjectName (subcommand name: **NewName**, always in position #0, but usually not displayed in a dialog) TXPROP_ObjectPosIndex (subcommand name: **NewIndex**, always in position #1, but usually not displayed in a dialog). To find out the argument names of the properties of an object, you can use the following script:

```
guiGetArgumentNames(“classname”)
```

The argument names are usually very similar to the corresponding prompts, so that figuring out which dialog field corresponds to which property should not be a problem.

Modifying ClassInfo Objects

ClassInfo objects can be modified using either programming commands, their property dialogs, or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If you are simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

Using Commands

The `guiModify` command is used to modify an existing ClassInfo object. Specify the Name of the ClassInfo object to modify, and the properties to modify with their new values.

```
guiModify(classname="ClassInfo", Name="lms",  
          DoubleClickAction="plot")
```

Using the Property Dialog

ClassInfo objects may be modified through the ClassInfo object property dialog.

To modify a ClassInfo object, open the Object Explorer to a page with filtering set to **ClassInfo**. Right-click on the ClassInfo object's icon in the right pane and choose Properties from the context menu. Refer to the previous sections for details on using the property dialog.

Using the Context Menu

ClassInfo objects can be modified with their context menus. The context menu for an object is launched by right-clicking on the object in the Object Explorer. The context menu provides options such as creating, copying, and pasting the object, as well as a way to launch the property dialog.

Example: Customizing the Context Menu

This example shows how to add to the context menu for objects of class `data.frame` displayed in the Object Explorer. The new item automatically computes summary statistics for the selected data frame. To begin, open an Object Explorer page and filter by ClassInfo and MenuItem.

1. Creating a ClassInfo object for the Class `data.frame`

1. Right-click on a ClassInfo object and select **Create ClassInfo** in its context menu.
2. Enter **data.frame** in the Name field. This represents the name of the object class in which objects will have the context menu item specified below.
3. Enter **dfMenu** in the Context Menu field. This will be the name of the context menu.

4. Click **OK**.

2. Creating the Context Menu

1. Right-click on any `MenuItem` object and select **Insert MenuItem** from its context menu.
2. Enter **dfMenu** in the Name field. This corresponds to the Context Menu name given in to the `ClassInfo` object above.
3. Enter **Menu** in the Type field.
4. Click **OK**.
5. Right-click on **dfMenu** in the left pane and select **Insert MenuItem** from the context menu.
6. Enter **desc** in the Name field. This name is not important, as long as it does not conflict with that of an existing object.
7. Select **MenuItem** from the Type field.
8. Enter **data.frame** in the Document Type field; do not choose from the dropdown box selections. This corresponds to the object class which will have this context menu.
9. Select **Function** from the Action field.
10. Enter the text **Summary....** in the MenuItem Text field. This text will appear in the context menu.
11. Move to the **Command** page of the dialog.

Tip...

A `FunctionInfo` object must exist for the function which is called by the context menu item. Otherwise, the default dialog for that function will not appear.

12. Enter **menuDescribe** in the Command field. This is the function which is executed by the dialog which appears with **Statistics ► Data Summaries ► Summary Statistics**. There is a built-in `FunctionInfo` object by the same name.
13. **Show Dialog On Run**. This should be checked.
14. The `MenuItem` object **desc** is now found alongside **dfMenu** in the **MenuItem** tree. To move it underneath **dfMenu**, hold down the ALT key and drag the **desc** icon onto the **dfMenu**

icon. To see the MenuItem object desc in its new position, click on the **dfMenu** icon in the left pane and look in the right pane.

3. Displaying and Testing the Context Menu

1. Click the Object Explorer button in the main toolbar to open a default Object Explorer window.
2. When data frame objects are visible in the right pane, right-click on any data frame. Choose **Properties**, which should appear in the context menu, as shown in Figure 8.18.

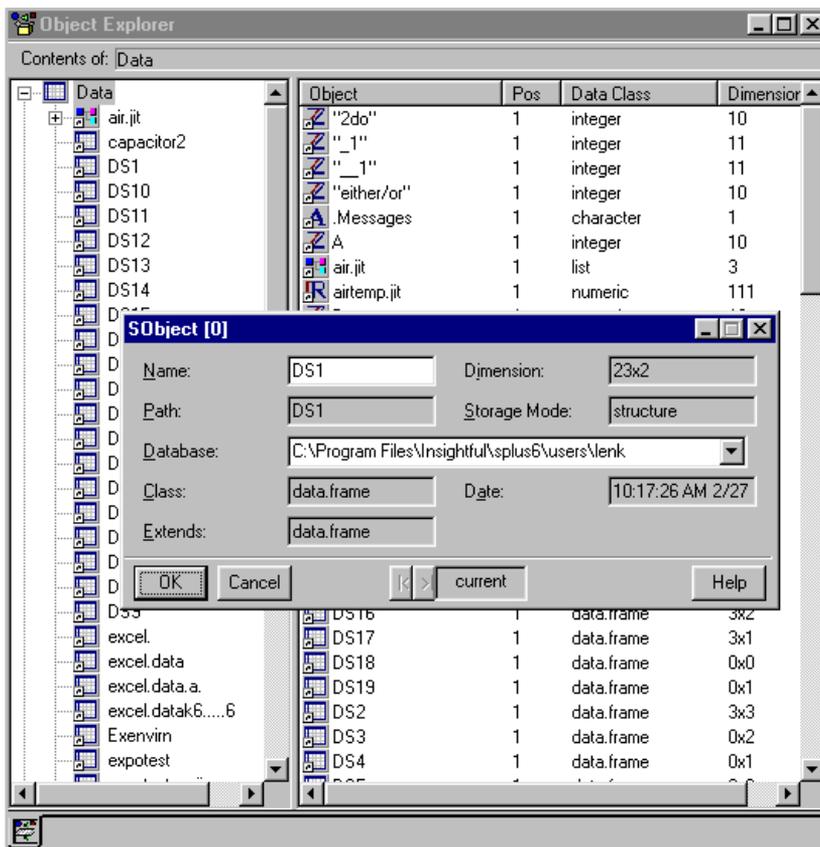


Figure 8.18: A context menu with the item Summary added.

By default, Data Frame is set to **air** in that dialog. Click **OK** and the statistics are sent to a Report window, unless the Command window is open to receive them.

Instead of the built-in `FunctionInfo` object `menuDescribe` and its associated built-in S-PLUS function, user-defined objects can also be used. The procedure for adding a context menu option is identical.

4. Applying the Context Menu to a Class which Inherits from `data.frame`

1. Use the **Select Data** dialog in the Data menu to select the **catalyst** data set. When it opens in the Data window, change the upper left cell to read "180", then change it back to 160. (This won't change the data, but it will write it to your working data, so it will appear in your Object Explorer.)
2. Right-click on the object `catalyst`. The context option `Summary` does not appear, because the object `catalyst` has class design, which inherits from `data.frame`. To confirm this, you can check `Data Class` and `Inheritance` in the `Right Pane` page of the `Object Explorer` property dialog, if this is not already done, and view the information in the right pane of the `Object Explorer`, as in Figure 8.19. Make sure that `Include Derived Classes` is checked in the `Object Explorer` property dialog.

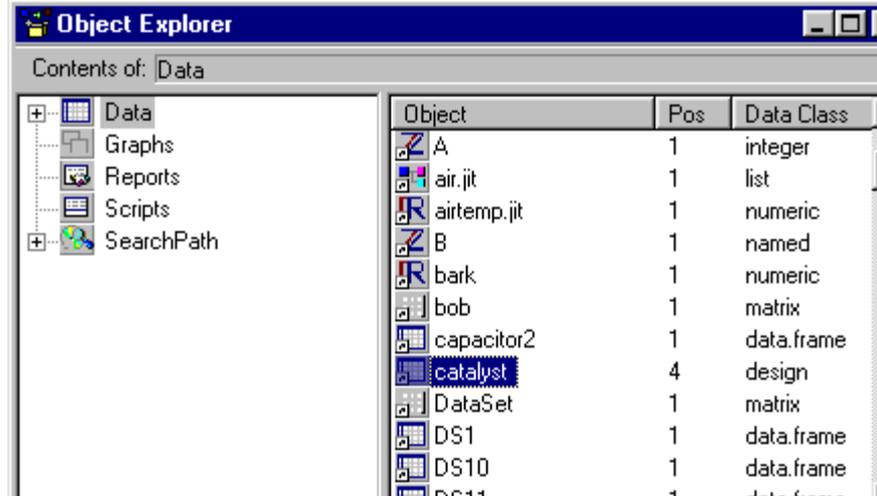


Figure 8.19: *The Object Explorer showing the class of the data.*

3. To enable the context menu for objects in the class design, open the property dialog for the `MenuItem desc`.
4. Enter `data.frame`, `design` in the `Document Type` field.

5. Click **OK**.
6. Return to the page showing data frames and right-click on the object catalyst. The context menu now contains Summary.

STYLE GUIDELINES

Typically Spotfire S+ programmers will begin by writing functions for use in scripts and at the command line. These functions will generally fall into one of the following classes:

- Functions which compute some quantities and return a vector, matrix, `data.frame`, or list. If the result is assigned these values are stored, and if not they are printed using the standard mechanism. Functions such as `mean` and `cor` are of this type.
- Functions which take data and produce plots. The returned value is typically not of interest. Functions such as `xyp1ot` and `pairs` are of this type.
- A set of functions including a modeling function which produces a classed object, and method functions such as `print`, `summary`, `plot`, and `predict`. Functions such as `lm` and `tree` are of this type.

The custom menu and dialog tools allow the creation of a dialog for any function. Hence the programmer may create a dialog which directly accesses a function developed for use at the command line. While this may be acceptable in some cases, experience has shown that it is generally preferable to write a wrapper function which interfaces between the dialog and the command line function.

This section discusses the issues that arise when creating a function for use with a dialog, and describes how these issues are handled by the built-in statistical dialog functions. In addition, we discuss basic design guidelines for statistical dialogs.

Basic Issues

Most functions will perform these steps:

- Accept input regarding the data to use.
- Accept input regarding computational parameters and options.
- Perform computations.
- Optionally print the results.
- Optionally store the results.
- Optionally produce plots.

Modeling functions have additional follow-on actions which are supported at the command line by separate methods:

- Providing additional summaries.
- Producing plots.
- Returning values such as fitted values and residuals.
- Calculating predicted values.

We will first discuss the basic steps performed by any function such as accepting input, performing calculations, printing results, saving results, and making plots. Then we will discuss the issues which arise for modeling functions with methods.

Basic Dialogs

We will begin by discussing the **Correlations and Covariances** dialog. Exploring this dialog and the related analysis and callback functions will display the key issues encountered when constructing functions for dialogs.

The Dialog

The **Correlations and Covariances** dialog is available from the **Statistics ► Data Summaries ► Correlations** menu item.

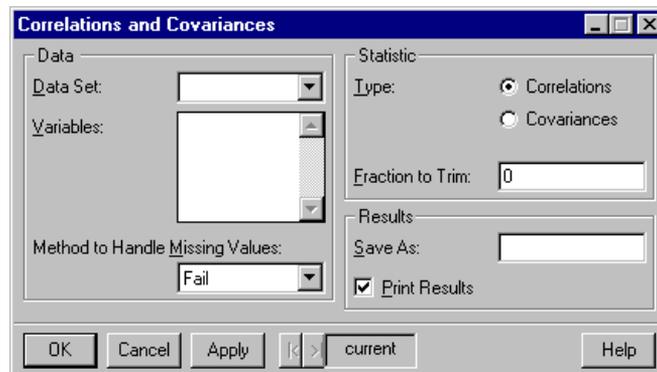


Figure 8.20: *The Correlations and Covariances dialog.*

This dialog provides access to the `cor` and `var` functions. It allows the user to specify the data to use, computation options, a name under which to save the results, and whether to print the results.

Note that the data to use is specified in the upper left corner of the dialog. The user first specifies which Data Frame to use, and then the variables of interest. (Some dialogs will accept matrices or vectors in the Data Frame field, but for simplicity users are encouraged to work with data frames.)

The Results group in the lower right corner of the dialog lets the user specify an object name under which to store the results, and provides a check box indicating whether the results should be printed.

Other options are placed between the Data group and the Results group.

The Function

When **OK** or **Apply** is pressed in the dialog, the `menuCor` function is called. The naming convention for functions called by dialogs is to append `menu` to the command line function name, such as `menuLm`, `menuTree`, and `menuCensorReg`.

The `menuCor` function is:

```
> menuCor
function(data, variables = names(data), cor.p = F, trim = 0,
  cov.p = F, na.method = "fail", print.it = T,
  statistic = "Correlations")
{
# Note cor.p and cov.p have been replaced with statistic.
# They are left in solely for backwards compatibility.
  data <- as.data.frame(data)
  data.name <- deparse(substitute(data))
  if(!missing(variables))
    variables <- sapply(unpaste(variables, sep = ","),
      strip.blanks)
  if(!is.element(variables[[1]], c("<ALL>", "(All
    Variables)"))) {
    if(!length(variables))
      stop("You must select at least one variable\n")
    data <- data[, variables, drop = F]
  }
  dropped.cols <- !sapply(data, is.numeric) | sapply(data,
    is.dates)
  if(all(dropped.cols))
    stop("No numeric columns specified.")
  if(any(dropped.cols)) {
```

```
        warning(paste("Dropping non-numeric column(s) ",
paste(names(data)[
        dropped.cols], collapse = ", "), ".", sep = ""))
        data <- data[, !dropped.cols, drop = F]
    }
    na.method <- casefold(na.method)
    if(statistic == "Correlations" || (cor.p && !cov.p)) {
        coeff <- cor(data, trim = trim, na.method = na.method)
        header.txt <- paste("\n\t*** Correlations for data
in: ", data.name,
        "***\n\n")
    }
    else {
        coeff <- var(data, na.method = na.method)
        header.txt <- paste("\n\t*** Covariances for data in:
", data.name,
        "***\n\n")
    }
    if(print.it) {
        cat(header.txt)
        print(coeff)
    }
    invisible(coeff)
}
```

Input Values

The function arguments are:

```
function(data, variables = names(data), cor.p = F, trim = 0,
cov.p = F, na.method = "fail", print.it = T,
statistic = "Correlations")
```

The function has one argument for each control in the dialog, with the exception of the **Save As** field specifying the name to which to assign the value returned by the function. Default values are present for all arguments except data. A default argument value will be used if the corresponding field in the dialog is left empty.

The first few lines in the function transform these inputs from a form preferable for a dialog field to the format expected by `cor` and `var`.

First the data is transformed to a data frame, to allow the handling of vectors and matrices. The name of the data is stored for use in printing the results:

```
data <- as.data.frame(data)
data.name <- deparse(substitute(data))
```

Next the function constructs the names of the variables of interest. The `variables` argument passed by the dialog is a single string containing a comma delimited list of column names, and perhaps the string “(All Variables)”. This string is broken into a character vector of variable names. If it does not include “(All Variables)” and is not empty, the specified columns of the data are extracted.

```
if(!missing(variables))
  variables <- sapply(unpaste(variables, sep = ","),
    strip.blanks)
if(!is.element(variables[[1]], c("<ALL>", "(All
  Variables)"))) {
  if(!length(variables))
    stop("You must select at least one variable\n")
  data <- data[, variables, drop = F]
}
```

Computations

After the desired set of data is constructed, the statistics are calculated:

```
if(statistic == "Correlations" || (cor.p && !cov.p)) {
  coeff <- cor(data, trim = trim, na.method = na.method)
  header.txt <- paste("\n\t*** Correlations for data
in: ", data.name,
  "***\n\n")
}
else {
  coeff <- var(data, na.method = na.method)
  header.txt <- paste("\n\t*** Covariances for data in:
", data.name,
  "***\n\n")
}
```

The `statistic` argument takes a string, either “Correlations” or “Covariances”; `cor.p` and `cov.p` arguments are logical values indicating whether to form the correlations or covariances which are supported for backward compatibility. The callback function (discussed later) enforces the constraint that only one of these is

TRUE. Note that this could also have been implemented using Radio Buttons passing a character string rather than as separate Check Boxes.

The `trim` and `na.method` arguments are passed directly to the computational functions.

A character string is also constructed for use as a header when printing the results.

Printing Results

The standard behavior in Spotfire S+ is to either print the results from a function or store them under a specified name using assignment. That is, a user may either see the results printed using

```
> cor(swiss.x)
```

save the results using

```
> swiss.cor <- cor(swiss.x)
```

or do both by saving the results and then printing the object

```
> swiss.cor <- cor(swiss.x)
> swiss.cor
```

Explicitly printing the results in a function is frowned upon unless the function is a `print` method for a classed object. The evaluation mechanism determines whether to print the result.

This convention is waived for the dialog functions, as it is necessary to provide a mechanism for both saving and printing the output within the function.

Another difference between using a function from the command line and from a dialog is that the command line alternates between an expression and the output related to that expression. Hence it is clear which expression and output go together. The output from a dialog is not preceded by an expression (the expression evaluated will be stored in the history log but is not printed to the output stream). Hence it is necessary to provide a header preceding the output which indicates the source of the output. The header lines also serve to separate subsequent sets of output.

If the user requests printed output, the header is printed with `cat`, and the result object with `print`:

```

    header.txt <- paste("\n\t*** Covariance for data in:
", data.name, "***\n\n")
    ...
    if(print.it) {
      cat(header.txt)
      print(coeff)
    }

```

Generally `cat` is used to print character strings describing the output, and `print` is used for other objects.

Note that that convention for header lines is to use a character string of the form:

```

"\n\t*** Output Description ***\n\n"

```

Saving Results

In this dialog, the results need not be explicitly saved within the function. The command is written such that the result is assigned to the name specified in **Save As** if a name is specified.

Note that the value is returned invisibly:

```
invisible(coeff)
```

As we have already printed the result if printing is desired, it is necessary to suppress the autoprinting which would normally occur if the result were returned without assignment.

In some cases it is necessary to assign the result within the function. In particular, this is required if the function is creating the data and then displaying it in a Data window. For example, this is done in the `menuFacDesign` function, which creates a data frame `new.design` containing a factorial design, and displays this in a Data window.

```

if(missing(save.name))
  return(new.design)
else {
  assign(save.name, new.design, where = 1,
         immediate = T)
  if(is.sgui.app() && show.p)
    guiOpenView(classname = "data.frame",
                Name = save.name)
  invisible(new.design)
}

```

If `save.name` is not specified, the result is simply returned. Otherwise, the result is immediately assigned to the working directory. Then the data frame is displayed in a Data window if the Windows Spotfire S+ GUI is being run and the user specifies `show.p=T` by checking the **Show in Data Window** box in the **Factorial Design** dialog.

The explicit assignment is necessary because the data frame must exist as a persistent object on disk before it can be displayed in a Data window.

Saving Additional Quantities

In some cases the user may want access to other quantities which are not part of the standard object returned by the function, such as residuals or predicted values. At the command line these functions can be accessed using extractor functions such as `resid` and `predict`. In dialogs it may be preferable to save these objects into specified data frames using the save mechanism as described above. The section Modeling Dialog Saved Results discusses this situation.

Plots

The Windows Spotfire S+ GUI supports multiple coexisting Graph sheets, each of which may have multiple tabbed pages. When a new graph is created it may do one of three things:

- Replace the current graph (typically the graph most recently created).
- Create a new tab on the current Graph sheet.
- Create a new Graph sheet.

The default behaviour is for a statistical dialog function to open a new Graph sheet before creating graphs. If the function produces multiple graphs, these appear on multiple tabs in the new Graph sheet.

This autocreation of new Graph sheets may annoy some users due to the proliferation of windows. The **Graphs Options** dialog has a **Statistics Dialogs Graphics: Create New Graph Sheet** check box which indicates whether or not to create a new Graph sheet for each new set of plots.

It is good form for any plots created from dialogs to follow the dictates of this option. This is done by calling `new.graphsheet` before plots are produced. This function will create a new Graph sheet if the abovementioned option specifies to do so. The `new.graphsheet`

function should only be called if plots are to be produced, and should only be called once within the function as calling it multiple times would open multiple new Graph sheets.

The `menuAcf` function provides an example of the use of `new.graphsheet`:

```
if(as.logical(plot.it)) {
  new.graphsheet()
  acf.plot(acf.obj)
}
```

The Callback Function

Most dialogs of any real complexity will have some interactions between the allowable argument values. In the **Correlations and Covariances** dialog the **Fraction to Trim** is only relevant for correlations. Hence this field should be disabled if **Variance/Covariance** is checked. The callback function `backCor` updates the values and enable status of controls based on actions in the dialog.

When the dialog is launched, **OK** or **Apply** is pressed, or a control is changed, the callback function is executed. The function is passed a data frame containing character strings reflecting dialog prompts, values, option lists, and enable status. These strings may be accessed and modified to make changes to the dialog.

This function starts by getting the name of the active property. This is the property which was last modified.

```
backCor <- function(data)
{
  activeprop <- cbGetActiveProp(data)
```

If the dialog has just been launched then **Fraction to Trim** should only be enabled if **Correlation** is checked. If **Correlation** is checked then **Variance/Covariance** should be unchecked, and vice versa. If which check box is checked changes, the enable status of **Fraction to Trim** must change. The next set of lines enforces these constraints.

```
if(cbIsInitDialogMessage(data) || activeprop ==
  "SPropCorrP" || activeprop == "SPropCovP") {
  if(activeprop == "SPropCorrP") {
    if(cbGetCurrValue(data, "SPropCorrP") ==
      "F") {
      data <- cbSetEnableFlag(data,
```

```
"SPropTrim", F)
data <- cbSetCurrValue(data,
  "SPropCovP", "T")
```

...

If the dialog has just been launched or the **Data Frame** has changed, the list of variables must be created. This is done by checking that an object of the specified name exists, and if so getting the object's column names and pasting them together with the (All Variables) string. Note that the list of variable names is passed as a single comma delimited string rather than as a vector of strings.

```
if(activeprop == "SPropDataX2" || cbIsInitDialogMessage(
  data)) {
  if(exists(cbGetCurrValue(data, "SPropDataX2")))
  {
    x.names <- names(get(cbGetCurrValue(data,
      "SPropDataX2")))
    x.names <- paste(c("(All Variables)",
      x.names), collapse = ",")
    data <- cbSetOptionList(data,
      "SPropVariableX2", x.names)
  }
}
```

Lastly, the data frame containing the dialog status information is returned.

```
invisible(data)
```

The most common uses of callback functions are to fill variable lists and to enable/disable properties as is done by `backAcf`. For further examples, search for functions whose names start with `back`, or look at the `FunctionInfo` for a dialog with callback behaviour of interest to determine the name of the relevant callback function.

Modeling Dialogs

A powerful feature of Spotfire S+ is the object-oriented nature of the statistical modeling functions. Statistical modeling is an iterative procedure in which the data analyst examines the data, fits a model, examines diagnostic plots and summaries for the model, and refines the model based on the diagnostics. Modeling is best performed interactively, alternating between fitting a model and examining the model.

This interactive modeling is supported in Spotfire S+ by its class and method architecture. Generally there will be a modeling function (such as `lm` for linear regression) which fits a model, and then a set of methods (such as `print`, `plot`, `summary`, and `anova`) which are used to examine the model. The modeling function creates a model object whose class indicates how it is handled by the various methods.

This differs from other statistical packages, in which all desired plots and summaries are typically specified at the time the model is fit. If additional diagnostic plots are desired the model must be completely refit with options indicating that the new plots are desired. In Spotfire S+ additional plots may be accessed by simply applying the plot method to the model object.

In moving from a set of command line functions to dialogs for statistical modeling, the desired level of granularity for action specification changes. At the command line the basic strategy would be to issue a command to fit the model, followed by separate commands to get the desired plots and summaries. The ability to use such follow-on methods is still desirable from a graphical user interface, but it should be a capability rather than a requirement. The user will generally want to specify options for fitting the model plus desired plots and summaries in a single dialog, with all results generated when the model is fit.

The design of the statistical modeling dialogs is such that the user may specify the desired summaries and plots at the time the model is fit, but it is also possible to right-click on a model object in the Object Explorer and access summary and plot methods as a follow-on action. Generally the Results, Plot, and Predict tabs on the modeling dialog are also available as separate dialogs from the model object context menu.

This section describes the general design of statistical modeling dialogs by examining the **Linear Regression** dialog. The following section describes the structure of the functions connected to the

modeling and follow-on method dialogs. The statistical modeling dialogs follow the same design principles as are described here, but details will vary.

Model Tab

The **Model** tab describes the data to use, the model to fit, the name under which to save the model object, and various fitting options. It is typical to have **Data**, **Formula**, and **Save Model Object** groups which are similar to those in the **Linear Regression** dialog.

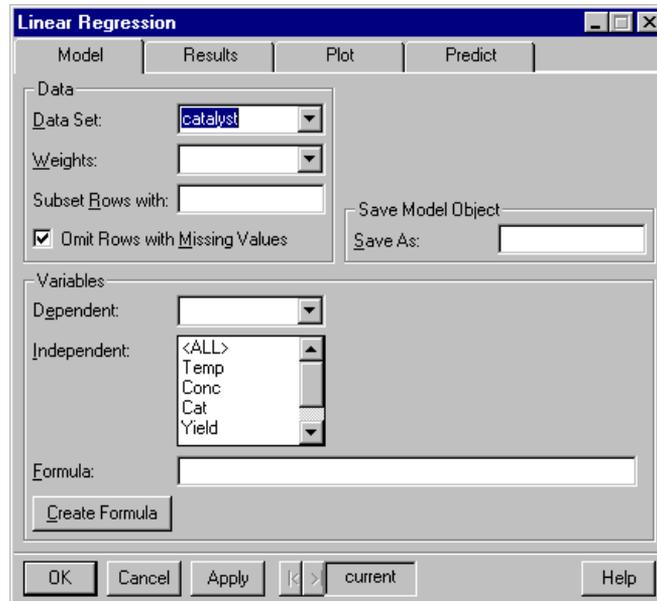


Figure 8.21: The Model tab of the Linear Regression dialog.

Data Group

The **Data Set** property is a drop-down list of available data sets. This list is filled with the data sets which are in the working database, or have been displayed by filtering on other databases in the Object Explorer. This specifies the data argument to the modeling function.

The **Weights** property is a list of columns in the selected data set. The selected column will be used as weights in the model. This specifies the weights argument to the modeling function.

The **Subset Rows with** property takes an expression which is used as the subset expression in the model. This specifies the subset argument to the modeling function.

The **Omit Rows with Missing Values** check box specifies how missing values are handled. Checking this box is equivalent to specifying `na.action=na.omit`, while leaving it unchecked is equivalent to `na.action=na.fail`. Some dialogs (such as **Correlations and Covariances**) instead have a **Method to Handle Missing Values** list box, which provides additional missing value actions.

Variables Group

The Variables group includes controls for specifying the **Dependent** and **Independent** variables. As you select or enter variables in these controls, they are echoed in the **Formula** control. The **Formula** specifies the form of the model, that is what variables to use as the predictors (independent variables) and the response (dependent variables). This specifies the `formula` argument to the modeling function.

Most modeling dialogs have a **Create Formula** button, which launches the Formula Builder dialog when pressed. This dialog allows point-and-click formula specification.

Dialogs in which the formula specifies a set of covariates rather than predictors and a response (such as **Factor Analysis**) have a **Variables** list rather than a **Create Formula** button.

Save Model Object Group

The **Save As** edit field specifies the name under which to save the model object. The returned value from the function called by the dialog is assigned to this name. Typically the default value is prefaced by `last` and is model specific, as in `last.lm`.

Options Tab

In the **Linear Regression** dialog, all of the necessary fitting options are available on the **Model** tab. Some other modeling dialogs, such as the **Logistic Regression** dialog, have more options which are placed on a separate tab. An **Options** tab may be useful either due to the availability of a large number of options, or to shelter the casual user from more advanced options.

Results Tab

The **Results** tab generally has groups for specifying the printed and saved results.

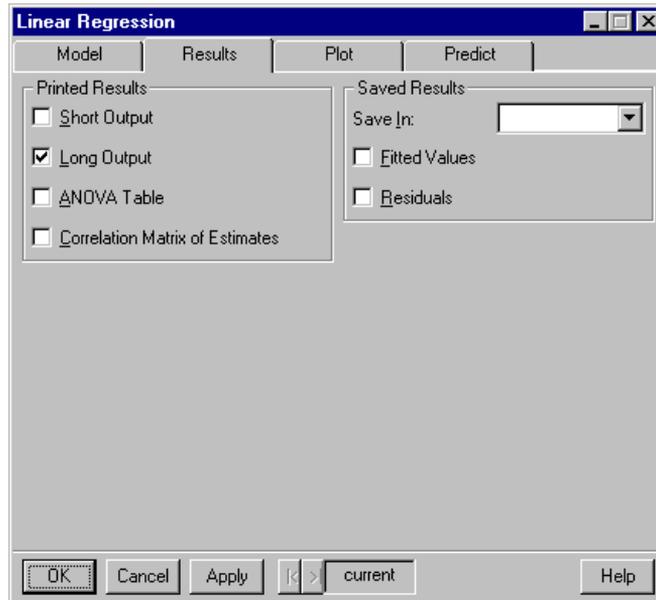


Figure 8.22: *The Results page of the Linear Regression dialog.*

Printed Results Group

The **Printed Results** specify the types of summaries to print. These summaries will be displayed in a Report window, or in another location as specified in the **Text Output Routing** dialog.

Checking the **Short Output** check box generally indicates that the `print` method for the model object will be called.

Checking the **Long Output** check box generally indicates that the `summary` method for the model object will be called.

Other options will vary based on the statistical model.

Saved Results Group

The **Saved Results** specify results to be saved separate from the model object, such as fitted values and residuals. These saved components are usually of the same length as the data used in the

model. It is often of interest to plot these quantities against the data used to construct the model, and hence it is convenient to save these values in a data frame for later use in plotting or analysis.

The **Save In** edit field takes the name of a data frame in which to save the results. This may be a new data frame or the data frame used to construct the model. If the data frame named exists and has a different number of rows than are in the results, then a new name will be constructed and the results saved in the new data frame.

Check boxes specify what results to save. Common choices include **Fitted Values** and **Residuals**.

Plot Tab

The Plot tab specifies which plots to produce and plotting options. Typically a **Plots** group provides check boxes to select plot types to produce. Other groups provide options for the various plots.

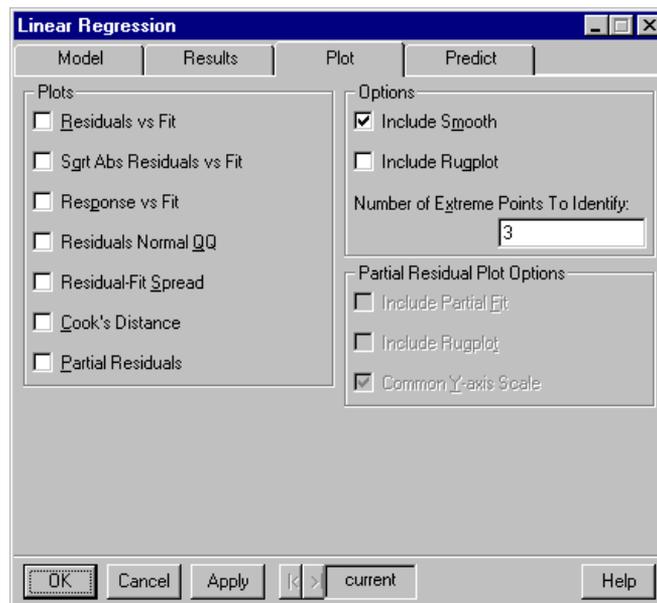


Figure 8.23: *The Plot page of the Linear Regression dialog.*

Predict Tab

The **Predict** tab specifies whether predicted values will be saved, using similar conventions as the **Saved Results** group on the **Results** tab.

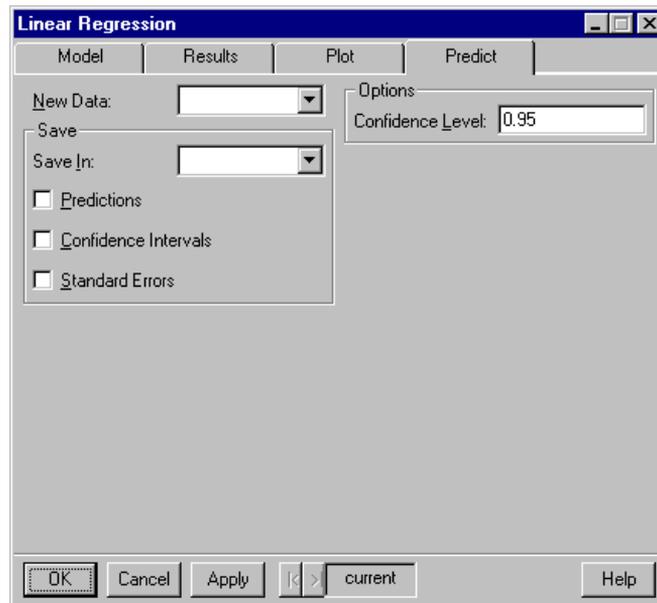


Figure 8.24: *The Predict page of the Linear Regression dialog.*

The **New Data** edit field accepts the name of a data frame containing observations for which predictions are desired. This specifies the `newdata` argument to `predict`. If this is left empty the data used to fit the model will be used.

The **Save In** edit field takes the name of a data frame in which to save the results. This may be a new data frame or the data frame used to construct the model. If the data frame named exists and has a different number of rows than are in the results, then a new name will be constructed and the results saved in the new data frame.

Check boxes specify what results to save. Common choices include **Predictions**, **Confidence Intervals**, and **Standard Errors**.

Other options related to prediction may also be present.

Other Tabs

A statistical model may have additional methods specific to that type of model. The dialog for this model may have additional tabs, such as the **Prune/Shrink** tab on the **Tree Regression** dialog. The only limitation on additional tabs is that each dialog is limited to at most five tabs.

Modeling Dialog Functions

Command line functions for statistical models generally consist of a fitting function (such as `lm`) and method functions for that model (such as `print.lm`, `summary.lm`, `plot.lm`, and `predict.lm`). Similarly, the wrapper functions called by the dialogs consist of a main dialog function (such as `menuLm`) and method functions (`tabSummary.lm`, `tabPlot.lm`, and `tabPredict.lm`). The method functions are called by the main function, and also from separate dialogs available by right-clicking on the model object in the Object Explorer. This structure allows the methods to be called either at the time of fitting the model, or later for use with a fitted model object.

The convention is to name the main dialog function with the preface `menu`, and the method functions with the preface `tab`. The suffix of the function reflects the name of the related command line function.

Main Function The main dialog function has two primary purposes: fitting the model and calling the method functions. The main dialog function for linear regression is `menuLm`:

```
menuLm <-  
function(formula, data, weights, subset, na.omit.p = T,  
  print.short.p = F, print.long.p = T, print.anova.p = T,  
  print.correlation.p = F, save.name = NULL, save.fit.p =  
  F, save.resid.p = F, plotResidVsFit.p = F,  
  plotSqrtAbsResid.p = F, plotResponseVsFit.p = F,  
  plotQQ.p = F, plotRFSpread.p = F, plotCooks.p = F,  
  smooths.p = F, rugplot.p = F, id.n = 3,  
  plotPartialResid.p = F, plotPartialFit.p = F,  
  rugplotPartialResid.p = F, scalePartialResid.p = T,  
  newdata = NULL, predobj.name = NULL, predict.p = F, ci.p  
  = F, se.p = F, conf.level = 0.95)  
{  
  fun.call <- match.call()  
  fun.call[[1]] <- as.name("lm")  
  if(na.omit.p)  
    fun.call$na.action <- as.name("na.omit")  
  else fun.call$na.action <- as.name("na.fail")  
  fun.args <- is.element(arg.names(fun.call), arg.names(  
    "lm"))  
  fun.call <- fun.call[c(T, fun.args)]  
  lobj <- eval(fun.call)#  
  # Call summary function:  
  tabSummary.lm(lobj, print.short.p, print.long.p,  
    print.correlation.p, print.anova.p, save.name,  
    save.fit.p, save.resid.p)#  
  # Call plot function:  
  if(any(c(plotResidVsFit.p, plotSqrtAbsResid.p,  
    plotResponseVsFit.p, plotQQ.p, plotRFSpread.p,  
    plotCooks.p, plotPartialResid.p))) tabPlot.lm(  
    lobj, plotResidVsFit.p,  
    plotSqrtAbsResid.p, plotResponseVsFit.p,  
    plotQQ.p, plotRFSpread.p, plotCooks.p,  
    smooths.p, rugplot.p, id.n,  
    plotPartialResid.p, plotPartialFit.p,  
    rugplotPartialResid.p,  
    scalePartialResid.p)#
```

```
# Call predict:
  if(any(c(predict.p, ci.p, se.p)))
    tabPredict.lm(lmobj, newdata, predobj.name,
                  predict.p, ci.p, se.p, conf.level)
  invisible(lmobj)
}
```

This function has one argument corresponding to each property in the dialog, with the exception of the **Save As** field and invisible fields used for formatting.

The first eight lines of the function are used to fit the model. The approach is to construct an expression which specifies the appropriate call to the model function, and then to evaluate this expression. The somewhat sophisticated syntax is used so that the `call` component in the model object has the appropriate names as specified in the dialog. This general recipe may be used for any function, with “`lm`” replaced by the name of the modeling function.

Note that `na.action` is specified as a check box in the dialog, but as a function name in `lm`. This necessitates modification of this argument before evaluating the call to `lm`.

After the model has been fit, the methods `tabSummary.lm`, `tabPlot.lm`, and `tabPredict.lm` are called. For efficiency, calls are only made to `tabPlot.lm` and `tabPredict.lm` if the options are specified such that calling these functions will produce some action.

Finally, the model object is returned. It is returned invisibly since any printing of the object has been handled by `tabSummary.lm`.

Summary Method The summary method produces printed summaries and saves specified results in a data frame separate from the model object. The summary method for the **Linear Regression** dialog is `tabSummary.lm`:

```
tabSummary.lm <-
function(lmobj, print.short.p = F, print.long.p = T,
        print.correlation.p = F, print.anova.p = F, save.name =
        NULL, save.fit.p = F, save.resid.p = F)
{
  if(print.short.p || print.long.p || print.anova.p) {
    cat("\n\t*** Linear Model ***\n")
    if(print.short.p) {
```

```
        print(lmobj)
    }
    if(print.long.p) {
        print(summary(lmobj, correlation =
            print.correlation.p))
    }
    if(print.anova.p) {
        cat("\n")
        print(anova(lmobj))
    }
    cat("\n")
}
# Save results if requested:
if(any(c(save.fit.p, save.resid.p)) && !is.null(
    save.name)) {
    saveobj <- list()
    if(save.fit.p)
        saveobj[["fit"]] <- fitted(lmobj)
    if(save.resid.p)
        saveobj[["residuals"]] <- residuals(
            lmobj)
    saveobj <- data.frame(saveobj)
    n.save <- nrow(saveobj)
    if(exists(save.name, where = 1)) {
        if(inherits(get(save.name, where = 1),
            "data.frame") && nrow(get(
                save.name, where = 1)) == n.save
        )
            assign(save.name, cbind(get(
                save.name, where = 1), saveobj
            ), where = 1)
    } else {
        newsave.name <- unique.name(
            save.name, where = 1)
        assign(newsave.name, saveobj,
            where = 1)
        warning(paste(
            "Fit and/or residuals saved in"
            ,
            newsave.name))
    }
}
```

```

    }
    else assign(save.name, saveobj, where = 1)
    invisible(NULL)
  }
  invisible(lmobj)
}

```

The first part of this function is responsible for printing the specified summaries. If any printed output is specified, a header will be printed demarcating the start of the output. Based on option values, the `print`, `summary`, and other methods for the model will be called.

The second part of the function concerns itself with saving the requested values. Extractor functions such as `fitted` and `residuals` are used to get the desired values. The remainder of the code specifies whether to add columns to an existing data frame, create a new data frame with the specified name, or create a new data frame with a new name to avoid overwriting an existing object.

The model object passed to this function is returned invisibly.

Plot Method

The `plot` function opens a new Graph sheet if necessary, and produces the desired plots. The plot method for the **Linear Regression** dialog is `tabPlot.lm`:

```

tabPlot.lm <-
function(lmobj, plotResidVsFit.p = F, plotSqrtAbsResid.p =
  F, plotResponseVsFit.p = F, plotQQ.p = F,
  plotRFSpread.p = F, plotCooks.p = F, smooths.p = F,
  rugplot.p = F, id.n = 3, plotPartialResid.p = F,
  plotPartialFit.p = F, rugplotPartialResid.p = F,
  scalePartialResid.p = T, ...)
{
  if(any(c(plotResidVsFit.p, plotSqrtAbsResid.p,
    plotResponseVsFit.p, plotQQ.p, plotRFSpread.p,
    plotCooks.p, plotPartialResid.p)))
    new.graphsheet()
  if(any(c(plotResidVsFit.p, plotSqrtAbsResid.p,
    plotResponseVsFit.p, plotQQ.p, plotRFSpread.p,
    plotCooks.p))) {
    whichPlots <- seq(1, 6)[c(plotResidVsFit.p,
      plotSqrtAbsResid.p, plotResponseVsFit.p,
      plotQQ.p, plotRFSpread.p, plotCooks.p)]
  }
}

```

```
        plot.lm(lmobj, smooths = smooths.p, rugplot =
              rugplot.p, id.n = id.n, which.plots =
              whichPlots, ...)
    }
    if(plotPartialResid.p) {
        partial.plot(lmobj, residual = T, fit =
                    plotPartialFit.p, scale =
                    scalePartialResid.p, rugplot =
                    rugplotPartialResid.p, ...)
    }
    invisible(lmobj)
}
```

If any plots are desired, the function first calls `new.graphsheet` to open a new Graph sheet if necessary. The plot method for `lm` is then called. Some additional plots useful in linear regression are produced by the `partial.plots` function, which is called if partial residual plots are desired.

The model object passed to this function is returned invisibly.

Predict Method

The `predict` function obtain predicted values for new data or the data used to fit the model. The `predict` method for the **Linear Regression** dialog is `tabPredict.lm`:

```
tabPredict.lm <-
function(object, newdata = NULL, save.name, predict.p = F,
        ci.p = F, se.p = F, conf.level = 0.95)
{
    if(is.null(newdata))
        predobj <- predict(object, se.fit = se.p || ci.p
                          )
    else predobj <- predict(object, newdata, se.fit = se.p ||

                           ci.p)
    if(ci.p) {
        if(conf.level > 1 && conf.level < 100)
            conf.level <- conf.level/100
        t.value <- qt(conf.level, object$df.residual)
        lower.name <- paste(conf.level * 100, "% L.C.L.",
                           sep = "")
        upper.name <- paste(conf.level * 100, "% U.C.L.",
```

```

        sep = "")
        predobj[[lower.name]] <- predobj$fit - t.value *
            predobj$se.fit
        predobj[[upper.name]] <- predobj$fit + t.value *
            predobj$se.fit
    }
# remove prediction column and se column if not requested:
if(!predict.p)
    predobj$fit <- NULL
if(!se.p)
    predobj$se.fit <- NULL
predobj$residual.scale <- NULL
predobj$df <- NULL
predobj <- as.data.frame(predobj)
n.predict <- nrow(predobj)
if(exists(save.name, where = 1)) {
    if(inherits(get(save.name, where = 1),
        "data.frame") && nrow(get(save.name,
            where = 1)) == n.predict)
        assign(save.name, cbind(get(save.name,
            where = 1), predobj), where = 1)
    else {
        newsave.name <- unique.name(save.name,
            where = 1)
        assign(newsave.name, predobj, where = 1)
        warning(paste("Predictions saved in",
            newsave.name))
    }
}
else assign(save.name, predobj, where = 1)
invisible(NULL)
}

```

The function calls `predict.lm` to obtain predicted values and standard errors. It then calculates confidence intervals, and removes any undesired components from the output. The predictions are then stored in a data frame using the same type of algorithm as in `tabSummary.lm`.

No value is returned.

- Other Methods** If the dialog has additional tabs, other dialog methods will be available. For example, the **Tree Regression** dialog has a **Prune/Shrink** tab with a corresponding function `tabPrune.tree`.
- Callback Functions** Modeling dialogs will have callbacks similar to those for a simple dialog. The callback function for **Linear Regression** is `backLm`.
- Method dialogs may also need callback functions for use when the related dialog is launched from the model object's context-menu. An example is the `tabPlot.princomp` callback function used by the **Principal Components** plot dialog. Method dialogs need callback functions less frequently than do the main modeling dialogs.
- Class Information** Every model object has a class indicating what type of model the object is. For example, linear regression model objects are of class `lm`. At the command line, functions such as `print` look to see if there is a special function `print.lm` to use when they are given an `lm` object, and if not they use the default `print` method for the object.
- Similarly, the Object Explorer has a limited set of actions it can perform on any object. In addition, it can allow class-specific actions. The `ClassInfo` object tells the Object Explorer what to do with objects of the specified class. In particular, the double-click action and the context menu may be specified.
- Double-Click Action** The double-click action is the action to perform when the model object is double-clicked in the Object Explorer. The convention for statistical models is to produce printed summaries. In linear regression the `tabSummary.lm` function is called.

Context Menu

The context menu is the menu launched when the user right-clicks on a model object in the Object Explorer. Figure 8.25 displays the context menu for linear regression (lm) objects:

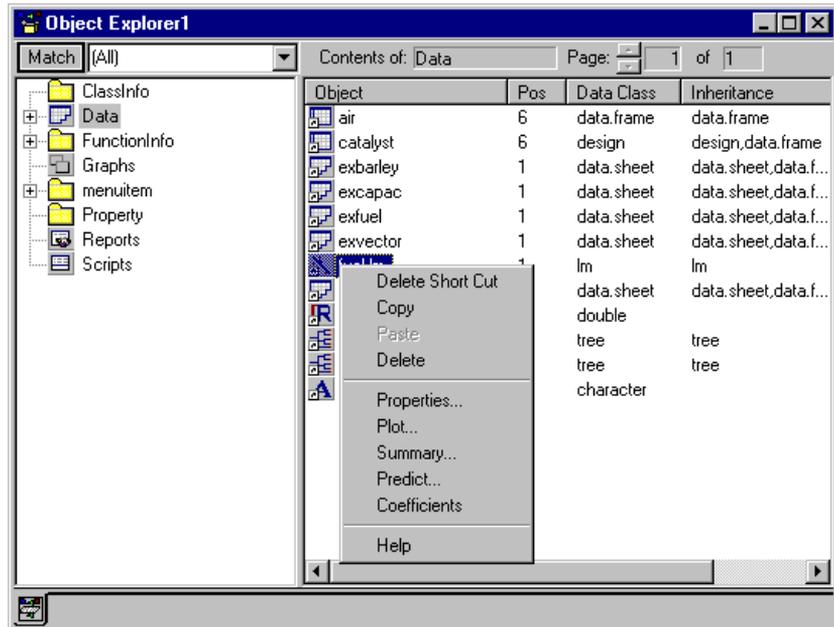


Figure 8.25: *The context menu for linear regression objects.*

This menu includes the methods made available through the dialog method functions such as **Summary**, **Plot**, and **Predict**. Each of these menu items launches a dialog which calls the respective function. For example, selecting **Summary** launches the **Linear Regression Results** dialog shown in Figure 8.26.

Some additional methods may be present, such as the **Coefficients** menu item which calls the `coef` function. This allows the user to quickly print the coefficients in the current Report window.

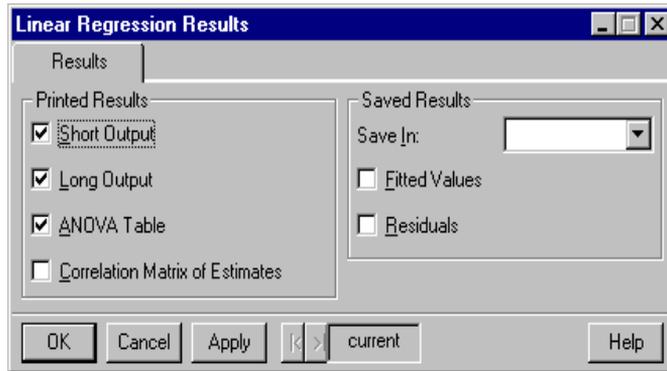


Figure 8.26: The *Linear Regression Results* dialog.

Method Dialogs

The dialog functions are designed to make it easy to have dialogs for the various method functions.

A `FunctionInfo` object for each of the method dialogs defines the relevant dialog and its relationship to the method function. Most of the properties will be exactly the same as the properties in the modeling dialog. The `SPropCurrentObject` property is used as the first argument to indicate that the currently selected object is the model object of interest, and `SPropInvisibleReturnObject` is used for the return value.

Look at the properties of the `FunctionInfo` object for `tabSummary.lm` for details on the `FunctionInfo` object properties for the dialog given above.

Dialog Help

The built-in statistical dialogs in Spotfire S+ have help entries in the main Spotfire S+ help file. As this is a compiled HTML help file, it may not be extended by the user. However, the user may still make help information available for their dialogs.

The **HelpCommand** property of a `FunctionInfo` object specifies a S-PLUS expression to evaluate when the dialog's Help button is pushed.

If the user has created a help file for the command line function, the help for this function may be launched using an expression such as `help(menuLm)`. The help for the dialog might also be placed in a separate text file and displayed using an expression such as `system("notepad C:/myhelp.txt")`. With creative use of `system`, the user is able to display help files in another format such as HTML or Word.

Introduction	370
Creating a Library	372
Chapters vs. Libraries	373
Steps in Creating a Library	373
Creating Directories	374
Storing Functions	374
Storing Interface Objects	375
Copying Help Files	376
Storing Other Files	377
Start-up and Exit Actions	377
Distributing the Library	379

INTRODUCTION

Preceding chapters described how to create functions, dialog boxes, and other tools. Spotfire S+ libraries are a convenient way to package such user-created functionality to share it with other users.

A library can contain the following types of objects:

- Functions.
- Example data sets.
- Graphical user interface objects such as menus and dialog boxes.
- Help files.
- Compiled C and Fortran objects ready for dynamic or DLL loading.

This chapter describes how to create and use libraries.

Example Functions

The example in Chapter 8, *Extending the User Interface*, demonstrated creating a function, `my.sqrt`, to calculate and print the square root of a number. The example also demonstrated creating a dialog box for this function and a menu item on the **Data** menu to launch the dialog box. In this chapter, create a library for these objects.

The following commands create the function and the interface objects:

```
my.sqrt <- function(x){
  y <- sqrt(x)
  cat("\nThe square root of ",x," is ",y, ".\n",sep="")
  invisible(y)
}

guiCreate(classname="Property", Name="MySqrtInput",
  DialogControl="String", UseQuotes=F,
  DialogPrompt="Input Value")

guiCreate(classname="FunctionInfo", Name="my.sqrt",
  DialogHeader="Calculate Square Root",
  PropertyList="SPropInvisibleReturnObject,
```

```
    MySqrtInput",  
    ArgumentList="#0=SPropInvisibleReturnObject,  
                #1=MySqrtInput")  
  
guiCreate(classname="MenuItem",  
          Name="$$SPlusMenuBar$Data$MySqrt",  
          Type="MenuItem",  
          MenuItemText="Square Root...",  
          Action="Function", Command="my.sqrt")
```

CREATING A LIBRARY

If you want to distribute functions and related files in an organized manner, create a library and give it to your users to attach in Spotfire S+. Attaching a library makes the objects in that library available to the Spotfire S+ user by adding the relevant **.Data** directory to the search list.

The library can contain a `.First.lib` function, specifying actions that Spotfire S+ performs when the user attaches the library. Typically, this function might include dynamically-loading C or Fortran code and code that creates GUI objects such as menus, toolbars, and dialog boxes. The library can also contain a `.Last.lib` function, which specifies actions to perform when the user either unloads the library or exits the application.

The user can use any directory containing a **.Data** directory as a library. If the **.Data** directory is a subdirectory of the **library** directory, which is located in the Spotfire S+ program directory, then the user can attach the library by using the following function call:

```
library(libraryname)
```

where *libraryname* is the name of the library to attach. For example, `library(cluster)` attaches the built-in `cluster` library. The specified library does not need to reside in the **library** directory under the Spotfire S+ program directory. If it resides elsewhere, just use the `lib.loc` argument to `library` to specify the path to the directory containing the library directory. For example, if the library is called `mylib`, and it exists as a directory in `/homes/rich`, then attach the library using the following function call:

```
> library(mylib, lib.loc = "/homes/rich")
```

To share a library with users who share a file system, and who can receive distributed routines, create the library anywhere on the file system, and then tell users to use the `library` function with the specified `lib.loc` argument.

The easiest way to distribute the library to users who do not share a file system is to create a **.zip** archive (on Windows[®]) or a **.tar** archive (on UNIX[®] or Linux[®]) containing the library folder and its contents, and then to distribute the archive to the users.

Chapters vs. Libraries

When we speak of “chapters” and “libraries” in Spotfire S+, in both cases, we are speaking essentially of directories containing objects, functions, image files, and related files (such as compiled C or Fortran code). The difference between chapters and libraries lies in how they are used:

- Chapters are directories containing objects, functions, and related files used for data analysis in particular projects. When you use a chapter, attach it to a project as writable, using the `attach` function.
- Libraries are directories containing objects, functions, and related files intended for ongoing use across different projects. When you use a library, attach it as read-only using the `library` function.

Steps in Creating a Library

Create the directory structure for the library in Windows Explorer, or by using system calls. The high-level steps, which are outlined in more detail below, are:

1. Create a main directory for the library in the **library** subdirectory of the program directory and initialize it for Spotfire S+ by using the function `createChapter`.
2. Source the objects into the **.Data** directory in the new directory.
3. Place any additional help files in the **__Hhelp** directory.
4. Optionally, write `.First.lib` and `.Last.lib` functions to specify actions to perform upon loading and unloading the library. For example, specify displaying GUI objects such as toolbars and menu options when the library loads.
5. Optionally, add a **.Prefs** directory parallel to the **.Data** directory that `createChapter` creates, and then copy GUI files such as toolbar bitmaps (***.bmp**) into the **.Prefs** directory.

6. Store menu and dialog box objects in ***.DFT** files placed in the **.Prefs** directory.

Creating Directories

In Windows, the easiest way to create directories is using Windows Explorer. If you prefer a programmatic approach, use the `mkdir` command.

First, create directories using the function `mkdir`. Use `getenv("SHOME")` to find the location of the Spotfire S+ program directory.

```
> mkdir(paste(getenv("SHOME"),
+ "\\library\\mylib", sep=""))
> mkdir(paste(getenv("SHOME"),
+ "\\library\\mylib\\.Data", sep=""))
> mkdir(paste(getenv("SHOME"),
+ "\\library\\mylib\\.Prefs", sep=""))
> mkdir(paste(getenv("SHOME"),
+ "\\library\\mylib\\.Data\\_Help", sep=""))
```

Storing Functions

Next, use `library` to attach the new `mylib` library. Use `pos=2` to specify that it should be attached as the second directory in the search list.

```
> library(mylib,pos=2)
```

Use `assign` to copy `mylib`-related objects into this directory. Make a vector of names of the functions to copy, and then loop over this vector.

```
> mylib.funcs<-c("my.sqrt")
> for (i in 1:length(mylib.funcs)){
+ assign(mylib.funcs[i],
+ get(mylib.funcs[i]),where=2)
+ }
```

(In this example, the loop is unnecessary, because the library consists of only a single function. The above syntax is provided to use with multiple objects.)

Note that running this function produces warnings indicating that you are creating objects in the second directory on the search list with the same names as objects in other directories on the search list. These

warning are expected, and, in this case, indicate that you should remove the objects from the first directory after making the copies to avoid duplicate names.

An alternative approach is to keep a text file, `mylib.ssc`, containing the functions, and then attach the library directory as the working directory and use `source` to create the functions in this directory.

```
> attach(paste(getenv("SHOME"),
+ "\\library\\mylib\\.Data", sep=""), pos=1)
> source("mylib.ssc")
```

Storing Interface Objects

User-created interface objects can include `MenuItem`, `ToolBar`, `ToolBarButton`, `Property`, `FunctionInfo`, and `ClassInfo` objects. In contrast to function and data objects, these objects are not stored in separate files contained in a `.Data` directory. Instead, they are serialized to class-specific files in the user's `.Prefs` directory.

`ClassInfo`, `FunctionInfo`, and `Property` objects are stored in `axclass.dft`, `axfunc.dft`, and `axprop.dft`. `ToolBar` and `ToolBarButton` objects are stored in `*.STB` files, with one file per toolbar. `MenuItem` objects are stored in `smenu.smn`.

When you create a library, package the `ClassInfo`, `FunctionInfo`, and `Property` objects in files separate from the built-in objects shipped with Spotfire S+ by specifying where to save each interface object that you create. This procedure is used only for `ClassInfo`, `FunctionInfo`, and `Property` objects. `ToolBar` and `ToolBarButton` objects are already in separate files, and `MenuItems` are best created on the fly when loading the module.

Save Path Name

Each interface object has a `SavePathName` property that can specify the name of the file in which to save the object. We recommend defining variables for the desired file names at the top of the script that you use to generate the interface objects, and then referring to this variable when you create the objects.

For example, add a `SavePathName` argument when you create the `Property` object `MySqrtInput` and the `FunctionInfo` object `my.sqrt`:

```
prop.file <- "props.dft"
funcinfo.file <- "funcinfo.dft"
guiCreate(classname="Property", Name="MySqrtInput",
  DialogControl="String", UseQuotes=F,
```

```
DialogPrompt="Input Value",
SavePathName= prop.file)

guiCreate(classname="FunctionInfo", Name="my.sqrt",
DialogHeader="Calculate Square Root",
PropertyList="SPropInvisibleReturnObject,
MySqrtInput",
ArgumentList="#0=SPropInvisibleReturnObject,
#1=MySqrtInput",
SavePathName= funcinfo.file)
```

Store Default Objects

The `guiStoreDefaultObjects` function stores all interface objects of a specific class in the files specified by the object's `SavePathName` property. If the object has no `SavePathName` property, then the interface objects are stored in the default file for objects of its class.

Store the `Property` and `FunctionInfo` objects, as follows:

```
> guiStoreDefaultObjects("Property")
> guiStoreDefaultObjects("FunctionInfo")
```

The `guiLoadDefaultObjects` function loads all objects of a specified class from a specified file. It updates the `SavePathName` property for the objects read from the file to reflect the current path to the file. This can differ from the path to the file when the objects were stored.

Copying Help Files

If you have help files to include in the library, copy the text files into **library\mylib\Data_Help**.

It is important to give each help file the same name as the object to which it is related. The name is the mechanism that the `help` function uses to determine the help file to display. The `true.file.name` function returns the name under which each function is stored on disk. This name is the name to use for the help file.

1. Specify the name of the function and the location in the search list of the library to get the file name under which to save the help file.

```
true.file.name("my.sqrt",where=2)
[1] "__3"
```

2. Save the help file for `my.sqrt` as `__3` in **library\mylib\Data_Help**.

Storing Other Files

Other files can also belong in the library; therefore, you should copy them to **library\mylib**. Other files can include:

- C and Fortran object files.
- Documentation for the library.

Copy files related to interface objects to **library\mylib\.Prefs**. These file types include:

- Toolbar files (*.STB).
- Bitmap files containing toolbar button images.

Start-up and Exit Actions

Attaching a library makes the functions and data sets in the library available. Additional steps are needed to create menus and dialog boxes, or to load object code.

The `.First.lib` function runs when a library is attached. You can use this function to perform actions such as loading interface objects and creating menu items when a library is attached:

```
.First.lib<-function(library, section){
  prefs.loc <- paste(getenv("SHOME"),
    "\\library\\mylib\\.Prefs", sep="")
  guiLoadDefaultObjects("Property",
    paste(prefs.loc,"\\prop.dft", sep=""))
  guiLoadDefaultObjects("FunctionInfo",
    paste(prefs.loc,"\\funcinfo.dft", sep=""))
  guiCreate(classname="MenuItem",
    Name="$$SPlusMenuBar$Data$MySqrt",
    Type="MenuItem", MenuItemText="Square Root...",
    Action="Function", Command="my.sqrt")
  invisible()
}
```

The `.Last.lib` function runs when a library is detached or the program exits normally. You can use this function to perform cleanup actions upon exit.

Note

If the `.Last` function contains errors, when you quit the Spotfire S+ GUI, these errors are reported in a dialog box. Click **OK** to close the GUI, and then restart to correct the errors in `.Last`.

DISTRIBUTING THE LIBRARY

After you create the library, you can package it as a compressed archive or a self-extracting executable. One approach is to use a utility such as WinZip[®] or PKZip[®] to compress the library directory, and include a **readme.txt** file indicating how to unzip the package as a subdirectory of the **library** directory. A more sophisticated approach is to use these tools to produce a self-extracting archive that unpacks to the proper location. Creating these archives is beyond the scope of this manual.

SPOTFIRE S+ DIALOGS IN JAVA

10

Overview	383
Background Reading	383
Motivation	384
Contrast with S-PLUS 4 Dialogs	384
Topics Covered	385
A Simple Example	385
Classes	387
Standard Spotfire S+ Dialog	387
Group Panels	388
Control Types	388
Layout	391
Standard Spotfire S+ Layout	391
Pages	391
Columns	391
Groups	392
Sizing	392
Actions	394
Using Listeners	394
Getting Information From Spotfire S+	394
Special Controls	394
Overriding Button Actions	395
Checking Completeness	395
Calling The Function	397
The Function Info Object	397
Formatting Information	397
Sending the Command	398
Modifying Menus	399
Style Guidelines	402
Basic Issues	402
Basic Dialogs	403

Modeling Dialogs	409
Modeling Dialog Functions	417
Example: Correlations Dialog	423
Example: Linear Regression Dialog	427

OVERVIEW

As dialog construction technology has evolved, so has the Spotfire S+ functionality for constructing custom dialogs. Previous systems for constructing dialogs in Spotfire S+ have been based on sets of Spotfire S+ commands which create, modify, and display dialog elements. These systems are platform-dependent, with different tools in Windows and UNIX.

The introduction of the Swing classes for interface development in Java 2 is a major step forward for graphical user interface development. The Swing classes provide a powerful set of cross-platform tools which are flexible and easy to use. S-PLUS 6 introduced a set of Swing classes and conventions for constructing dialogs using Java which communicate with Spotfire S+.

S-PLUS 6 introduced Java connectivity on Linux[®] and Solaris[®] platforms.

Background Reading

This chapter is written for programmers with an understanding of dialog construction in Java using Swing. Most general books on Java will have some discussion of Swing.

Two good introductions to Java are the following:

Horton, Ivor. (1999). *Beginning Java 2*. Wrox Press. Birmingham, UK.

van der Linden, Peter. (1999) *Just Java 2*. Sun Microsystems Press. Palo Alto, CA.

More detail on the specifics of Swing is available in:

Pantham, Stayaraj. (1999) *Pure JFC Swing*. Sams. Indianapolis, IN.

These are the books used by the developer of the tools discussed here. Numerous other books on Java and Swing are available.

On the Web, the primary site for information on Java is:

<http://www.javasoft.com>

This site contains a variety of useful documentation and tutorials. An indispensable resource is the Java 2 Platform API Specification at:

<http://www.javasoft.com/products/jdk/1.2/docs/api/index.html>

Motivation

The emphasis of this chapter is on creating dialogs using Java that construct calls to S-PLUS functions. The design goal for the system described here was to provide a system which would allow both TIBCO Software Inc. developers and users to create full-featured dialogs which could be used on either Windows[®] or UNIX[®]. The look of the controls and layout of the dialogs was designed for consistency with the dialogs in Spotfire S+ for Windows.

Contrast with S-PLUS 4 Dialogs

The Java dialog system provides a number of benefits not present in the S-PLUS 4 dialog system. Benefits include:

- Layout controls as desired in a dialog rather than being restricted to a two-column layout with automatic positioning.
- Specify callback information (information on how to modify the dialog when a control is changed) at the control level instead of in a single dialog-level callback function.
- Fully specify the action to perform when **OK**, **Apply**, or **Cancel** is pressed.
- Use the wealth of documentation on Java dialog programming.

The S-PLUS 4 dialog system does have some features which are useful at times and which are absent from the Java system:

- The S-PLUS 4 system automatically lays out the controls on the dialog. The Java system requires more layout specification.
- The S-PLUS 4 system uses Spotfire S+ commands, while the Java system requires the programmer to learn some Java.
- The S-PLUS 4 system has a dialog-rollback system which allows the user to roll the values of controls in a dialog back to a previous state. The caveat is that the dialog callback function must include code to update option lists and enabled status on rollback. The Java dialogs do not currently support rollback.

Topics Covered This chapter covers:

- A simple example of a custom dialog.
- The new Java classes for constructing Spotfire S+ dialogs.
- Layout of the dialog.
- Performing actions when control values are modified.
- Calling the S-PLUS function.
- Single-page dialog example.
- Multiple-page dialog example.
- Dynamically modifying the Spotfire S+ menu.

A Simple Example

Below is Java code to create a dialog that queries the user for the name of a `data.frame`, and then runs the `summary()` function on the specified `data.frame`. Comments indicate the basic purpose of each expression.

An extended version of this example which includes code for creating a menu item that launches the dialog is provided on page 399.

```
// Import the necessary swing and S-PLUS classes
import com.insightful.splus.*;
import com.insightful.controls.*;
import javax.swing.*;

public class SummaryDialogExample extends SplusDialog {

    public SummaryDialogExample(){
        // Specify the title in the SplusDialog constructor
        super("Summary Dialog");

        // Specify the function to call on OK or Apply
        SplusFunctionInfo funcInfo = new
            SplusFunctionInfo("summary");
        setFunctionInfo(funcInfo);

        // Create a panel to contain the dialog controls
        JPanel dataPage = new JPanel();

        // Create a combo box control
```

```
SplusDataSetComboBox dataSet = new
    SplusDataSetComboBox();

// Add the combo box to the dialog panel
dataPage.add(dataSet);

// Add the combo box to the list of function arguments
funcInfo.add(dataSet, "object");

// Add the panel to the dialog and perform dialog sizing
setCenterPanel(dataPage);
}
}
```

CLASSES

The `com.insightful.controls` package contains the basic set of controls used to construct standard Spotfire S+ dialogs. Some additional controls which communicate with the Spotfire S+ engine to autoconstruct their option lists or values are available in the `com.insightful.splus` package.

This section provides details on the architecture and behavior of the custom dialog system.

Standard Spotfire S+ Dialog

All of the dialogs available from the Statistics and Graph menus are examples of a standard Spotfire S+ dialog. For each dialog there is a Java class representing the dialog which extends `SplusDialog`. `SplusDialog` is a `JDialog` with a standard set of buttons (**OK**, **Cancel**, **Apply**, and **Help**) and standard sizing. The `SplusDialog` class takes care of sizing the dialog and reacting to button presses.

If **OK** or **Apply** is pressed, a function call is constructed (based upon the values of the controls) and is sent to the S-PLUS engine for evaluation. If **OK** or **Cancel** is pressed, the dialog is dismissed. If **Help** is pressed, the help information for the dialog is displayed. Dialogs which extend `SplusDialog` may change the behavior of the buttons, as is discussed in the section *Overriding Button Actions* (page 395).

To create a standard Spotfire S+ dialog, create a class which extends `SplusDialog`. The class constructor should:

- Create an `SplusFunctionInfo` object giving the name of the function to use when **OK** or **Apply** is pressed. In addition to adding controls to panels in the dialog for display, each control will be added to the `SplusFunctionInfo` object to indicate that it is to be used when constructing the function call.
- Create a panel which will contain the controls. For a single-page dialog use a `JPanel`. For a multi-page dialog use a `JTabbedPane` containing one `JPanel` for each page.
- Add additional panels to each page to organize the layout. This is discussed in the section *Standard Spotfire S+ Layout* (page 391).

- Add `SplusGroupPanel` objects to the panels used for layout. These are `JPanel` objects with a border and label specified.
- Add controls implementing `SplusControlMethods` such as `SplusListBox` and `SplusComboBox` to the group panels. Also add each control to the `SplusFunctionInfo` for the dialog. When defining a control you specify information related to the content of the control. When adding the control to the `SplusFunctionInfo` object you specify information on how the control is used when constructing the function call.
- After all groups and controls have been added to this panel, use `setCenterPanel()` to add the panel to the center of the dialog. This method will also size the dialog and set the dialog's location so that it is centered in the application window.

This chapter discusses the Java classes for standard Spotfire S+ controls and their use in constructing dialogs with a consistent look and feel. You are not limited to using these controls or this layout. Ways to use alternate layouts and controls are discussed in the relevant sections.

Group Panels

Typically multiple controls will be grouped together in a bordered panel with a label describing the purpose of the set of controls. The `SplusGroupPanel` control provides a `JPanel` with a text label and a border. It uses a `Box` layout in which controls are stacked vertically. Controls are added to the panel with the `add()` method.

The `SplusGroupPanel` is sized based on the size of the controls it contains. If the panel contains an `SplusWideTextField`, it will be sized as a wide group.

Control Types

Table 10.1 lists the Java classes for controls used in standard Spotfire S+ dialogs. Each control is a Swing `JPanel` containing other Swing components, with a uniform sizing specified by `SplusControlMetrics`

and a set of methods for getting and setting various control information. (An additional class `SplusButton` provides a button which is a `JButton` sized to match these controls.)

Table 10.1: *Standard Spotfire S+ Controls*

Control Name	Swing Components
<code>SplusCheckBox</code>	<code>JCheckBox</code>
<code>SplusComboBox</code>	<code>JLabel</code> , <code>JComboBox</code>
<code>SplusDoubleSpinner</code> , <code>SplusIntegerSpinner</code>	<code>JLabel</code> , <code>JTextField</code> , <code>JScrollBar</code>
<code>SplusInvisibleControl</code>	None
<code>SplusListBox</code>	<code>JLabel</code> , <code>JList</code> , <code>JScrollPane</code>
<code>SplusRadioButtonGroup</code>	<code>JLabel</code> , <code>JButtonGroup</code> , multiple <code>JRadioButtons</code>
<code>SplusTextField</code>	<code>JLabel</code> , <code>JTextField</code>
<code>SplusWideTextField</code>	<code>JLabel</code> , <code>JTextField</code>

In Swing, a `JLabel` and a corresponding `JTextField` are two separate controls. The `SplusTextField` control incorporates the two elements into a single control. All of the controls presented in the table above implement the `SplusControlMethods` interface. This common interface to disparate control types simplifies dialog programming.

To implement the `SplusControlMethods` interface, a class must have methods to:

- Get and set the text and mnemonic for the control using `getText()`, `setText()`, `getMnemonic()`, and `setMnemonic()`. This is typically the text for a label, although in the `SplusCheckBox` it is the text for the check box.

- Get and set the option list for the control using `getOptionList()` and `setOptionList()`. `SplusComboBox`, `SplusListBox`, and `SplusRadioButtonGroup` all have meaningful option lists. For other controls the `getOptionList()` and `setOptionList()` methods have no effect.
- Get and set whether the control is enabled using `getEnabled()` and `setEnabled()`. If the control contains multiple Swing controls (such as a label and a text field) all of the elements are enabled or disabled together.

Typically each control will also support a set of listener methods which will pass registration of a listener down to the related Swing control. The listener methods available vary by control type, and match to listeners available for the underlying Swing control. Most controls support `addActionListener()` and perhaps `addChangeListener()` or `addItemListener()`. The exception is `SplusListBox`, which supports `addListSelectionListener()`.

Some controls will have additional methods suited to the underlying Swing control. For example, `SplusListBox` supports `clearSelection()` and `isSelectionEmpty()`.

If a programmer wants to use a control other than one of the standard controls provided, one way to do so is by defining a new control class which implements `SplusControlMethods`.

Another way to pass a value from any Java control to a S-PLUS function call is to instantiate an `SplusInvisibleControl` which listens for changes in the control of interest and updates its value based on the status of the control. Add the `SplusInvisibleControl` to the `SplusFunctionInfo` object to pass the value in the function call.

LAYOUT

Standard Spotfire S+ Layout

In contrast to the dialog system introduced in S-PLUS 4.0 for Windows, the Java dialog system does not attempt to automatically lay out controls on the dialog. It is the programmer's responsibility to specify the layout of the controls. Although any Java layout may be used, we encourage use of the Box layout with nested panels to organize the elements on the dialog.

The one restriction is that if an `SplusDialog` is used, the programmer should only add a panel to the center of the dialog using `setCenterPanel()`. The bottom panel is used by the **OK**, **Apply**, **Cancel**, and **Help** buttons, and the other panels are left empty.

To see the actual commands needed to perform layout, look at the Java code used to create particular dialogs in Spotfire S+. The `CorrelationsDialog` class contains an example of a single-page dialog. The `LinearRegressionDialog` class contains an example of a multiple-page dialog. The Java code for these classes is included at the end of this chapter.

Pages

If the dialog is a single-page dialog, then a single `JPanel` should be added to the `SplusDialog` using `setCenterPanel()`. Panels for columns are then added to this panel.

If the dialog is a multiple-page dialog, then a `JTabbedPane` should be added to the `SplusDialog` using `setCenterPanel()`. Each page is then specified by a `JPanel` which is placed in the `JTabbedPane` using the `addTab()` method. Panels for columns are then added to each page panel.

Columns

The standard layout for Spotfire S+ dialogs contains two columns of controls. This layout is created by specifying that the `JPanel` for the page contain two other `JPanels`. The page panel will have a `BoxLayout` with a horizontal orientation, while each column will have a `BoxLayout` with a vertical orientation.

Some more sophisticated dialogs will have a wide group along the bottom of a page. This layout is achieved by giving the page panel a vertical orientation and placing two `JPanels` in the panel to represent the top and bottom sections of the dialog. The top panel is then given

a horizontal orientation and two `JPanel`s are added for the columns. The bottom panel fills both columns. An example of this layout is in the `LinearRegressionDialog` class.

Note that in a `Box` layout the groups will expand to fill all of the space available unless `Box.Filler` components are added. `SplusBoxFiller` and `SplusWideBoxFiller` are `Box.Filler` components sized to fill a single column or the full center panel width of an `SplusDialog`. These components are added after the last group in each column to tell the layout to leave as much space as possible at the end of the columns.

Groups

For proper layout and margins, all controls must be placed within `SplusGroupPanel` objects, which are then added to the columns in the dialog. If a control is added directly to the column, the margin spacing is likely to be off.

To add a control to a dialog with no visible group, create an `SplusGroupPanel` for the control with an empty border. The panel will have an empty border if no title is specified in the `SplusGroupPanel` constructor.

Sizing

The `SplusControlMetrics` class provides standard control sizing information. This class has static methods which are used by the controls to obtain uniform sizing. The method `getControlDimension()` returns the dimension of a standard list field or combo box. The function `getFullDimension()` returns a value which is twice this size plus a margin of 10 in width, which represents the size of a text field with a label. Each control sets its minimum size and preferred size such that all controls have a common width. The height of each control depends upon its contents, with `SplusRadioButtonGroup` and `SplusListBox` taller than the other controls.

Controls such as `SplusTextField` and `SplusComboBox` that consist of a text label on the left and an active control on the right typically extend `SplusLabelBox`. This class handles sizing for this kind of control. Other controls such as `SplusCheckBox` have their own code for sizing.

One wider control is available. The `SplusWideTextField` control is an `SplusTextField` which spans both columns. This control is used to display a value which is too long to fit in a standard text box, such as a statistical modeling formula.

ACTIONS

Using Listeners In dialogs it is often necessary to change the value or state of one control based upon the state of another control. Swing supports this through the use of listeners and events.

For example, we might want to disable a text field when a check box is changed. To do so, we add an action listener to the check box which specifies code to run whenever the check box is acted upon. This code would then check the value of the check box, and disable or enable the text field according to its value.

Most interactions needed in a dialog may be accomplished with listeners and the `SplusControl` methods such as `getValue()`, `setValue()`, `setOptionList()`, and `setEnabled()`.

Getting Information From Spotfire S+

Some dialog actions require information that is not available to the dialog, such as the names of the columns in a data set. In these cases the S-PLUS engine may be called to get the information. This is done by constructing a S-PLUS expression which is sent to the S-PLUS engine and evaluated to get the desired information. The `SplusDataSetComboBox` class contains example code for this purpose.

Special Controls

For a few cases the special action desired for a control is common enough that it is useful to have a special class for the control which takes care of performing the action. Special cases of `SplusComboBox`, `SplusListBox`, and `SplusTextField` with particular uses are:

- `SplusDataSetComboBox`: This is a combo box with an option list providing the names of all `data.frames` in the user's working database. It also keeps track of the column names for the `data.frame` which is selected.
- `SplusDataColumnComboBox`: This is a combo box with an option list providing the column names of a particular data set. When the control is created the `SplusDataSetComboBox` containing the name of the corresponding `data.frame` must be specified. This combo box listens to the data set control and updates the column list if the selected data set changes.
- `SplusDataColumnListBox`: This is a list box with the same behavior as an `SplusDataColumnComboBox`.

- `SplusSubsetField`: This is a text field representing a subset of rows in a data set. If rows are currently selected as the rows of interest, the row numbers will be filled into this field when the dialog is displayed. The intent is that row selection in a Data Window will be used to specify the rows to use. As this has not been implemented, the field is currently a standard `SplusTextField`.
- `SplusWideFormula`: This is a wide text field representing a model formula. The intent is that column selection in a Data Window will be used to specify the columns to use, and a formula will be constructed based on the selected columns. As this has not been implemented, the field is currently a standard `SplusWideTextField`.

These classes are in the `com.insightful.splus` package.

While `SplusDataColumnComboBox` and `SplusDataColumnListBox` are typically used with an `SplusDataSetComboBox`, they can be used with some other control implementing the `SplusDataColumnListProvider` interface. This capability is used in the time series graph dialogs to create column lists for a `timeSeries` object named in an `SplusTimeSeriesComboBox`.

Overriding Button Actions

At times it is useful to perform extra actions when **OK** or **Apply** is pressed. In particular, it may be necessary to check the values of fields for consistency to determine whether to proceed with the function call or display a warning dialog requesting more information.

The `SplusDialog` takes care of performing actions when **OK**, **Cancel**, **Apply**, or **Help** is pressed. It does so by calling `performOk()`, `performCancel()`, `performApply()`, or `performHelp()`. To specify an alternate action, override the relevant function in the class for your dialog (which extends `SplusDialog`). After performing whatever extra actions are desired, use `sendCommand()` to create and evaluate the function call, and if necessary `dispose()` to dismiss the dialog.

Checking Completeness

When **OK** or **Apply** is pressed, it is often useful to check whether required arguments are present before formulating the function call and dismissing the dialog. The `performOk()` and `performApply()` methods will call the function `isComplete()`, which returns a boolean

indicating whether to continue with the button's action. Override `isComplete()` to check a condition such as whether all required fields have values.

The `warnIfEmpty()` method will check whether a control's value is an empty string, and if so throw up a warning dialog and return false. This method can be used in `isComplete()` to check whether a required field is empty.

CALLING THE FUNCTION

The Function Info Object

The way dialogs extending `SplusDialog` perform actions is by using the values of controls to build up a function call which is then sent to the S-PLUS engine for evaluation. Each `SplusDialog` will have an `SplusFunctionInfo` object which contains the information on how to construct the function call. An `SplusFunctionInfo` object contains the name of the function to call and the name of the control containing a name under which to save the result of the function call (if any). It also contains a list of the controls containing the values to use in the function call, the corresponding argument names, and details on how to format each value.

`SplusDialog` has classes `getFunctionInfo()` and `setFunctionInfo()` to access the corresponding function information. An `SplusFunctionInfo` object must be set for the dialog in order for it to have any action on **OK** or **Apply**.

The `SplusFunctionInfo` object has methods `getFunctionName()` and `setFunctionName()` to get and set a `String` giving the name of the function to call. The methods `getResultControl()` and `setResultControl()` will get and set the reference to the control containing the name under which to save the returned value.

Objects implementing `SplusControlMethods` are added to the list of controls using the `add()` methods.

The method `getFunctionCall()` looks at the current values of the controls in the `SplusFunctionInfo` object and uses them to construct a function call, which is returned as a string.

Formatting Information

When a control is added to the `SplusFunctionInfo` object, information is specified which is used in formatting the argument in the function call:

- The argument name specifies the name of the argument corresponding to the control. If the control has an empty argument name (e.g., ""), it will not be used in the function call.
- The quote format specifies whether quotes should be placed around the value.

- The list format specifies whether to add `list()` around the value, as in `list("Height", "Weight")`. Typically this is used with a quote format of "true" to produce a list of strings.

The `add()` method of `SplusFunctionInfo` requires the name of the control and a `String` giving the argument name. Booleans specifying whether quote format and list format are to be used may also be provided. The default is to not add any extra formatting.

Controls without argument names are typically present because they are used in the dialog to build a string in a text field. They are added to the control list so that we may keep track of them, with the intent being to add rollback to the dialogs at some point and use the control list to save the current values of all controls when **OK** is pressed.

Only non-empty arguments will be added to the function call. If `getValue()` for a control returns an empty string, the argument will not be specified in the function call.

Sending the Command

Once the dialog has obtained the function call by using `getFunctionCall()` on its `SplusFunctionInfo` object, it typically passes this `String` on to `SplusGui` for evaluation.

If the dialog is in a user-constructed application which is using `SplusUserApp` to communicate with the S-PLUS engine, then the string may instead be passed to the engine using `SplusUserApp`'s `eval()` method. This will return an `SplusDataResult` object which has methods to obtain the output, warnings, and errors produced by evaluating the function call.

MODIFYING MENUS

The standard Spotfire S+ graphical user interface has a predefined menu defined in the class `SplusMainMenuBar`. This default menu can be dynamically modified using a static Java method called from Spotfire S+ with the `.JavaMethod` function.

Most of the built-in menu items generate an `ActionEvent` that is caught and acted upon by the `SplusMainActionListener`. A new menu item may use either this action listener or some other action listener.

To create a new dialog with its own menu item and display the menu item, we will need to do the following:

1. Create a Java class for the dialog by extending `SplusDialog`.
2. Add a static method to this class which adds a menu item to launch the dialog.
3. Write an S-PLUS function which calls this static method.

We will of course need to compile the Java code, possibly package it for distribution, and assure that it is included in the Java classpath. These steps are discussed elsewhere.

As there are standard Java methods for creating and modifying menu items, we do not need to use any methods specific to Spotfire S+ for this purpose. We use `SplusDialog.getMainMenuBar()` to obtain a reference to the menu bar. We then use `JMenu`, `JMenuBar`, and `JMenuItem` methods to create and modify menu items.

For example, suppose we want to add a new menu item after the Statistics menu for a simple dialog that calls `summary()` on a `data.frame`.

The S-PLUS expression to call the Java method is:

```
.JavaMethod("SummaryDialogWithMenu", "addItem", "()V")
```

The Java code defining the dialog and the `addItem()` method is:

```
import com.insightful.splus.*;
import com.insightful.controls.*;
import javax.swing.*;
```

```
public class SummaryDialogWithMenu extends SplusDialog {

    // Constructor for dialog

    public SummaryDialogWithMenu(){
        super("Summary Dialog");

        SplusFunctionInfo funcInfo = new
            SplusFunctionInfo("summary");
        setFunctionInfo(funcInfo);

        JPanel dataPage = new JPanel();
        SplusDataSetComboBox dataSet = new
            SplusDataSetComboBox();

        dataPage.add(dataSet);
        funcInfo.add(dataSet, "object");

        setCenterPanel(dataPage);
    }

    // Method to modify menu

    // Have a flag indicating whether menu already added.
    // Another approach is to check the menu item text.

    static boolean menuHasBeenAdded = false;

    public static void addMenuItem() {

        if (!menuHasBeenAdded){
            menuHasBeenAdded = true;

            JMenu myMenu = new JMenu("Other");
            myMenu.setMnemonic('O');

            JMenuItem menuItem = new
                JMenuItem("Simple Summary...");
            menuItem.setMnemonic('S');

            // The menu action listener launches a dialog if it
```

```
// gets an action command starting with "D:".

menuItem.setActionCommand("D:SummaryDialogWithMenu");
menuItem.addActionListener(getMainActionListener());
myMenu.add(menuItem);

// Add myMenu after the Statistics menu.

JMenuBar menuBar = getMainMenuBar();

int numItems = menuBar getMenuCount();

// Find Statistics menu item

int index = -1;

for (int i = 0; index < 0 && i < numItems; i++){
    JMenuItem item = (JMenuItem)
        menuBar.GetComponent(i);
    if (item.getText().equals("Statistics"))
        index = i + 1;
}

menuBar.add(myMenu, index);

// revalidate to get menu to redisplay
menuBar.revalidate();
}
}
}
```

STYLE GUIDELINES

Typically Spotfire S+ programmers will begin by writing functions for use in scripts and at the command line. These functions will generally fall into one of the following classes:

- Functions which compute some quantities and return a vector, matrix, `data.frame`, or list. If the result is assigned these values are stored, and if not they are printed using the standard mechanism. Functions such as `mean` and `cor` are of this type.
- Functions which take data and produce plots. The returned value is typically not of interest. Functions such as `xyp1ot` and `pairs` are of this type.
- A set of functions including a modeling function which produces a classed object, and method functions such as `print`, `summary`, `plot`, and `predict`. Functions such as `lm` and `tree` are of this type.

The custom dialog tools allow the creation of a dialog for any function. Hence the programmer may create a dialog which directly accesses a function developed for use at the command line. While this may be acceptable in some cases, experience has shown that it is generally preferable to write a wrapper function which interfaces between the dialog and the command line function.

This section discusses the issues that arise when creating a function for use with a dialog, and describes how these issues are handled by the built-in statistical dialog functions. In addition, we discuss basic design guidelines for statistical dialogs.

Basic Issues

Most functions will perform these steps:

- Accept input regarding the data to use.
- Accept input regarding computational parameters and options.
- Perform computations.
- Optionally print the results.
- Optionally store the results.

- Optionally produce plots.

Modeling functions have additional follow-on actions which are supported at the command line by separate methods:

- Providing additional summaries.
- Producing plots.
- Returning values such as fitted values and residuals.
- Calculating predicted values.

We will first discuss the basic steps performed by any function such as accepting input, performing calculations, printing results, saving results, and making plots. Then we will discuss the issues which arise for modeling functions with methods.

Basic Dialogs

We will begin by discussing the **Correlations and Covariances** dialog. Exploring this dialog and the related analysis functions will display the key issues encountered when constructing functions for dialogs.

The Dialog

The **Correlations and Covariances** dialog is available from the **Statistics | Data Summaries:Correlations** menu item.

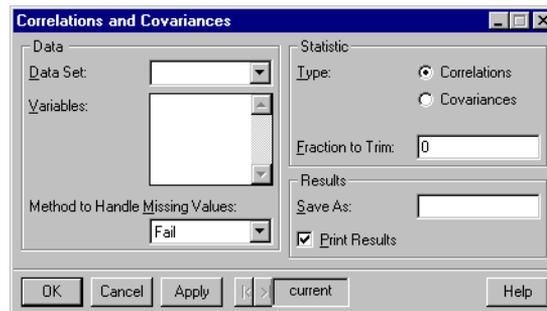


Figure 10.1: *The Correlations and Covariances dialog.*

This dialog provides access to the `cor` and `var` functions. It allows the user to specify the data to use, computation options, a name under which to save the results, and whether to print the results.

Note that the data to use is specified in the upper left corner of the dialog. The user first specifies which data set to use, and then the variables of interest.

The **Results** group in the lower right corner of the dialog lets the user specify an object name under which to store the results, and provides a check box indicating whether the results should be printed.

Other options are placed between the **Data** group and the **Results** group.

The Function

When **OK** or **Apply** is pressed in the dialog, the `menuCor` function is called. The naming convention for functions called by dialogs is to append `menu` to the command line function name, such as `menuLm`, `menuTree`, and `menuCensorReg`.

The `menuCor` function is:

```
> menuCor
function(data, variables = names(data), cor.p = F, trim = 0,
  cov.p = F, na.method = "fail", print.it = T,
  statistic = "Correlations")
{
# Note cor.p and cov.p have been replaced with statistic.
# They are left in solely for backwards compatibility.
  data <- as.data.frame(data)
  data.name <- deparse(substitute(data))
  if(!missing(variables))
    variables <- sapply(unpaste(variables, sep = ","),
      strip.blanks)
  if(!is.element(variables[[1]], c("<ALL>", "(All
    Variables)"))) {
    if(!length(variables))
      stop("You must select at least one variable\n")
    data <- data[, variables, drop = F]
  }
  dropped.cols <- !sapply(data, is.numeric) | sapply(data,
    is.dates)
  if(all(dropped.cols))
    stop("No numeric columns specified.")
  if(any(dropped.cols)) {
    warning(paste("Dropping non-numeric column(s) ",
      paste(names(data)[
        dropped.cols], collapse = ", ", ".", sep = ""))
    data <- data[, !dropped.cols, drop = F]
  }
  na.method <- casefold(na.method)
```

```

if(Statistic == "Correlations" || (cor.p && !cov.p)) {
  coeff <- cor(data, trim = trim, na.method = na.method)
  header.txt <- paste("\n\t*** Correlations for data
in: ", data.name,
  "****\n\n")
}
else {
  coeff <- var(data, na.method = na.method)
  header.txt <- paste("\n\t*** Covariances for data in:
", data.name,
  "****\n\n")
}
if(print.it) {
  cat(header.txt)
  print(coeff)
}
invisible(coeff)
}

```

Input Values

The function arguments are:

```

function(data, variables = names(data), cor.p = F, trim = 0,
  cov.p = F, na.method = "fail", print.it = T,
  statistic = "Correlations")

```

The function has one argument for each control in the dialog, with the exception of the **Save As** field specifying the name to which to assign the value returned by the function. Default values are present for all arguments except data. A default argument value will be used if the corresponding field in the dialog is left empty.

The first few lines in the function transform these inputs from a form preferable for a dialog field to the format expected by `cor` and `var`.

First the data is transformed to a data frame, to allow the handling of vectors and matrices. The name of the data is stored for use in printing the results:

```

data <- as.data.frame(data)
data.name <- deparse(substitute(data))

```

Next the function constructs the names of the variables of interest. The `variables` argument passed by the dialog is a single string containing a comma delimited list of column names, and perhaps the

string "(All Variables)". This string is broken into a character vector of variable names. If it does not include "(All Variables)" and is not empty, the specified columns of the data are extracted.

```
if(!missing(variables))
  variables <- sapply(unpaste(variables, sep = ","),
    strip.blanks)
if(!is.element(variables[[1]], c("<ALL>", "(All
  Variables)"))) {
  if(!length(variables))
    stop("You must select at least one variable\n")
  data <- data[, variables, drop = F]
}
```

Computations

After the desired set of data is constructed, the statistics are calculated:

```
if(statistic == "Correlations" || (cor.p && !cov.p)) {
  coeff <- cor(data, trim = trim, na.method = na.method)
  header.txt <- paste("\n\t*** Correlations for data
in: ", data.name,
  "\n\t***\n\n")
}
else {
  coeff <- var(data, na.method = na.method)
  header.txt <- paste("\n\t*** Covariances for data in:
", data.name,
  "\n\t***\n\n")
}
```

The statistic argument takes a string, either "Correlations" or "Covariances"; cor.p and cov.p arguments are logical values indicating whether to form the correlations or covariances which are supported for backward compatibility.

The trim and na.method arguments are passed directly to the computational functions.

A character string is also constructed for use as a header when printing the results.

Printing Results

The standard behavior in Spotfire S+ is to either print the results from a function or store them under a specified name using assignment. That is, a user may either see the results printed using

```
> cor(swiss.x)
```

save the results using

```
> swiss.cor <- cor(swiss.x)
```

or do both by saving the results and then printing the object

```
> swiss.cor <- cor(swiss.x)
> swiss.cor
```

Explicitly printing the results in a function is frowned upon unless the function is a `print` method for a classed object. The evaluation mechanism determines whether to print the result.

This convention is waived for the dialog functions, as it is necessary to provide a mechanism for both saving and printing the output within the function.

Another difference between using a function from the Command line and from a dialog is that the Command line alternates between an expression and the output related to that expression. Hence it is clear which expression and output go together. The output from a dialog is not preceded by an expression (the expression evaluated will be stored in the history log but is not printed to the output stream). Hence it is necessary to provide a header preceding the output which indicates the source of the output. The header lines also serve to separate subsequent sets of output.

If the user requests printed output, the header is printed with `cat`, and the result object with `print`:

```
header.txt <- paste("\n\t*** Covariance for data in:
", data.name, "***\n\n")
...
if(print.it) {
  cat(header.txt)
  print(coeff)
}
```

Generally `cat` is used to print character strings describing the output, and `print` is used for other objects.

Note that the convention for header lines is to use a character string of the form:

```
"\n\t*** Output Description ***\n\n"
```

Saving Results In this dialog, the results need not be explicitly saved within the function. The command is written such that the result is assigned to the name specified in **Save As** if a name is specified.

Note that the value is returned invisibly:

```
invisible(coeff)
```

As we have already printed the result if printing is desired, it is necessary to suppress the autoprinting which would normally occur if the result were returned without assignment.

In some cases it is necessary to assign the result within the function. In particular, this is required for the Windows GUI if the function is creating the data and then displaying it in a Data window. This is handled by the function `spropSaveResultData`. For example, `menuFacDesign` creates an object `new.design` and then calls:

```
spropSaveResultData(new.design, save.name, show.p)
```

If `save.name` is not specified, a name is constructed. The result is immediately assigned to the working directory. Then the data frame is displayed in a Data window if the Windows Spotfire S+ GUI is being run and the user specifies `show.p=T` by checking the **Show in Data Window** box in the **Factorial Design** dialog. In the Java GUI, the printed output includes a comment indicating the name under which the data is saved.

The explicit assignment is necessary because the data frame must exist as a persistent object on disk before it can be displayed in a Data window.

If an object named `save.name` already exists, it will be replaced by the new object by the call to `spropSaveResultData`. To add columns to an existing data set use `spropSaveResultColumns`. If the object `save.name` exists and has the same number of rows as the new columns, then the columns will be appended to the data. Otherwise the columns will be saved under a new name. See `tabSummary.lm` for an example of a call to this function.

Saving Additional Quantities In some cases the user may want access to other quantities which are not part of the standard object returned by the function, such as residuals or predicted values. At the command line these functions can be accessed using extractor functions such as `resid` and `predict`.

In dialogs, it may be preferable to save these objects into specified data frames using the save mechanism as described above. The section Summary Method (page 419) discusses this situation.

Plots

The Spotfire S+ GUI supports multiple coexisting Graph sheets, each of which may have multiple tabbed pages. When a new graph is created it may do one of three things:

- Replace the current graph (typically the graph most recently created).
- Create a new tab on the current Graph sheet.
- Create a new Graph sheet.

The default behaviour is for a statistical dialog function to open a new Graph sheet before creating graphs. If the function produces multiple graphs, these appear on multiple tabs in the new Graph sheet.

This autocreation of new Graph sheets may annoy some users due to the proliferation of windows. In the Windows GUI, the **Graphs Options** dialog has a **Statistics Dialogs Graphics: Create New Graph Sheet** check box which indicates whether or not to create a new Graph sheet for each new set of plots. In the Java GUI, this option is in the **Dialog Options** dialog.

It is good form for any plots created from dialogs to follow the dictates of this option. This is done by calling `new.graphsheet` before plots are produced. This function will create a new Graph sheet if the aforementioned option specifies to do so. The `new.graphsheet` function should only be called if plots are to be produced, and should only be called once within the function as calling it multiple times would open multiple new Graph sheets.

The `menuAcf` function provides an example of the use of `new.graphsheet`:

```
if(as.logical(plot.it)) {
  new.graphsheet()
  acf.plot(acf.obj)
}
```

Modeling Dialogs

A powerful feature of Spotfire S+ is the object-oriented nature of the statistical modeling functions. Statistical modeling is an iterative procedure in which the data analyst examines the data, fits a model,

examines diagnostic plots and summaries for the model, and refines the model based on the diagnostics. Modeling is best performed interactively, alternating between fitting a model and examining the model.

This interactive modeling is supported in Spotfire S+ by its class and method architecture. Generally there will be a modeling function (such as `lm` for linear regression) which fits a model, and then a set of methods (such as `print`, `plot`, `summary`, and `anova`) which are used to examine the model. The modeling function creates a model object whose class indicates how it is handled by the various methods.

This differs from other statistical packages, in which all desired plots and summaries are typically specified at the time the model is fit. If additional diagnostic plots are desired the model must be completely refit with options indicating that the new plots are desired. In Spotfire S+, additional plots may be accessed by simply applying the plot method to the model object.

In moving from a set of command line functions to dialogs for statistical modeling, the desired level of granularity for action specification changes. At the command line the basic strategy would be to issue a command to fit the model, followed by separate commands to get the desired plots and summaries. The ability to use such follow-on methods is still desirable from a graphical user interface, but it should be a capability rather than a requirement. The user will generally want to specify options for fitting the model plus desired plots and summaries in a single dialog, with all results generated when the model is fit.

In Windows, the design of the statistical modeling dialogs is such that the user can specify the desired summaries and plots at the time the model is fit, but it is also possible to right-click on a model object in the Object Explorer and access summary and plot methods as a follow-on action. Generally the Results, Plot, and Predict tabs on the modeling dialog are also available as separate dialogs from the model object context menu.

This section describes the general design of statistical modeling dialogs by examining the **Linear Regression** dialog. The following section describes the structure of the functions connected to the modeling and follow-on method dialogs. The statistical modeling dialogs follow the same design principles as are described here, but details will vary.

Model Tab

The **Model** tab describes the data to use, the model to fit, the name under which to save the model object, and various fitting options. It is typical to have **Data**, **Formula**, and **Save Model Object** groups which are similar to those in the **Linear Regression** dialog.

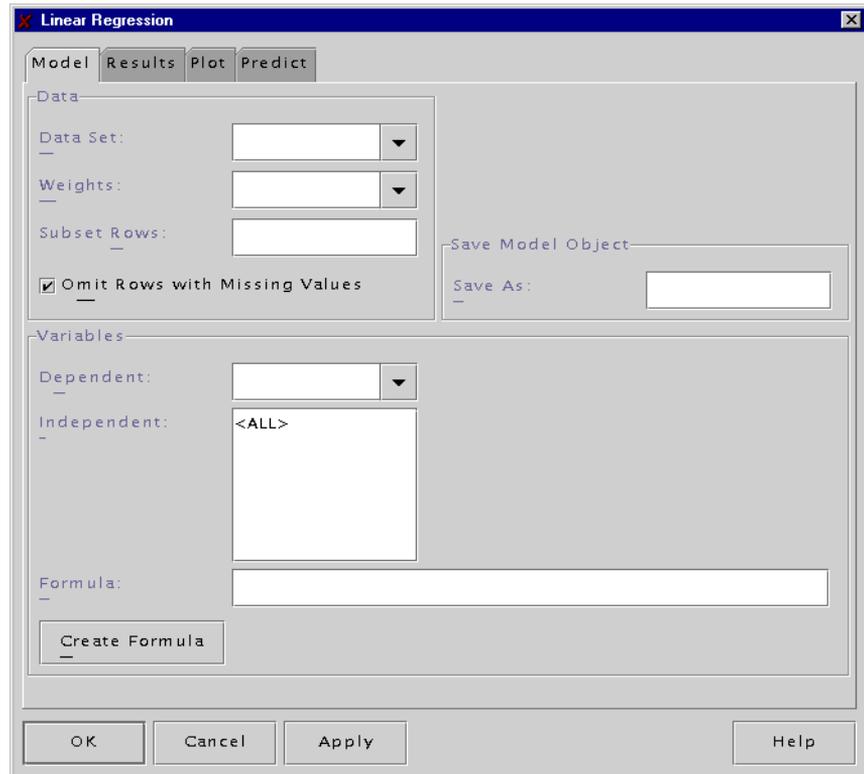


Figure 10.2: *The Model tab of the Linear Regression dialog.*

Data Group

The **Data Set** property is a drop-down list of available data sets. This list is filled with the data sets which are in the working database. This specifies the data argument to the modeling function.

The **Weights** property is a list of columns in the selected data set. The selected column will be used as weights in the model. This specifies the `weights` argument to the modeling function.

The **Subset Rows with** property takes an expression which is used as the subset expression in the model. This specifies the subset argument to the modeling function.

The **Omit Rows with Missing Values** check box specifies how missing values are handled. Checking this box is equivalent to specifying `na.action=na.exclude`, while leaving it unchecked is equivalent to `na.action=na.fail`. Some dialogs (such as **Correlations and Covariances**) instead have a **Method to Handle Missing Values** list box, which provides additional missing value actions.

Variables Group

The Variables group includes controls for specifying the **Dependent** and **Independent** variables. As you select or enter variables in these controls, they are echoed in the **Formula** control. The **Formula** specifies the form of the model, that is what variables to use as the predictors (independent variables) and the response (dependent variables). This specifies the formula argument to the modeling function.

Most modeling dialogs have a **Create Formula** button, which launches the Formula Builder dialog when pressed. This dialog allows point-and-click formula specification.

Dialogs in which the formula specifies a set of covariates rather than predictors and a response (such as **Factor Analysis**) have a **Variables** list rather than a **Create Formula** button.

Save Model Object Group

The **Save As** edit field specifies the name under which to save the model object. The returned value from the function called by the dialog is assigned to this name.

Options Tab

In the **Linear Regression** dialog, all of the necessary fitting options are available on the **Model** tab. Some other modeling dialogs, such as the **Logistic Regression** dialog, have more options which are placed on a separate tab. An **Options** tab may be useful either due to the availability of a large number of options, or to shelter the casual user from more advanced options.

Results Tab

The **Results** tab generally has groups for specifying the printed and saved results.

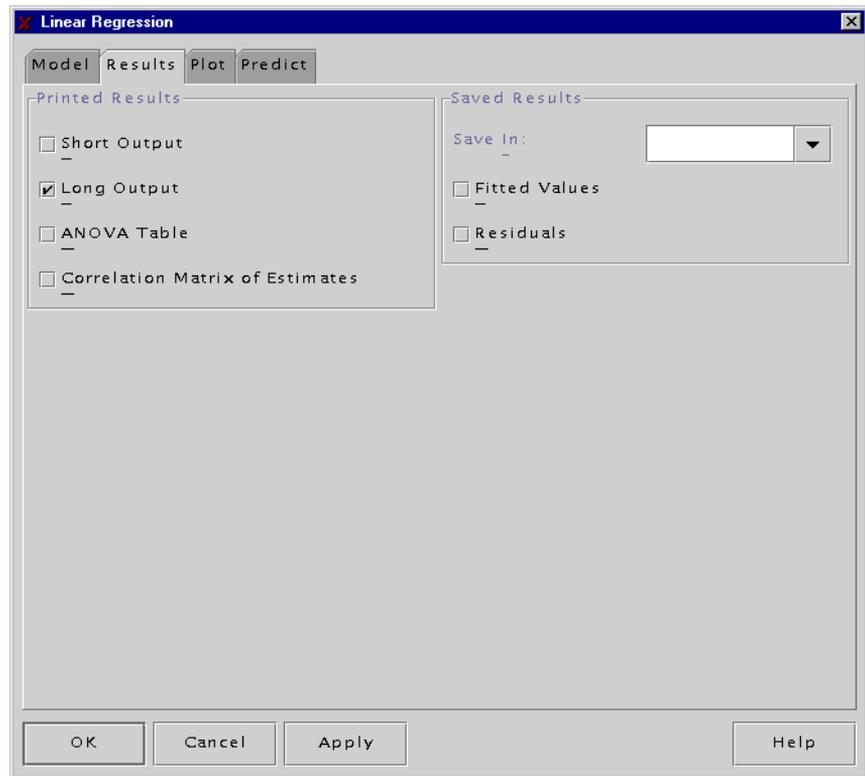


Figure 10.3: *The Results tab of the Linear Regression dialog.*

Printed Results Group

The **Printed Results** specify the types of summaries to print. These summaries will be displayed in a Report window, or in another location as specified in the **Text Output Routing** dialog.

Checking the **Short Output** check box generally indicates that the print method for the model object will be called.

Checking the **Long Output** check box generally indicates that the summary method for the model object will be called.

Other options will vary based on the statistical model.

Saved Results Group

The **Saved Results** specify results to be saved separate from the model object, such as fitted values and residuals. These saved components are usually of the same length as the data used in the model. It is often of interest to plot these quantities against the data used to construct the model, and hence it is convenient to save these values in a data frame for later use in plotting or analysis.

The **Save In** edit field takes the name of a data frame in which to save the results. This may be a new data frame or the data frame used to construct the model. If the data frame named exists and has a different number of rows than are in the results, then a new name will be constructed and the results saved in the new data frame.

Check boxes specify what results to save. Common choices include **Fitted Values** and **Residuals**.

Plot Tab

The Plot tab specifies which plots to produce and plotting options. Typically a **Plots** group provides check boxes to select plot types to produce. Other groups provide options for the various plots.

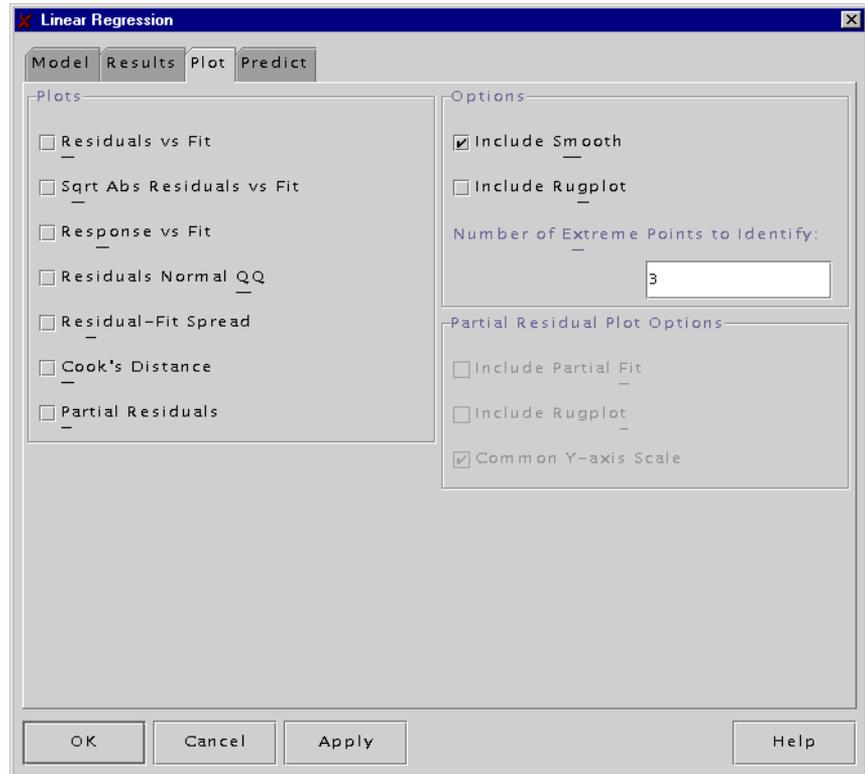


Figure 10.4: *The Plot tab of the Linear Regression dialog.*

Predict Tab

The **Predict** tab specifies whether predicted values will be saved, using similar conventions as the **Saved Results** group on the **Results** tab.

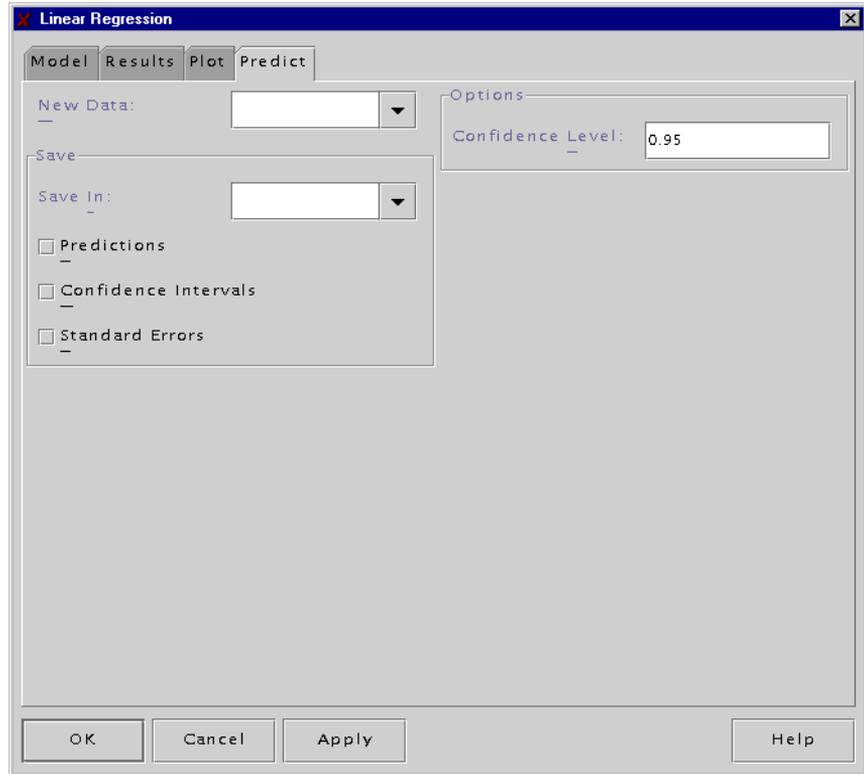


Figure 10.5: *The Predict tab of the Linear Regression dialog.*

The **New Data** edit field accepts the name of a data frame containing observations for which predictions are desired. This specifies the `newdata` argument to `predict`. If this is left empty the data used to fit the model will be used.

The **Save In** edit field takes the name of a data frame in which to save the results. This may be a new data frame or the data frame used to construct the model. If the data frame named exists and has a different number of rows than are in the results, then a new name will be constructed and the results saved in the new data frame.

Check boxes specify what results to save. Common choices include **Predictions**, **Confidence Intervals**, and **Standard Errors**.

Other options related to prediction may also be present.

Other Tabs

A statistical model may have additional methods specific to that type of model. The dialog for this model may have additional tabs, such as the **Prune/Shrink** tab on the **Tree Regression** dialog. In the Windows GUI, each dialog is limited to at most five tabs.

Modeling Dialog Functions

Command line functions for statistical models generally consist of a fitting function (such as `lm`) and method functions for that model (such as `print.lm`, `summary.lm`, `plot.lm`, and `predict.lm`). Similarly, the wrapper functions called by the dialogs consist of a main dialog function (such as `menuLm`) and method functions (`tabSummary.lm`, `tabPlot.lm`, and `tabPredict.lm`). The method functions are called by the main function.

In the Windows GUI, the method functions are also called from separate dialogs available by right-clicking on the model object in the Object Explorer. This structure allows the methods to be called either at the time of fitting the model, or later for use with a fitted model object.

The convention is to name the main dialog function with the preface `menu`, and the method functions with the preface `tab`. The suffix of the function reflects the name of the related command line function.

Main Function

The main dialog function has two primary purposes: fitting the model and calling the method functions. The main dialog function for linear regression is `menuLm`:

```
menuLm <-
function(formula, data, weights, subset, na.omit.p = T,
  print.short.p = F, print.long.p = T, print.anova.p = T,
  print.correlation.p = F, save.name = NULL, save.fit.p =
  F, save.resid.p = F, plotResidVsFit.p = F,
  plotSqrtAbsResid.p = F, plotResponseVsFit.p = F,
  plotQQ.p = F, plotRFSpread.p = F, plotCooks.p = F,
  smooths.p = F, rugplot.p = F, id.n = 3,
  plotPartialResid.p = F, plotPartialFit.p = F,
  rugplotPartialResid.p = F, scalePartialResid.p = T,
  newdata = NULL, predobj.name = NULL, predict.p = F, ci.p
  = F, se.p = F, conf.level = 0.95)
{
  fun.call <- match.call()
  fun.call[[1]] <- as.name("lm")
}
```

```
    if(na.omit.p)
      fun.call$na.action <- as.name("na.exclude")
    else fun.call$na.action <- as.name("na.fail")
    fun.args <- is.element(arg.names(fun.call), arg.names(
      "lm"))
    fun.call <- fun.call[c(T, fun.args)]
    lmbj <- eval(fun.call)#
# Call summary function:
    tabSummary.lm(lmbj, print.short.p, print.long.p,
      print.correlation.p, print.anova.p, save.name,
      save.fit.p, save.resid.p)#
# Call plot function:
    if(any(c(plotResidVsFit.p, plotSqrtAbsResid.p,
      plotResponseVsFit.p, plotQQ.p, plotRFSpread.p,
      plotCooks.p, plotPartialResid.p))) tabPlot.lm(
      lmbj, plotResidVsFit.p,
      plotSqrtAbsResid.p, plotResponseVsFit.p,
      plotQQ.p, plotRFSpread.p, plotCooks.p,
      smooths.p, rugplot.p, id.n,
      plotPartialResid.p, plotPartialFit.p,
      rugplotPartialResid.p,
      scalePartialResid.p)#
# Call predict:
    if(any(c(predict.p, ci.p, se.p)))
      tabPredict.lm(lmbj, newdata, predobj.name,
        predict.p, ci.p, se.p, conf.level)
    invisible(lmbj)
  }
```

This function has one argument corresponding to each property in the dialog, with the exception of the **Save As** field and invisible fields used for formatting.

The first eight lines of the function are used to fit the model. The approach is to construct an expression which specifies the appropriate call to the model function, and then to evaluate this expression. The somewhat sophisticated syntax is used so that the `call` component in the model object has the appropriate names as specified in the dialog. This general recipe may be used for any function, with `lm` replaced by the name of the modeling function.

Note that `na.action` is specified as a check box in the dialog, but as a function name in `lm`. This necessitates modification of this argument before evaluating the call to `lm`.

After the model has been fit, the methods `tabSummary.lm`, `tabPlot.lm`, and `tabPredict.lm` are called. For efficiency, calls are only made to `tabPlot.lm` and `tabPredict.lm` if the options are specified such that calling these functions will produce some action.

Finally, the model object is returned. It is returned invisibly since any printing of the object has been handled by `tabSummary.lm`.

Summary Method The summary method produces printed summaries and saves specified results in a data frame separate from the model object. The summary method for the **Linear Regression** dialog is `tabSummary.lm`:

```
tabSummary.lm <-
function(lmobj, print.short.p = F, print.long.p = T,
        print.correlation.p = F, print.anova.p = F, save.name =
        NULL, save.fit.p = F, save.resid.p = F)
{
  if(print.short.p || print.long.p || print.anova.p) {
    cat("\n\t*** Linear Model ***\n")
    if(print.short.p) {
      print(lmobj)
    }
    if(print.long.p) {
      print(summary(lmobj, correlation =
        print.correlation.p))
    }
    if(print.anova.p) {
      cat("\n")
      print(anova(lmobj))
    }
    cat("\n")
  }
  # Save results if requested:
  if(any(c(save.fit.p, save.resid.p)))
    saveobj <- list()
    if(save.fit.p)
      saveobj[["fit"]] <- fitted(lmobj)
    if(save.resid.p)
      saveobj[["residuals"]] <- residuals(lmobj)
}
```

```

        saveobj <- guiDefaultDataObject(saveobj)
        spropSaveResultColumns(saveobj, save.name, show.p)
    }
    invisible(lmobj)
}

```

The first part of this function is responsible for printing the specified summaries. If any printed output is specified, a header will be printed demarcating the start of the output. Based on option values, the `print`, `summary`, and other methods for the model will be called.

The second part of the function concerns itself with saving the requested values. Extractor functions such as `fitted` and `residuals` are used to get the desired values. The remainder of the code calls function to save the columns and possibly display them in a Data window.

The model object passed to this function is returned invisibly.

Plot Method

The `plot` function opens a new Graph sheet if necessary, and produces the desired plots. The plot method for the **Linear Regression** dialog is `tabPlot.lm`:

```

tabPlot.lm <-
function(lmobj, plotResidVsFit.p = F, plotSqrtAbsResid.p =
  F, plotResponseVsFit.p = F, plotQQ.p = F,
  plotRFSpread.p = F, plotCooks.p = F, smooths.p = F,
  rugplot.p = F, id.n = 3, plotPartialResid.p = F,
  plotPartialFit.p = F, rugplotPartialResid.p = F,
  scalePartialResid.p = T, ...)
{
  if(any(c(plotResidVsFit.p, plotSqrtAbsResid.p,
    plotResponseVsFit.p, plotQQ.p, plotRFSpread.p,
    plotCooks.p, plotPartialResid.p)))
    new.graphsheet()
  if(any(c(plotResidVsFit.p, plotSqrtAbsResid.p,
    plotResponseVsFit.p, plotQQ.p, plotRFSpread.p,
    plotCooks.p))) {
    whichPlots <- seq(1, 6)[c(plotResidVsFit.p,
      plotSqrtAbsResid.p, plotResponseVsFit.p,
      plotQQ.p, plotRFSpread.p, plotCooks.p)]
    plot.lm(lmobj, smooths = smooths.p, rugplot =
      rugplot.p, id.n = id.n, which.plots =

```

```

        whichPlots, ...)
    }
    if(plotPartialResid.p) {
        partial.plot(lmobj, residual = T, fit =
            plotPartialFit.p, scale =
            scalePartialResid.p, rugplot =
            rugplotPartialResid.p, ...)
    }
    invisible(lmobj)
}

```

If any plots are desired, the function first calls `new.graphsheet` to open a new Graph Window if necessary. The plot method for `lm` is then called. Some additional plots useful in linear regression are produced by the `partial.plots` function, which is called if partial residual plots are desired.

The model object passed to this function is returned invisibly.

Predict Method

The `predict` function obtain predicted values for new data or the data used to fit the model. The `predict` method for the **Linear Regression** dialog is `tabPredict.lm`:

```

tabPredict.lm <-
function(object, newdata = NULL, save.name, predict.p = F,
  ci.p = F, se.p = F, conf.level = 0.95)
{
  if(is.null(newdata))
    predobj <- predict(object, se.fit = se.p || ci.p
      )
  else predobj <- predict(object, newdata, se.fit = se.p ||
    ci.p)
  if(ci.p) {
    if(conf.level > 1 && conf.level < 100)
      conf.level <- conf.level/100
    t.value <- qt(conf.level, object$df.residual)
    lower.name <- paste(conf.level * 100, "% L.C.L.",
      sep = "")
    upper.name <- paste(conf.level * 100, "% U.C.L.",
      sep = "")
    predobj[[lower.name]] <- predobj$fit - t.value *

```

```
        predobj$se.fit
        predobj[[upper.name]] <- predobj$fit + t.value *
        predobj$se.fit
    }
# remove prediction column and se column if not requested:
if(!predict.p)
    predobj$fit <- NULL
if(!se.p)
    predobj$se.fit <- NULL
predobj$residual.scale <- NULL
predobj$df <- NULL
predobj <- as.data.frame(predobj)
n.predict <- nrow(predobj)
if(exists(save.name, where = 1)) {
    if(inherits(get(save.name, where = 1),
               "data.frame") && nrow(get(save.name,
               where = 1)) == n.predict)
        assign(save.name, cbind(get(save.name,
               where = 1), predobj), where = 1)
    else {
        newsave.name <- unique.name(save.name,
               where = 1)
        assign(newsave.name, predobj, where = 1)
        warning(paste("Predictions saved in",
               newsave.name))
    }
}
else assign(save.name, predobj, where = 1)
invisible(NULL)
}
```

The function calls `predict.lm` to obtain predicted values and standard errors. It then calculates confidence intervals, and removes any undesired components from the output. The predictions are then stored in a data frame using the same type of algorithm as in `tabSummary.lm`.

No value is returned.

Other Methods

If the dialog has additional tabs, other dialog methods will be available. For example, the **Tree Regression** dialog has a **Prune/Shrink** tab with a corresponding function `tabPrune.tree`.

EXAMPLE: CORRELATIONS DIALOG

Below is the Java code used to create the Correlations dialog. This is a single-page dialog with the standard two-column layout. It is part of the `com.insightful.splus.statdlg` package.

```
package com.insightful.splus.statdlg;

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.insightful.controls.*;
import com.insightful.splus.SplusDataSetComboBox;
import com.insightful.splus.SplusDataColumnComboBox;
import com.insightful.splus.SplusSubsetField;
import com.insightful.splus.SplusDataColumnListBox;

public class CorrelationsDialog extends SplusDialog {

    // Define any controls with listeners

    SplusDataSetComboBox dataSet;
    SplusDataColumnListBox dataColumnList;

    SplusRadioButtonGroup typeButtons;
    SplusTextField trimField;

    public CorrelationsDialog(){
        super("Correlations and Covariances");

        // A one page dialog with two columns
        // Controls and groups get added to the columns

        JPanel mainPanel = new JPanel();

        Box columnOne = new Box(BoxLayout.Y_AXIS);
        Box columnTwo = new Box(BoxLayout.Y_AXIS);
        mainPanel.setLayout(new BoxLayout(mainPanel,
```

```
        BorderLayout.X_AXIS));
mainPanel.add(columnOne);
mainPanel.add(columnTwo);

/* Information needed to construct function call */
SplusFunctionInfo funcInfo = new
    SplusFunctionInfo("menuCor");
setFunctionInfo(funcInfo);

// Data group

SplusGroupPanel dataGroup = new SplusGroupPanel("Data");
columnOne.add(dataGroup);

dataSet = new SplusDataSetComboBox();
dataGroup.add(dataSet);
funcInfo.add(dataSet, "data");

dataColumnList = new SplusDataColumnListBox(dataSet);
dataColumnList.setPrepend(new String [] {"<ALL>"});
dataColumnList.setValue("<ALL>");
dataGroup.add(dataColumnList);
funcInfo.add(dataColumnList, "variables", true, true);

SplusComboBox naMethod = new
    SplusComboBox("Method to Handle Missing Values", 'M',
        new String[] {"Fail", "Omit", "Include",
            "Available"});
naMethod.setValue("Fail");
naMethod.setEditable(false);
dataGroup.add(naMethod);
funcInfo.add(naMethod, "na.method", true);

// Spacer for column one

columnOne.add(new SplusBoxFiller());

// Statistic group

SplusGroupPanel statGroup = new
    SplusGroupPanel("Statistic");
```

```
columnTwo.add(statGroup);

typeButtons = new
    SplusRadioButtonGroup("Type", 'T',
        new String [] {"Correlations", "Covariances"});
typeButtons.setValue("Correlations");
statGroup.add(typeButtons);
funcInfo.add(typeButtons, "statistic", true);

// Listen to disable trimming for Covariances

typeButtons.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent ae){
        trimField.setEnabled(

typeButtons.getValue().equals("Correlations"));
    }
});

trimField = new SplusTextField("Fraction to Trim",
    'F');
trimField.setValue("0");
statGroup.add(trimField);
funcInfo.add(trimField, "trim");

// Results group

SplusGroupPanel resultsGroup =
    new SplusGroupPanel("Results");
columnTwo.add(resultsGroup);

SplusTextField saveAs =
    new SplusTextField("Save As", 'S');
resultsGroup.add(saveAs);
funcInfo.setResultControl(saveAs);

SplusCheckBox printResults =
    new SplusCheckBox("Print Results", 'P');
printResults.setValue("T");
resultsGroup.add(printResults);
funcInfo.add(printResults, "print.it");
```

```
// Spacer for column two

columnTwo.add(new SplusBoxFiller());

// Add mainPanel to CENTER panel of dialog

setCenterPanel(mainPanel);

    }

// Redefine isComplete() to check for required arguments.

public boolean isComplete() {
    return (warnIfEmpty(dataSet));
}

}
```

EXAMPLE: LINEAR REGRESSION DIALOG

Below is the Java code used to create the Linear Regression dialog. This is a multiple-page dialog with a wide-group on the first page, and standard two-column layout on the other pages. It is part of the **com.insightful.splus.statdlg** package.

```
package com.insightful.splus.statdlg;

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.insightful.controls.*;
import com.insightful.splus.SplusDataSetComboBox;
import com.insightful.splus.SplusDataColumnComboBox;
import com.insightful.splus.SplusSubsetField;
import com.insightful.splus.SplusDataColumnListBox;
import com.insightful.splus.SplusWideFormulaField;

public class LinearRegressionDialog extends SplusDialog {

    // Any controls referenced by listeners must be defined
    // here.
    // Be sure not to declare their class in the
    // constructor.

    SplusDataSetComboBox dataSet;
    SplusDataColumnComboBox dependentColumn, weightsColumn;
    SplusDataColumnListBox independentColumnList;
    SplusWideFormulaField formulaField;
    SplusButton formulaButton;

    SplusCheckBox longOutput, correlationMatrix,
        plotPartialResid, includeFitPartialResid,
        rugplotPartialResid, scalePartialResid;

    public LinearRegressionDialog() {
        super("Linear Regression");
    }
}
```

```
// A multiple page dialog with two columns.
// The first page has a wide group along the bottom which
// spans both columns.

// Controls and groups get added to the columns

JTabbedPane tabbedPane = new JTabbedPane();

/* Information needed to construct function call */
SplusFunctionInfo funcInfo = new
    SplusFunctionInfo("menuLm");
setFunctionInfo(funcInfo);

/* Model Page */

JPanel modelPage = new JPanel();
tabbedPane.addTab("Model", modelPage);
Box modelTopPanel = new Box(BoxLayout.X_AXIS);
Box modelBottomPanel = new Box(BoxLayout.X_AXIS);
Box modelTopLeftColumn = new Box(BoxLayout.Y_AXIS);
Box modelTopRightColumn = new Box(BoxLayout.Y_AXIS);
modelTopPanel.add(modelTopLeftColumn);
modelTopPanel.add(modelTopRightColumn);

modelPage.setLayout(new BoxLayout(modelPage,
    BoxLayout.Y_AXIS));
modelPage.add(modelTopPanel);
modelPage.add(modelBottomPanel);

// Data group

SplusGroupPanel dataGroup = new SplusGroupPanel("Data");
modelTopLeftColumn.add(dataGroup);

dataSet = new SplusDataSetComboBox();
dataGroup.add(dataSet);
funcInfo.add(dataSet, "data");

weightsColumn = new
```

```
        SplusDataColumnComboBox("Weights", 'W', dataSet);
weightsColumn.setValue("");
dataGroup.add(weightsColumn);
funcInfo.add(weightsColumn, "weights", true);

SplusSubsetField subsetRows = new SplusSubsetField();
dataGroup.add(subsetRows);
funcInfo.add(subsetRows, "subset");

SplusCheckBox omitMissing = new
    SplusCheckBox("Omit Rows with Missing Values", 'M');
omitMissing.setValue("T");
dataGroup.add(omitMissing);
funcInfo.add(omitMissing, "na.omit.p");

// Right column

// Spacer for right column

modelTopRightColumn.add(new SplusBoxFiller(8));

// Save Model Object group

SplusGroupPanel saveModelGroup = new
    SplusGroupPanel("Save Model Object");
modelTopRightColumn.add(saveModelGroup);

SplusTextField saveAsField = new
    SplusTextField("Save As", 'S');
saveModelGroup.add(saveAsField);
funcInfo.setResultControl(saveAsField);

// Variables group

// Dependent and Independent columns are used to build
// formula

SplusGroupPanel variablesGroup = new
    SplusGroupPanel("Variables");
modelBottomPanel.add(variablesGroup);
```

```
dependentColumn = new
    SplusDataColumnComboBox("Dependent", 'E', dataSet);
dependentColumn.setValue("");
variablesGroup.add(dependentColumn);
funcInfo.add(dependentColumn, "");

independentColumnList = new
    SplusDataColumnListBox("Independent", 'I', dataSet);
independentColumnList.setPrepend(new String []{"<ALL>"});
variablesGroup.add(independentColumnList);
funcInfo.add(independentColumnList, "");

formulaField = new SplusWideFormulaField();
variablesGroup.add(formulaField);
funcInfo.add(formulaField, "formula");

// Add listeners to Dependent and Independent to build
// formula

dependentColumn.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent ae){
        formulaField.setDependentValue(
            dependentColumn.getValue());
    }
});

independentColumnList.addListSelectionListener(new
    ListSelectionListener() {
        public void valueChanged(ListSelectionEvent e){
            formulaField.setIndependentValue(
                independentColumnList.getValue());
        }
    });

// Have button disable relevant fields and launch formula
// builder.

formulaButton = new SplusButton("Create Formula", 'C');

formulaButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent ae){
```

```
(new RegressionFormulaDialog(formulaField,
    dataSet,
    RegressionFormulaDialog.TYPE_LM)).show();
}
});

// Put button in a box for layout

Box formulaButtonBox = new Box(BoxLayout.X_AXIS);
formulaButtonBox.add(Box.createHorizontalStrut(5));
formulaButtonBox.add(formulaButton);
formulaButtonBox.add(Box.createHorizontalGlue());

variablesGroup.add(Box.createVerticalStrut(5));
variablesGroup.add(formulaButtonBox);
variablesGroup.add(Box.createVerticalStrut(5));

// bottom spacer
modelPage.add(new SplusWideBoxFiller(10));

/* Results page */

JPanel resultsPage = new JPanel();
tabbedPane.addTab("Results", resultsPage);
Box resultsColumnOne = new Box(BoxLayout.Y_AXIS);
Box resultsColumnTwo = new Box(BoxLayout.Y_AXIS);

resultsPage.setLayout(new BoxLayout(resultsPage,
    BoxLayout.X_AXIS));
resultsPage.add(resultsColumnOne);
resultsPage.add(resultsColumnTwo);

// Printed Results group

SplusGroupPanel printedResultsGroup =
    new SplusGroupPanel("Printed Results");
resultsColumnOne.add(printedResultsGroup);

SplusCheckBox shortOutput = new
    SplusCheckBox("Short Output", 'S');
```

```
printedResultsGroup.add(shortOutput);
funcInfo.add(shortOutput, "print.short.p");

longOutput = new SplusCheckBox("Long Output", 'L');
longOutput.setValue("T");
printedResultsGroup.add(longOutput);
funcInfo.add(longOutput, "print.long.p");

SplusCheckBox anovaTable = new
    SplusCheckBox("ANOVA Table", 'A');
printedResultsGroup.add(anovaTable);
funcInfo.add(anovaTable, "print.anova.p");

correlationMatrix = new
    SplusCheckBox("Correlation Matrix of Estimates",
        'C');
printedResultsGroup.add(correlationMatrix);
funcInfo.add(correlationMatrix, "print.correlation.p");

// Disable correlationMatrix if no Long Output

longOutput.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent ae){
        correlationMatrix.setEnabled(
            longOutput.getValue().equals("T"));
    }
});

// spacer

resultsColumnOne.add(new SplusBoxFiller());

// Saved Results group

SplusGroupPanel savedResultsGroup = new
    SplusGroupPanel("Saved Results");
resultsColumnTwo.add(savedResultsGroup);

SplusDataSetComboBox saveResultsIn = new
    SplusDataSetComboBox("Save In", 'I');
saveResultsIn.setValue("");
```

```
savedResultsGroup.add(saveResultsIn);
funcInfo.add(saveResultsIn, "save.name", true);

SplusCheckBox saveFittedValues = new
    SplusCheckBox("Fitted Values", 'F');
savedResultsGroup.add(saveFittedValues);
funcInfo.add(saveFittedValues, "save.fit.p");

SplusCheckBox saveResiduals = new
    SplusCheckBox("Residuals", 'R');
savedResultsGroup.add(saveResiduals);
funcInfo.add(saveResiduals, "save.resid.p");

// spacer

resultsColumnTwo.add(new SplusBoxFiller());

/* Plot page */

JPanel plotPage = new JPanel();
tabbedPane.addTab("Plot", plotPage);
Box plotColumnOne = new Box(BoxLayout.Y_AXIS);
Box plotColumnTwo = new Box(BoxLayout.Y_AXIS);

plotPage.setLayout(new BoxLayout(plotPage,
    BoxLayout.X_AXIS));
plotPage.add(plotColumnOne);
plotPage.add(plotColumnTwo);

// Which Plots group

SplusGroupPanel whichPlotsGroup =
    new SplusGroupPanel("Plots");
plotColumnOne.add(whichPlotsGroup);

SplusCheckBox plotResidVsFit = new
    SplusCheckBox("Residuals vs Fit", 'R');
whichPlotsGroup.add(plotResidVsFit);
funcInfo.add(plotResidVsFit, "plotResidVsFit.p");
```

```
SplusCheckBox plotSqrtAbsResid = new
    SplusCheckBox("Sqrt Abs Residuals vs Fit", 'Q');
whichPlotsGroup.add(plotSqrtAbsResid);
funcInfo.add(plotSqrtAbsResid, "plotSqrtAbsResid.p");

SplusCheckBox plotResponseVsFit = new
    SplusCheckBox("Response vs Fit", 'P');
whichPlotsGroup.add(plotResponseVsFit);
funcInfo.add(plotResponseVsFit, "plotResponseVsFit.p");

SplusCheckBox plotQQ = new
    SplusCheckBox("Residuals Normal QQ", 'Q');
whichPlotsGroup.add(plotQQ);
funcInfo.add(plotQQ, "plotQQ.p");

SplusCheckBox plotRFSpread = new
    SplusCheckBox("Residual-Fit Spread", 'S');
whichPlotsGroup.add(plotRFSpread);
funcInfo.add(plotRFSpread, "plotRFSpread.p");

SplusCheckBox plotCooks = new
    SplusCheckBox("Cook's Distance", 'C');
whichPlotsGroup.add(plotCooks);
funcInfo.add(plotCooks, "plotCooks.p");

plotPartialResid = new SplusCheckBox("Partial Residuals",
    'P');
whichPlotsGroup.add(plotPartialResid);
funcInfo.add(plotPartialResid, "plotPartialResid.p");

// spacer

plotColumnOne.add(new SplusBoxFiller());

// Options group

SplusGroupPanel plotOptionsGroup = new
    SplusGroupPanel("Options");
plotColumnTwo.add(plotOptionsGroup);
```

```
SplusCheckBox optionIncludeSmooth = new
    SplusCheckBox("Include Smooth", 'M');
optionIncludeSmooth.setValue("T");
plotOptionsGroup.add(optionIncludeSmooth);
funcInfo.add(optionIncludeSmooth, "smooths.p");

SplusCheckBox optionIncludeRugplot = new
    SplusCheckBox("Include Rugplot", 'G');
plotOptionsGroup.add(optionIncludeRugplot);
funcInfo.add(optionIncludeRugplot, "rugplot.p");

SplusTextField optionNumberId = new
    SplusTextField(
        "Number of Extreme Points to Identify:", 'X');
optionNumberId.setValue("3");
plotOptionsGroup.add(optionNumberId);
funcInfo.add(optionNumberId, "id.n");

// Partial Residual Plot Options group

SplusGroupPanel partialResidualOptionsGroup = new
    SplusGroupPanel("Partial Residual Plot Options");
plotColumnTwo.add(partialResidualOptionsGroup);

includeFitPartialResid = new SplusCheckBox(
    "Include Partial Fit", 'F');
includeFitPartialResid.setEnabled(false);
partialResidualOptionsGroup.add(includeFitPartialResid);
funcInfo.add(includeFitPartialResid, "plotPartialFit.p");

rugplotPartialResid = new SplusCheckBox(
    "Include Rugplot", 'T');
rugplotPartialResid.setEnabled(false);
partialResidualOptionsGroup.add(rugplotPartialResid);
funcInfo.add(rugplotPartialResid,
    "rugplotPartialResid.p");

scalePartialResid = new
    SplusCheckBox("Common Y-axis Scale");
scalePartialResid.setValue("T");
scalePartialResid.setEnabled(false);
```

```
partialResidualOptionsGroup.add(scalePartialResid);
funcInfo.add(scalePartialResid, "scalePartialResid.p");

// Enable options only when plotPartialResid

plotPartialResid.addActionListener( new ActionListener(){
    public void actionPerformed(ActionEvent ae){
        boolean enableState =
            plotPartialResid.getValue().equals("T");
        includeFitPartialResid.setEnabled(enableState);
        rugplotPartialResid.setEnabled(enableState);
        scalePartialResid.setEnabled(enableState);
    }
});

// spacer

plotColumnTwo.add(new SplusBoxFiller());

/* Predict page */

JPanel predictPage = new JPanel();
tabbedPane.addTab("Predict", predictPage);
Box predictColumnOne = new Box(BoxLayout.Y_AXIS);
Box predictColumnTwo = new Box(BoxLayout.Y_AXIS);

predictPage.setLayout(new BoxLayout(predictPage,
    BoxLayout.X_AXIS));
predictPage.add(predictColumnOne);
predictPage.add(predictColumnTwo);

// New data field has no group

SplusGroupPanel predictNewDataGroup = new
    SplusGroupPanel("");
predictColumnOne.add(predictNewDataGroup);

SplusDataSetComboBox predictNewData = new
    SplusDataSetComboBox("New Data", 'N');
```

```
predictNewData.setValue("");
predictNewDataGroup.add(predictNewData);
funcInfo.add(predictNewData, "newdata");

// Save group

SplusGroupPanel predictSaveGroup = new
    SplusGroupPanel("Save");
predictColumnOne.add(predictSaveGroup);

SplusDataSetComboBox predictSaveIn = new
    SplusDataSetComboBox("Save In", 'I');
predictSaveIn.setValue("");
predictSaveGroup.add(predictSaveIn);
funcInfo.add(predictSaveIn, "predobj.name", true);

SplusCheckBox predictSavePredictions = new
    SplusCheckBox("Predictions", 'P');
predictSaveGroup.add(predictSavePredictions);
funcInfo.add(predictSavePredictions, "predict.p");

SplusCheckBox predictSaveConfInt = new
    SplusCheckBox("Confidence Intervals", 'C');
predictSaveGroup.add(predictSaveConfInt);
funcInfo.add(predictSaveConfInt, "ci.p");

SplusCheckBox predictSaveStdError = new
    SplusCheckBox("Standard Errors", 'S');
predictSaveGroup.add(predictSaveStdError);
funcInfo.add(predictSaveStdError, "se.p");

// spacer

predictColumnOne.add(new SplusBoxFiller());

// Options group

SplusGroupPanel predictOptionsGroup = new
    SplusGroupPanel("Options");
predictColumnTwo.add(predictOptionsGroup);
```

```
SplusTextField predictConfLevel = new
    SplusTextField("Confidence Level", 'L');
predictConfLevel.setValue("0.95");
predictOptionsGroup.add(predictConfLevel);
funcInfo.add(predictConfLevel, "conf.level");

// spacer

predictColumnTwo.add(new SplusBoxFiller());

// add center panel

setCenterPanel(tabbedPane);

    }

// Redefine isComplete() to check for required arguments.

public boolean isComplete() {
    return (warnIfEmpty(dataSet) &&
        warnIfEmpty(formulaField));
}

}
```

Introduction to Creating Help Files in Windows	440
Required Software and Scripts	442
Downloading and Running Cygwin	442
Downloading and Running HTML Help Workshop	443
Creating, Editing, and Distributing a Help File in Windows	444
Step 1: Creating the Help File	444
Step 2: Editing the Help File	445
Step 3: Editing and Running the Script	446
Step 4: Checking the .chm File	449
Step 5: Distributing the .chm File	449
Errors	449
Introduction to Creating Help Files in UNIX	451
Creating, Editing, and Distributing a Help File in UNIX	453
Step 1: Creating the Help File	453
Step 2: Copying the Help File to a “Clean” Directory	454
Step 3: Running the CHAPTER Utility	454
Step 4: Editing the Help File	454
Step 5: Running <code>Splus make install.help</code>	455
Step 6: Viewing the Help File	457
Step 7: Distributing the Help File	457
Common Text Formats	458
Contents of Help Files	460
Descriptions of Fields	460
Special Help Files	474

INTRODUCTION TO CREATING HELP FILES IN WINDOWS

When a S-PLUS function is made available system-wide, a help file should always accompany it. Without a help file, use of the function is likely to be limited to those in close contact with the person who wrote it.

Note

Creating Help using this system is deprecated as of Spotfire S+ 8.2.

Rather than using the prompt command and SGML to create Help, We recommend using the Spotfire S+ Packages system, because it bundles your functions, data sets, C code, and help files in a more systematic way.

For details, refer to the *Guide to Packages* in your **SHOME/help** (Windows[®]) or **SHOME/doc** (Solaris/Linux[®]) directory.

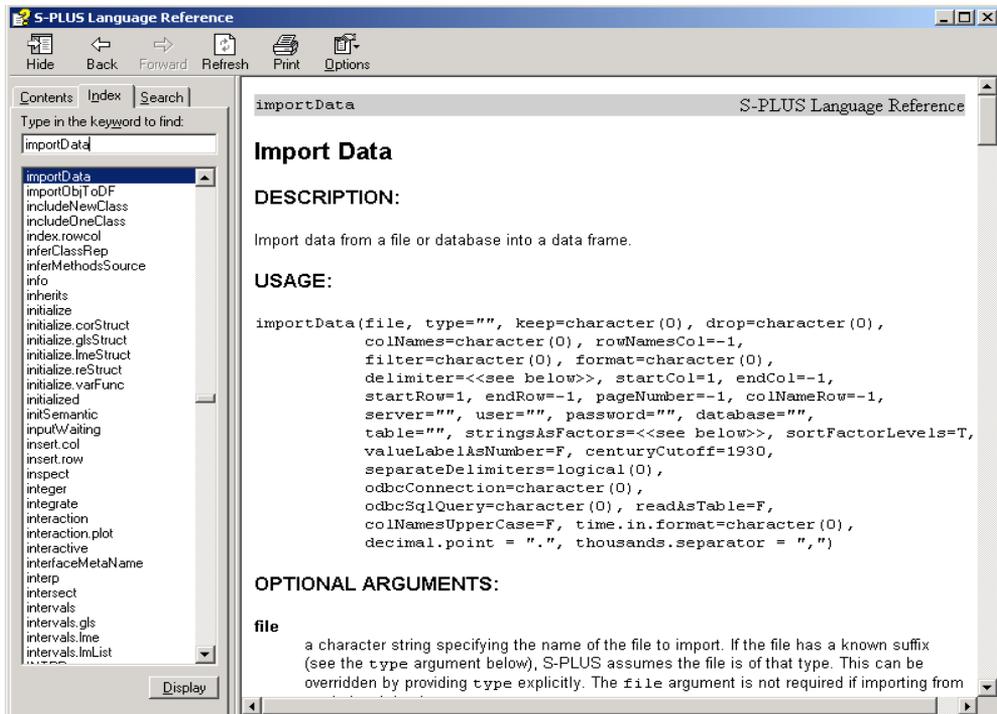


Figure 11.1: An example of a compiled HTML Help project, *splus.chm*.

Starting with S-PLUS 6, the S Language version 4 provided tools for creating and processing SGML (Standard Generalized Markup Language) help files to document your user-defined S-PLUS functions. Using the built-in functions available in your Spotfire S+ installation plus two other downloaded programs, you can create and distribute user-defined help files in **.chm** format, a compiled form of HTML. The HTML Help (**.chm**) format is the online help standard from Microsoft used in Spotfire S+. Figure 11.1 shows an example of the HTML Help viewer displaying **splus.chm**, the Spotfire S+ Language Reference.

The S-PLUS `prompt` function generates a help file outline for any S-PLUS function or data object you specify. The outline includes the correct syntax for the function, as well as the current definition of the function, and headings under which you can supply the following information:

- What the function does
- A brief description of the function
- A description of each required and optional argument
- A description of the value returned by the function
- *Side effects*, which are any effects of the function besides the return of a value
- The *method* behind the function; that is, how the function works
- References to any relevant external literature
- Cross-references to related online help files
- One or more Keywords (used by the Spotfire S+ help system in assembling its **Category** and **Function** menus)

The following sections guide you through the process of generating an HTML Help help **.chm** file and distributing it with your S-PLUS functions. We include descriptions and locations of the software you need to download, the elements of the build script, and the components used to create, edit, view, and distribute your customized help files on a Windows platform.

Required Software and Scripts

Before you can create HTML Help for your S-PLUS functions, you need to have the following software installed:

1. **Cygwin** The Cygwin tools are ports of the popular GNU development tools for Microsoft Windows. It is a UNIX environment for Windows that consists of two parts:
 - **cygwin1.dll**, which acts as a UNIX emulation layer providing substantial UNIX API functionality.
 - A collection of tools ported from UNIX which provide UNIX/Linux[®] look and feel.
2. **HTML Help Workshop** This is a help authoring tool that provides an easy-to-use system for creating and managing help projects.
3. **The BuildHelpFiles directory** This contains the build scripts used for converting the SGML files to HTML, creating the index and TOC, and compiling the HTML to produce a **.chm** file and generating the **.chm**. It is included in your Spotfire S+ installation and is located in:

SHOME\help\BuildHelpFiles

Downloading and Running Cygwin

After you create and edit your Spotfire S+ SGML help files, use Cygwin to run the scripts that convert your SGML files into HTML format, create an index and table of contents (TOC), and fix broken links before you create the **.chm** file. A free download of Cygwin is available from:

www.cygwin.com

Notes

In the **Select Install Root Directory** dialog, the Cygwin default setting for **Default Text File Type** is UNIX. Be sure to set the **Default Text File Type** to **DOS** when installing to a Windows system.

Because the full Cygwin installation requires a large amount of disk space, by default the Cygwin Set up program installs only the minimal base packages. In the **Select Packages** dialog, be sure to select the GNU Bourne Again Shell (**bash**) for installation.

Refer to the Cygwin web site and installation instructions for more information.

To use Cygwin from within a bash shell, double-click the **Cygwin** icon on your desktop.

To use Cygwin from a Windows Command Prompt, you must first add the **cygwin\bin** directory to your path:

1. Right-click the **My Computer** icon on your desktop and select **Properties**.
2. Select the **Advanced** tab.
3. Click the **Environment Variables** button.
4. In the **System variables** group, select the **Path** variable and click **Edit**. If there is no **Path** variable, click **New** and enter **Path** for the new variable name.
5. Edit the value of the **Path** environment variable so that the full path to **cygwin\bin** is included, for example:

c:\cygwin\bin;%system%;c:\foobar

If there are other directories you want to search first, make sure they are positioned ahead of **cygwin\bin** in the **path** variable.

Downloading and Running HTML Help Workshop

After your SGML files have been converted to HTML and an index and TOC have been generated, these files are compiled into a single **.chm** file, which can be distributed. The required HTML compiler is component of the Microsoft HTML Help Workshop, available from:

www.msdn.microsoft.com/library/en-us/htmlhelp/html/hwMicrosoftHTMLHelpDownloads.asp

Choose the defaults for the installation.

CREATING, EDITING, AND DISTRIBUTING A HELP FILE IN WINDOWS

The following is an overview of the process for creating and distributing user-defined help for Windows:

1. In Spotfire S+, use the `prompt` function to generate a template SGML help file. If you need multiple help files, run `prompt` once for each file you need.
2. Edit your SGML files with your Windows text editor of choice.
3. Customize and then run the **build_helpchm.cyg** script (either from Cygwin or from a Windows Command Prompt) to create a **.chm** file.
4. Double-click the **.chm** file to view and verify the content.
5. Distribute the **.chm** file by copying it to the appropriate directory.

Repeat steps 2-4 until you are satisfied with the results.

The following sections describe the detailed steps of the entire process.

Step I: Creating the Help File

Let's suppose you create a new S-PLUS function named `myfunc`:

```
> myfunc <- function(x) return(x * 2)
```

Use the `prompt` function to create a template help file named **myfunc.sgml**:

```
> prompt(myfunc)
created file named myfunc.sgml in the current directory
edit the file according to the directions in the file.
```

By default, this file is placed in your Spotfire S+ working directory.

To create multiple help files, run `prompt` for each function or data object you want to document.

Step 2: Editing the Help File

Once you have created one or more SGML files, you can edit them in your text editor of choice, such as Notepad or Wordpad. The following are the first few lines from the template SGML help file for myfunc:

```
<!doctype s-function-doc system "s-function-doc.dtd" [  
<!entity % S-OLD "INCLUDE">  
]  
>  
<s-function-doc>  
<s-topics>  
<s-topic> myfunc </s-topic>  
</s-topics>  
<s-title>
```

The first four lines of **myfunc.sgml** are required in all Spotfire S+ SGML help files. For this reason, we recommend that you always use prompt to create a template file rather than write your SGML code from scratch.

The `<s-function-doc>` tag begins the contents of the help file. The end tag `</s-function-doc>` should appear at the end of the file. The start and end tags for most fields in a Spotfire S+ help file are included in the template files that prompt creates. If you do not want to include a particular field in your help file, you can delete the start and end tags for that field.

The meaning and use of each SGML tag in Spotfire S+ help files is described in the section Common Text Formats and the section Contents of Help Files later in this chapter.

Notes

In order to properly deploy your online help such that it can be found via `help(myfunc)` within Spotfire S+, you must include at least one keyword in the SGML file. For example: `<s-keyword>dp1ot</s-keyword>`. For details about Spotfire S+ keywords, see section Keywords on page 470.

The SGML parser cannot handle extremely long lines of text. If the text for a given section is especially long, be sure to break it into shorter lines. If the parser encounters a line that is too long, it truncates the help file and inserts the following error message:

Input string too long

Step 3: Editing and Running the Script

The Spotfire S+ Windows installation places the **build_helpchm.cyg** script in your **\$HOME\help\BuildHelpFiles** directory. This serves as a template that you will need to customize for your particular environment.

To customize the script, we recommend that you leave the original read-only template file intact and create a writable copy in your **\$HOME\help\BuildHelpFiles** directory, named for the help system you are creating (e.g., **build_helpchm_myfunc.cyg**).

Continuing with the `myfunc` example from the previous steps, the following procedure describes the essential script edits required to build a `myfunc.chm` file. For more detailed information about each script setting, refer to the comments in the script template.

Note

As a Cygwin script, **build_helpchm.cyg** requires that all paths be specified using DOS short path names with *forward* slash (/) separators instead of the normal DOS back slash (\) separators. See examples below.

1. Set the path to your Cygwin bin directory. Specify the same location you added to your path in section Downloading and Running Cygwin earlier in this chapter, but be sure to use forward slashes. For example:

```
CYGWIN_BIN=C:/cygwin/bin
```

2. Using DOS short name syntax, set the path to the directory that contains **hhc.exe**, the Microsoft HTML Help compiler. For example:

```
HELPSHOP_DIR="C:/PROGRA~1/HTMLHE~1"
```

To see the short names, at the Windows Command Prompt run `dir` with the `/x` option. For example:

```
C:\>dir /x
<DIR>      PROGRA~1      Program Files
```

3. Set the path to the directory where you want the script to place your output **.chm** and **help.log** files. For example:

```
CHM_DIR=C:/PROGRA~1/INSIGH~2/spplus80/users/myname
```

4. Specify a base filename for the **.chm** file you are building. The script will automatically add the **.chm** filename extension. For example:

```
CHM_NAME=myhelp
```

5. Specify a header to appear in the upper-right corner of each help topic in the **.chm** file. For example:

```
HEADER="myfunc Language Reference"
```

6. Specify the complete filename for the help topic you want displayed by default when your **.chm** file is first opened in the HTML Help viewer. For example:

```
DEFAULTFILE=myfunc.html
```

Note that the base filename of an **.sgm** file can change when converted to an HTML file. If you get a compilation error from HTML Help Workshop, make sure the default file you specify is in your `__HTML` directory.

7. The **build_helpchm.cyg** script template includes the following Cygwin command to set the current directory as a short DOS path with forward slashes:

```
CURRENT_DIR=`pwd | cygpath -ms -f -`
```

8. Set the path to the directory that contains the **.chm** build tools. For example:

```
CHM_TOOLS=$CURRENT_DIR
```

9. If the **.chm** file you are building contains links to the Spotfire S+ Language Reference (**splus.chm**), set

```
SEARCH_STOPIC_LIST=T. Otherwise, set it to F as follows:
```

```
SEARCH_STOPIC_LIST=F
```

```
STOPIC_LIST_SPLUS=$CHM_TOOLS/stopicList.out
```

Do not remove or comment out these two lines.

10. If you want your **.chm** file to include **.html** files that are *not* being converted from SGML during this build, set `GUI_HELP_DIR` to the directory where the **.html** files are located. If your external **.html** (help topic) files call any **.htm** (popup) files, set `HAVE_HTM_FILES=T`. For example:

```
GUI_HELP_DIR=$CHM_DIR/html_include  
HAVE_HTML_FILES=T
```

If you do not want to include external **.html** or **.htm** files, comment out these two lines.

11. Set the path to the directory that contains the Spotfire S+ SGML help files you have prepared for this build. Also, ensure that `HTML_DIR_ROOT` is set to `$SGML_DIR`. For example:

```
SGML_DIR=$CHM_DIR/sgml  
HTML_DIR_ROOT=$SGML_DIR
```

Do *not* set `SGML_DIR` to a path that include directory names that begin with a period (e.g., a path under your `.Data` directory). Doing so produces HTML Help Workshop errors.

12. The last executable line of the script builds the **.chm** file. In most cases you do not need to edit this line.
13. Save your **build_helpchm_myfunc.cyg** script in your **\$HOME\help\BuildHelpFiles** directory.
14. To execute your script, either from Cygwin or a Windows Command Prompt, `cd` to the **\$HOME\help\BuildHelpFiles** directory and enter:

```
bash build_helpchm_myfunc.cyg
```

If the build is successful, a success message appears in your command window and the **.chm** and **help.log** files are written to your **CHM_DIR**.

After a successful build, the `SGML_DIR` specified in your **build_helpchm.cyg** script will contain the following subdirectories:

- **__Shelp** contains temporary working versions of your `.sgml` help files renamed with an **.sgm** file extension.
- **__Hhelp** contains the converted **.html** files that are input to the HTML Help Workshop, as well as the other files used in the compilation: **.hhc** (contents), **.hhk** (index), and **.hhp** (project).

Note that **__Shelp** and **__Hhelp** are temporary working directories that are removed and recreated each time you run your **build_helpchm.cyg** script.

Step 4: Checking the .chm File

Go to the target directory for your **.chm** file (in the example above, **C:\Documents and Settings\myname\My Documents\Spotfire S+ Projects**). Double-click the **.chm** file. You should see a help topic similar to the one in Figure 11.1.

Check the various components of your output **.chm**:

- Title of the HTML Help Viewer
- TOC
- Index
- Default help file
- Header string, located in the upper-right and the title bar of the HTML Help Viewer window
- Any other components specified in the **build_helpchm.cyg** script (e.g., included **.html** and **.htm** files)

If you want to make any changes to the **.chm** file, edit your original SGML files and rebuild the **.chm** until you are satisfied with the results.

Step 5: Distributing the .chm File

When the **.chm** file is ready for deployment, copy it to the directory that contains the function or functions you are deploying, for example, **SHOME\library\myfunc**.

Alternatively, you can copy it to your **SHOME\cmd** directory, the same location as the Spotfire S+ Language Reference (**splus.chm**) and the Spotfire S+ Graphical User Interface (**gui.chm**) help files.

To access your help file, enter the following the Spotfire S+ command:

```
> help(myfunc)
```

Spotfire S+ searches through all attached libraries to find the help file for **myfunc**, then displays it in your HTML Help viewer.

Errors

The **help.log** file (written to the same directory as the output **.chm** file) logs each step executed by the **build_helpchm.cyg** script. Refer to this file if you encounter errors.

For more detailed information on the structure of the Spotfire S+ SGML help files, refer to section Common Text Formats on page 458.

INTRODUCTION TO CREATING HELP FILES IN UNIX

When an S-PLUS function is made available system-wide, a help file should always accompany it. Without a help file, use of the function is likely to be limited to those in close contact with the person who wrote it.

Starting with S-PLUS 5.1, the S Language version 4 provides the flexibility of creating and editing your own SGML (Standard Generalized Markup Language) help files to document your user-defined functions in Spotfire S+. Using the built-in functions distributed in your Spotfire S+ installation, you can use and distribute help via JavaHelp™, the help system from Sun Microsystems included with Spotfire S+ 8. JavaHelp is used to display the `importData` help file, as shown in Figure 11.2.

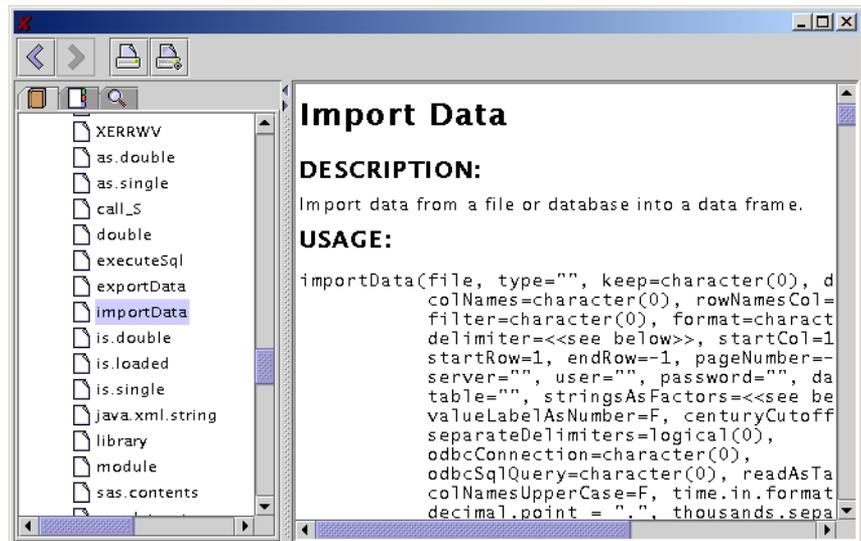


Figure 11.2: JavaHelp is used to invoke the `importData` help file in Spotfire S+.

The S-PLUS function prompt generates a help file outline for any S-PLUS function or data object you specify. The outline includes the correct syntax for the function, as well as the current definition of the function, and headings under which you can supply the following information:

- What the function does.
- A brief description of the function.
- A description of each argument, with the option of specifying both required and optional arguments.
- A description of the value returned by the function.
- *Side effects*, which are any effects of the function besides the return of a value.
- The *method* behind the function; that is, how the function works.
- Any references to the literature.
- Cross-references to other help files.
- Keywords. Keywords are used by the Spotfire S+ help system in assembling its **Category** and **Function** menus.

The following sections describe the steps involved in creating, editing, viewing, and distributing your customized help files.

CREATING, EDITING, AND DISTRIBUTING A HELP FILE IN UNIX

Creating a help file for distribution in S-PLUS 6 for UNIX involves the use of Spotfire S+ commands and commands from a UNIX prompt. In general, the steps are as follows:

1. Use the prompt function to generate a SGML file in your working directory.
2. Copy the SGML file to a new directory.
3. Run `Sp1us CHAPTER` on this directory to create a Spotfire S+ chapter.
4. Edit the SGML file with your editor of choice.
5. Run `Sp1us make install.help` to install the file so it can be accessed by JavaHelp.
6. Launch Spotfire S+, attach the Spotfire S+ chapter, and test the help file by typing

```
> help(myfunc)
```

Repeat steps 4-6 until you are satisfied with the results. If you have a number of files you want to add to your distribution, you can also run this process on a number of help files at once.

Your working directory may change, but as long as you attach the chapter that contains your help files, you can always access them.

To illustrate this process, we create a sample help file, and detail each of the steps involved in distributing it in the following sections.

Step 1: Creating the Help File

Let's suppose you create a new S-PLUS function named `myfunc`:

```
> myfunc <- function(x) return(x * 2)
```

Use the prompt function to create a template help file named **myfunc.sgml**:

```
> prompt(myfunc)
created file named myfunc.sgml in the current directory
edit the file according to the directions in the file.
```

By default, this file is placed in your Spotfire S+ working directory. If you have a number of help files to create, use prompt for each function.

Step 2: Copying the Help File to a “Clean” Directory

The next step is to copy the SGML file(s) to a directory that will become a Spotfire S+ chapter. This chapter becomes the chapter you attach in your Spotfire S+ session to access your user-defined help files.

From a UNIX prompt, create a new directory and copy the SGML files from the working directory:

```
mkdir myfuncdir
cp myfunc.sgml myfuncdir
```

If you have a number of help files, type

```
cp *.sgml myfuncdir
```

to copy all the SGML files to the directory. Change directories to **myfuncdir** and proceed to the next step.

Step 3: Running the CHAPTER Utility

As mentioned in the previous step, the CHAPTER utility must be run on the directory so **myfuncdir** can be recognized as a Spotfire S+ chapter:

```
Splus CHAPTER
```

This process creates the chapter's **.Data** directory, which includes the **__Shelp** and **__Hhelp** directories that are required in step 5 to install the files so they can be accessed by JavaHelp.

Step 4: Editing the Help File

You can view and edit the file in your text editor of choice, such as emacs or vi. For example, if you want to invoke the emacs editor to edit your file, type the following command:

```
emacs myfunc.sgml
```

Your skeleton help file should contain text similar to the following:

```
<!doctype s-function-doc system "s-function-doc.dtd" [
<!entity % S-OLD "INCLUDE">
]
>
<s-function-doc>
```

```
<s-topics>  
<s-topic> myfunc </s-topic>  
</s-topics>  
<s-title>  
. . .
```

The first four lines of **myfunc.sgml** are required in all SGML help files for Spotfire S+. For this reason, we recommend that you use `prompt` to create a template file rather than write SGML code from scratch.

The `<s-function-doc>` tag begins the contents of the help file; you should see the end tag `</s-function-doc>` at the end of the file. The start and end tags for most fields in a Spotfire S+ help file are included in the skeleton files that `prompt` creates. If your function does not require a certain field in its help file, you can delete the corresponding tags.

If the text for a given section is particularly long, be sure to start sentences or major phrases on new lines. The SGML parser tends to break when a single line contains too much text. When this occurs, the help file is truncated at the point where the parser breaks and an error message is inserted:

```
Input string too long
```

There is a great deal of information on the meaning and use of the SGML tags in Spotfire S+ help files, and this is described at length in the section `Common Text Formats` and the section `Contents of Help Files` that follow. These sections have been omitted here for brevity.

Step 5: Running

```
Splus make  
install.help
```

Now that the SGML files have been edited, view them in Spotfire S+ to verify they have the proper content and formatting. This requires another Spotfire S+ utility to be run, so at a UNIX prompt, type

```
Splus make install.help
```

Running the `Splus make install.help` utility invokes two other processes:

- `HINSTALL`: This copies **myfunc.sgml** to **.Data/___Shelp/myfunc.sgm** and translates this **.sgm** file to HTML and stores it as **.Data/___Hhelp/myfunc.html**. Note the suffix is **.sgm** and not **.sgml**, which preserves the original file.

- **BUILD_JHELP:** This creates the **__Jhelp** directory and populates the directory with the XML files required to catalog and dispatch JavaHelp. These XML files are described in Table 11.1.

Table 11.1: Files within the **__Jhelp** directory.

File	Description
*.hs	The help set file, which acts as an identifier to JavaHelp.
*Index.xml	The index file, which lists all the topics in the help set and is used as the text for the Index tab.
*TOC.xml	The Table-of-Contents file, which lists the topics in the help set by category and is used as the text for the Contents tab.
*Map.jhm	The mapping file, which maps topic names to specific URLs. This is the most important file in the help set; both the Index file and the TOC file rely on it.
mapsHelpSearch	A directory containing the files used by JavaHelp's full text search facility on the Search tab.

This process also reveals any errors in SGML and warns you if there are any deficiencies. If the `Splus make install.help` should fail in this event, edit your SGML file before proceeding.

Now that the help files can be accessed by JavaHelp, you can view your files in Spotfire S+ and determine if any changes need to be made.

Step 6: Viewing the Help File

To view the edits in your help file, first launch Spotfire S+. Then, attach the chapter and invoke your help file from the **Commands** window:

```
> attach("myfuncdir")  
> help(myfunc)
```

You should see the help file for `myfunc` displayed in the help viewer, and you can check for any formatting or content errors. If you need to correct anything, repeat steps 4 through 6 by editing the file, running `Splus make install.help`, and viewing the file.

Step 7: Distributing the Help File

You can modify your **S.init** file to attach the chapter where you created your help files each time you start Spotfire S+. You can now distribute the files as necessary, which you could package as a compressed archive or self-extracting executable. A utility such as `compress` or `gzip` could be used for this purpose.

The remaining sections in this chapter are devoted to the use and format of SGML tags in Spotfire S+ help files, which are necessary to understand the editing phase of step 4. The same tags apply to SGML help files whether they are destined for Windows or UNIX platforms.

COMMON TEXT FORMATS

Table 11.2 lists the most common text formats for Spotfire S+ SGML files. Note that this list is not exhaustive; in general, there might be many tags to format text in a particular way. We suggest using tags with names that describe the formats themselves. Thus, instead of using `<tt>` and `</tt>` for code, we encourage you to use `<s-expression>` and `</s-expression>` (or `<code>` and `</code>` to save typing).

Table 11.2: *Common text formats in Spotfire S+ SGML files.*

Format	SGML Tags	Notes
Fixed-width code font	<code><s-expression></code> , <code></s-expression></code> <code><code></code> , <code></code></code> <code><tt></code> , <code></tt></code>	Although <code><tt></code> and <code></tt></code> still exist in some older help files, please use the other tags instead.
Preformatted text	<code><pre></code> , <code></pre></code>	This is usually for code samples that extend over multiple lines. With <code><pre></code> , you are responsible for line breaks and blank space, as SGML does no formatting for you.
Italics Emphasis	<code><it></code> , <code></it></code> <code></code> , <code></code>	
Bold	<code><bf></code> , <code></bf></code>	
Lists Enumerated lists	<code><descrip></code> , <code></descrip></code> <code><enum></code> , <code></enum></code> <code><item></code> , <code></item></code> <code><tag></code> , <code></tag></code>	The <code><descrip></code> and <code><enum></code> tags create the lists. Use <code><item></code> and <code></item></code> , or <code><tag></code> and <code></tag></code> , to format the individual list elements. See the <code>n1minb</code> help file for an example.

Table 11.2: Common text formats in Spotfire S+ SGML files. (Continued)

Format	SGML Tags	Notes
Line breaks	 	To include a blank line in a help file, must be used twice: once for the text line and once for the blank line. The tag is not needed in preformatted sections.
Paragraph breaks	<p>	This is the recommended tag for including blank lines in a help file.

CONTENTS OF HELP FILES

Descriptions of Fields Each Spotfire S+ help file is composed of titled sections that appear in the order listed in Table 11.3. We discuss each of these sections below.

Table 11.3: *Titled sections in S-PLUS engine help files. Titles in all capitals appear in a formatted help file. Titles in lowercase letters do not appear in a help file, but the information in those sections do.*

Section Title	Quick Description	SGML Tags
Topic	The name of the function.	<s-topics> </s-topics>
Title	The title that appears at the top of a formatted help file.	<s-title> </s-title>
DESCRIPTION	A short description of the function.	<s-description> </s-description>
USAGE	The function call with all of its arguments.	<s-usage> </s-usage>
REQUIRED ARGUMENTS	Descriptions of arguments that are required by the function.	<s-args-required> </s-args-required>
OPTIONAL ARGUMENTS	Descriptions of arguments that are optional.	<s-args-optional> </s-args-optional>
VALUE	The return value from the function.	<s-value> </s-value>
SIDE EFFECTS	Side effects from the function.	<s-side-effects> </s-side-effects>
GRAPHICAL INTERACTION	A description of graphical interactions expected of the user.	See below.
CLASSES	A description of the classes the function is applicable to, if it is a default method.	See below.

Table 11.3: Titled sections in S-PLUS engine help files. Titles in all capitals appear in a formatted help file. Titles in lowercase letters do not appear in a help file, but the information in those sections do. (Continued)

Section Title	Quick Description	SGML Tags
WARNING	Anything the user should be warned about when using the function.	See below.
DETAILS	Descriptions of algorithmic details and implementation issues.	<s-details> </s-details>
BACKGROUND	Background information on the function or method.	See below.
NOTE	Any information that does not fit into the above categories.	See below.
REFERENCES	Available texts and papers the user can refer to for additional information.	See below.
BUGS	Descriptions of known bugs in the function.	See below.
SEE ALSO	Links to related S-PLUS functions.	<s-see> </s-see>
EXAMPLES	Coded Spotfire S+ examples.	<s-examples> </s-examples>
Keywords	A list of keywords that place the help file in the Contents topics of the help system.	<s-keywords> </s-keywords>

Topic

The topic section contains the function name wrapped in the tags <s-topic> and </s-topic>. In the **myfunc.sgml** example, the topic looks like:

```
<s-topics>
<s-topic> myfunc </s-topic>
</s-topics>
```

In help files containing multiple functions, each function name should be wrapped in the `<s-topic>` tags. For example, the following is an excerpt from the common SGML file for `cor` and `var`:

```
<s-topics>
  <s-topic>cor</s-topic>
  <s-topic>var</s-topic>
</s-topics>
```

The topic section is not visible in a formatted help file, but is used to index the file in the help system.

Title

The title section contains the title that appears at the top of a formatted help file. For example, the title from the `coxph` SGML file is:

```
<s-title>
Fit Proportional Hazards Regression Model
</s-title>
```

All words in a title should begin in uppercase letters. For clarity, avoid Spotfire S+ jargon such as function names and class names in the title section. A title should be short enough to fit on one line in the help file.

DESCRIPTION

This section contains a short description of the function. The description in the `coxph` SGML file is:

```
<s-description>
Fits a Cox proportional hazards regression model.
Time dependent variables, time dependent strata, multiple
events per subject, and other extensions are incorporated
using the counting process formulation of Andersen and
Gill.
</s-description>
```

USAGE

This section includes the function call with all of its arguments. You should list optional arguments with the form `name=default`. If a default argument is complicated, use the form `name=<<see below>>` instead and describe the default in the argument's section of the SGML file. Because the angle brackets `<` and `>` signify tags in SGML,

however, it is safest to type them as `<` and `>` when tags are not intended. Thus, the form for a complicated default is `name=<<see below>>`.

If the help file describes more than one function, the usage for each function should be listed on separate lines. If the argument list for a function is more than one line long, subsequent lines should be indented to one space past the opening parentheses. Use spaces to indent each line instead of tabs. For example, the following is the usage section from the SGML file for `lm`:

```
<s-usage>
<s-old-style-usage>
lm(formula, data=&lt;&lt;see below&gt;&gt;,
   weights=&lt;&lt;see below&gt;&gt;,
   subset=&lt;&lt;see below&gt;&gt;, na.action=na.fail,
   method="qr", model=F, x=F, y=F, contrasts=NULL, ...)
</s-old-style-usage>
</s-usage>
```

The tag `<s-old-style-usage>` preformats the section so that it prints in a fixed-width font. This also causes newlines and blank lines to be recognized in the text. Because of the preformatting, no space is needed around the equals signs for each of the arguments. In addition, please ensure that the width of each line is no more than 60 characters so that the usage section displays nicely in conveniently-sized help windows.

REQUIRED ARGUMENTS

This section lists each required argument wrapped in the tags `<s-arg name="name">` and `</s-arg>`. The first word in the description of each argument should not be capitalized. For example, the following excerpt lists the three required arguments from the SGML file for `ifelse`:

```
<s-args-required>
<s-arg name="test">
logical object. Missing values <s-object>(NA)</s-object>
are allowed.
</s-arg>
<s-arg name="yes">
vector containing values to be returned for elements with
<s-expression>test</s-expression> equal to
<s-expression>TRUE</s-expression>.
```

```
</s-arg>
<s-arg name="no">
vector containing values to be returned for elements with
<s-expression>test</s-expression> equal to
<s-expression>FALSE</s-expression>.
</s-arg>
</s-args-required>
```

In descriptions of arguments, you should always state whether exceptional values (NA, NaN, Inf, etc.) are treated specially.

OPTIONAL ARGUMENTS

This section lists each optional argument wrapped in the tags `<s-arg name="name">` and `</s-arg>`. As in the section for required arguments, the first word in each argument's description should not be capitalized, and you should document whether exceptional values are accepted.

VALUE

This section describes the return value of the function. The first word in the description should not be capitalized. Often, the description of a function's return value begins with a phrase similar to:

a list containing the following components:

In this case, you can use the tags `<s-return-component name="name">` and `</s-return-component>` to format the individual components of the return list. The first word in the description for each return component should not be capitalized. As an illustration, the value section of the density SGML file is provided below.

```
<s-value>
a list with two components, <s-expression>x</s-expression>
and <s-expression>y</s-expression>, suitable for giving as
an argument to one of the plotting functions.
<s-return-component name="x">
a vector of <s-expression>n</s-expression> points at which
the density is estimated.
</s-return-component>
<s-return-component name="y">
the density estimate at each <s-expression>x</s-expression>
point.
</s-return-component>
</s-value>
```

You should include articles in the uncapitalized sentences of the value section. That is, you should write

`a list with two components`

instead of simply

`list with two components`

SIDE EFFECTS

Side effects of the function (plotting, changing graphics devices, changing session options, etc.) are described in this section. This is also the appropriate place to describe the lack of side effects if a user might expect one. For instance, the fact that many S-PLUS editing functions do not actually change an object can be documented in the side effects sections of their SGML files.

Any function that updates the object `.Random.seed` must include the following message in its side effects section:

The function *name* causes the creation of the data set `.Random.seed` if it does not already exist. Otherwise, the value of `.Random.seed` is updated.

GRAPHICAL INTERACTION

If the user is expected to interact with a graphical display, the interaction is described in this section. Help files that currently contain this section include `burl.tree`, `hist.tree`, and `snip.tree`. The graphical interaction section in Spotfire S+ SGML files does not have a specific tag. Instead, the tags `<s-section name="name">` and `</s-section>` are used. For example, the following excerpt is from the `hist.tree` SGML file.

```
<s-section name="GRAPHICAL INTERACTION">
```

```
This function checks that the user is in split-screen mode.
A dendrogram of <s-expression>tree</s-expression> is
expected to be visible on the current active screen, and a
graphics input device (for example, a mouse) is required.
Clicking the selection button on a node results in the
additional screens being filled with the information
described above. This process may be repeated any number of
times. Warnings result from selecting leaf nodes. Clicking
the exit button will stop the display process and return the
list described above for the last node selected. See
<s-expression>split.screen</s-expression> for specific
details on graphic input and split-screen mode.
</s-section>
```

CLASSES

This section lists the classes the function is applicable to, if it is a default method. Like the graphical interaction section, the classes section in Spotfire S+ SGML files does not have a specific tag. Instead, the tags `<s-section name="name">` and `</s-section>` are used. For example, the classes section in the gamma help file is:

```
<s-section name="CLASSES">
This function will be used as the default method for classes
that do not inherit a specific method for the function or
for the <tt>Math</tt> group of functions. The result will
retain the class and the attributes. If this behavior is
<em>not</em> appropriate, the designer of the class should
provide a method for the function or for the <tt>Math</tt>
group
</s-section>
```

WARNING

Anything the user should be warned about when using the function should be described here. The warning section in Spotfire S+ SGML files does not have a specific tag, but uses `<s-section name="name">` and `</s-section>` instead. The following is the warning section from the gamma help file:

```
<s-section name="WARNING">
<s-expression>gamma(x)</s-expression> increases very
rapidly with <s-expression>x</s-expression>. Use
<s-expression>lgamma</s-expression> to avoid overflow.
</s-section>
```

DETAILS

Algorithmic details and implementation issues are described in this section. For example, the details section of the density help file explains the smoothing algorithm implemented in the function:

```
<s-details>
. . .
These are kernel estimates. For each
<s-expression>x</s-expression> value in the output, the
window is centered on that <s-expression>x</s-expression>
and the heights of the window at each datapoint are summed.
This sum, after a normalization, is the corresponding
<s-expression>y</s-expression> value in the output. Results
are currently computed to single-precision accuracy only.
</s-details>
```

Details that apply to only one argument should be part of the affected argument's description, and not part of the details section.

BACKGROUND

Background information on the function or method is described in this section. The text here should be aimed at those with complete ignorance on the subject. The background section in Spotfire S+ SGML files does not have a specific tag, but uses

`<s-section name="name">` and `</s-section>` instead. For example, the background section in the `hclust` help file provides general information on clustering algorithms:

```
<s-section name="BACKGROUND">
Cluster analysis divides datapoints into groups of points
that are "close" to each other. The
<s-expression>hclust</s-expression> function continues to
aggregate groups together until there is just one big
group. If it is necessary to choose the number of groups,
this can be decided subsequently. Other methods (see
<s-expression>kmeans</s-expression>) require that the
number of groups be decided from the start.
<p>
By changing the distance metric and the clustering method,
several different cluster trees can be created from a
single dataset. No one method seems to be useful in all
situations. Single linkage
(<s-expression>"connected"</s-expression>) can work poorly
if two distinct groups have a few "stragglers" between
them.
</s-section>
```

NOTE

Anything that does not fit into one of the above categories can be described in this section. The note section in Spotfire S+ SGML files does not have a specific tag, but uses `<s-section name="name">` and `</s-section>` instead. The following is a note from the `gamma` help file:

```
<s-section name="NOTE">
See <s-expression>family</s-expression> for the family
generating function <s-expression>Gamma</s-expression> used
with the <s-expression>glm</s-expression> and
<s-expression>gam</s-expression> functions. See
<s-expression>GAMMA</s-expression> for the functions
```

```
related to the gamma distribution:  
<s-expression>dgamma</s-expression> (density),  
<s-expression>pgamma</s-expression> (probability),  
<s-expression>qgamma</s-expression> (quantile),  
<s-expression>rgamma</s-expression> (sample).  
</s-section>
```

REFERENCES

References for functions are listed alphabetically and should follow *The Chicago Manual of Style*. The format for a book reference is similar to:

Venables, W.N. and Ripley, B.D. (1999). *Modern Applied Statistics with S-PLUS* (3rd ed.). New York: Springer-Verlag, Inc.

The format for a journal article is similar to:

Andersen, P. and Gill, R. (1982). Cox's regression model for counting processes, a large sample study. *Annals of Statistics* **10**: 1100-1120.

The references section in Spotfire S+ SGML files does not have a specific tag, but uses `<s-section name="name">` and `</s-section>` instead. The following is an excerpt from the references in the SGML help file for density, which cites a number of different works:

```
<s-section name="REFERENCES">  
Silverman, B.W. (1986).  
<it>Density Estimation for Statistics and Data  
Analysis.</it> London: Chapman and Hall.  
<p>  
Wegman, E.J. (1972).  
Nonparametric probability density estimation.  
<it>Technometrics</it> <bf>14</bf>: 533-546.  
<p>  
Venables, W.N. and Ripley, B.D. (1997)  
<it>Modern Applied Statistics with S-PLUS</it> (2nd ed.).  
New York: Springer-Verlag.  
</s-section>
```

The tag for paragraph breaks `<p>` should be used to separate multiple references. Please try to cite works that are widely available to users.

BUGS

In this section, you should document any known bugs a user might encounter while using the function. The bugs section in Spotfire S+ SGML files does not have a specific tag, but uses `<s-section name="name">` and `</s-section>` instead. For example, the following is a bug from the SGML help file for `subplot`:

```
<s-section name="BUGS">
  If you request it to draw outside the figure region,
  <s-expression>subplot</s-expression> gets very confused
  and typically puts the subplot across the whole region.
</s-section>
```

SEE ALSO

This section provides links to related functions. In general, any function you reference in the text of a help file should be linked in this section. The name of each function is wrapped in the tags `<s-function name="filename">` and `</s-function>`; this provides the hyperlinks in the formatted help file. The *filename* is the name of the installed help file, which is the linked function followed by `.sgm`. For example, the links in the SGML file for `subplot` are:

```
<s-see>
<s-function name="symbols.sgm">symbols</s-function>,
<s-function name="locator.sgm">locator</s-function>,
<s-function name="par.sgm">par</s-function>.
</s-see>
```

Functions that rely on the self-doc mechanism for their help files cannot be linked in this section.

Although newlines are not recognized in the see also section of Spotfire S+ SGML files, spaces are. Thus, be sure to include spaces between each link, even if you type them on separate lines. In the SGML file for `subplot`, two spaces are included at the end of each line, immediately after the `</s-function>` tags.

EXAMPLES

The examples in this section should help the user understand the function better. The goal is to provide the user with clear examples that are easily copied and run, either from the commands window or from a script window. Therefore, do not include the Spotfire S+ prompt character `>` in your examples and comment any output you include. So that the examples are self-contained, use built-in data sets

or create simple data sets in the code. For clarity, do not abbreviate argument names in your code, and be sure to test your examples before including them in a help file.

The following is an excerpt from the examples section of the `coxph` help file:

```
<s-examples>
<s-example type=text>
# Create the simplest test data set
test1 &lt;- list(time=c(4,3,1,1,2,2,3),
               status=c(1,1,1,0,1,1,0),
               x=c(0,2,1,1,1,0,0),
               sex=c(0,0,0,0,1,1,1))
# Fit a stratified model
coxph(Surv(time, status) ~ x + strata(sex), test1)
. . .
</s-example>
</s-examples>
```

The tag `<s-example type=text>` preformats the examples section so that it prints in a fixed-width font. This also causes newlines and blank lines to be recognized in the text. Thus, you can include spaces between different examples to enhance readability. So that the examples display nicely in conveniently-sized help windows, please ensure that the width of each line is no more than 60 characters.

In your examples, always use the left assignment operator `<-` instead of the underscore `_` for assignments. Because the angle bracket `<` signifies a tag in SGML, it is safest to type the left assignment operator using `<-` instead. Thus, the operator is `<-` in SGML.

Keywords

All help files should have keywords listed at the bottom, immediately before the closing tag `</s-function-doc>`. The help system uses keywords to organize the engine functions in the **Contents** tab of the help window; open the **Language Reference** from the Spotfire S+ menu and click on the **Contents** tab to see this.

Each keyword should be wrapped in the tags `<s-keyword>` and `</s-keyword>`. For example, the keywords section in the SGML file for `coxph` is:

```
<s-keywords>
<s-keyword>models</s-keyword>
<s-keyword>regression</s-keyword>
<s-keyword>survival4</s-keyword>
</s-keywords>
```

This places the `coxph` function in the following **Contents** topics: Statistical Models, Regression, and Survival Analysis. Table 11.4 lists the current keywords and the help topics they map to.

Table 11.4: Current keywords and the *Contents* topics they map to.

Keyword	Topic	Keyword	Topic	Keyword	Topic
<code>aplot</code>	Add to Existing Plot	<code>design</code>	ANOVA Models	<code>bootstrap</code>	Bootstrap Methods
<code>category</code>	Categorical Data	<code>character</code>	Character Data Operations	<code>cluster</code>	Clustering
<code>complex</code>	Complex Numbers	<code>dplot</code>	Computations Related to Plotting	<code>menudata</code>	Data Menu Functions
<code>wdialogs</code>	Customizable Dialogs (pre-S-PLUS 4.0)	<code>wmenus</code>	Customizable Menus (pre-S-PLUS 4.0)	<code>attribute</code>	Data Attributes
<code>data</code>	Data Directories	<code>manip</code>	Data Manipulation	<code>sysdata</code>	Data Sets
<code>classes</code>	Data Types	<code>chron</code>	Dates Objects	<code>debugging</code>	Debugging Tools
<code>defunct</code>	Defunct Functions	<code>deprecated</code>	Deprecated Functions	<code>documentation</code>	Documentation
<code>dynamic</code>	Dynamic Graphics	<code>error</code>	Error Handling	<code>device</code>	Graphical Devices

Table 11.4: Current keywords and the *Contents* topics they map to. (Continued)

Keyword	Topic	Keyword	Topic	Keyword	Topic
hplot	High-Level Plots	file	Input/Output Files	iplot	Interacting With Plots
interface	Interfaces to Other Languages	jackknife	Jackknife Methods	libchron	Library of Chronological Functions
libcluster	Library of Clustering Methods	libmaps	Library of Maps	algebra	Linear Algebra
list	Lists	loess	Loess Objects	logic	Logical Operators
iteration	Looping and Iteration	math	Mathematical Operations	array	Matrices and Arrays
methods	Methods and Generic Functions	misc	Miscellaneous	missing	Missing Values
mixed	Mixed Effects Models (version 2)	nlme3	Mixed Effects Models (version 3)	multivariate	Multivariate Techniques
nonlinear	Nonlinear Regression	nonparametric	Nonparametric Statistics	optimize	Optimization
ode	Ordinary Differential Equations	print	Printing	distribution	Probability Distributions and Random Numbers
programming	Programming	qc	Quality Control	regression	Regression
tree	Classification and Regression Trees	release	Release Notes	resample	Resampling (Bootstrap, Jackknife, and Permutations)

Table 11.4: Current keywords and the **Contents** topics they map to. (Continued)

Keyword	Topic	Keyword	Topic	Keyword	Topic
robust	Robust/ Resistant Techniques	environment	Spotfire S+ Session Environment	smooth	Smoothing Operations
hstest	Statistical Inference	menustat	Statistics Menu Functions	models	Statistical Models
survival4	Survival Analysis	ts	Time Series	trellis	Trellis Displays Library
guifun	User Interface Programming	utilities	Utilities	DOX	Design of Experiments, Response Surfaces, and Robust Design
fracfac	Fractional Factorial Experiments	rsm	Response Surfaces	taguchi	Robust Experimental Design
modgarch	GARCH Module for Modeling Time Series Volatility	geostat	Geostatistical Data Analysis	hexbin	Hexagonal Binning
lattice	Lattice Data Analysis	pointpattern	Point Pattern Analysis	spatialreg	Spatial Regression
spatialstats	Spatial Statistics Module	WAVELETS	Wavelet Analysis of Data, Signals, and Images	swt	Discrete Wavelet Transform Analysis
transform1d	1-D Wavelet and Cosine Transforms	transform2d	2-D Wavelet and Cosine Transforms	conv	Wavelet Convolutions and Filters
cpt	Cosine Packet Analysis	wpt	Wavelet Packet Analysis	crystal	Wavelet Crystals

Table 11.4: Current keywords and the *Contents* topics they map to. (Continued)

Keyword	Topic	Keyword	Topic	Keyword	Topic
molecule	Wavelet Molecules and Atoms	wavemake	Creating Wavelets, Wavelet Packets, and Cosine Packets	wavelets	Wavelets Module Functions
wavedata	Wavelets Module Signals, Images, and Datasets				

Special Help Files

Some Spotfire S+ help files do not fit into the general format described above, and instead require special fields. The two most common types of special help files are those for class objects and data sets. In this section, we briefly list the fields in these types of help files. We do not discuss each field in detail, but refer you to specific help files for more information and example SGML code. As with help files for functions, you can use prompt to create template SGML files and delete the tags that are not applicable to your objects.

SV4 Class Objects The SGML sections in an SV4 `class.type` help file are listed in Table 11.5. For more details, see the SGML files for `class.timeSeries`, `class.vector`, and `class.matrix`.

Table 11.5: Titled sections in Spotfire S+ help files for SV4 class objects.

Section Title	Quick Description	SGML Tags
Topic	The name of the object.	<s-topics> </s-topics>
Title	The title that appears at the top of a formatted help file.	<s-title> </s-title>
DESCRIPTION	A short description of the object.	<s-description> </s-description>

Table 11.5: Titled sections in Spotfire S+ help files for SV4 class objects. (Continued)

Section Title	Quick Description	SGML Tags
CLASS SLOTS	A list of descriptions for the slots in the object. Each slot can be formatted with the list tags <s-class-slot name=> and </s-class-slot>.	<s-slots> </s-slots>
EXTENDS	A list of classes the object extends. Each class is formatted with <s-contains-class name=> and </s-contains-class>.	<s-contains> </s-contains>
DETAILS	Descriptions of implementation issues.	<s-details> </s-details>
NOTE	Any information that does not fit into the above categories.	<s-section name=> </s-section>
REFERENCES	Available texts and papers the user can refer to for additional information.	<s-section name=> </s-section>
SEE ALSO	Links to related Spotfire S+ functions. The function that creates the object should be included in the links.	<s-see> </s-see>
Keywords	A list of keywords that place the help file in the Contents topics of the help system.	<s-keywords> </s-keywords>

Data Sets The SGML sections in help files for data sets are listed in Table 11.6. For more details, see the SGML files for solder and kyphosis.

Table 11.6: Titled sections in Spotfire S+ help files for data sets.

Section Title	Quick Description	SGML Tags
Topic	The name of the data object.	<s-topics> </s-topics>
Title	The title that appears at the top of a formatted help file. This should not be the name of the object itself.	<s-title> </s-title>

Table 11.6: Titled sections in Spotfire S+ help files for data sets. (Continued)

Section Title	Quick Description	SGML Tags
SUMMARY	A brief description of the experiment that produced the data. The name of the object should be included in this section.	<s-section name=> </s-section>
DATA DESCRIPTION	A short description of each of the variables in the object.	<s-description> </s-description>
SOURCE	The original references for the data.	<s-section name=> </s-section>
WARNING	Anything the user should be warned about when using the data.	<s-section name=> </s-section>
NOTE	Any information that does not fit into the above categories.	<s-section name=> </s-section>
SEE ALSO	Links to related Spotfire S+ functions and data sets.	<s-see> </s-see>
EXAMPLES	Coded Spotfire S+ examples using the data.	<s-examples> </s-examples>
Keywords	A list of keywords that place the help file in the Contents topics of the help system.	<s-keywords> </s-keywords>

Introduction	478
Working With Locales	479
Setting a Locale	480
Changing a Locale	482
Changing the Collation Sequence	483
Using Extended Characters	484
In Variable Names	484
In PostScript Output	485
Importing, Exporting, and Displaying Numeric Data	486
Importing and Exporting Data	486
Displaying Data	489

INTRODUCTION

Spotfire S+ includes enhancements designed to improve global installation and use:

- New functions have been added that allow you to specify a locale to be used by Spotfire S+. Specifically, Spotfire S+ now supports French and German locales.
- Sorting and collating functions have been modified to support different locales.
- The use of 8-bit characters (Latin1/Western European character set) has been enabled. The 8-bit characters can now be used in variable names and will properly display in data.
- Functions that import, export, and display numeric data have been modified to support different locales.
- Spotfire S+ Setup has been improved to test for local settings and install localized DLLs as necessary.

In the sections that follow, we describe these enhancements in greater detail.

WORKING WITH LOCALES

The concept of a “locale” in Spotfire S+ is akin to that of **Regional Options** (or **Regional Settings**) in Windows[®]. A locale determines which characters are considered printable or alphabetic, the format for numerals, and the collating sequence of characters.

Note

The implementation of locales in Spotfire S+ does not cover date formats (date formats are handled through a separate mechanism) and messaging (all messages in Spotfire S+ are in the English language).

The default locale is C, the locale used by the C language (7-bit ASCII characters and U.S.-style numbers). The C locale conforms to the behavior of earlier versions of Spotfire S+, that is:

- No accented characters are displayed (backslash octal instead).
- The decimal marker is a period.
- The sort order is all uppercase followed by all lowercase.

In European-based locales, however, Spotfire S+ prints accented characters and uses a more natural sort order (“a”, then “A”, then “à” (“\340”), then “à” (“\300”). In continental-based locales, the decimal marker is a comma. In English-based (and French Canadian) locales, the decimal marker is a period.

Not every locale is supported on every computer. To see which locales are supported on a computer running a version of Windows, look in **Control Panel ► Regional and Language Options**. On UNIX, run `locale -a`.

Hint

Choose a locale with “8859-1” in the name if there is a choice between this and “Roman8,” as Spotfire S+ assumes you are using the ISO 8859-1 character set. Your terminal emulator should be using an 8859-1 font. (This is an issue on HP/UX where the default seems to be “Roman8.”)

Setting a Locale

Setting a locale gives you a way to tell Spotfire S+ which character sets, number formats, and collation sequences to use. The primary function for setting a locale is the `Sys.setlocale()` function. This function allows you to set `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC`, or `LC_TIME`.

In the Commands Window

To use the default regional settings for locale, open the **Commands** window and type the following:

```
> Sys.setlocale(locale="")
```

This command tells Spotfire S+ to follow your Windows **Regional Options** (or **Regional Settings**).

Hint

If you put the desired `Sys.setlocale()` command into the `\local\S.init` file under your Spotfire S+ home directory, the command will be run automatically each time you launch Spotfire S+.

In the General Settings Dialog

In Windows, on the **Startup** page of the **Options ► General Settings** dialog, the fields for controlling regional settings have been replaced with a single check box called **Set Region-specific defaults**, as shown in Figure 12.1. (This option is not selected by default.)

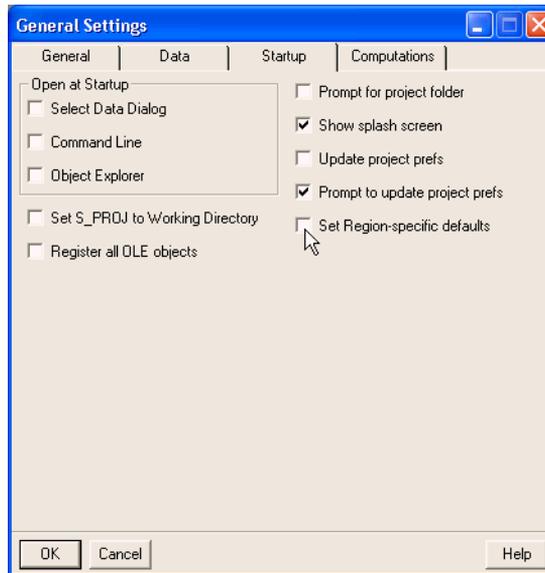


Figure 12.1: The **Startup** page of the **General Settings** dialog.

Selecting this check box sets Spotfire S+'s default locale according to your Windows **Regional Options** (or **Regional Settings**) in effect at the time of startup.

- If this check box is selected, the next time you start Spotfire S+ it sets the `time.in.format`, `time.out.format`, and `time.zone` options (shown in **.Options**) according to your Windows **Regional Options** (or **Regional Settings**). It also gets the currently selected Windows regional language setting and calls `Sys.setlocale(locale=language)` where `language` is the setting from Windows.

Note

When you select this option, you must restart Spotfire S+ for the change to take effect.

- If this check box is *not* selected, the `time.zone` option is set to the default of GMT, and `Sys.setlocale()` is set to the C locale.

Note

Selecting the **Set Region-specific defaults** check box also includes setting `LC_NUMERIC`, which causes numbers to print with a comma as the decimal marker in regions where that is the convention. This may break existing code that relies on the format of printed numbers.

If you select the **Set Region-specific defaults** option but want to use the period as the decimal marker instead, add the command `Sys.setlocale("LC_NUMERIC","C")` to the **\local\S.init** file under your Spotfire S+ home directory.

In UNIX, To use the default regional settings for locale, type the following:

```
> Sys.setlocale(locale="")
```

This command tells Spotfire S+ to follow the UNIX environment variables (`LANG`, `LC_{CTYPE,COLLATE,NUMERIC}`).

Hint

If you put the desired `Sys.setlocale()` command into your **S.init** file, the command will be run automatically each time you launch Spotfire S+.

Changing a Locale

Use the `Sys.setlocale()` and `Sys.getlocale()` functions to set and get locales, respectively. Use the `Sys.withlocale()` function if you want to evaluate an expression within a specified locale.

For example:

```
> Sys.getlocale()           # Default
[1] "C"
> Sys.setlocale(locale="en_US")
[1] "en_US"
> Sys.getlocale()
[1] "en_US"
```

Changing the Collation Sequence

“Collation sequence” refers to the ordering of characters by the `sort()` and `order()` functions. For example, in the `C` locale, uppercase (capital) letters collate before lowercase letters. In almost all other locales, however, collation sequence ignores case.

For example:

```
> Sys.setlocale(locale="C")
[1] "C"
> sort(c("as", "Axe"))
[1] "Axe" "as"
> Sys.setlocale(locale="English (United States)")
[1] "English_United_States.1252"
> sort(c("as", "Axe"))
[1] "as"  "Axe"
```

Changing `LC_COLLATE` affects the way Spotfire S+ interprets the ordering of character data and thus can be used to change the behavior of functions like `sort()` and `order()`.

USING EXTENDED CHARACTERS

Spotfire S+ accepts more “alphabetic” characters than in previous versions. This set of “alphabetic” characters is fixed and is the set of characters considered alphabetic in ISO 8859-1 (Latin1/Western European character set).

The set of characters that are displayed as octal codes changes depending on the current locale (specifically, `LC_CTYPE`). For example, in all locales that use Latin characters, “\341” is displayed as “á”. This affects how functions such as `format()`, `print()`, and `cat()` display character data.

In Variable Names

Because Spotfire S+ now supports Western European character sets (ISO 8859-1), you can include 8-bit ASCII characters (ASCII codes 128-255) within character data and within the names of S-PLUS objects. For example, you can include names such as Furtwängler and García Íñiguez in your character data and they will properly display. Spotfire S+ uses your locale settings to determine the appropriate character set to use in printing output.

The characters allowed in S-PLUS object names are:

```
alphanumerics <- c(".", "0", "1", "2", "3", "4", "5", "6",
  "7", "8", "9", "A", "B", "C", "D", "E", "F", "G", "H",
  "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S",
  "T", "U", "V", "W", "X", "Y", "Z", "a", "b", "c", "d",
  "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o",
  "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z",
  "\212", "\214", "\216", "\232", "\234", "\236", "\237",
  "\300", "\301", "\302", "\303", "\304", "\305", "\306",
  "\307", "\310", "\311", "\312", "\313", "\314", "\315",
  "\316", "\317", "\320", "\321", "\322", "\323", "\324",
  "\325", "\326", "\330", "\331", "\332", "\333", "\334",
  "\335", "\336", "\337", "\340", "\341", "\342", "\343",
  "\344", "\345", "\346", "\347", "\350", "\351", "\352",
  "\353", "\354", "\355", "\356", "\357", "\360", "\361",
  "\362", "\363", "\364", "\365", "\366", "\370", "\371",
  "\372", "\373", "\374", "\375", "\376", "\377")
```

The characters not allowed in names are:

```
non.alphanumerics <- c("\001", "\002", "\003", "\004",
  "\005", "\006", "\007", "\b", "\t", "\n", "\013", "\014",
  "\r", "\016", "\017", "\020", "\021", "\022", "\023",
  "\024", "\025", "\026", "\027", "\030", "\031", "\032",
  "\033", "\034", "\035", "\036", "\037", " ", "!", "\'",
  "#", "$", "%", "&", "'", "(", ")", "*", "+", ",", "-",
  "/", ":", ";", "<", "=", ">", "?", "@", "[", "\\", "]",
  "^", "_", "`", "{", "|", "}", "~", "\177", "\200",
  "\201", "\202", "\203", "\204", "\205", "\206", "\207",
  "\210", "\211", "\213", "\215", "\217", "\220", "\221",
  "\222", "\223", "\224", "\225", "\226", "\227", "\230",
  "\231", "\233", "\235", "\240", "\241", "\242", "\243",
  "\244", "\245", "\246", "\247", "\250", "\251", "\252",
  "\253", "\254", "\255", "\256", "\257", "\260", "\261",
  "\262", "\263", "\264", "\265", "\266", "\267", "\270",
  "\271", "\272", "\273", "\274", "\275", "\276", "\277",
  "\327", "\367")
```

Note

The set of characters allowed or disallowed in object names is not affected by the current locale. The functions `deparse()`, `parse()`, `dump()`, `dput()`, `source()`, `data.dump()`, and `data.restore()` are also unaffected by the choice of locale.

In PostScript Output

The `postscript()` function now uses the Latin1 encoding of its standard fonts by default. This means you can more easily use non-English, Western European characters. If you previously used octal escapes like `"\267"` to get characters in the upper half of the standard PostScript encoding (`"\267"` was the bullet character), you must either change such code (`"\200"` is now the bullet) or use the arguments `setfont=ps.setfont.std` and `bullet=ps.bullet.std` in calls to `postscript()`. The Symbol and Zapf family of fonts are not changed. The Latin1 encoding is not quite the ISO 8859-1 standard, in that the bullet character was added at position `"\200"`.

IMPORTING, EXPORTING, AND DISPLAYING NUMERIC DATA

Spotfire S+ supports importing, exporting, and displaying numeric data written in regional notation. This means you can import, export, and display numbers written using decimal markers and thousands separators other than the period and comma, respectively. For example, you can specify a comma as your decimal marker and a period as your digit-grouping symbol, or you can use a period as the decimal marker and an apostrophe as your digit-grouping symbol.

This feature is supported for the functions `scan()`, `print()`, `read.table()`, and `write.table()`. It is also supported for tick labels in graphics and for display in the **Data** window (in Windows). Note, however, that it is *not* supported for use within S-PLUS expressions, within `xlab` and `ylab` in graphics, or by default within the `importData()` and `exportData()` functions.

Hint

The default values for `xlab` and `ylab` do not use the numeric locale information because the default values are made with `deparse()`. However, if you make up your labels with `as.character()` or `paste()`, the locale information *is* used.

Importing and Exporting Data

When importing and exporting data, you can either use the command-line functions or, in Windows or using the Java GUI in Linux[®] or Solaris[®], the import/export dialogs.

In the Commands Window

The functions `importData()` and `exportData()` now have two additional arguments for use in reading and writing ASCII numbers:

- The `decimal.point` argument controls the single character used to mark the decimal place in ASCII numbers. The default is the period (`.`).
- The `thousands.separator` argument controls the single character used as the digit-grouping symbol in ASCII numbers. The default is the comma (`,`).

You can also specify which locale to use within `importData()` and `exportData()` by setting the argument `use.locale` to `T` (the default is `use.locale=F`.)

In the Import/ Export Dialogs

In Windows, both the **Import From File** dialog and the **Export To File** dialog provide two new fields for specifying the decimal character and the digit-grouping symbol. (See Figure 12.2 and Figure 12.3.)

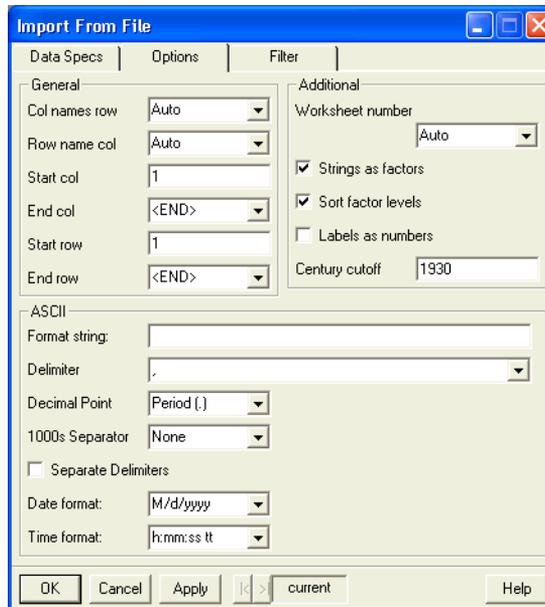


Figure 12.2: *The Options page of the Import From File dialog.*

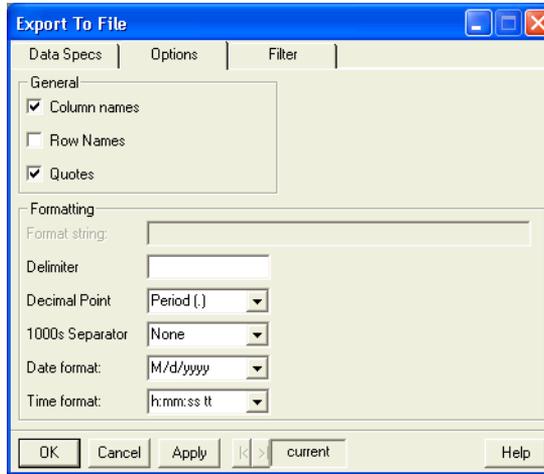


Figure 12.3: *The Options page of the Export To File dialog.*

On the **Options** page, make your selections as follows:

- **Decimal Point** Select **Comma (,)** or **Period (.)** from the drop-down list. The default is **Comma (,)**.
- **1000s Separator** Select **Comma (,)**, **None**, or **Period (.)** from the drop-down list. The default is **None**.

Note

These two fields are grayed out if the selected file format is not an ASCII file type. Note also that neither option is affected by the selection of delimiter or by any locale or regional settings.

Two additional changes have been made to the import/export dialogs:

- A **Semicolon (;)** selection has been added to the **Delimiter** field drop-down list.
- On the **Data Specs** page, the list of ASCII file formats has been updated to the following:
 - **ASCII file - whitespace delim (asc; dat; txt; prn)**
 - **ASCII file - whitespace delim (comma-decimal) (asc; dat; txt; prn)**
 - **ASCII file - comma delimited (csv)**

- **ASCII file - semicolon delimited (comma-decimal) (csv)**
- **ASCII file - user-defined delimiter (asc; dat; txt; prn)**

Note

ASCII file - delimited, **ASCII file - space delimited**, and **ASCII file - tab delimited** have been removed from the list. However, the function `guiImportData()` will still accept these strings as valid values for its `FileTypes` argument.

The **ASCII file - semicolon delimited (comma-decimal) (csv)** option sets **Delimiter** to **Semicolon (;)**, selects the **Separate Delimiters** check box, and sets **Decimal Point** to **Comma (,)**.

The **ASCII file - whitespace delim (comma-decimal) (asc; dat; txt; prn)** option behaves like the **ASCII file - whitespace delim (asc; dat; txt; prn)** option but also sets **Decimal Point** to **Comma (,)**.

Displaying Data

The decimal marker used when displaying or reading numbers changes depending on `LC_NUMERIC`. For example, in many European locales, the number pi is written 3,14159 (approximately). This affects how functions such as `format()`, `print()`, `cat()`, `write.table()`, and `html.table()` display numeric data. It also affects how `as.numeric()` and `as.character()` convert between strings and numbers. Other functions like `cut()` and `factor()` (in the labeling of levels) are also affected.

VERBOSE LOGGING

13

Overview	492
Logged Information	492
Verbose Batch Execution	494
Windows	494
UNIX	498
Details	500
Example	501
Input File	501
Output File	501
Log File	503

OVERVIEW

Batch processing support to run Spotfire S+ scripts non-interactively has been a key feature of Spotfire S+, and this feature has been available for a number of releases. Often this is performed to execute Spotfire S+ scripts as part of a production process.

For example, in the medical and pharmaceutical industries, it is vital that data management and statistical tasks be validated and documented. All steps involved in experimentation and validation must be documented and preserved in order to demonstrate to oversight bodies that the results are complete, accurate, valid, and reproducible. In addition to source files and generated output, log files are typically retained for these purposes.

The standard batch processing can record standard input, output, and errors to one or more files. This is the same output that the user would see executing the commands in the script at the command line.

In production use, it is valuable to include detailed information on specific tasks, such as when the commands were executed, what files were created, and the circumstances under which errors occurred. The information should be available in a text file that is readily understood and has tags to indicate the type of information in each line. This allows tools such as Perl to automatically extract information from the file.

The verbose logging function in Spotfire S+ provides this type of information regarding the execution of a Spotfire S+ script.

Logged Information

There are many features of the verbose log file:

- The log file has a structured format with the first seven columns reserved for line numbers and a line description tag. A colon in the eighth column separates these tags from the rest of the text; lines without colons in the eighth column are continuation of the previous tagged line. This makes parsing of the log file by Perl or other languages easy.
- There is a header with information about the user, the machine, the input and output files.
- If a file gets sourced in by the input file, it is logged (e.g., **S.init** is sourced in at the start of every Spotfire S+ session).

- Each database that gets attached is logged.
- Each line from the input file appears in the log file, identified by its line number.
- Each complete input expression is timed.
- A summary is logged when data is read in. The summary includes the disk location of the file, the resulting data frame size, the types of columns in the data frame, and missing value counts. Data frame reading and writing are also logged.
- Errors are logged but (by default) processing of the script continues.
- On completion of the script, a summary of the entire processing is written.

VERBOSE BATCH EXECUTION

Verbose batch execution can be invoked in a number of different ways, depending upon the platform you're using. We discuss the different methods by platform below.

Windows

In Windows[®], you can do batch execution using three unique sources:

- GUI, via the **BATCH** dialog
- Command Prompt, entering commands
- Spotfire S+ command line, with the **Target** field of a Spotfire S+ shortcut.

GUI

On Windows, the Spotfire S+ program group menu (from **Start ► All Programs ► TIBCO ► Spotfire S+ 8.2**) includes a menu item for the **BATCH** executable. This menu item launches the **BATCH**

processing dialog, which is used to specify the Spotfire S+ script to execute, and has options specific to batch operation. This dialog is shown in Figure 13.1.

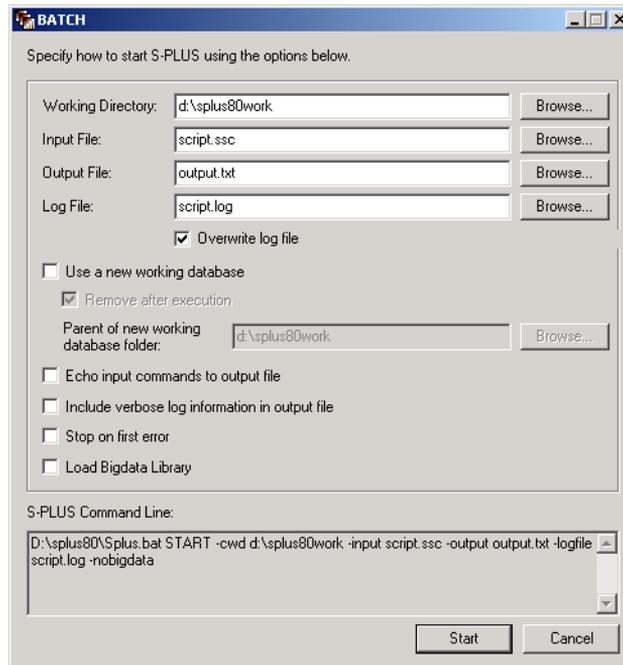


Figure 13.1: The *BATCH* dialog, which executes the batch script in Windows.

Working Directory The directory where Spotfire S+ reads and writes files. This directory typically contains a **.Data** directory with S-PLUS objects; see the description for **Use a new working database** for details.

Input File Contains the Spotfire S+ script commands to be run. If you enter just a relative file name, the **BATCH** utility looks for that file relative to the **Working Directory**.

Output File The name of the file that contains the output generated by the Spotfire S+ script. If you enter just a relative file name, the **BATCH** utility creates the file relative to the **Working Directory**.

Log File The name of the file that contains the log information generated by the Spotfire S+ script. If you enter just a relative file name, the **BATCH** utility creates the file relative to the **Working Directory**.

Overwrite log file Select this if you want to overwrite the contents of an existing log file.

Use a new working database Enables you to run the script under a newly created **.Data** directory. This directory is created under the directory specified in the **Parent of new working database folder** specified in the field below (typically the **Working Directory**), and is empty except a new `.Random.seed`.

Remove after execution Removes the new working database after the Spotfire S+ script is finished. This option is only available if you have checked the **Use a new working database** checkbox, and it does not remove any files if you are using the working database specified in the **Working Directory**.

Echo input commands to output file Combines the input commands with the output lines in the output file.

Include verbose logging information in output file Allows you to have the verbose log information mixed in with the script output. This is very useful for debugging and support.

Stop on first error Stop execution when the first error is found.

Load Big Data Library Load the bigdata library before executing the script.

Spotfire S+ Command Line Contains a copy of the command that executes the batch process when the **Start** button is pressed. The user can copy this command and paste it into a ***.bat** file or create a shortcut for use later.

The **Spotfire S+ Command Line** is the executable that runs the batch script and contains options, as in this example:

```
C:\Program Files\TIBCO\splus82\Splus.bat START
--work D:\spluswork
--input D:\SplusDaily\users\johnd\Script1.ssc
--output D:\SplusDaily\users\johnd\Script1.txt
--logfile D:\SplusDaily\users\johnd\Script1.log --echo --verbose
```

Note that when you enter filenames in the **Working Directory**, **Input File**, **Output File**, or **Log File** fields in the GUI, the content of the **Spotfire S+ Command Line** is updated with this information.

Command Prompt

To run a Windows batch file from a Command Prompt, the syntax is

```
> Splus SBATCH [args] inputfile
```

where [args] are those described in the section **Command Line in UNIX** (below), except that the `-j` and `-headless` options are not enabled on Windows.

Spotfire S+ Command Line

You can also run the batch file using the **Target** field in a Spotfire S+ command line, as shown in Figure 13.2.



Figure 13.2: Running a batch script in a Spotfire S+ command line.

You must supply a complete path to the **Splus.bat** script, including the **.bat** file extension. For example:

```
"C:\Program Files\TIBCO\splus82\Splus.bat" SBATCH
--echo
```

Note that you can use the **Start in** field to include the path to the file name, so the **Target** field only has to include **Splus.bat** and any flags.

UNIX

Command Line In UNIX, use `Splus SBATCH` to execute a script under verbose logging. This command has a number of optional arguments:

```
% Splus SBATCH --help
Syntax : SBATCH [args] inputfile
Optional flags are:
```

<code>--help</code>	Print this message and quit
<code>-j</code>	Enable java (not supported on Win32)
<code>--cwd directory</code>	Change directory to this directory before anything else
<code>--input inputfile</code>	Read input from inputfile (or specify inputfile as last argument)
<code>--output file</code>	Send output to file (default <basename inputfile>.txt)
<code>--logfile file</code>	Write session log to file (default <basename inputfile>.slg) By <basename inputfile> we mean inputfile stripped of possible period and extension
<code>--appendlog</code>	Append to log file (default is to overwrite)
<code>--nologfile</code>	Do not make a session log
<code>--work directory</code>	Use directory for working database (<code>where=1</code>), making it if needed. If work is not specified we will make new randomly named directory for the working database.

<code>--noclean</code>	Do not remove the new randomly named directory used for a working database
<code>--newworkparent</code>	Directory in which to make the new randomly named working database (value of <code>project directory</code> , whose default =.)
<code>--echo</code>	Echo command lines to output (sets <code>options(echo=T)</code>)
<code>--noecho</code>	Do not set <code>options(echo=T)</code>
<code>--verbose</code>	Echo verbose logging information to output (sets <code>options(verbose=T)</code>)
<code>--quitonerror</code>	Terminate immediately on engine error. (Default continues processing)
<code>--background</code>	Run Spotfire S+ in the background
<code>--vanilla</code>	Do not run user-defined startup script or functions (<code>-work dir</code> is ignored)
<code>--console</code>	Run only the console version, not the GUI (Windows only).
<code>--nobigdata</code>	Do not load the bigdata library when starting (the default action)
<code>--bigdata</code>	Load the bigdata library when starting
<code>--headless</code>	Do not use X Windows connection if running java (i.e., <code>DISPLAY</code> need not be set - Unix only)
<code>name=value</code>	Will be added to environment so <code>getenv("name")</code> returns <code>value</code> . Note that cannot have any spaces between <code>name</code> and <code>value</code> .

Details

On Windows and UNIX, setting

```
> options(verbose=T)
```

after starting Spotfire S+ or setting the environment variable `S_VERBOSE = yes` causes most of this information to be printed to the standard output, either the commands on a script window or the output file.

You may use the `logcat()` function to write more logging information. It does nothing if logging is not turned on, but prints one tagged line if it is turned on.

Note that `SBATCH` is designed for production use of verbose logging. To run batch files interactively on Windows or UNIX, use

```
Splus START
```

More details are available by typing `Splus START --help`.

EXAMPLE

The following is a short sample batch script, along with the output generated and the log file:

Input File

```
# Import bad file, has blank line at top:
ffDF <- importData("ffBad.txt", separateDelim=T)
# Import correct file
ffDF <- importData("ffGood.txt", separateDelim=T)
summary(ffDF)
# Incorrect model formula:
attach(fuel.frame)
reg1 <- lm(Fuel ~ Wt + Disp., data=ffDF, na.action=na.omit)
# Correct model formula:
reg1 <- lm(Fuel ~ Weight + Disp., data=ffDF,
na.action=na.omit)
reg1
# Write out a data set of results:
exportData(data.frame(fit=fitted(reg1), resid=resid(reg1)),
file="reg1.sas7bdat")
```

Output File

```
S-PLUS : Copyright (c) 1988, 2010 TIBCO Software Inc.
Version 8.2.0 for Microsoft Windows : 2010
Log file will be \\PUSHPULL\d\username\SP6\log.txt
Working data will be in d:\name\sp8\Sc000534.tmp
```

```
> # Import bad file, has blank line at top:
> ffDF <- importData("ffBad.txt", separateDelim=T)
> # Import correct file
> ffDF <- importData("ffGood.txt", separateDelim=T)
> summary(ffDF)
      Weight      Disp.      Mileage      Fuel
  Min.:1845    Min.: 73.0    Min.:18.0
  Min.:2.702703
  1st Qu.:2624  1st Qu.:114.5  1st Qu.:21.0  1st
  Qu.:3.703704
  Median:2903  Median:146.0  Median:23.0
  Median:4.347826
  Mean:2927    Mean:152.4    Mean:24.4
  Mean:4.226319
```

Chapter 13 Verbose Logging

```
3rd Qu.:3310 3rd Qu.:180.0 3rd Qu.:27.0 3rd
Qu.:4.761905
Max.:3855 Max.:305.0 Max.:37.0
Max.:5.555556 NA's: 2.0 NA's: 2.0
NA's:2.000000
```

```
Type
Compact:15
Large: 3
Medium:14
Small:13
Sporty:11
Van: 8
```

```
> # Incorrect model formula:
> reg1 <- lm(Fuel ~ Wt + Disp., data=ffDF,
na.action=na.omit)
Problem: Object "Wt" not found
Use traceback() to see the call stack
> # Correct model formula:
> reg1 <- lm(Fuel ~ Weight + Disp., data=ffDF,
na.action=na.omit)
```

```
Call:
lm(formula = Fuel ~ Weight + Disp., data = ffDF, na.action =
na.omit)
```

```
Coefficients:
(Intercept)      Weight      Disp.
  0.484787  0.001239589  0.0008637803
```

```
Degrees of freedom: 61 total; 58 residual
Dropped 3 cases due to missing values
Residual standard error: 0.3870086
```

```
> # Write out a data set of results:
> exportData(data.frame(fit=fitted(reg1),
resid=resid(reg1)),
+           file="reg1.sas7bdat")
[1] 60
```

Log File

START: Spotfire S+ Version 8.2 for WIN386 started at Fri
Oct 17 15:14:09
2010

NOTE: User=*name* Machine=PUSHPULL
Directory=\\PUSHPULL\d*name*\SP8

NOTE: Spotfire S+ is installed in directory
\\PUSHPULL\d\splus82

NOTE: Input file is \\HOMER*name*\ff.ssc

NOTE: Output file is \\PUSHPULL\d*name*\SP8\out.txt

PARSING: \\PUSHPULL\d\splus82\S.init

DATA: Attaching directory
\\PUSHPULL\d*name*\sp8\Sc000534.tmp\.Data in
position 1 with name d:*name*\sp8\Sc000534.tmp

DATA: Attaching directory
\\PUSHPULL\d\splus82\library\nlme3\.Data in
position 6 with name nlme3

DATA: Attaching directory
\\PUSHPULL\d\splus82\library\menu\.Data in
position 7 with name menu

DATA: Attaching directory
\\PUSHPULL\d\splus82\library\sgui\.Data in
position 8 with name sgui

TIME: Task #1 succeeded (Seconds = 1.531 CPU, 13.721
elapsed)

PARSING: \\HOMER*name*\FF.SSC

1 : # Import bad file, has blank line at top:
TIME: Task done (Seconds = 0 CPU, 0 elapsed)

2 : ffDF <- importData("ffBad.txt", separateDelim=T)
NOTE: Imported 62 by 5 data.frame with column names Col1,
Col2, Col3,
Col4, Col5

NOTE: Counts of column types: factor:5

NOTE: There are 5 missing values

NOTE: 1 rows have some missing values

DATA: Storing 62x5 data.frame 'ffDF' on database
d:*name*\sp8\Sc000534.tmp

TIME: Task done (Seconds = 0.484 CPU, 0.734 elapsed)

3 : # Import correct file

TIME: Task done (Seconds = 0 CPU, 0 elapsed)

```
4      : ffDF <- importData("ffGood.txt", separateDelim=T)
      NOTE: Imported 64 by 5 data.frame with column names
      Weight, Disp.,
      Mileage,
      Fuel, Type
      NOTE: Counts of column types: factor:1, numeric:4
      NOTE: There are 6 missing values
      NOTE: 4 rows have some missing values
      DATA: Storing 64x5 data.frame 'ffDF' on database
d:\name\sp8\Sc000534.tmp
      TIME: Task done (Seconds = 0.391 CPU, 0.625 elapsed)

5      : summary(ffDF)
      DATA: Reading 64x5 data.frame 'ffDF' on database
d:\name\sp8\Sc000534.tmp
      TIME: Task done (Seconds = 0.593 CPU, 0.656 elapsed)

6      : # Incorrect model formula:
      TIME: Task done (Seconds = 0 CPU, 0 elapsed)

7      : reg1 <- lm(Fuel ~ Wt + Disp., data=ffDF,
na.action=na.omit)
      DATA: Reading 64x5 data.frame 'ffDF' on database
d:\name\sp8\Sc000534.tmp
      ERROR: Problem: Object "Wt" not found
      TIME: Task done (Seconds = 0.157 CPU, 0.282 elapsed)

8      : # Correct model formula:
      TIME: Task done (Seconds = 0 CPU, 0 elapsed)

9      : reg1 <- lm(Fuel ~ Weight + Disp., data=ffDF,
na.action=na.omit)
      DATA: Reading 64x5 data.frame 'ffDF' on database
d:\name\sp8\Sc000534.tmp
      TIME: Task done (Seconds = 0.203 CPU, 0.265 elapsed)

10     : reg1
      TIME: Task done (Seconds = 0.078 CPU, 0.078 elapsed)

11     : # Write out a data set of results:
      TIME: Task done (Seconds = 0 CPU, 0 elapsed)
```

```
12      : exportData(data.frame(fit=fitted(reg1),  
resid=resid(reg1)),
```

```
13      : file="reg1.sas7bdat")
```

```
TIME: Task done (Seconds = 0.141 CPU, 0.203 elapsed)
```

```
NOTE: There were 1 errors and 0 warnings in this session
```

```
TIME: Session done (Seconds = 7.483 CPU, 36.109 elapsed)
```

```
QUIT: End session at Fri Nov 12 15:14:45 2010
```


XML Overview	508
XML and SPXML Library Overview	509
The SPXML Library	510
Reading and Writing XML Using the SPXML Library	511
Examples of XSL Transformations	512
Example 1: Creating a Vector	512
Example 2: Creating a Named Vector	514
Example 3: A List of Data Frames	516
Example 4: Import SAS XML as a Data Frame	519

XML OVERVIEW

Extensible Markup Language (XML) provides a mechanism for storing information and exchanging information between applications. Spotfire S+ includes two libraries of functions for working with XML: the SPXML library from TIBCO Software Inc., and the XML library from Bell Laboratories.

Using XML effectively involves more than just the XML code. An application using XML requires definitions for writing and reading XML. XML uses a *Document Type Definition* (DTD) file to define the elements and attributes allowed in an XML document. DTD files create specific markup languages that can be used for specific tasks. For example, *Predictive Modeling Markup Language* (PMML) has become an industry-standard for describing statistical models and serves as a way to exchange model information between applications. After the language is defined, an XML parser can read the XML text file and interpret the tags to determine its type of data. The user can then put the data into an internal data structure.

One of the largest benefits of XML is that it provides ways to exchange data between applications whose internal data representations are different.

- The *Extensible Stylesheet Language* (XSL) provides standards and tools describing how to create other documents from an XML file.
- *XSL transforms* (XSLT) can be used to create an HTML document, or to transform XML to another style of XML that uses different tags and organizes the data differently.
- *XSL formatting objects* (XSL-FO) describe how to create formatted documents such as PostScript, PDF, and RTF documents based on formatting specified in the XSL file and information specified in the XML file.

The next section describes the differences between the XML and SPXML libraries. The remainder of the document provides more details about the SPXML library and shows examples of using its functions for data exchange.

XML AND SPXML LIBRARY OVERVIEW

As a convenience for Spotfire S+ users, the Spotfire S+ installation includes a copy of the XML library. This library is written and maintained by Duncan Temple Lang of Bell Laboratories.

The SPXML library supports reading and writing XML by specifying a set of XML tags describing XML objects, and using C code that writes and parses these tags. There is no such thing as an XML object within S-PLUS, and Spotfire S+ does not manipulate XML. In this library, XSL manipulates XML.

The XML library takes a different approach: It introduces a set of S-PLUS classes representing XML objects such as `XMLNode` and `XMLComment`. A *Document Object Model* (DOM) parser is used to create a S-PLUS object from the information in an XML file. This object can be traversed using Spotfire S+ subscripting, manipulated using S-PLUS functions, and written back to a file if desired.

For more information about the XML library, see the description of its functions in the CRAN package, located at

<http://cran.r-project.org/web/packages/XML/XML.pdf>

Determining which library is most appropriate to use depends on the purpose and user preferences:

- The SPXML library is more efficient for serialization, because it uses a SAX parser and does computations almost exclusively in C code.
- When translating between S-PLUS objects and XML from other applications, the SPXML library does the transformations using XSL. This is preferable for users familiar with XSL.
- When translating between S-PLUS objects and XML from other applications, the XML library builds a S-PLUS object representing the XML document. S-PLUS functions are used to create a standard S-PLUS object such as a data frame from the XML document object. Spotfire S+ programmers might prefer to use S-PLUS functions rather than XSL.

The rest of this chapter focuses on the SPXML library. Consult the URL above for more information about the XML library.

THE SPXML LIBRARY

The SPXML library contains a small number of powerful functions:

- To write S-PLUS objects to a file as XML, use `createXMLFile`.
- To read S-PLUS objects from a file, use `parseXMLFile`.
- To transform XML files with XSLT or XSL-FO, use `javaXMLTransform`.

These functions are all you need to create a file with an XML description of an S-PLUS object and generate a report from the information in this XML file. The formatting information is specified in an XSL file.

The SPXML library keeps the S-PLUS functions simple and focused on XSL as a transformation mechanism because XSL is an industry standard with a wealth of documentation on its usage. The existing documentation provides extensive reference materials that are not included as part of the Spotfire S+ documentation set.

In addition to the functions listed above, the SPXML library provides some useful utility functions:

- The functions `xml2html`, `xml2pdf`, `xml2ps`, `xml2rtf`, and `xml2xml` are wrappers for `javaXMLTransform` that specify the corresponding output file type.
- The `createXMLString` function returns the generated XML as a string rather than writing it to a file.
- The functions `parseXMLPathFile` and `parseXMLPathString` parse the entire contents of an XML file or the specified contents of a file (respectively) and return them as character vectors. For example, using these functions, you can read `<a b c>` as "a", "b", "c", rather than "a b c".
- The `summaryReport` function implements a particular kind of summary report. This function, along with the corresponding XSL files, are good examples of how to combine Spotfire S+ and XSL to create a sophisticated report.

This document focuses on using XML as a tool for data exchange. The `summaryReport` function and report generation in general are discussed in the separate technical paper titled *XML Reporting*.

READING AND WRITING XML USING THE SPXML LIBRARY

The `createXMLFile` and `parseXMLFile` functions provide a simple mechanism for S-PLUS objects to be saved and restored using XML.

In the simplest case, you can save any S-PLUS object to a readable file and access it at a future time.

The following shows a simple example:

```
> library(SPXML)
> xmlFile <- "output.xml"
> orig.list <- list(fuel.frame, c(1:50), hist)
> createXMLFile(orig.list, xmlFile)
> new.list <- parseXMLFile(xmlFile)
> all.equal(orig.list, new.list)
[1] T
```

The XML tags for S-PLUS objects are described in the **SPLUS_1_0.dtd** file in the **library/SPXML/xml** directory of your Spotfire S+ installation.

The objective in defining these tags is to provide a means to read and write all S-PLUS objects (data frames, matrices, vectors, and multidimensional arrays). The XML descriptions of these objects are rectangular and hyper-rectangular data structures, which can be easily interpreted.

EXAMPLES OF XSL TRANSFORMATIONS

To import XML from another application into Spotfire S+, you must transform the application's XML into XML describing S-PLUS objects. You can do this by using an XSL file. This section describes four examples to illustrate using XSL.

These examples start with PMML files generated by TIBCO Spotfire Miner, a data mining and statistical data analysis product designed to handle large amounts of data. Spotfire Miner stores information on fitted models in its own XML format and generates industry-standard PMML, describing the models as a way to exchange model information with other applications. Spotfire Miner also uses the Extension mechanism to include additional information about the models.

One type of output that Spotfire Miner computes for models is a column importance measure for each independent column used in the model. By reading this information into Spotfire S+, you can use S-PLUS functions to create plots and reports that are not available in Spotfire Miner.

Example 1: Creating a Vector

In this example, the PMML file describes a logistic regression model. The file is located in the **library/SPXML/examples/xml_generation** directory and is named **logRegPMML.xml**. The column importance measure used for logistic regression is the Wald statistic.

PMML

In the XML for the logistic regression model, the column importance information is stored in one place:

```
<Extension extender="Insightful"  
  name="X-IMML-GeneralRegressionModel-Importance">  
<X-IMML-GeneralRegressionModel-Importance count="29"  
  targetCategory="credit.card.owner">  
<X-IMML-GeneralRegressionModel-Effect name="mean.check.cash.withdr"  
  value="169.99539009465" df="1" Pr="0"/>  
<X-IMML-GeneralRegressionModel-Effect name="mean.check.debits"  
  value="165.004662158705" df="1" Pr="0"/>  
  ...
```

XSL

To create a vector of column importances in Spotfire S+, first design an XSL transformation file that extracts the desired information from the PMML file and writes it to a new XML file using Spotfire S+ XML tags.

The file `library/SPXML/examples/xml_generation/logReg_ColImp_Vec.xml` contains the following XSL to create a S-PLUS vector of column importance values:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" standalone="yes" indent="yes"/>
  <xsl:template match="PMML">
    <xsl:apply-templates select="GeneralRegressionModel/Extension/
      X-IMML-GeneralRegressionModel-Importance"/>
  </xsl:template>

  <xsl:template match="X-IMML-GeneralRegressionModel-Importance">
    <xsl:element name="S-PLUS">
      <xsl:element name="Vector">
        <xsl:attribute name="length"><xsl:value-of select="@count"/></xsl:attribute>
        <xsl:attribute name="type">numeric</xsl:attribute>
        <xsl:element name="Items">
          <xsl:for-each select="X-IMML-GeneralRegressionModel-Effect">
            <xsl:element name="Item">
              <xsl:value-of select="@value"/>
            </xsl:element>
          </xsl:for-each>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

S-PLUS Functions To create the S-PLUS vector, use `xml2xml` to transform the file, and then use `parseXMLFile` to create a S-PLUS object:

```
> library(SPXML)
> logRegPMMLFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation
+   /logRegPMML.xml", sep="")
> logRegXSLFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation
+   /logReg_ColImp_Vec.xml", sep="")
> splusVecXMLFile <- "Splus_ColImp_Vec.xml"
> xml2xml(logRegPMMLFile, splusVecXMLFile, logRegXSLFile)
> colImpVec <- parseXMLFile(splusVecXMLFile)
> colImpVec

[1] 169.995390095 165.004662159 109.405532767 97.18724276
[5] 67.254514683 31.503267187 20.055698417 14.51615874
[9] 9.359599672 8.369635041 7.632052511 4.176602535
...

```

Example 2: Creating a Named Vector

While it is useful to have the column importance values in Spotfire S+, it would be even more practical to include the column names. Using a more detailed XSL, create a named vector with the column names and importance values.

XSL

The `library/SPXML/examples/xml_generation/logReg_ColImp_NamedVec.xml` file contains XSL to create a named vector from the logistic regression PMML. The resulting S-PLUS object is represented in the XML as a `Generic` object with class named.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" standalone="yes" indent="yes"/>
  <xsl:template match="PMML">
    <xsl:apply-templates select="GeneralRegressionModel/Extension/
      X-IMML-GeneralRegressionModel-Importance"/>
  </xsl:template>

  <xsl:template match="X-IMML-GeneralRegressionModel-Importance">
    <xsl:element name="S-PLUS">
      <xsl:element name="Generic">
        <xsl:attribute name="class">named</xsl:attribute>
        <xsl:element name="Attrs">
          <xsl:attribute name="length">1</xsl:attribute>
          <xsl:element name="Attr">
            <xsl:attribute name="name">.Names</xsl:attribute>
            <xsl:element name="Vector">
              <xsl:attribute name="length">
                <xsl:value-of select="@count"/></xsl:attribute>
              <xsl:attribute name="type">character</xsl:attribute>
              <xsl:attribute name="name">.Names</xsl:attribute>
              <xsl:element name="Items">
                <xsl:for-each select=
                  "X-IMML-GeneralRegressionModel-Effect">
                  <xsl:element name="Item">
                    <xsl:value-of select="@name"/>
                  </xsl:element>
                </xsl:for-each>
              </xsl:element>
            </xsl:element>
          </xsl:element>
        </xsl:element>
      <xsl:element name="Vector">
        <xsl:attribute name="length">
          <xsl:value-of select="@count"/></xsl:attribute>
        <xsl:attribute name="type">numeric</xsl:attribute>
        <xsl:element name="Items">
          <xsl:for-each select=
            "X-IMML-GeneralRegressionModel-Effect">
            <xsl:element name="Item">
              <xsl:value-of select="@value"/>
            </xsl:element>
          </xsl:for-each>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

```
</xsl:element>  
</xsl:template>
```

S-PLUS Functions Use `xml2xml` to translate the XSL and `parseXMLFile` to create the object:

```
> logRegXSLFile2 <- paste(getenv("SHOME"),  
+   "/library/SPXML/examples/xml_generation  
+   /logReg_ColImp_NamedVec.xsl", sep="")  
> splusNamedVecXMLFile <- "Splus_ColImp_NamedVec.xml"  
> xml2xml(logRegPMMLFile, splusNamedVecXMLFile,  
+   logRegXSLFile2)  
> colImpNamedVec <- parseXMLFile(splusNamedVecXMLFile)  
> colImpNamedVec  
  
mean.check.cash.withdr mean.check.debits cust.age  
169.9954 165.0047 109.4055  
  
mean.saving.balance mean.amnt.atm.withdr  
97.18724 67.25451  
...
```

Example 3: A List of Data Frames

In the logistic regression PMML, the column importance is already computed for each column and is stored in a single place in the XML. This makes it easy to access and transform.

Spotfire Miner also has an *ensemble tree* model. In this model, multiple classification (or regression) trees are fit, and the results averaged together for prediction. This is similar to using multiple models in bagging or boosting.

With the ensemble tree, determine the column importance measure by taking the change in goodness-of-fit at each split, and attributing that change as being due to the independent column used for the split. These changes in goodness-of-fit are summed over all splits and all trees to get a single change in goodness-of-fit attributed to each independent column. This is used as the importance measure.

Unlike logistic regression, this column importance is not a quantity that is stored in the PMML. Instead, it is computed by Spotfire Miner from information stored at the split level of the XML tree description. For example, the `library/SPXML/examples/xml_generation/ensembleTreePMML.xml` file has the following:

```
<Node score="0" recordCount="6806">
  <Extension extender="Insightful" name="X-IMML-XTProps">
    <X-IMML-XTProps>
      <X-IMML-Property name="id" value="1"/>
      <X-IMML-Property name="group" value="0"/>
      <X-IMML-Property name="deviance"
        value="5750.79288765702"/>
      <X-IMML-Property name="risk" value="1020"/>
      <X-IMML-Property name="yprob"
        value="0.850132236262122 0.149867763737878"/>
      <X-IMML-Property name="improvement"
        value="435.586807320647"/>
    </X-IMML-XTProps>
  </Extension>
  <SimplePredicate field="mean.salary.deposits"
    operator="lessThan" value="4426.431109375"/>

```

In this example, create an S-PLUS list of data frame objects, with one data frame for each tree in the ensemble. The data frame has two columns:

- The name of the independent column used to split.
- The change in goodness-of-fit.

After you have this information in Spotfire S+, you could use other S-PLUS functions to create separate column importance vectors for each tree, a single column importance vector over all trees, or some other quantity.

XSL

The file `library/SPXML/examples/xml_generation/tree_Collmp_DFList.xml` contains the following XSL transform:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" standalone="yes" indent="yes"/>
  <xsl:template match="PMML">
    <xsl:element name="S-PLUS">
      <xsl:element name="List">
        <xsl:attribute name="length"><xsl:value-of select="count(../TreeModel)"/></xsl:attribute>
        <xsl:attribute name="named">F</xsl:attribute>
        <xsl:element name="Components">
          <xsl:apply-templates select="../TreeModel"/>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
  <xsl:template match="TreeModel">
    <xsl:param name="numSplits" select="count(../Node[count(../True)=0])"/>
    <xsl:element name="Component">
      <xsl:element name="DataFrame">
        <xsl:attribute name="numRows"><xsl:value-of select="$numSplits"/>
        </xsl:attribute>
        <xsl:attribute name="numCols">2</xsl:attribute>
        <xsl:element name="RowNames">
          <xsl:attribute name="length"><xsl:value-of select="$numSplits"/>
          </xsl:attribute>
          <xsl:element name="Items">
            <xsl:for-each select="../Node[count(../True)=0]">
              <xsl:element name="Item">
                <xsl:value-of select="position()"/>
              </xsl:element>
            </xsl:for-each>
          </xsl:element>
        <xsl:element name="Columns">
          <xsl:element name="Column">
            <xsl:attribute name="length"><xsl:value-of select="$numSplits"/></xsl:attribute>
            <xsl:attribute name="type">character</xsl:attribute>
            <xsl:attribute name="name">Split.Column</xsl:attribute>
            <xsl:element name="Items">
              <xsl:for-each select="../Node[count(../True)=0]">
                <xsl:element name="Item">
                  <xsl:if test="count(SimplePredicate) &gt; 0">
                    <xsl:value-of select="SimplePredicate[1]/@field"/>
                  </xsl:if>
                  <xsl:if test="count(CompoundPredicate) &gt; 0">
                    <xsl:value-of select="CompoundPredicate/SimplePredicate[1]/@field"/>
                  </xsl:if>
                </xsl:element>
              </xsl:for-each>
            </xsl:element>
          </xsl:element>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

```

</xsl:element>
<xsl:element name="Column">
  <xsl:attribute name="length"><xsl:value-of
    select="$numSplits"/></xsl:attribute>
  <xsl:attribute name="type">numeric</xsl:attribute>
  <xsl:attribute name="name">Change.in.Fit</xsl:attribute>
  <xsl:element name="Items">
    <xsl:for-each select="//Node[count(./True)=0]">
      <xsl:element name="Item">
        <xsl:for-each select="Extension/X-IMML-XTProps
          /X-IMML-Property[@name='improvement']">
          <xsl:value-of select="@value"/>
        </xsl:for-each>
      </xsl:element>
    </xsl:for-each>
  </xsl:element>
</xsl:element>
</xsl:element>
</xsl:element>
</xsl:element>
</xsl:element>
</xsl:template>
</xsl:stylesheet>

```

Note that you can create a complex S-PLUS object from a complex XML document in a little over a single page of XSL.

S-PLUS Functions Once the XSL file has been created, the following code can be run from Spotfire S+:

```

> ensembleTreePMMLFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation
+   /ensembleTreePMML.xml", sep="")
> treeXSLFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation
+   /tree_ColImp_DFList.xsl", sep="")
> splusDFListXMLFile <- "Splus_ColImp_DFList.xml"
> xml2xml(ensembleTreePMMLFile, splusDFListXMLFile,
+   treeXSLFile)
> colImpDFList <- parseXMLFile(splusDFListXMLFile)
> length(colImpDFList)
[1] 2
> colImpDFList[[1]][1:5,]

```

```

          Split.Column Change.in.Fit
1      mean.salary.deposits      435.58681
2 mean.num.check.cash.withdr      90.42567
3                      gender      110.75756
4      mean.saving.balance      46.77268
5      mean.amnt.transfers      16.38389

```

Example 4: Import SAS XML as a Data Frame

SAS can create data sets from XML. Here is an example of SAS-friendly XML code:

```
<?xml version="1.0" encoding="UTF-8"?>
<LIBRARY>
  <STUDENTS>
    <ID> 0755 </ID>
    <NAME> Brad Martin </NAME>
    <CITY> Huntsville </CITY>
    <STATE> Texas </STATE>
  </STUDENTS>
  <STUDENTS>
    <ID> 1522 </ID>
    <NAME> Michelle Harvell </NAME>
    <CITY> Houston </CITY>
    <STATE> Texas </STATE>
  </STUDENTS>
  <STUDENTS>
    <ID> 1523 </ID>
    <NAME> Terry Glynn </NAME>
    <CITY> Chicago </CITY>
    <STATE> Illinois </STATE>
  </STUDENTS>
  <PROFESSORS>
    <ID> 9122 </ID>
    <NAME> Sue Clayton </NAME>
    <TENURE> YES </TENURE>
    <CITY> Huntsville </CITY>
    <STATE> Texas </STATE>
  </PROFESSORS>
  <PROFESSORS>
    <ID> 9453 </ID>
    <NAME> Todd Cantrell </NAME>
    <TENURE> NO </TENURE>
    <CITY> Houston </CITY>
    <STATE> Texas </STATE>
  </PROFESSORS>
  <PROFESSORS>
    <ID> 9562 </ID>
    <NAME> Larry Anders </NAME>
    <TENURE> YES </TENURE>
    <CITY> Chicago </CITY>
    <STATE> Illinois </STATE>
  </PROFESSORS>
</LIBRARY>
```

In this example, SAS uses the second-level tags (STUDENTS and PROFESSORS) to denote individual data sets. Repeated second-level tags (note that STUDENTS is repeated three times) represent separate rows in the resulting data set. Among the repeated second-level tags, common children represent data set columns (ID, NAME, CITY and STATE in STUDENTS).

When this XML file is imported into SAS, it creates two data sets, STUDENTS and PROFESSORS, with several columns ID, NAME, TENURE (for PROFESSORS), CITY, and STATE.

XSL

To import this file into Spotfire S+, create an XSL file that creates XML versions of data frames (analogous to the SAS data sets). Here is an example of such an XSL file:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" standalone="yes" indent="yes"/>

  <xsl:template match=".">
    <xsl:variable name="root" select="." />
    <xsl:variable name="level1Children" select="/*" />
    <xsl:variable name="level1Child1" select="$level1Children[1]" />

    <xsl:variable name="children" select="$level1Child1/*" />
    <xsl:variable name="nUniqueNames">
      <xsl:call-template name="CountLevel2Nodes">
        <xsl:with-param name="level2" select="$children" />
      </xsl:call-template>
    </xsl:variable>
    <SPLUS>
      <xsl:choose>
        <xsl:when test="$nUniqueNames > 1">
          <xsl:element name="List">
            <xsl:attribute name="length">
              <xsl:value-of select="$nUniqueNames"/>
            </xsl:attribute>
            <xsl:attribute name="named">F</xsl:attribute>

            <Components>
              <xsl:call-template name="Level2Nodes">
                <xsl:with-param name="level2" select="$children"/>
                <xsl:with-param name="needList" select="1" />
              </xsl:call-template>
            </Components>
          </xsl:element>
        </xsl:when>
        <xsl:otherwise>
          <xsl:call-template name="Level2Nodes">
            <xsl:with-param name="level2" select="$children"/>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </SPLUS>
  </xsl:template>

  <xsl:template name="CountLevel2Nodes">
    <xsl:param name="level2" select=""/>
    <xsl:param name="rCount" select="1"/>

    <xsl:variable name="firstChild" select="$level2[1]" />
    <xsl:variable name="child1Name" select="name($firstChild)" />
    <xsl:variable name="same" select="$level2[name(.) = $child1Name]" />
    <xsl:variable name="other" select="$level2[name(.) != $child1Name]" />

    <xsl:choose>
      <xsl:when test="boolean($other)">
        <xsl:call-template name="CountLevel2Nodes">
          <xsl:with-param name="level2" select="$other"/>
          <xsl:with-param name="rCount" select="$rCount + 1" />
        </xsl:call-template>
      </xsl:when>
    </xsl:choose>
  </xsl:template>

```

```

        <xsl:otherwise>
            <xsl:value-of select="$rCount"/>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>

<xsl:template name="Level2Nodes">
    <xsl:param name="level2" select=""/>
    <xsl:param name="needList" select=""/>

    <xsl:variable name="firstChild" select="$level2[1]"/>
    <xsl:variable name="child1Name" select="name($firstChild)" />
    <xsl:variable name="same" select="$level2[name(.) = $child1Name]" />
    <xsl:variable name="other" select="$level2[name(.) != $child1Name]" />

    <xsl:if test="boolean($same)">
        <xsl:choose>
            <xsl:when test="boolean($needList)">
                <Component>
                    <xsl:call-template name="NodesOfOneName">
                        <xsl:with-param name="nodes" select="$same"/>
                    </xsl:call-template>
                </Component>
            </xsl:when>
            <xsl:otherwise>
                <xsl:call-template name="NodesOfOneName">
                    <xsl:with-param name="nodes" select="$same"/>
                </xsl:call-template>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:if>

    <xsl:if test="boolean($other)">
        <xsl:call-template name="Level2Nodes">
            <xsl:with-param name="level2" select="$other"/>
            <xsl:with-param name="needList" select="$needList"/>
        </xsl:call-template>
    </xsl:if>
</xsl:template>

<xsl:template name="NodesOfOneName">
    <xsl:param name="nodes" select=""/>

    <xsl:variable name="firstChild" select="$nodes[1]"/>
    <xsl:variable name="child1Name" select="name($firstChild)" />
    <xsl:variable name="firstChildChildren" select="$firstChild/*" />

    <xsl:variable name="nRows" select="count($nodes)"/>
    <xsl:variable name="nCols" select="count($firstChildChildren)" />

    <xsl:element name="DataFrame">
        <xsl:attribute name="numRows">
            <xsl:value-of select="$nRows"/>
        </xsl:attribute>
        <xsl:attribute name="numCols">
            <xsl:value-of select="$nCols"/>
        </xsl:attribute>

        <xsl:element name="RowNames">
            <xsl:attribute name="length">
                <xsl:value-of select="$nRows"/>
            </xsl:attribute>
            <Items>

```

```

        <xsl:for-each select="$nodes">
          <Item><xsl:value-of select="position()"/></Item>
        </xsl:for-each>
      </Items>
    </xsl:element>

    <Columns>
      <xsl:for-each select="$firstChildChildren">
        <xsl:variable name="cName" select="name(.)" />
        <xsl:element name="Column">
          <xsl:attribute name="length">
            <xsl:value-of select="$nRows"/>
          </xsl:attribute>
          <xsl:attribute name="type">character</xsl:attribute>
          <xsl:attribute name="name">
            <xsl:value-of select="$cName"/>
          </xsl:attribute>

          <Items>
            <xsl:for-each select="$nodes">
              <xsl:variable name="allColumns" select="./**"/>
              <xsl:variable name="thisColumn" select="$allColumns[name(.) = $cName]" />
              <Item>
                <xsl:choose>
                  <xsl:when test="not(boolean($thisColumn))">
                    <xsl:text></xsl:text>
                  </xsl:when>
                  <xsl:otherwise>
                    <xsl:value-of select="$thisColumn/text()"/>
                  </xsl:otherwise>
                </xsl:choose>
              </Item>
            </xsl:for-each>
          </Items>
        </xsl:element>
      </xsl:for-each>
    </Columns>
  </xsl:element>
</xsl:template>

</xsl:stylesheet>

```

S-PLUS Functions After you create this XSL file, run the following Spotfire S+ commands to import the SAS XML into Spotfire S+:

```

> sasFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation/ImportSAS.xml",
+   sep="")
> xlsFile <- paste(getenv("SHOME"),
+   "/library/SPXML/examples/xml_generation/ImportSAS.xls",
+   sep="")
> xmlFile <- "ImportSPLUS.xml"
> xml2xml(sasFile, xmlFile, xlsFile)
> parseXMLFile(xmlFile)
[[1]]:
      ID          NAME          CITY          STATE
1  0755      Brad Martin  Huntsville    Texas

```

Examples of XSL Transformations

```
2 1522 Michelle Harvell Houston Texas
3 1523 Terry Glynn Chicago Illinois
```

[[2]]:

```
      ID      NAME TENURE      CITY      STATE
1  9122  Sue Clayton  YES  Huntsville  Texas
2  9453  Todd Cantrell   NO   Houston    Texas
3  9562  Larry Anders   YES   Chicago    Illinois
```


Overview	526
What is XSL?	527
Custom Reports	529
Summary Reports	533
Modifying Colors	538
Changing the Fonts	540
Outputting Positive Values in Red	541
Example of Modified XSL	544
Character Substitutions	545

OVERVIEW

The SPXML library introduces a rich report generation system for creating custom reports in a wide array of document formats. In this chapter, use the Spotfire S+ data frame `fuel.frame` to create report examples that are output in Portable Document Format (PDF), HTML, PostScript, and Rich Text Format (RTF).

This reporting system uses the Extensible Stylesheet Language (XSL) to create PDF, HTML, PostScript, and RTF reports from an XML description of an S-PLUS object. This system provides a maximum degree of freedom for specifying what is included or excluded, and the manner of presentation. It also removes nearly all dependencies of a specific report format, so you can create a PDF report from a report created in RTF.

The SPXML library contains functions used to generate these reports. Two functions provide the XML reporting infrastructure:

- `createXMLFile` Writes S-PLUS objects to a file in XML.
- `javaXMLTransform` Transforms XML files with XSLT or XSL-FO.

These functions are all you need to create a file with an XML description of an S-PLUS object, and to generate a report from the information in this XML file. The formatting information is specified in an XSL file.

The S-PLUS functions are simple, focusing on XSL as a transformation mechanism. Because XSL is an industry standard with a wealth of documentation on its usage, its extensive reference materials are not included as part of the Spotfire S+ documentation set.

This chapter focuses on using XML as a reporting tool. For more information on XML for data exchange, see Chapter 14, XML Generation.

While the report generation tools described in Chapter 14 are useful for creating custom reports, the SPXML library also includes a `summaryReport` function that provides a specific set of summary statistics by groups. This function, combined with the corresponding XSL files, provides examples of how you can create sophisticated reports.

WHAT IS XSL?

XSL is a specification for transforming XML to other types of XML or to other markup formats. You can divide XSL into two components:

Table 15.1: *XSL Components*

XSL Component	Description
XSLT	Used for transforms. Converts an XML document from one series of element types to another. The resulting document is usually XML or HTML.
XSL-FO	<p>XSL formatting objects. Describes how to use a set of XML elements to create a formatted document with elements such as titles, sections, tables, and page breaks. You can use this XSL description to create PDF, PostScript, or RTF files.</p> <p>The translation from a conceptual tag, such as <code><fo:table-header></code>, to the appropriate PDF description is generated automatically, so you do not need to know about the markup characters that a specific file format uses.</p>

The tags used for XSL-FO are a superset of those used for XSLT. Typically, you have

- An XSL file describing the HTML generation.
- An XSL file containing additional formatting objects tags for the other report formats (PDF, PostScript, or RTF).

The Spotfire S+ directory *[SHOME]/library/SPXML/xml* includes the following XSL files:

Table 15.2: *library/SPXML/xml files*

File name	Description
SplusObjects.xsl	A simple transformation file that creates a formatted HTML version of the following S-PLUS objects: data frames, lists, arrays, vectors, matrices, functions, call objects, and named objects.
SplusObjects_FO.xsl	A simple transformation file that creates a formatted PDF, PostScript or RTF version of the following S-PLUS objects: data frames, lists, arrays, vectors, matrices, functions, call objects, and named objects.
ColumnReport.xsl	A detailed transformation file that creates an HTML report in conjunction with the S-PLUS function <code>summaryReport</code> .
ColumnReport_FO.xsl	A detailed transformation file that creates a PDF, PostScript or RTF report in conjunction with the S-PLUS function <code>summaryReport</code> .

CUSTOM REPORTS

To create a report from a S-PLUS object, first save the object as XML, and then transform the XML with XSL. For example, create simple reports of the data frame `fuel.frame`:

```
library(SFXML)
xsltFile <- paste(getenv("SHOME"),
  "/library/SFXML/xml/SplusObjects.xsl", sep="")
xmlFile <- "temp.xml"
splusObject <- fuel.frame
createXMLFile(splusObject, xmlFile)
javaXMLTransform(xmlFile, "fuelReport.htm", xsltFile)
```

Note

The following command yields the same results, because the default XSL file is **library/SFXML/xml/SplusObjects.xsl**.

```
> javaXMLTransform(xmlFile, "fuelReport.htm")
```

By default, the report is created in your working directory.

In this example, you use **SplusObjects.xsl** as the XSL transform. This is a generic transformation that you can use for data frames, lists, arrays, vectors, matrices, functions, call objects, and named objects. In the example above, you could substitute a list, array, vector, matrix, function, call object, or name for the data frame. For example, any of the following work:

```
splusObject <- hist
splusObject <- list(c(1:100), fuel.frame, hist)
splusObject <- c("A", "AA", "AAA", "AAAA")
```

To create a report in PDF, PostScript, and RTF, use **SplusObjects_FO.xsl**, as follows, respectively:

```
foFile <- paste(getenv("SHOME"),
  "/library/SFXML/xml/SplusObjects_FO.xsl", sep="")
javaXMLTransform(xmlFile, "fuelReport.pdf", foFile)
javaXMLTransform(xmlFile, "fuelReport.ps", foFile)
javaXMLTransform(xmlFile, "fuelReport.rtf", foFile)
```

Note

If you specify no XSL file, Spotfire S+ uses the file **library/SPXML/xml/SplusObjects_FO.xsl** by default for output file names with extensions **.pdf**, **.rtf** and **.ps**.

To customize the formatting of the report, copy the XSL files, and then modify them appropriately. Examples of modifying XSL are presented in the section Summary Reports.

While you specify the XSL file explicitly in these examples, you do not need to specify the XSL file if you want to use one of the standard XSL files. If you specify no XSL file, then Spotfire S+ uses **SplusObjects.xsl** for HTML or XML transforms, and **SplusObjects_FO.xsl** for other types of transforms (PDF, PostScript, or RTF).

Creating your own Reporting XSL File

You can create your own XSL file for reporting purposes. As a simple example, suppose you want to create an HTML table from a S-PLUS matrix.

1. Output the specified matrix to an XML file:

```
splusObject <- format(cor(state.x77))
xmlFile <- "corXMLFile.xml"
createXMLFile(splusObject, xmlFile)
```

2. Create an XSL file capable of transforming the XML into an HTML table. The following code is an example of such an XSL file. (You can find the entire file in **[SHOME]/library/SPXML/examples/xml_reporting/CorrelationMatrix.xsl**):

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="S-PLUS">
    <html>
      <!-- create title for html page -->
      <title>Correlation Matrix for state.x77</title>

      <body>
        <!-- create a title for the top of the page -->
        <h2>Correlation Matrix for:</h2>
        <!-- create html table version of matrix -->
        <xsl:call-template name="Matrix">
```

```

        <xsl:with-param name="element" select="./Matrix"/>
    </xsl:call-template>
</body>
</html>
</xsl:template>

<xsl:template name="Matrix">
    <xsl:param name="element" select=""/>

    <div>
        <!-- create table for display of data frame or of matrix -->
        <table cellspacing="1" cellpadding="5" border="1">
            <!-- create a row of column headers -->
            <tr>
                <th></th>
                <xsl:for-each select="$element/Columns/Column">
                    <th><xsl:value-of select="@name"/></th>
                </xsl:for-each>
            </tr>

            <!-- create an html row for each row of data in the matrix -->
            <xsl:for-each select="$element/RowNames/Items/Item">
                <xsl:variable name="rowNumber" select="position()"/>
                <tr>
                    <!-- first item is row header that contains row name -->
                    <th><xsl:value-of select="."/text()></th>

                    <!-- add actual data...watch for factors and missing values -->
                    <xsl:for-each select="./../Columns/Column">
                        <xsl:variable name="colType" select="./@type"/>
                        <xsl:variable name="value" select="./Items/Item[$rowNumber]/text()"/>
                        <xsl:variable name="levels" select="./Attrs/Attr[@name = '.Label']"/>

                        <td align="right">
                            <xsl:choose>
                                <!-- character value -->
                                <xsl:when test="$colType= 'character'">
                                    <xsl:value-of select="$value"/>
                                </xsl:when>
                                <!-- factor value -->
                                <xsl:when test="$colType= 'factor'">
                                    <xsl:value-of select="$levels/Vector/Items/Item[position() = $value]/text()"/>
                                </xsl:when>
                                <!-- non-factor value -->
                                <xsl:when test="count($levels) = 0">
                                    <xsl:value-of select="$value"/>
                                </xsl:when>
                                <!-- missing factor -->
                                <xsl:when test="$value = 'NA'">
                                    <xsl:text/>
                                </xsl:when>
                            </xsl:choose>
                        </td>
                    </xsl:for-each>
                </tr>
            </xsl:for-each>
        </table>
    </div>
</xsl:template>

</xsl:stylesheet>

```

- Convert the XML file into an HTML table using these commands:

```
xslFile <- paste(getenv("SHOME"),
  "/library/SPXML/examples/xml_reporting",
  "/CorrelationMatrix.xsl", sep="")
htmlFile <- "CorrelationTable.html"
xml2html(xmlFile, htmlFile, xslFile)
```

The HTML output file should look like Figure 15.1:

Correlation Matrix for:

```
createXMLFile(format(cor(state.x77)), "corXMLFile.xml")
```

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area
Population	1.00000000	0.20822756	0.10762237	-0.06805195	0.34364275	-0.09848975	-0.33215245	0.02254384
Income	0.20822756	1.00000000	-0.43707519	0.34025534	-0.23007761	0.61993232	0.22628218	0.36331544
Illiteracy	0.10762237	-0.43707519	1.00000000	-0.58847793	0.70297520	-0.65718861	-0.67194697	0.07726113
Life Exp	-0.06805195	0.34025534	-0.58847793	1.00000000	-0.78084575	0.58221620	0.26206801	-0.10733194
Murder	0.34364275	-0.23007761	0.70297520	-0.78084575	1.00000000	-0.48797102	-0.53888344	0.22839021
HS Grad	-0.09848975	0.61993232	-0.65718861	0.58221620	-0.48797102	1.00000000	0.36677970	0.33354187
Frost	-0.33215245	0.22628218	-0.67194697	0.26206801	-0.53888344	0.36677970	1.00000000	0.05922910
Area	0.02254384	0.36331544	0.07726113	-0.10733194	0.22839021	0.33354187	0.05922910	1.00000000

Figure 15.1: *CorrelationTable.html*, the generated HTML table.

SUMMARY REPORTS

The `summaryReport` function provides summary statistics by groups for data frame objects. Use the `args` function to list the arguments:

```
> args(summaryReport)
function(data, file, variables = c(), grouping.variables =
c(), mean = T, median = T, stdev = T, range = T, quartile =
T, st.error = T, missing = T, precision = 2, nbins = 5,
minLevels = 5, type = NULL, xslFile= NULL, title = "",
maxColumnsPerTable = 0, loggingLevel = 4)
```

These arguments are described in Table 15.3.

Table 15.3: *Arguments for summaryReport.*

Argument	Required/ Optional	Description
<code>data</code>	Required	Specifies data frame object or set of columns.
<code>file</code>	Required	Specifies generated report's output location.
<code>variables</code>	Optional	Specifies report columns. Use this option to target specific columns for report. If unspecified, the default is to include all non-grouping columns in data.
<code>grouping.variables</code>	Optional	Specifies grouping. Use to group or sort the report's statistics by certain variable values.
<code>mean</code>	Optional	Specifies numeric mean.
<code>median</code>	Optional	Specifies numeric median.
<code>stdev</code>	Optional	Specifies numeric standard deviation.

Table 15.3: Arguments for `summaryReport`. (Continued)

Argument	Required/ Optional	Description
<code>range</code>	Optional	Specifies numeric range.
<code>quartile</code>	Optional	Specifies numeric quartile.
<code>st.error</code>	Optional	Specifies numeric standard error.
<code>missing</code>	Optional	Specifies numeric missing count.
<code>precision</code>	Optional	Specifies output precision.
<code>nbins</code>	Optional	Specifies bin count for numeric columns in groups argument.
<code>minLevels</code>	Optional	Specifies minimum number of unique values in numeric grouping column.
<code>type</code>	Optional	Specifies format: “ xml ”, “ pdf ”, “ rtf ”, “ ps ”, or “ html ”. If not specified, the extension of the filename is used to determine the output format, if possible.
<code>xslFile</code>	Optional	Specifies user XSL file for report generation. The default XSL used is ColumnReport.xsl (for HTML) and ColumnReport_FO.xsl (for PDF, PostScript, and RTF).

Table 15.3: Arguments for `summaryReport`. (Continued)

Argument	Required/ Optional	Description
<code>title</code>	Optional	Specifies a title to appear at the top of the report.
<code>maxColumnsPerTable</code>	Optional	Specifies the maximum number of data columns per table. Controls the overall width of the report by breaking wide tables into several smaller tables.
<code>loggingLevel</code>	Optional	Specifies the desired level of logging during XSL transform.

To create a very simple report of a `data.frame` object:

```
reportFilename <- "fuel.report.html"
summaryReport(fuel.frame, reportFilename)
```

This creates a report that looks like Figure 15.2:

		Statistic
Weight	mean	2,900.83
	median	2,885.00
	stdev	495.87
	range	(1,845.00 , 3,855.00)
	quartile	(2,571.25 , 3,231.25)
	st.error	64.02
	missing	0 (0.00 %)
Disp.	mean	152.05
	median	144.50
	stdev	54.16
	range	(73.00 , 305.00)

Figure 15.2: *The output of `fuel.report.html`*

You can create a more complicated report. For example, run the following code to create a PDF report of the `fuel.frame` variable `Weight` grouped by `Type`:

```
summaryReport(data=fuel.frame,
  file="fuel.report.pdf",
  variables=c("Weight"),
  grouping.variables=c("Type"))
```

This code creates a report that looks like Figure 15.3:

Grouping Description
Number of Categories:
Type : 6 categories

Type	Value
Compact	15
Large	3
Medium	13
Small	13
Sporty	9
Van	7

Column Summary

Weight		Type			
		Compact	Large	Medium	Sr
	mean	2,821.00	3,676.67	3,195.77	2,257.00
	median	2,780.00	3,850.00	3,200.00	2,295.00
	stdev	168.92	304.56	264.01	203.00
	range	(2,575.00 , 3,110.00)	(3,325.00 , 3,855.00)	(2,765.00 , 3,610.00)	(1,845.00 , 2,580.00)
	quartile	(2,662.50 , 2,927.50)	(3,587.50 , 3,852.50)	(2,975.00 , 3,450.00)	(2,260.00 , 2,350.00)
	st.error	43.61	175.84	73.22	56.00
	missing	0 (0.00 %)	0 (0.00 %)	0 (0.00 %)	0 (0.00 %)

Figure 15.3: The PDF output in *fuel.report.pdf*.

The `summaryReport` function generates a summary of each column in the input. For greater detail, you can segment the output by grouping information supplied as an argument. The column summary depends on column type:

- **Categorical columns** Outputs a count and percentage for each category.
- **Numeric columns** Outputs mean, range, median, quartile, standard deviation, error, and missing (as appropriate).

The default behavior outputs all information possible, but you can eliminate each of the numeric items from the output by setting the appropriate argument to F (for example, eliminate range by setting the argument `range=F`).

`summaryReport` uses **ColumnReport.xml** (for HTML reports) and **ColumnReport_FO.xml** (for PDF, PostScript, and RTF reports) using XSL transforms. These files are located in the Spotfire S+ installation directory **[SHOME]/library/SPXML/xml**.

To modify the report, you can either modify these files or make copies of the files, and then specify the XSL file name when using `summaryReport`.

The following examples demonstrate modifying the XSL to customize reports, as follows:

- Change the table colors.
- Change the font used in the table cells.
- Use a different color for positive numbers in the table.

Modifying Colors

To modify the XSL that `summaryReport` uses

1. Copy the **ColumnReport.xsl** and **ColumnReport_FO.xsl** files in the Spotfire S+ installation directory `[SHOME]/library/SPXML/xml`.

Note

To follow this example in an already-modified XSL file, use **EditedColumnReport.xsl** (for HTML and XML files) and **EditedColumnReport_FO.xsl** (for PDF, PostScript, and RTF files), which are located in the directory `[SHOME]/library/SPXML/examples/xml_reporting`.

The beginning of the XSL file defines table color and font attribute sets. To change colors and fonts, just change the values in these definitions.

There are four (4) colors used to create the tables:

- **g_tableBG** The background color of the table.
- **g_headerBG** The background color of the table headers (row and column).
- **g_rowBG_1, g_rowBG_2** The alternating background colors (each row).

These colors are located beginning on line 5 of the XSL transform files. Their default values are listed as follows:

```
<xsl:variable name="g_tableBG" select="#82C0FF" />
<xsl:variable name="g_headerBG" select="#CCE6FF" />
<xsl:variable name="g_rowBG_1" select="#EEEEEE" />
<xsl:variable name="g_rowBG_2" select="#FFFFFF" />
```

- Run the `summaryReport` command. The default report resembles the output in Figure 15.4:

Column Summary

		V1	V2
variety	V1	12 (100.00 %)	0 (0.00 %)
	V2	0 (0.00 %)	12 (100.00 %)
	V3	0 (0.00 %)	0 (0.00 %)
	V4	0 (0.00 %)	0 (0.00 %)
	V5	0 (0.00 %)	0 (0.00 %)
	V6	0 (0.00 %)	0 (0.00 %)
	V7	0 (0.00 %)	0 (0.00 %)
	V8	0 (0.00 %)	0 (0.00 %)

Figure 15.4: *The default report HTML file.*

In this example, the table has a background color of blue, each header is a lighter variation of blue, and the rows alternate between white and light grey.

The values stored are standard hexcode versions of RGB.

- The first two characters represent the red component (for example, in `g_tableBG: 82`).
 - The second two characters represent the green component (for example, in `g_tableBG: C0`).
 - The last two characters represent the blue component (for example, in `g_tableBG: FF`).
- Create a simple S-PLUS function to convert RGB values into hexcode:

```
char2hex <- function(x) {
  if (x < 0 || x > 255) stop
    ("x must be between 0 and 255")
  a <- floor(x/16)
  b <- x-16*a
  if (a>9) a <- LETTERS[a-9]
  if (b>9) b <- LETTERS[b-9]
  paste(as.character(a), as.character(b), sep="")
}
```

4. To change these colors, adjust the RGB hex code to the desired color, or use the specified HTML color tags (such as **RED**, **BLUE**, **YELLOW**, **GREEN**, and so on). For example, change the lines in **ColumnReport.xml** to the following:

```
<xsl:variable name="g_tableBG" select="yellow"/>
<xsl:variable name="g_headerBG" select="silver"/>
<xsl:variable name="g_rowBG_1" select="#0099cc"/>
<xsl:variable name="g_rowBG_2" select="#00cccc"/>
```

The `summaryReport` command produces a report (Figure 15.5).

Column Summary

		V1	V2
treatment	T1	3 (25.00 %)	3 (25.00 %)
	T2	3 (25.00 %)	3 (25.00 %)
	T3	3 (25.00 %)	3 (25.00 %)
	T4	3 (25.00 %)	3 (25.00 %)
reps	1	4 (33.33 %)	4 (33.33 %)
	2	4 (33.33 %)	4 (33.33 %)
	3	4 (33.33 %)	4 (33.33 %)
plants	mean	24.67	26.83
	median	12.50	16.50
	stdev	25.27	23.96
	range	(6.00 , 70.00)	(4.00 , 77.00)

Figure 15.5: Customizing the table colors of the output HTML file.

Changing the Fonts

In the XSL, you can modify the font for several regions (lines 13-32 in **ColumnReport.xml** and lines 12-43 in **ColumnReport_FO.xml**):

- `title` Controls the major section title font.
- `subtitle` Controls the minor section title font.
- `bold-emph` Controls the minor section title font.
- `small-cap` Controls the minor section title font.
- `table-content` Controls all the table data font.
- `row-header` Controls the table row header font.

- `column-header` Controls the table column header font.

For example, to set the table data font to Courier in an HTML report, you need to add the line

```
<xsl:attribute name="face">Courier</xsl:attribute>
```

to **ColumnReport.xsl** within the `table-content` attribute-set that starts on line 21. Running the `summaryReport` command creates a report like Figure 15.6:

2	4 (33.33 %)	4 (33.33 %)
3	4 (33.33 %)	4 (33.33 %)
mean	24.67	26.83
median	12.50	16.50
stdev	25.27	23.96

Figure 15.6: *Changing the fonts in the HTML output table.*

To make a similar change to the PDF, PostScript, and RTF reports, add the line

```
<xsl:attribute name="font-family">Courier</xsl:attribute>
```

to **ColumnReport_FO.xls** within the `table-content` attribute-set that starts on line 37.

Note

To commit a format change to a PDF, PostScript, or RTF file that you specified in the XSL-FO file, you must restart your Spotfire S+ session.

Outputting Positive Values in Red

You can change the fonts and colors for the whole table by changing the definitions at the top of the XSL file. For example, you might be interested in more specific formatting, such as using a different color for the positive numeric values. You can specify this change by editing the XSL in the places where you format the table cell values.

Because the summary report is a sophisticated report, it has a complex XSL file. The numbers in the table cells are formatted in two ways:

- Numeric data, such as mean, median, or range.
- Categorical data, such as counts and percentages.

The two templates that output table data are **OutputCountAndPercent** (line 692) and **OutputNumericItem** (line 573). Modify them in both places:

1. Modify **OutputNumericItem** by replacing the following code:

```
<xsl:call-template name="roundNumber">
  <xsl:with-param name="string" select="./text()"/>
  <xsl:with-param name="decimalPattern" select="$decimalPattern"/>
  <xsl:with-param name="scientificPattern" select="$scientificPattern"/>
</xsl:call-template>
```

with this code:

```
<xsl:element name="font">
  <xsl:if test="./text() &gt; 0">
    <xsl:attribute name="color">RED</xsl:attribute>
  </xsl:if>

  <xsl:call-template name="roundNumber">
    <xsl:with-param name="string" select="./text()"/>
    <xsl:with-param name="decimalPattern" select="$decimalPattern"/>
    <xsl:with-param name="scientificPattern" select="$scientificPattern"/>
  </xsl:call-template>

</xsl:element>
```

2. Modify **OutputCountAndPercent**, as follows:

```
<xsl:template name="OutputCountAndPercent">
  <xsl:param name="count" select=""" />
  <xsl:param name="percent" select=""" />
  <xsl:param name="decimalPattern" select=""" />
  <xsl:param name="scientificPattern" select=""" />
  <font xsl:use-attribute-sets="table-content">
    <xsl:element name="font">
      <xsl:if test="$count &gt; 0">
        <xsl:attribute name="color">RED</xsl:attribute>
      </xsl:if>

      <xsl:value-of select="$count"/>
      <xsl:if test="boolean($percent)">
        (<xsl:call-template name="roundNumber">
          <xsl:with-param name="string" select="$percent"/>
          <xsl:with-param name="decimalPattern" select="$decimalPattern"/>
          <xsl:with-param name="scientificPattern" select="$scientificPattern"/>
        </xsl:call-template>%)
      </xsl:if>
    </xsl:element>
  </font>
</xsl:template>
```

- Using these modifications and running the `summaryReport` command produces a report resembling Figure 15.7:

Column Summary

		V1	V2
variety	V1	12 (100.00%)	0 (0.00%)
	V2	0 (0.00%)	12 (100.00%)
	V3	0 (0.00%)	0 (0.00%)
	V4	0 (0.00%)	0 (0.00%)
	V5	0 (0.00%)	0 (0.00%)
	V6	0 (0.00%)	0 (0.00%)
	V7	0 (0.00%)	0 (0.00%)
	V8	0 (0.00%)	0 (0.00%)
treatment	T1	3 (25.00%)	3 (25.00%)
	T2	3 (25.00%)	3 (25.00%)
	T3	3 (25.00%)	3 (25.00%)
	T4	3 (25.00%)	3 (25.00%)
reps	1	4 (33.33%)	4 (33.33%)
	2	4 (33.33%)	4 (33.33%)
	3	4 (33.33%)	4 (33.33%)
	mean	24.67	26.83
	median	12.50	16.50
	stdev	25.27	23.96

Figure 15.7: *Outputting positive values in red.*

To modify the PDF, PostScript, or RTF reports, in the XSL-FO file, you must similarly modify the same templates, **OutputCountAndPercent** (line 789) and **OutputNumericItem** (line 678). In this case, there are two differences between the changes described above and changes to be made to **ColumnReport_FO.xsl**:

- Instead of creating a font element, create an `fo:inline` element.
- XSL-FO does not recognize color names. Instead of `RED`, you must use the hexadecimal RGB code `#FF0000`.

Thus, the modified template **OutputCountAndPercent** looks like this:

```
<xsl:template name="OutputCountAndPercent">
  <xsl:param name="count" select="" />
  <xsl:param name="percent" select="" />
  <xsl:param name="decimalPattern" select="" />
  <xsl:param name="scientificPattern" select="" />

  <xsl:element name="fo:inline">
    <xsl:if test="$count > 0">
      <xsl:attribute name="color">#FF0000</xsl:attribute>
    </xsl:if>

    <xsl:value-of select="$count"/>
    <xsl:if test="boolean($percent)">
      (<xsl:call-template name="roundNumber">
        <xsl:with-param name="string" select="$percent"/>
        <xsl:with-param name="decimalPattern" select="$decimalPattern"/>
        <xsl:with-param name="scientificPattern"
          select="$scientificPattern"/>
        </xsl:call-template>%)
    </xsl:if>
  </xsl:element>
</xsl:template>
```

Example of Modified XSL

Spotfire S+ includes example files that demonstrate all the changes described above:

- Modified Table Colors
- Use of Courier Font
- Using Red for Positive Numbers

For HTML reports, Spotfire S+ includes **EditedColumnReport.xml** in the **library/SPXML/examples/xml_reporting** directory. For PDF, PostScript, and RTF reports, Spotfire S+ includes **EditedColumnReport_FO.xml**.

CHARACTER SUBSTITUTIONS

Notice that in the XSL code, “greater than” is represented by `>`; rather than the standard `>`, because certain characters are reserved by XSL and must be substituted. Table 15.4 lists reserved characters and their corresponding substitutions

Table 15.4: *Substitutions in XML for reserved characters.*

Character	XSL Substitution
Less than (<)	<code>&lt;</code>
Greater than (>)	<code>&gt;</code>
Equals (=)	<code>=</code>
Less than or equal	<code>&lt;=</code>
Greater than or equal	<code>&gt;=</code>

INDEX

Symbols

.Call interface 38, 41, 43, 44, 47, 52, 58
 .chm file 441, 442, 443, 444, 449, 377
 JavaField 70, 89
 JavaMethod 70, 71, 75, 89, 377

Numerics

1000s Separator field 488

A

Adding an ActiveX control to a dialog 308
 addMenuItem method 399
 application object
 methods for 214
 properties of 219
 arithmetic operators 54
 ASCII file formats 488
 Assign 39
 assignment operators 49, 50, 51
 attach function 25
 automation
 client examples 239
 creating unexposed objects 209
 definition of 198
 embedding Graph Sheets 223
 exposing functions 206
 exposing objects 205, 206, 209
 high-level functions for 230

HTML-based object help
 system 199, 207, 208, 210
 refreshing 208
 methods
 common 211
 for application object 214
 for function objects 217
 for graph objects 216
 passing data to functions 220
 properties 219
 reference counting 232
 server examples 235
 type library 199, 207
 refreshing 208
 removing 208
 automation objects
 common methods for 211
 registering 205
 unregistering 205
 automation server objects
 passing null parameters 227, 232

B

batch mode 4
 batch processing 18, 19
 BorderLayout 391
 build_helpchm.cyg 444, 446, 448, 449
 BUILD_JHELP 456
 BuildHelpFiles 442
 buttons

- modifying 268
- C**
- C++
 - See* CONNECT/C++
 - call.ole.method 232
 - specifying null parameters 227
 - callback function 327
 - catch block 47
 - cat function 29
 - CF_TEXT format 244
 - channel numbers 244
 - CHAPTER utility 20
 - characters
 - allowed in variable names 484
 - not allowed in variable names 485
 - character sets 478, 480, 484
 - ISO 8859-1 479, 484, 485
 - classes
 - CONNECT/C++ 36, 38, 42, 46, 49, 50, 53, 54, 56, 58, 59
 - connection 47, 48
 - data object 46
 - evaluator 47, 48
 - function evaluation 46
 - class library 36, 37, 46, 49
 - CLASSPATH 72, 73
 - C locale 479, 482, 483
 - collating 478
 - collation sequences 479, 480, 483
 - com.TIBCO.controls 387
 - com.TIBCO.splus 395
 - command line 5
 - script files 5
 - switches 5, 18
 - tokens 7
 - variables 5
 - Commit 57
 - Common error conditions when
 - using ActiveX controls in Spotfire S+ 311
 - CONNECT/C++ 36
 - .Call interface 38, 41, 43, 44, 47, 52, 58
 - Assign 39
 - catch block 47
 - classes 36, 38, 42, 46, 49, 50, 53, 54, 56, 58, 59
 - connection 47, 48
 - data object 46
 - evaluator 47, 48
 - function evaluation 46
 - class library 36, 37, 46, 49
 - Commit 57
 - constructors 46, 49, 50, 57, 58
 - converting objects 52
 - Create 39, 46, 56, 57, 58
 - CSParray 53
 - CSPengineConnect 38, 47, 57, 58
 - CSPevaluator 47, 48, 58
 - CSPnumeric 38, 52, 53, 56, 57, 59
 - CSPobject 46, 49, 50, 51, 52, 54, 56, 57, 58, 59
 - CSPproxy 54
 - CSPvector 53
 - destructors 49
 - generating functions 49
 - IsValid 49, 56
 - make utility 44
 - methods 42, 46, 47, 49, 57
 - objects
 - named 56, 57, 58
 - unnamed 58
 - OnModify 56, 57, 58
 - OnRemove 56, 57, 58
 - operators
 - arithmetic 54
 - assignment 49, 50, 51
 - conversion 52
 - overloading 51, 53, 54
 - subscripting 53, 54
 - printing 56
 - reference counts 49, 50, 51, 52
 - Remove 56, 57
 - S_EVALUATOR 43

- s_object 43, 46, 52, 56, 57, 58
 - S.so 44
 - sconnect.h 38, 42, 62
 - sconnectlib 44
 - SyncParseEval 39
 - try block 43, 47
 - try-catch block 43
 - CONNECT/Java 66
 - connection classes 47, 48
 - constructors 46, 49, 50, 57, 58
 - conversion operators 52
 - converting objects 52
 - Copy/Paste Link 243
 - copying external GUI files 375
 - copying help files 376
 - CorrelationsDialog class 391
 - cran 509
 - Create 39, 46, 56, 57, 58
 - creating and modifying buttons 268
 - creating and modify toolbars 264, 278
 - creating directories 374
 - CSParray 53
 - CSPengineConnect 38, 47, 57, 58
 - CSPevaluator 47, 48, 58
 - CSPnumeric 38, 52, 53, 56, 57, 59
 - CSPobject 46, 49, 50, 51, 52, 54, 56, 57, 58, 59
 - CSPproxy 54
 - CSPvector 53
 - Cygwin 442, 443, 444
- D**
- data directory 25
 - data object classes 46
 - DDE
 - CF_TEXT format 244
 - channel numbers 244
 - clients 243
 - definition of 242
 - functions
 - Advise 243
 - Connect 243
 - Execute 243
 - Poke 243
 - Request 243
 - Terminate 243
 - Unadvise 243
 - server names 244
 - servers 243
 - topics 244
 - using Copy/Paste Link 243
 - DDE requests
 - enabling and disabling response to 250
 - decimal.point argument 486
 - decimal markers 479, 482, 486, 487, 489
 - Decimal Point field 488, 489
 - delimiter, semicolon 488
 - Designing ActiveX controls that support Spotfire S+ 312
 - destination programs 243
 - destructors 49
 - dialog callback 327
 - example 329
 - Dialog Controls 333
 - digit-grouping symbols 486, 487
 - displaying numerics 478, 479, 486, 489
 - distributing the library 379
 - dos function 32
 - DOS interface 32
 - Dynamic Data Exchange 242
- E**
- ed function 29
 - edit function 29
 - environment variables 5, 8, 14
 - S_CMDFILE 11
 - S_CMDSAVE 12
 - S_CWD 13
 - S_DATA 13
 - S_NOAUDIT 15
 - S_PATH 15
 - S_PREFS 16
 - S_PROJ 16
 - S_SCRSAVE 17

- S_SILENT_STARTUP 17
- S_TMP 17, 18
- SHOME 14
- eval 94
- evalDataQuery 83
- evaluator classes 47, 48
- exportData function 486, 487
- exporting numerics 478, 486
- Export To File dialog 487, 488

F

- file expansion 7
- file formats, ASCII 488
- FileUtilities 95, 372
- fix function 29
- function evaluation classes 46
- function objects
 - methods for 217
 - passing data to 220

G

- Gauss-Seidel method 41, 42, 43
- General Settings dialog 481
- generating functions 49
- getControlDimension method 392
- getData 85
- getFileInputStream 95
- getFileOutputStream 95
- getFullDimension function 392
- graph objects
 - methods for 216
- Graph Sheets
 - embedding in automation clients 223
- guiImportData function 489

H

- help 451
 - Language Reference 470
- help.log 449
- help files, user-defined 441, 451, 454
 - .chm file 441, 442, 443, 444, 449

- build_helpchm.cyg 444, 446, 448, 449
- BUILD_JHELP 456
- BuildHelpFiles 442
- Cygwin 442, 443, 444
- help.log 449
- HINSTALL 455
- HTML Help Workshop 442, 443, 447
- keywords 441, 452, 461, 470, 471, 475, 476
- prompt function 441, 444, 445, 452, 453, 454, 455, 474
- Splus6 make install.help 455
- Splus8 make install.help 455, 456, 457
- Splus make install.help 453
- text formats for 458
- titled sections in 460, 474, 475

376

HINSTALL 455

- HTML-based object help system
 - 199, 207, 208, 210
 - refreshing 208

- HTML Help Workshop 442, 443, 447

I

- ImageCalculatorExample 95
- importData function 486, 487
- Import From File dialog 487
- importing numerics 478, 486
- install.help, Splus6 make 455
- install.help, Splus8 make 455, 456, 457
- install.help, Splus make 453
- ISO 8859-1 character set 479, 484, 485
- IsValid 49, 56

J

- jar 72
- java.graph 81, 86

java.util.Random 73
 javac 73, 87
 javaGetResultSet 93
 Java Native Interface 67, 71
 javap 69, 71
 Java Virtual Machine 66, 69
 JComponent 86
 JDBC 93
 JFileChooser 74
 JLabel object 389
 JNI 68, 71
 JRE 66
 JTabbedPane object 391
 JTextField object 389
 JVM 69

K

keywords 441, 452, 461, 470, 471,
 475, 476

L

Language Reference 470
 last.lib fz 372
 Latin1 478, 484, 485
 lib.loc 372
 libraries 372
 Linear Regression dialog 427
 LinearRegressionDialog class 391,
 392
 locales 478, 479, 484, 485

- C 479, 482, 483
- changing 482
- setting 480
 - in the Commands window
480
 - in the General Settings
dialog 481
- Sys.getlocale function 482
- Sys.setlocale function 480, 481,
482
- Sys.withlocale function 482

M

make 44
 makefile 44
 make install.help, Splus 453
 make install.help, Splus6 455
 make install.help, Splus8 455, 456,
 457
 markers, decimal 479, 482, 486, 487,
 489
 menuCor function 404
 menuLm function 417
 methods

- common, for automation
objects 211
- for application object 214
- for function objects 217
- for graph objects 216

 methods, CONNECT/C++ 42, 46,
 47, 49, 57

N

named objects 56, 57, 58
 names, variables

- characters allowed in 484
- characters not allowed in 485

 New Toolbar dialog 264
 numerics

- displaying 478, 479, 486, 489
- exporting 478, 486
- importing 478, 486

O

object help system 199, 207, 208,
 210

- refreshing 208

 object hierarchy 199, 207, 210
 objects

- named 56, 57, 58
- unnamed 58

 OnModify 56, 57, 58
 OnRemove 56, 57, 58
 operators

- arithmetic 54

- assignment 49, 50, 51
- conversion 52
- overloading 51, 53, 54
- subscripting 53, 54
- order, sort 479
- order function 483
- overloading operators 51, 53, 54

P

- paste function 29
- postscript function 485
- predict.lm 422
- preferences 24
- projects 24
- prompt function 441, 444, 445, 452, 453, 454, 455, 474
- putting functions in the library 374

R

- RandomNormalExample 88
- reference counts 49, 50, 51, 52
- referencing counting 232
- Regional Options, Windows 479, 480, 481
- regional settings 482
- Regional Settings, Windows 479, 480, 481
- Registering an ActiveX control 310
- Remove 56, 57
- ResultSet 93
- ResultSetUtilities 93
- R project 509

S

- S_CMDFILE variable 11
- S_CMDSAVE variable 12
- S_CWD variable 13
- S_DATA variable 13
- S_EVALUATOR 43
- S_FIRST 14
- S_FIRST variable 14
- S_NOAUDIT variable 15

- s_object 43, 46, 52, 56, 57, 58
- S_PATH variable 15
- S_PREFS variable 16
- S_PROJ variable 16
- S_SCRSAVE variable 17
- S_SILENT_STARTUP variable 17
- S_TMP variable 17, 18
- S.init file 457, 480, 482
- S.so 44
- sconnect.h 38, 42, 62
- search command 26
- semicolon delimiter 488
- sequences, collation 479, 480, 483
- server names 244
- setCenterPanel object 388, 391
- Set Region-specific defaults check box 481, 482
- settings, regional 482
- Setup, Spotfire S+ 478
- SHOME variable 14
- sort function 483
- sorting 478
- sort order 479
- source programs 243
- splus.exe executable 4
- splus.shome 82
- SplusBadDataException 85
- SplusControlMethods 397
- SplusControlMethods interface 389
- SplusControlMethods object 388, 390
- SplusControlMetrics class 392
- SplusDataResult 82, 83, 84, 85, 86, 89, 94
- SplusDataResult object 398
- SplusDataSetComboBox class 394
- SplusDialog 387
- SplusDialog.getMainMenuBar object 399
- SplusDialog object 391, 395, 397, 399
- SplusFunctionInfo object 387, 388, 397
- SplusGroupPanel object 388, 392
- SplusGroupPanel objects 388

- SplusIncompleteExpressionExcepti
on 82
 - SplusInvisibleControl object 390
 - SplusMainMenuBar class 399
 - SplusSession 83, 95
 - SplusTextField object 389
 - SplusUserApp 81, 86
 - SplusUserApp.eval 82, 89
 - SplusWideTextField object 388
 - Spotfire S+
 - Setup 478
 - Standard Generalized Markup
Language (SGML) 441, 451
 - statistic argument 406
 - steps in creating a library 373
 - subscripting operators 53, 54
 - summary method 419
 - Swing 388, 390
 - Swing classes 383
 - switch
 - HKEY_CURRENT_USER 22
 - MULTIPLEINSTANCES 21
 - Q 22
 - REGISTEROLEOBJECTS 22
 - REGKEY 22
 - TRUNC_AUDIT 21
 - UNREGISTEROLEOBJECTS
22
 - switches 18
 - symbols, digit-grouping 486, 487
 - SyncParseEval 39
 - Sys.getlocale function 482
 - Sys.setlocale function 480, 481, 482
 - Sys.withlocale function 482
 - system 28
- T**
- tabPlot.lm method 420
 - tabSummary.lm 422
 - tabSummary.lm method 419
 - TextOutputExample 87
 - thousands.separator argument 486
 - ToolBarButton property dialog 269
 - toolbars and palettes
 - customizing 264
 - topics, DDE 244
 - transferBytes 95, 376
 - try block 43, 47
 - try-catch block 43
 - type library 199, 207
 - refreshing 208
 - removing 208
- U**
- unnamed objects 58
 - user preferences 24
- V**
- variable names
 - characters allowed in 484
 - characters not allowed in 485
 - variables argument 405
- W**
- Where can the PROGID for the
control be found? 309
 - Why only “OCX String”? 311
 - Windows applications, Calculator
28
 - Windows applications, Notepad 29
 - Windows interface 28
 - Windows Regional Options 479,
480, 481
 - Windows Regional Settings 479,
480, 481

