



TIBCO Spotfire S+[®] 8.1

Guide to Packages

November 2008

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 THE *TIBCO SPOTFIRE S+® INSTALLATION AND ADMINISTRATION GUIDE*). 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+NuParam, S+SeqTrial, S+SpatialStats, S+Wavelets, S-PLUS Graphlets, Graphlet, Spotfire S+ FlexBayes, Spotfire S+ Resample, TIBCO Spotfire Miner, TIBCO Spotfire S+ Server, 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 © 1996-2008 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+® 8.1 Guide to Packages TIBCO Software Inc.

Technical Support

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

ACKNOWLEDGMENTS

TIBCO Spotfire S+ would not exist without the pioneering research of the Bell Labs S team at AT&T (now Lucent Technologies): John Chambers, Richard A. Becker (now at AT&T Laboratories), Allan R. Wilks (now at AT&T Laboratories), Duncan Temple Lang, and their colleagues in the statistics research departments at Lucent: William S. Cleveland, Trevor Hastie (now at Stanford University), Linda Clark, Anne Freeny, Eric Grosse, David James, José Pinheiro, Daryl Pregibon, and Ming Shyu.

TIBCO Software Inc. thanks the following individuals for their contributions to this and earlier releases of TIBCO Spotfire S+: Douglas M. Bates, Leo Breiman, Dan Carr, Steve Dubnoff, Don Edwards, Jerome Friedman, Kevin Goodman, Perry Haaland, David Hardesty, Frank Harrell, Richard Heiberger, Mia Hubert, Richard Jones, Jennifer Lasecki, W.Q. Meeker, Adrian Raftery, Brian Ripley, Peter Rousseeuw, J.D. Spurrier, Anja Struyf, Terry Therneau, Rob Tibshirani, Katrien Van Driessen, William Venables, and Judy Zeh.

TIBCO SPOTFIRE S+ BOOKS

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...
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>

Spotfire S+ documentation. (Continued)

Information you need if you...	See the...
<p>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.</p>	<p><i>Guide to Graphics</i></p>
<p>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.</p>	<p><i>Big Data User's Guide</i></p>
<p>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.</p>	<p><i>Guide to Packages</i></p>
<p>Are looking for categorized information about individual Spotfire S+ functions.</p>	<p><i>Function Guide</i></p>
<p>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.</p>	<p><i>Guide to Statistics, Vol. 1</i></p>
<p>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+.</p>	<p><i>Guide to Statistics, Vol. 2</i></p>

CONTENTS

Chapter 1 Spotfire S+ Guide to Packages	1
Overview of Spotfire S+® Packages	3
"Quick Start" to Packages	8
Required Tools for Creating Packages	13
Package Details	19
Example: Creating a Spotfire S+® Package	22
How to Submit a Package to CSAN	29
Components of a Source Spotfire S+ Package	30
Converting a Package from R to Spotfire S+	38
Differences between R and Spotfire S+	47
Chapter 2 Adding A GUI To a Windows® Package	49
Overview	50
Components of A Spotfire S+ GUI	51
GUI Example	55
Extending The Example	67
Building The GUI During Installation	72
Index	75

Contents

SPOTFIRE S+ GUIDE TO PACKAGES

1

Overview of Spotfire S+® Packages	3
Package Types	4
Spotfire S+ Package Structure	4
Location of User-Installed Spotfire S+ Packages	5
Location of Packages on the Spotfire S+ Server	6
"Quick Start" to Packages	8
Installing the pkgutils Library Section	8
Finding Packages on CSAN	9
Downloading Packages from CSAN	10
Installing and Loading a Package	11
Creating a Package	12
Submitting a Package	12
Required Tools for Creating Packages	13
Windows	13
UNIX/Linux	17
Package Details	19
Installing the pkgutils Library Section	19
Browsing Packages	19
Example: Downloading and Installing the rpart Package	20
Example: Creating a Spotfire S+® Package	22
Soundex Example	22
Building, Checking, and Installing the Package	25
How to Submit a Package to CSAN	29
Components of a Source Spotfire S+ Package	30
DESCRIPTION File	30
data Directory	31
R Directory	32
man Directory	32

src Directory	35
java Directory	35
Converting a Package from R to Spotfire S+	38
Getting An R Source Package	38
Creating a Spotfire S+ Package from an R Package	39
Build Scripts	40
Differences Between Spotfire S+ and R Packages	41
Porting Tools	43
Trouble-Shooting Porting R Packages	45
Missing C/FORTRAN Functions	46
Differences between R and Spotfire S+	47

OVERVIEW OF SPOTFIRE S+® PACKAGES

A TIBCO Spotfire S+® package is a collection of S functions, data, help files, and other associated files (C, C++, or FORTRAN code) that have been combined into a single entity you can distribute to other Spotfire S+ users. These packages offer you and other Spotfire S+ users a mechanism to distribute user-generated functions quickly. You can download and install Spotfire S+ packages from an TIBCO-maintained Web site, or you can create Spotfire S+ packages that you can submit for potential distribution.

The Spotfire S+ package system is modeled after the package system in R. The R system has package repositories available via the Internet, and has seen huge success in distributing new statistics and data analysis functionality to R users.

This document contains guidance in the following areas:

- Browsing, downloading, and installing packages from a centralized repository.
- Downloading the tools necessary to create packages.
- Building your own packages to distribute and submit to the repository.

This overview contains introductory information on the following:

- Accessing the TIBCO package archive repository.
- Finding and downloading packages.
- Discussing package types.
- Listing package components.
- Creating a package.
- Submitting a package for posting on the Comprehensive S-PLUS Archive Network (CSAN).

TIBCO hosts the CSAN site at **<http://spotfire.tibco.com/csan>** to facilitate Spotfire S+ package distribution. This Web site serves as a centralized repository for Spotfire S+ packages, and for information about creating, installing, and using packages.

To maintain as much compatibility with R packages as possible, we adapted and used many of the functions and scripts from R for the Spotfire S+ package system. The code is distributed separately from Spotfire S+ under the GPL license.

Package Types Packages on CSAN are available as either Windows[®] *binary* or *source*. By default, the functions for downloading and installing packages from CSAN use binary packages, while UNIX/Linux uses source packages. You can also add Java code to a package, following the steps outlined on section java Directory on page 33.

Installing a package from *source* requires additional software tools, such as compilers and Perl, which are not available on a typical Windows installation (see the section Windows on page 12). However, with the proper tools installed, Windows users can build and install packages from source, or create binary packages that can be distributed to other Windows users.

Spotfire S+ Package Structure

A package is a collection of S functions, help files, and possibly C or FORTRAN source code combined in a single archive (**.zip** in Windows or **.tar.gz** in UNIX/Linux). The archive can be either a source archive or a platform-specific binary archive.

When unpacked, a source package contains a directory (with the same name as the package) and the following subdirectories:

- **data:** Contains data files, in dump format, or as a delimited (space or semi-colon) text file. (Optional.)
- **man:** Contains help files which use the **.Rd** help file format that R uses.
- **R:** Contains any S language functions as ASCII files.
- **src:** Contains C, C++, or FORTRAN source code as ASCII files (this directory is optional).
- **java:** Contains two subdirectories:
 - src:** Directory for Java source code.
 - prebuiltjars:** Precompiled **.jar** files.

(This directory is optional). See the section java Directory on page 35 for details.

- **inst**: Contains files and directories to be copied, recursively, into the main package directory when the package is compiled. Any informational files that the end user should see should be included in the **inst** directory. For example, if you have a PDF containing a vignette, include it in the **inst/doc** directory.

The package also contains, at the top level, the **DESCRIPTION**, which is text file containing information about the package (see the section **DESCRIPTION File** on page 30). This is the only file required in a package.

Note that a data-only package contains only the **data** and **man** directories, whereas an S-code-only package contains only the **R** and **man** directories. Similarly, a package can contain an **inst** directory containing a **CITATION** file (which you can create by calling `citation()`) and a **doc** subdirectory for any document files, such as vignettes.

A package can also have a **tests** directory to contain package-specific tests. This directory can contain test code (that is, **.S**, **.ssc**, **.q**, **.R**). When you run the tests in this file, the results are written to a **.Sout** file. For more information about running tests using a **tests** directory in your package, see the section **Checking the Package** on page 26.

When a binary package archive is unpacked, the S functions are already in binary form in a **.Data** directory, the help files are already in a form accessible from within Spotfire S+, and the source code has already been compiled into a shared library object (**S.dll** on Windows or **S.so** on UNIX/Linux). The unpacked binary package also includes the **DESCRIPTION** text file.

Location of User-Installed Spotfire S+ Packages

Starting with Spotfire S+ 8, user-installed packages (referred to as "library sections" in older manuals) have a platform-dependent default location :

Windows XP:

**C:\Documents and Settings\username\
Application Data\TIBCO\splus81_WIN386\library**

Windows Vista:

**C:\Users\username\AppData\Local\TIBCO\
splus81_WIN386\library**

UNIX[®]/Linux[®] : The following is an example on Linux:

\$HOME/MySwork/splus81_LINUX/library

Note the Spotfire S+ version and platform designation are included in these default package locations for all platforms. This allows packages for multiple platforms and Spotfire S+ versions to be installed; e.g., both Linux 32-bit and Linux 64-bit packages could be used on the same Linux-64 bit machine. Note the Spotfire S+ version in the directory name (e.g., **splus81_WIN386**) will be updated to match the Spotfire S+ version.

The `library` function searches this location before it searches **\$HOME/library** when it looks for a library section to load. The package manipulation functions use the new location as the default for installing packages.

One advantage of this new default is that packages the user installs are separate from those installed with Spotfire S+. This simplifies creating the same Spotfire S+ environment on another computer: after you install Spotfire S+ on the new computer, simply copy over the local directory.

Location of Packages on the Spotfire S+ Server

It is recommended that only system administrators install packages on the Spotfire S+ Server when you want to deploy packages on a system-wide basis. The following steps explain how to specify directory locations and install packages on a server:

1. Specify a directory to be used for package installation on the server by setting the `S_USER_APPDATA_DIR` environment variable to the directory.
2. Run Spotfire S+ to install the package.
3. Set the `splus.appdata.dir` property in

TOMCAT_HOME/conf/Catalina/localhost/SplusServer.xml

(where **TOMCAT_HOME** is your Tomcat installation) to this same directory, so the server engines can access this directory as well:

```
<Environment name="splus.appdata.dir"
              value=""
              type="java.lang.String"
              override="false"/>
```

If the `splus.appdata.dir` property is not given or is an empty string, the `<splus.webdav.root>/appdata` (i.e., the "appdata" directory in the directory specified by `splus.webdav.root`) is used.

Note that setting the `splus.appdata.dir` property only affects the operation of the Spotfire S+ Server and not a normal desktop Spotfire S+ installation.

If a system administrator creates their own `S_USER_APPDATA_DIR` directory, it is recommended the directory name include the Spotfire S+ version number and platform (e.g., **splus81_WIN386**) to keep track of the packages being installed, which are specific to a given Spotfire S+ version and platform.

See the *Spotfire S+ Server Administration Guide* for more details.

Caution: Be Careful When Using Client Code to Install Packages on a Server

All packages should be installed by a system administrator to control what packages are available system wide. There is nothing to prevent client code from trying to installing a package into the `S_USER_APPDATA_DIR` directory, allowing users to run Spotfire S+ code that could potentially compromise your system.

"QUICK START" TO PACKAGES

Installing the pkgutils Library Section

Before you can do any work with packages in Spotfire S+, you must download and install the pkgutils library section. This contains functions and scripts for downloading, installing, building and checking packages. The pkgutils library section contains code distributed under the GPL license, and thus is not included as part of the Spotfire S+ distribution.

On UNIX/Linux, the pkgutils library section should be installed when Spotfire S+ is installed and configured. Run the script

```
./INSTALL.PKGUTILS
```

in the top level directory of Spotfire S+ to download and install pkgutils in **library/pkgutils** under the top level directory of Spotfire S+. The same individual should install Spotfire S+ and the pkgutils library section, because you need to have the appropriate permissions to install in the Spotfire S+ directory.

On Windows, anyone can install the pkgutils library section, because it gets installed in the individual user's **Application Data** directory:

Windows XP

```
C:\Documents and Settings\username\  
Application Data\TIBCO\splus81_WIN386\library
```

Windows Vista:

```
C:\Users\username\AppData\Local\TIBCO\  
splus81_WIN386\library
```

To install pkgutils in Windows, in the Spotfire S+ **Commands** window type

```
install.pkgutils(update=T)
```

The update=T argument updates the pkgutils library in case you have already installed it and want to make sure you have the latest version of all the functions.

From this point forward, the steps are the same for both platforms to attach the library and install and run library functions. Typing

```
library(pkgutils)
```

loads the pkgutils library so all functions in the library are available for your current Spotfire S+ session. Typing

```
available.packages()
```

displays the packages currently available from CSAN. If we want to install (for example) the rpart package, enter the name in quotation marks:

```
install.packages("rpart")
```

This installs the rpart package in your package library directory by default, which is a platform-dependent location (see the next section for details). You can now load the rpart library to access its functions:

```
library(rpart)
```

Details on creating packages can be found in the section Creating a Package on page 11.

Finding Packages on CSAN

Use the following functions in Spotfire S+ to help you discover available packages on the CSAN site (<http://spotfire.tibco.com/csan>).

Table 1.1: *Package browsing functions.*

Spotfire S+ function	Description
available.packages	Use available.packages to determine which Spotfire S+ packages are available for download from the CSAN site.
new.packages	Use new.packages to discover any Spotfire S+ packages on CSAN that you have not yet installed.

Downloading Packages from CSAN

Use the following functions in Spotfire S+ to help you download and install packages from the CSAN site (for more detailed information about downloading packages, see the section Package Details on page 18).

Table 1.2: *Package downloading functions.*

Spotfire S+ function	Description
<code>install.packages</code>	Use <code>install.packages</code> to download and install packages from CSAN in a single step.
<code>download.packages</code>	Use <code>download.packages</code> to download a package from CSAN, for later installation. You must provide a destination directory (the <code>destdir</code> argument) or an error results.

Do not use **SHOME/library** when you use `download.packages`, because that path is reserved for base packages. See page 5 for the default locations by platform.

Note that `download.packages` is useful if you want to work on the source code or if you want to host a CSAN mirror; in most cases, `install.packages` is more appropriate, since you can download and install a package from CSAN in one step.

Note

Alternatively, you can use the Spotfire S+Spotfire S+ GUI menu to find and download packages.

- On the menu, click **File ► Find Packages**, and then select a repository and a package to download.

Also, you can update new versions of packages you have previously downloaded.

- On the menu, click **File ► Update Packages**, and then select a repository and the package to update.

If you are using the Spotfire S+Spotfire S+ Workbench on either Windows or UNIX[®]/Linux[®], you can find the **Find Packages** and **Update Packages** dialogs on the Spotfire S+ menu. See the *Spotfire S+ Workbench Guide* for more information.

Installing and Loading a Package

You can install a package under a **library** location on your system.

- Installing a binary package consists of unpacking the binary archive in the appropriate location.
- Installing from a source archive involves sourcing the S functions, converting the help files into a format that can be displayed within Spotfire S+, and compiling any source code into shared library object. The resulting pieces are then copied to the specified location.

After you have installed a package, load the package in a running Spotfire S+ session with the `library()` command. If package `xyzy` was installed under the standard package location of your Spotfire S+ installation, then you need only to enter

```
library(xyzy)
```

to load the package into your current Spotfire S+ session. If you installed the library in another location, you must specify that location in the `lib.loc` argument to `library`. For example, if you install all your packages under **D:\swork\lib**, then to load package `xyzy`, you must type

```
library(xyzzy, lib.loc="D:/swork/lib")
```

or

```
.libPaths("D:/swork/lib")  
library(xyzzy)
```

Creating a Package

Use the `package.skeleton` function to specify the name of the package you want to create and which functions to include in the package. By default, this function creates an empty package in your current working directory. You can control which objects you want to include in your package using the `list` argument.

The `package.skeleton` function also generates template help files in **.Rd** format, which you can edit to document your functions. After you have edited or added any associated files, you can run a Spotfire S+ check against the package to verify completeness, and then build the package in a compressed format (**.zip** for Windows[®] or **.tar.gz** for UNIX/Linux) for distribution. \

Note that you can also create a Java package by following the steps outline in the section `java Directory` on page 33.

Submitting a Package

To share your Spotfire S+ package with the Spotfire S+ user community, send it to TIBCO for posting on the CSAN site. TIBCO checks the package and if it is accepted, it is posted to the CSAN site in both source and Windows[®] binary form. Note that you only need to submit a source package archive, and TIBCO creates and posts the Windows binary. These Spotfire S+ packages are then posted to CSAN by TIBCO and become available for download.

REQUIRED TOOLS FOR CREATING PACKAGES

Downloading or installing Spotfire S+[®] packages requires the pkgutils library (described in the previous section). Editing and compiling a package requires the pkgutils library and a tool set appropriate for your platform. This section discusses these required tools and where to find them. You must be connected to the Internet to download the tools you need to edit and compile packages.

All tools discussed in this section are available for free download and installation.

Windows

Spotfire S+ for Windows[®] users must install additional software components to build and install packages from source code. These components are available for free download, and you can get detailed information on all required components by navigating to CSAN at

<http://spotfire.tibco.com/csan>

and clicking the **Windows Tools** link (under **Resources**, on the lower left side of the page).

Note

The following tools are required if you are creating or installing *source* packages. You do not need additional software if you are only installing binary packages for Windows from CSAN using Spotfire S+ functions (e.g. `install.packages()` and `update.packages()`). However, you do need Perl (and possibly other tools) if you use the scripts (invoked using `Sp1us CMD`) on Windows.

Table 1.3 describes the tools the package system expects to find in your PATH. For more information about these tools and others that you might need for package creation, see the sections following the table.

Table 1.3: *Tools the package system expects to find in Windows path.*

Tool	Comment
perl	Version 5.8 or later. (On Windows [®] , you must have Active State perl.)
hhc.exe	From Microsoft Help Workshop. Not included or supported on Vista.
nmake	Required if the package contains C or C++ code. Included in Visual C++ [®]
cl	Required if the package contains C or C++ code. Included in Visual C++.
link	Required if the package contains C or C++ code. Included in Visual C++.
javac	Required if the package contains java code. Included in the Java Development Kit (JDK).
jar	Required if the package contains java code. Included the JDK.
df.exe	Required if the package contains fortran code.

Perl

The scripts for creating, building, and installing packages from source are written in the Perl scripting language. We require (and have tested with) the Perl 5.8 for Windows implementation from ActiveState, a freely-available download.

Note that versions after ActiveState Perl 5.8 were not tested with this release of Spotfire S+, so the compatibility is not known.

Microsoft HTML Help Workshop

To create compiled help (**CHM**) files for your package, you need HTML Help Workshop. Compiled Help created with HTML Help Workshop is the only help format supported in Spotfire S+ packages.

tar and gzip

If you are starting with a package source archive that has a **tar.gz** or **gzip** extension, you need **tar** and/or **gzip** utilities to unpack the archive. These utilities are freely available from many locations (for example, www.cygwin.org).

Compilers

You must have a C/C++ compiler if your package includes C or C++ code. The Spotfire S+ package system currently supports the Microsoft Visual C++[®] compiler. You might already have the Microsoft Visual C++ compiler installed. If not, you can install Visual C++ 2005 Express Edition (which is free).

Spotfire S+ supports FORTRAN code compiled with Visual Fortran[®]. At this time, there is no free version of Visual Fortran available.

If you have any Java source code that is not pre-compiled and you want to include in a package, you must have the Java Development Kit to perform the compilation. The version of the Java Development Kit you install should be the same as the version of JRE that Spotfire S+ uses.

Note about Perl and Visual Studio Compiler Installations

The installer typically asks if you want the global PATH updated; it is generally easiest if you let the installer update the PATH. For Visual Studio, it is convenient to also copy LIB and INCLUDE into the global environment, taking values set in **vcvar32.bat**.

Access to Windows Tools

For the Spotfire S+ package build and install scripts to function properly, you must put the tools listed above in your path after you have installed them. The installation system for the particular tool may have updated your path for you. To check and update your path:

1. Right-click **My Computer** and click **Properties**.
2. In the **System Properties** window, click the **Advanced** tab.
3. In the **Advanced** section, click the **Environment Variables** button.

4. In the **Environment Variables** window, highlight the path variable in the **System variables** (or **User variables**) section and click **Edit**.
5. Check to confirm the path to the Windows tools are present and correct. You can modify the path lines as desired, separating with semicolons, e.g.

d:\Perl\bin;d:\htmlhelp;d:\VC\bin;d:\jdk1.6.0\bin

for Perl, HTML Help Workshop, the Visual Studio compiler, and the Java compiler, respectively (note the Java compiler is optional if you do not have any Java code to include in a package). If you installed a Windows compiler, confirm that it is present. Click **OK**.

Note

If you do not want to change your environment variables permanently, you can run a script in the command window that sets these variables just for the current session.

To confirm the path has been set correctly:

1. From the **Start** menu, click **Run**.
2. In the text box, type `cmd`, and then click **OK**.
3. From the `cmd` shell window, check to make sure the tools are working by trying the following commands:

```
perl --version
tar --help
gzip --help
hhc /help
javac -version
```

4. If you are using the Visual C++ compiler, type the following:

```
cl /help
```

5. If you are using Java, check that you have `javac`, as described above.
6. Make sure the following environment variables include the appropriate directories from your Visual C++ installation:

- LIB
- INCLUDE

The file **vcvars32.bat**, created when the Visual C++ compiler is installed, should set the necessary compiler variables. You can find this file in the **bin** subdirectory in your Visual C++ installation. Run **vcvars32.bat** from the **cmd** shell window every time before you run any package creation script.

UNIX/Linux

Creating packages on the UNIX[®]/Linux[®] platforms requires these additional software tools, in addition to Spotfire S+.

Table 1.4: *Tools the package system expects to find in PATH.*

Tool	Comment
cc	On Solaris, Required if package contains C code.
gcc	On Linux, required if the package contains C code.
CC	On Solaris, Required if package contains C++ code.
g++	On Linux, required if the package contains C++ code.
f77	On Solaris, required if the package contains fortran code
gfortran	On Linux, required if the package contains fortran code
jar	Required if the package contains java code.
javac	Required if the package contains java code.

Table 1.4: Tools the package system expects to find in PATH. (Continued)

Tool	Comment
perl	Version 5.8 or later.
grep, sh, other standard UNIX tools	
make	A standard tool, like grep, but it might be in another directory.

PACKAGE DETAILS

Installing the pkgutils Library Section

Before you download or install Spotfire S+ packages, you must download and install the Spotfire S+ pkgutils library section. See the section Installing the pkgutils Library Section on page 8 for more details.

Browsing Packages

Using your Web browser, you can browse for available packages on the CSAN site at <http://spotfire.tibco.com/csan>. From your browser, you can download a Spotfire S+ package and save the package archive on your local machine. You can use scripts from a shell, or functions within Spotfire S+ to install the package archive.

Alternatively, you can get a listing of the packages on CSAN using the function `available.packages` within Spotfire S+. You can then download and install the packages from within Spotfire S+. For example:

```
# attach the pkgutils library
library(pkgutils)
# get a list of available packages
ap <- available.packages()
```

This function call returns only packages from CSAN that match the `options("pkgType")` value. The default value is set to "win.binary" for Windows and "source" for UNIX/Linux. See the section Package Types on page 4 for more information on package types.

The return value is a character matrix, one row for each package returned, with the columns as values from the package's **DESCRIPTION** file. The first column is the names of all the packages.

**Example:
Downloading
and Installing
the rpart
Package**

When you find a Spotfire S+ package you want to try, you can download and install the package on your local machine. Alternatively, you can download and install a package in two separate steps, if needed.

The following is an example of downloading and installing `rpart` from CSAN as either a binary package on Windows or a source package on UNIX/Linux:

1. After you have loaded the `pkgutils` library and determined which package to install, run `install.packages`. In the following example, download and install the `rpart` package from CSAN in your default package directory:

```
install.packages("rpart")
```

Note that if you are an administrator, you can use the `lib` argument to install packages in a location where all users of a computer can access it:

```
install.packages("rpart",  
  lib=file.path(Sys.getenv("SHOME"),"local",  
  "library"))
```

2. Attach the library and check the objects:

```
library(rpart)  
objects("rpart")
```

3. With the library attached, you can get help on the `rpart` library or any functions within the `rpart` library:

```
help(rpart)
```

4. Now you can access any of the `rpart` functions. Here, we fit a classification tree to the `kyphosis` data set:

```
fit1 <- rpart(kyphosis ~ Age + Number + Start,  
  data=kyphosis)
```

If you just want to download the package without installing it, run

```
dp <- download.packages("rpart", destdir=".")
```

Note that you must supply the path for the `destdir` argument. The `download.packages` function returns a two-column matrix:

- The number of rows in the matrix is the number of packages downloaded. (In this example, only one package.)

- The first column of the matrix is the name of the package. ("rpart" in this example.)
- The second column of the matrix is the destination file name for the download archive. (In this example, ".\\rpart_3.0.zip" on Windows and "./rpart_3.0.tar.gz" on UNIX/Linux. The version number you see may be different if updated). To install from an archive that has been downloaded, call `install.packages` with the name of the archive file and set the `repos` argument to `NULL`, so the function does not attempt to get the file from the CSAN repository. To install the rpart archive, call:

```
install.packages(dp[1,2], repos=NULL)
```

EXAMPLE: CREATING A SPOTFIRE S+® PACKAGE

A package is a collection of S functions, C/C++/FORTRAN code, data sets, and documentation that you can share. The package has a specific organization of the files into subdirectories.

Before you start creating a package, make sure you have the tools required for your platform. For more information, see the section Required Tools for Creating Packages on page 12.

The easiest way to create a package is to use the `package.skeleton` function in Spotfire S+. The `package.skeleton` function, which is in the `pkgutils` library, creates an appropriate package directory with the same name as the package. Within that package directory, `package.skeleton` creates files and subdirectories; this directory structure is discussed in the section Components of a Source Spotfire S+ Package on page 30.

Soundex Example

The following example creates a Soundex example package using `package.skeleton`.

Note on Soundex

Soundex is a phonetic algorithm for indexing names by their sound when pronounced in English. Each name is converted to a string consisting of an initial letter followed by three numbers. Details of the algorithm are available at

<http://en.wikipedia.org/wiki/Soundex>

or at

<http://www.genealogyandhow.com/lib/soundex/codes.htm>

As noted in the references, the Soundex algorithm has several definitions. We show one implementation in the following example.

1. In Spotfire S+, load the `pkgutils` library:

```
library(pkgutils)
```

2. In Spotfire S+, define a `soundex` function:

```
"soundex"<-  
function(x) {
```

```
# 1. extract the last word of surnames and translate
#     to all upper case
base <- gsub("[^A-Z]", "", toupper(gsub("^.*[ \\t]",
    "", gsub("[ \\t]*$", "", x))))
# 2. encode the surnames (last word) using the
#     soundex algorithm

basecode <- gsub("[AEIOUY]", "", gsub("[R]+", "6",
    gsub("[MN]+", "5", gsub("[L]+", "4",
    gsub("[DT]+", "3", gsub("[CGJKQSXZ]+", "2",
    gsub("[BFPV]+", "1", gsub("[HW]", "", base)))))))))
# 3. deal with the 1st letter and generate the
#     final coding padded with 0
sprintf("%4.4s", paste(substring(base, 1, 1),
    ifelse(regexpr("^[HWAIEIOUY]", base) == 1,
    basecode, substring(basecode, 2)),
    "000", sep = ""))
}
```

The above function is the shortest and fastest implementation of a soundex function resulting from a contest held at Insightful. The code uses several functions that were new in S-PLUS 8.0.

Some data to test the function:

```
sample.surnames <- c("Ashcroft", "Asicroft",
    "de la Rosa", "Del Mar", "Eberhard",
    "Engebretson", "O'Brien", "Opnian", "van Lind",
    "Zita", "Zitzmeinn")
```

Try out the function:

```
soundex(sample.surnames)
```

3. Call the `package.skeleton` function in Spotfire S+ to create an initial package:

```
package.skeleton("soundex", list=c("soundex",
    "sample.surnames"))
```

This function call creates a directory called **soundex** under the current directory containing the initial package files. See the section Spotfire S+ Package Structure on page 4 for more information about the package files and subdirectories.

4. From the command shell (or from your favorite text editor), edit the help file templates in **soundex/man**, providing the details for the function and data set.
5. If your package includes any C/C++/FORTRAN code, you would put the source files in **soundex/src**. (This example contains no source code.)
6. Again, using your favorite text editor, edit the **DESCRIPTION** file, **soundex/DESCRIPTION**, adding values for the appropriate keywords. Be sure to complete the **Author**, **Maintainer**, **Title**, **Version** and **License** values. Note that any line starting with <letters><colon> starts a new section, and the colon should come immediately after the letters, with no space between them.

At this point you have a basic package directory called **soundex**.

If you want to add S functions to this package, you can add them to the **R** subdirectory with the `dump` function. To add help files for any added S functions, call `prompt.Rd`. This function creates **.Rd** help file templates in the **man** directory.

For example, if you have another `soundex` function called `soundex2`, you would add it to the package:

```
library(pkgutils)
dump("soundex2", "soundex/R/soundex2.q")
prompt.Rd("soundex2", "soundex/man/soundex2.Rd")
```

(The above example assumes you are running Spotfire S+ from the same location where you initially called the `package.skeleton` function.)

You can call the `package.skeleton` function without specifying any S objects in the `list` argument. Doing so creates the package directory structure with no files in the **man** or **R** subdirectory. This strategy can be useful if you already have functions and help files stored in ASCII files elsewhere, and you want to add them to your package. You would copy the S object files (in `dump` format) to the **R** or **data** subdirectory of the package and the **.Rd** files into the **man** subdirectory.

Building, Checking, and Installing the Package

Spotfire S+ includes utilities that you can run on the package (listed in Table 1.5), and you can run them from a command shell on Windows and UNIX/Linux. As noted in the section Windows on page 12, using these scripts requires additional software components. The scripts also require that the pkgutils library be installed.

Table 1.5: *Package utilities.*

Utility	Description
build	Creates an archive of the package source. By default, the archive is a source archive; however, there is an option to create a binary archive. A binary archive is platform-specific (that is, Windows or UNIX/Linux). A user installing a binary package archive does not need additional tools to install and use the package.
check	Checks the package source to ensure that all necessary files are included, that it can be built, and so on.
INSTALL	Installs the package on the system such that users can load the package with a call to the <code>library</code> function. You can also use <code>install.packages()</code> from a Commands window.
REMOVE	Removes the package from the system. You can also use <code>remove.packages()</code> from a Commands window.

Invoke the scripts in the command shell with a command of the form:

```
Spplus CMD utility.name options ...
```

You can get help on these scripts by entering this in a command shell:

```
Spplus CMD utility.name --help
```

Note

These scripts are named `build`, `check`, `INSTALL`, and `REMOVE` for compatibility with R, located in your **\$HOME/cmd** directory. On UNIX/Linux, there is a separate `INSTALL` utility (at the top level of your Spotfire S+ installation directory).

Building the Package Archive

To build a source archive from a package directory, run the `build` script from the directory containing the package (*not* from within the package directory). If you have the **soundex** package directory from the above example, run:

```
Splus CMD build soundex
```

This creates a source package archive file called **soundex_1.0.zip** on Windows or **soundex_1.0.tar.gz** on UNIX/Linux.

If you include the `-binary` flag in the call to `build`, you create a binary package archive file. The name of this archive file includes the platform in the name (for example, **soundex_1.0_WIN386.zip**). That package archive can be installed only on the same platform (that is, Windows or Linux) that it was created on.

Checking the Package

Before distributing a package archive to others, run the `Splus CMD check` utility on the package. This utility performs the following checks:

- Verifies the package structure (that is, checks that all required files and directories exist and are in the appropriate formats).
- Installs all S code to check for syntax errors.
- Compiles any C, Fortran, and Java code.
- Builds all help files in the **man** directory and (for Windows[®] binary packages) compiles the **package.chm**.
- Extracts and runs the **Examples** section of all help files and ensures that the code runs.
- Runs package tests to ensure that the package can be built and installed.

If you have code that you want to test iteratively, create a package **tests** directory and include the test files in it.

To create a package test

1. Create a **tests** directory in your package source. (See section Spotfire S+ Package Structure on page 4 for more information about this directory.)

2. In the **tests** directory, place any package-specific test files. These files can have any extension that Spotfire S+ recognizes (that is, **.S**, **.ssc**, **.q**, and **.R**).
3. Run `Splus CMD check` to run a check on the packages. As part of the check, all files in **tests** are run and the corresponding results files (**.Sout**) are created in your *package.Scheck/tests* directory.
4. Review the **.Sout** files and if you are satisfied with the results, rename the files to **Sout.save** and place them in your package's **tests** directory.
5. Run `Splus CMD check` again. This utility compares **.Sout** and **.Sout.save**. It should result in no differences between the resulting **.Sout** and the **.Sout.save** you just created.
6. If you change your code in the **tests** directory, run the `Splus CMD check` utility again to create another updated **.Sout** file. The utility reports any differences to the results. (It does not report errors.)

If the package includes a **tests** directory containing files with the extension **.t**, they are run using the `do.test()` function. Any tests that do not result in `TRUE` are reported as errors during a check. See the `do.test` help file for details.

To check the `soundex` package, run the `check` script from the directory containing the package (*not* from within the package directory).

```
Splus CMD check soundex
```

You can also check a source package archive directly. For example:

```
Splus CMD check soundex_1.0.tar.gz
```

Installing the Package

Use the `INSTALL` script to install a package. For the `soundex` package example, from the directory containing `soundex` (*not* from within **soundex**), run the following:

```
Splus CMD INSTALL soundex
```

By default, this command installs the package in your package directory.

Next, from within Spotfire S+, you can load the package with the following command:

```
library(soundex)
```

The `library` function searches the package library location by default. You can install the package in another location by providing that location with the `-l` flag:

```
mkdir mylib  
Splus CMD INSTALL -l mylib soundex
```

In Spotfire S+, assuming the working directory is the directory containing the **mylib** directory you just created, you can load the `soundex` package with the following command:

```
library(soundex, lib.loc="mylib")
```

You can install from a source or binary package archive. Instead of specifying the package directory in the call to the `INSTALL` script, pass it the package archive name. For example:

```
Splus CMD INSTALL soundex_1.0.tar.gz
```

or

```
Splus CMD INSTALL soundex_1.0.zip
```

As an alternative to using the `install.packages()` function in Spotfire S+, you can use the `INSTALL` script to install packages obtained from CSAN.

On Windows, you must use the `INSTALL` script to install a source package. You cannot install a source package on Windows with the `install.packages()` function.

HOW TO SUBMIT A PACKAGE TO CSAN

You can share your Spotfire S+ package with other users within your department, company, or university by sending them the package archive. Others can install them using the `INSTALL` script or the `install.packages` function (setting `repos=NULL`).

To share your package with the entire Spotfire S+ community, you can submit your package for inclusion in the Comprehensive S-PLUS Archive Network (CSAN). To submit a package, upload the source package archive (the result of running `Splus CMD build`) to:

`ftp://ftp.insightful.com/public/incoming/packages`

After you have uploaded your file, send a message to

`packages@tibco.com`

stating the name of the package archive you submitted.

Before you submit a package for inclusion in CSAN, be sure it passes the `check` utility. Also, make sure these key fields in the **DESCRIPTION** file have appropriate values: Package, Title, Version, Author, Maintainer, and License. If any of these are missing, your package cannot be posted to CSAN.

TIBCO Spotfire engineers review your submitted package, run the `check` utility, and create a Windows binary archive and then post the to the CSAN site (**`http://spotfire.tibco.com/csan`**). If the engineers find problems with the package, they alert the package submitter.

COMPONENTS OF A SOURCE SPOTFIRE S+ PACKAGE

The `package.skeleton` function automates some of the setup for a new source Spotfire S+ package. It creates directories, saves the specified functions and data to appropriate places, and creates skeleton help files, as well as **README** files describing further steps in packaging. The six main subdirectories and files generated in the working directory under the package name are as follows:

1. **DESCRIPTION** file: Lists package title, author, version, contact information, and other details specific to the package.
2. **man** subdirectory: Contains help file templates in **.Rd** format for Spotfire S+ functions, datasets, classes, etc. For example, **fun1.Rd** and **fun2.Rd** get generated if functions `fun1` and `fun2` are in your working directory when you run `package.skeleton`.
3. **README**: Provides details for each directory/file generated by `package.skeleton`. These files contain information for the package creator. They should be removed before the package is built or installed.
4. **R**: Directory containing text dumps of the package functions.
5. **src**: Directory holding C/C++/FORTRAN code. (Optional).
6. **java**: Directory to hold Java source code (located in **java/src**) and prebuilt **.jar** files (**java/prebuiltjars**). See the section `java Directory` on page 33 for details.
7. **data** - directory containing data files in dump or **CSV** format.

When you run `Splus CMD build packagename` to build a package, you concatenate all these parts into one compressed file for ease of distribution.

The following example uses the `rpart` package from CSAN to discuss the contents of each of these files/directories.

DESCRIPTION File

The **DESCRIPTION** file contains key information about the package including the package name, title, version, author, license, package date, and build date. If you find a bug or error in the package, contact information for the package's author should be included. It is in the Debian Control File format, where each line

consists of a keyword, colon, and description of the keyword. Description fields can continue on the next line if that next line starts with a space. The **DESCRIPTION** files are a key part of the package system: They are checked for available and installed packages, and which packages to update. See the help files for the functions `read.dcf` and `packageDescription` for more information about the **DESCRIPTION** files.

The `DESCRIPTION` file is an ASCII text file. Each line starts with a `KEYWORD: <space>` followed by the description for that keyword. The keyword list for `rpart` has this content:

```
Package: rpart
Type: Package
Title: Recursive Partitioning Tree Models
Version: 3.0
Date: Thu Mar  2 22:30:36 PST 2006
Author: Terry M. Therneau and Beth Atkinson
<atkinson@mayo.edu>
Description: Recursive partitioning and regression trees
License: GPL2, see Readme
Dialect: S-PLUS
Packaged: Sun Jul 30 10:10:10 2006; spk
```

The `Package` entry is written by the package build procedure, while the `Version` information is read when specific functions are run, including `available.packages()`.

A new function, `packageDescription`, reads an installed package's `DESCRIPTION` file and returns a named list, with keywords as names, and each component the value associated with that keyword:

```
packageDescription("rpart", lib=libhome)
```

data Directory The **data** directory is a subdirectory containing dumps of the data objects specified in the `package.skeleton` list argument.

This directory can also contain ASCII data files with particular file extensions (currently, **.csv**, **.CSV**, **.tab**, **.TAB**, **.txt** and **.TXT**). The `installFromDataFiles` function is used to process all of the files, and the name of the resulting data frame is the name of the file without the extension (e.g., the file **xyzyzy.txt** results in creation of a data frame called `xyzyzy` in the package. See the `installFromDataFiles` help files for more information.

For example, in the section Soundex Example on page 21, when you created the soundex package by calling

```
package.skeleton("soundex", list=c("soundex",  
  "sample.surnames"))
```

a data directory containing the dump file **sample.surnames.S** was created. The contents of this file looks like the following:

```
"sample.surnames" <- c("Ashcroft", "Asicroft", "De La Rosa",  
  "De1 Mar", "Eberhard", "Engebrethson", "O'Brien",  
  "Opnian", "van Lind", "Zita", "Zitzmeinn")
```

R Directory

The **R** subdirectory contains ASCII dumps of all the functions included in your Spotfire S+ package. For example, if you define fun1 as follows:

```
fun1 <- function(x) x^2
```

and you specify this function in your list argument in package.skeleton, then a file called **R/fun1.S** is generated in the package subdirectory. This file contains an ASCII version of the function. This design allows you to access and edit your package functions easily.

You can add functions to your package by copying the ASCII source to the **R** directory. The files should have an **.S** or **.ssc** (for S code), a **.q**, or an **.R** file extension; if not, they are ignored in the package build. From within Spotfire S+, you can add to the **R** directory with a call like the following:

```
dump("funabc",  
  "<path_to_packagedir>/<packagename>/R/funabc.s")
```

man Directory

The **man** subdirectory contains **.Rd** format documentation files for the objects in the package. That is, if you use the package.skeleton function to create your package tree, the **man** directory contains a template **.Rd** file for each object specified in the list argument. These are created by calling the prompt.Rd function on each object. The documentation files to be installed with the package must also start with a (lower or upper case) letter, and have the extension **.Rd**. Note that all user-level objects in a Spotfire S+ package should be documented; if a package **pkg** contains user-level objects which are

for internal use only, it should provide a file **pkg-internal.Rd** which documents all such objects, and clearly states that these are not meant to be called by the user.

You can create help files for functions or data sets (if you have any to include). We discuss each in the following sections.

Creating Help Files for Functions

After you generate the help files, you can edit them in your favorite text editor. The **fun1.Rd** help file looks like the following, with a description following the tag.

```
\name{fun1}
    the basename of the .Rd file.
\alias{fun1}
    the topics (or functions) the file documents. Note there must
    be an \alias entry for each topic.
\title{title information for fun1}
    the title information for the help file.
\description{description of what fun1 does}
    a concise (1-5 lines) description of the function.
\usage{fun1(x)}
    a synopsis of the function(s) and variables documented in the
    file. You can include usage for other objects documented here.
\arguments{\item{arg_i}{Description of arg_i}}
    a description of each of the function's arguments, using an
    entry of this form.
\details{more details than the description above}
    include more details, if relevant.
\value{value returned}
    short description of the value returned. If it is a list, use
    \item{comp1 }{Description of 'comp1'}
    \item{comp2 }{Description of 'comp2'}
\references{put references to the literature/web site here}
    include any URLs or other relevant information.
\author{who you are}
```

While not required, we encourage you to use this tag to correctly attribute your work to yourself (and co-authors).

```
\note{further notes}
```

```
make other sections like "Warning" with
\section{Warning}{....}
```

```
\seealso{objects to See Also}
```

```
pointers to related Spotfire S+ objects, using
\code{\link{...}}
```

```
\examples{example code}
```

```
this should be directly executable. This includes a definition
of the function as currently defined
```

```
\keyword{kwd}
```

```
at least one, from doc/KEYWORDS. This kwd string maps to
the Table of Contents for Windows files.
```

When you install a package (or build a binary package), the **.Rd** files are converted to the appropriate format for the particular platform. On Windows, a compiled help object (**.CHM**) file is created in the top level directory of the package. On UNIX/Linux, the **.Rd** files are converted to HTML, and they appear under **.Data/___Hhelp** in the top level directory of the package.

Creating Help Files for Data Sets

An **.Rd** help file is created for data sets listed in the `package.skelton` list argument.

If `dataset` is a data frame, you get a different skeletal help file generated. For example, look at the help file for `rivers`:

```
\name{rivers}
```

```
\docType{data}
```

```
\alias{rivers}
```

```
\title{Lengths of Major North American Rivers}
```

```
\description{
```

```
  This data set gives the lengths (in miles) of 141
  \dQuote{major} rivers in North America, as compiled by the
  US Geological Survey
}
```

```
\usage{data(rivers)}
```

```
\format{A vector containing 141 observations.}
```

```
\source{World Almanac and Book of Facts, 1975, page 406.}
```

```
\references{
  McNeil, D. R. (1977) \emph{Interactive Data Analysis}.
  New York: Wiley.
}
\keyword{datasets}
```

Note

Package authors also should consider creating vignettes for their packages. CSAN provides a package for SWeave, an application for integrating Latex and Spotfire S+ documentation, reports, and analyses for packages. For more information, and to download the SWeave package, see the CSAN Web site (<http://spotfire.tibco.com/csan>).

src Directory

C, C++, or FORTRAN code is stored in the **src** directory. The recognized file extensions are:

- **.c** - C code.
- **.cxx** - C++ code.
- **.f** - FORTRAN code.

When you install the package (or build a binary archive using the `--binary` option to the `build` utility), the code in **src** is compiled and linked into a shared library called **s.dll** on Windows or **S.so** on UNIX/Linux. The shared library is moved to the top level directory of the package, and note that it is automatically loaded into Spotfire S+ when the package is attached with the `library` command.

java Directory

The following details how to include Java code in a package, and how **.jar** files are put into `<pkg>/jar`. When this **.jar** file is created, it is loaded when `library(pkg)` is invoked:

1. Put your **.java** files in a directory `<pkg>/java/src`.

Add your **.properties** files, **.gif** files, or any other types of files you want to include in the same directory. Note that you likely will have to make subdirectories to conform to Java conventions.

2. If you have pre-built **.jar** files that your code depends upon, put these **.jar** files in `<pkg>/java/prebuiltjars`.
3. `Splus CMD build --binary`

or

```
Splus CMD INSTALL
```

compiles the **.java** files under **src** (putting the **.jar** files in **prebuiltjars** in the classpath of the compiler). It then puts all the **.class** files and any non-**.java** files under **src** into a new **.jar** file called **<pkg>.jar**. It then copies **<pkg>.jar** and all the **.jar** files in **prebuiltjars** to a newly made directory, **<pkg>/jars**.

When `library(pkg)` loads a package containing a **jars** subdirectory, it puts each **.jar** file into the Spotfire S+ classpath (starting Java if needed), so that calls to `.JavaMethod()` finds the methods.

The package system expects that your Java compiler is called `javac`, and the **.jar** file creator is called `jar` (same as Sun's JDK). It expects that the JDK **bin** directory containing these commands is in your path so that `javac --help` and `jar --help` function correctly.

If you want to add extra arguments to the Java compiler, include a makefile called **<pkg>/java/Makevars** (and not **<pkg>/src/Makevars**).

The build system uses the following two variables (if defined):

```
PKG_JAVACFLAGS
```

Can contain extra compiler flags (e.g., `-verbose` and `-g`), so the syntax looks like this:

```
PKG_JAVACFLAGS=-verbose -g.
```

Note these flags tend to be compiler-specific, so using them in a distributed package makes it less portable but more useful during development.

```
PKG_JAVAC_CLASSPATH
```

Lists **.jar** files not in **prebuiltjars** that are needed during the build. The variable `$HOME` is predefined so `PKG_JAVAC_CLASSPATH` can be used to refer to **.jar** files distributed with Spotfire S+. For example:

```
PKG_JAVAC_CLASSPATH=$(HOME)/java/jre/libext/jaxp.jar;  
$(HOME)/java/jre/libext/batik.jar
```

You can define other variables to use in `PKG_JAVAC_CLASSPATH`:

```
MY_JAR_DIR=C:/My jars;  
PKG_JAVAC_CLASSPATH=$(MY_JAR_DIR)/myFirstClass.jar;  
$(MY_JAR_DIR)/mySecondClass.jar
```

The build system splits PKG_JAVAC_CLASSPATH by semicolons (;), colons (:), or commas (,) before expanding the variables so the above works on Windows.

You can edit SHOME/cmd/Makevars_unix or SHOME/cmd/Makevars_windows_MS to redefine the values of JAVAC (the compiler), JAVAC_CLASSPATH_FLAG (the name of compiler flag used to introduce list of **.jar** files in the classpath), and JAVAC_CLASSPATH_SEP (the symbol used to separate entries in the classpath list, generally colon (:)) on Unix and semicolon (;) on Windows).

As of this release, TIBCO has limited support for using another compiler.

CONVERTING A PACKAGE FROM R TO SPOTFIRE S+

R packages from the Comprehensive R Archival Network (CRAN) can be Spotfire S+ converted to Spotfire S+ packages.

In some cases, the source package from CRAN installs and runs under Spotfire S+ without any changes; in general, some changes are required. However, a Windows binary zip archive of an R package will not run under Spotfire S+.

Getting An R Source Package

R packages can be found at the CRAN site:

<http://cran.r-project.org>

The left sidebar has a **Packages** link, and you can download the **.tar.gz** source file for the package.

The contents of CRAN are mirrored at many sites around the world. You are encouraged to download files from one of the mirror sites, and the location of these sites is available at

<http://cran.r-project.org/mirrors.html>

You can get a listing of all the packages in CRAN and download the source archive from within Spotfire S+:

1. Load the pkgutils library:

```
library(pkgutils)
```

2. List the available packages, using the repos argument to specify CRAN as the URL and type as source:

```
ap <- available.packages  
  (repos="http://cran.r-project.org", type="source")
```

The type argument is set by default in UNIX[®]/Linux[®] to source.

3. Look at package names:

```
ap[, 1]
```

4. Download a package source archive and save it to the current directory. Set type to source which is the default on UNIX/Linux (win.binary for Windows):

```
download.packages(repos="http://cran.r-project.org",  
"fBasics",destdir='.',type="source")
```

where `fBasics` is the name of the package to download.

In this example, this downloads **fBasics_221.10065.tar.gz** (the version available as of this writing) to the current directory.

Creating a Spotfire S+ Package from an R Package

1. After downloading an R source package from CRAN, unpack the package with the `tar` command. Example:

```
tar -xzf fBasics_221.10065.tar.gz
```

This creates a directory called **fBasics** that contains subdirectories **man**, **R**, and possibly other subdirectories.

As noted in the section Windows on page 12, you will likely need to install the `tar` utility.

2. Modify the files under **fBasics** to run under Spotfire S+. Modifications depend on what the package contains. The construct

```
if(is.R())
```

can be used to specify code for conditional execution in R or Spotfire S+.

3. The section Differences Between S-PLUS and R Packages on page 39 and section Trouble-Shooting Porting R Packages to S-PLUS Packages on page 43 indicate some areas to look out for.
4. Update the **DESCRIPTION** file to indicate changes made, porting to Spotfire S+, and so on. Add the `Dialect` flag if it does not exist, setting `S-PLUS` as the value. (If you modified the code so it still runs under R, then also add `R` as a value for the tag.)
5. After making any changes you can then run the check utility on the package directory (while in the directory that contains **fBasics**). This check utility also runs `*.t` files in the `[packagename]/tests` directory, and reports if `do.test()` makes any comments on them:

```
Splus CMD check fBasics
```

If everything is OK, you can build a source package archive:

```
Splus CMD build fBasics
```

This creates a compressed archive called **fBasics_version.zip** (if running Windows) or **fBasics_version.tar.gz** (UNIX/Linux), where **version** is the version number from the `Version` line in the package's **DESCRIPTION** file.

You can install from this compressed tar archive with this:

```
Splus CMD INSTALL  
-l mylib fBasics_version.zip
```

on Windows or

```
Splus CMD INSTALL  
-l mylib fBasics_version.tar.gz
```

on UNIX/Linux, where `mylib` is an existing directory in which you can install the package. You must have permission to write in that directory.

If you are satisfied with the conversion of the package to Spotfire S+, you may want to submit the package to the Spotfire S+ CSAN site. See section [Submitting a Package](#) on page 11 for information on how to do this.

Build Scripts

The following scripts are used to build pieces of a package. The `INSTALL` script (and `build` script with the `-binary` flag) calls these scripts. When porting a package from R to Spotfire S+, you might find these scripts useful to do the porting work incrementally. They are all called from a command shell with the syntax:

```
Splus CMD (scriptname) <scriptargs>
```

Most of these scripts work by default within the package directory, not in the directory containing the package directory. See their individual help topics for more information.

Table 1.6: *Package compilation scripts.*

Script name	Description
src2bin	Creates the binary version of a package from source files. This utility installs S code and data, compiles C, C++ and FORTRAN code to create a shared/dynamic library, and formats and installs help files.
HELPIINSTALL	Installs Spotfire S+ help files from .Rd source files into the .Data directory of a specified destination directory.
SINSTALL	Installs Spotfire S+ code or data objects from source files into the .Data directory of a specified destination directory.
DATAINSTALL	Install Spotfire S+ code or data objects from source files into the .Data directory of a specified destination directory.
SHLIB	Creates a shared library from C, C++, or FORTRAN source files. The source files are compiled and resulting object files are linked to make a shared library. A shared library is also known as a shared object, dynamic library, or dynamic shared object.

Differences Between Spotfire S+ and R Packages

There are specific differences between R and Spotfire S+ packages that do not make them completely interchangeable. The key differences:

- To do almost any work with Spotfire S+ packages, you need to load the pkgutils library section.
- At this time, there is only one repository for Spotfire S+ packages, CSAN (<http://spotfire.tibco.com/csant>). There is no system in place to select a mirror as there is in R.
- Spotfire S+ package system does not support bundles, translations, or front ends.

- C and FORTRAN code is automatically loaded in Spotfire S+ packages when the package is attached with the library function. This is done through the Spotfire S+ feature that automatically loads the file named **S.so** or **S.dll** in the directory being attached. This means there is no need to explicitly write a **.First.lib** function that loads the packages shared library.
- Spotfire S+ package system uses only the current Spotfire S+ help system.

The **.Rd** files are converted to HTML on UNIX/Linux, and to a **.chm** file on Windows. LaTeX help files are not supported at this time.

- The Sweave system is not supported.
- The **data** directory in a Spotfire S+ source package can only contain ASCII data objects created with `dump()`, space-delimited data files (**.txt**), or comma-delimited files (**.CSV**). R binary objects (**.rda** files) are ignored by Spotfire S+.
- Spotfire S+ does not have NAMESPACES so any references to them in a package needs to be modified to work in Spotfire S+.
- The first argument to `.Call` must be a a string in Spotfire S+. R allows the first argument to `.Call` to be a variable.
- The default storage mode for a numeric value with no decimal place in Spotfire S+ is an integer, while in R it is a double. For example, in Spotfire S+:

```
x <- 3
storage.mode(x)
[1] "integer"
```

While in R:

```
x <- 3
storage.mode(x)
[1] "double"
```

If you parse the R functions with `set.parse.mode("R")`, the numeric values without decimal points in the R functions are parsed as doubles, for example, in Spotfire S+:

```
set.parse.mode("R")
x <- 3
storage.mode(x)
[1] "double"
x
```

If you have a ***.R** file that can only be read correctly when parsed in R mode (because it uses underscores in the name or it relies on "1", a double precision number), you can parse it and deparse it (with `dump` or `deparse`) to make a new file that can be read identically in either R or Spotfire S+ mode.

- The first component in the return list from the `integrate` function is named "integral" in Spotfire S+ and "value" in R. Portable code that uses `integrate` should access the first value in the list by position (`z[[1]]`) instead of by name (`z$integral` or `z$value`).

Porting Tools

A useful tool for porting R packages to Spotfire S+ is the `unresolvedGlobalReferences` function introduced in Spotfire S+ 8.0. This function looks for undefined functions and data in Spotfire S+ or R source files. It returns the names of all undefined items and the names of the files and functions where they are referenced.

The `unresolvedGlobalReferences` function can look at list of source files, or you can point the function to a directory containing the source files. The function analyzes all files in the directory that end with **.q**, **.ssc**, **.S**, or **.R**.

When you port a package from R to Spotfire S+, you can call `unresolvedGlobalReferences` with the `dir` argument set to the R subdirectory in the package source tree.

The following example shows a partial listing of calling the function `unresolvedGlobalReferences(dir="R")` in the **randomForest** package directory that was just downloaded from CRAN:

```
unresolvedGlobalReferences("R")
.
.
$"R/classCenter.R#classCenter":
[1] "max.col" "mapply"
.
.
```

```

$"R/classCenter.R#classCenter#<anonymous-1>":
[1] "cls"

$"R/classCenter.R#classCenter#<anonymous-2>":
[1] "idx"  "label"

$"R/classCenter.R#classCenter#<anonymous-3>":
[1] "x"
.
.
```

This shows the `classCenter` function defined in the file **classCenter.R** referencing the functions `max.col` and `mapply`. These functions are not defined within the package files, nor do they appear in the current Spotfire S+ search path. The `mapply` function is contained in the `pkgutils` library; If you attached the library before running `unresolvedGlobalReferences`, `mapply` would not be flagged.

The `max.col` function is defined in the `MASS` library that ships with Spotfire S+. To make a portable `classCenter` function that would run in both Spotfire S+ and R, one would add the following lines before the two functions were called in the `classCenter` function:

```

if(!is.R()) {
  if(!existsFunction("max.col")) library(MASS)
  if(!is.R() && !existsFunction("mapply")) library(pkgutils)
}
```

The `<anonymous-1>`, `<anonymous-2>` and `<anonymous-3>` references in `classCenter` typically indicate use of R scoping rules in calls to a function in the `apply` family (`lapply`, `sapply`, `apply`, and so on).

Typically, you can fix these by passing the function arguments explicitly to the function being called by the `apply` function.

To make a portable fix for the unresolved `cls` object, use the following code:

```

if(is.R()) {
  ncls <- sapply(clsLabel, function(x)
    rowSums(cls == x))
} else {
  ncls <- sapply(clsLabel, function(x, cls = cls)
    rowSums(cls == x), cls = cls)
}
```

See the help file for `unresolvedGlobalReferences` for more information and examples.

Trouble-Shooting Porting R Packages

R and Spotfire S+ are different dialects of the S language; each dialect has capabilities that are not implemented in the other dialect. The "R and S" section of the R FAQ, available at

<http://www.ci.tuwien.ac.at/~hornik/R/R-FAQ.html#R-and-S>

has a long section on the differences. Some of the key differences that you might encounter when trying to get an R package to work in Spotfire S+ are listed here.

Scoping Rules

Spotfire S+ functions search for objects in the current frame, frame 1, frame 0, and then the attached databases. R searches for objects in the following order:

1. Current frame ("environment")
2. The environment of the function in which the current function is *defined*, not called.
3. The environment in which the definer of the current function was defined, etc., until it gets to the global environment.

This difference often causes problems with calls to the `lapply` family of functions. In Spotfire S+, you need to pass all objects included in the FUN function as arguments to FUN, and include those arguments by name in the `lapply` call. For example:

```
nsamp <- 10
lapply(z, function(x, nsamp) {
  mean(sample(x, size=nsamp, replace=T))
}, nsamp=nsamp)
```

The scoping rule difference also shows up in calls to optimization functions, e.g., `optim()` inside of functions. You need to pass all objects referenced inside the function being optimized as arguments to that function.

For more information about dealing with scoping problems, see the section Porting Tools on page 41.

**Missing C/
FORTRAN
Functions**

Spotfire S+ and R contain different internal C functions and FORTRAN subroutines.

Sometimes the underlying code is the same but the function or subroutine name differs between the two systems. Any calls to C or FORTRAN code not included as source in the package should be checked.

Some known missing C code:

- R has a collection of *bessel* functions from *netlib* that are not yet in Spotfire S+.
- R has its *exponential random number generator* available for calling from C, but Spotfire S+ does not. The *uniform and gaussian random number generators* are available in Spotfire S+.

DIFFERENCES BETWEEN R AND SPOTFIRE S+

R and Spotfire S+ have certain differences that you should be aware of if you are working in both, or if you are translating packages from one to another.

You can find information about the differences on the Spotfire S+ Support Web site:

<http://support.tibco.com>

Also, you can use your Internet search engine to find other resources that list differences.

ADDING A GUI TO A WINDOWS[®] PACKAGE

2

Overview	50
Components of A Spotfire S+ GUI	51
Menus and Menu Items	52
Control Properties	53
FunctionInfo Object	53
The menu Function	53
The callback Function	53
GUI Example	55
Creating the Menu	55
Creating the Calling Function	57
Creating the Dialog	58
Creating the FunctionInfo Object	61
Creating the Callback Function	62
Loading the GUI at Startup	64
Removing the Menu at Shutdown	65
The Resulting Dialog	66
Extending The Example	67
Additional Properties	67
Dialog Pages	68
Updating the Menu Function	68
Updating the FunctionInfo Object	69
The Resulting Dialog	71
Building The GUI During Installation	72
Basic Code Structure	72

OVERVIEW

You can program the Spotfire S+ for Windows Graphical User Interface (GUI) using the Spotfire S+ language; therefore, you can add a GUI to a package to give users access to the functions you have created via a GUI as well as via the command line interface. Using the functions that control GUI customization, you can create menus, menu items, and control properties (such as drop-down lists for selecting options, check boxes, input fields, and grouping structures for grouping properties on the pages of a dialog).

- The special structure, the `FunctionInfo` object, aligns the control properties in the dialog. These controls accept input from the user, with the arguments of a function that is called when the user clicks **OK** or **Apply**.
- The special function, `callback`, is associated with the dialog that manages information in the control properties. For example, if you select a data frame as the object to apply your function to, you might want to display the column names in a drop-down list for selection. In this case, the `callback` function is used to change the content of a drop-down list dynamically, depending on the content of another property.

This section is designed to:

- Give you an introduction to building Spotfire S+ GUIs.
- Describe the required directory structure where your GUI creation code goes to ensure it is loaded during package build and installation.

If you have created a custom GUI for your package already, there is a simple directory and file structure that you must follow in order for it to be included in the installed package. To understand that structure see section Building The GUI During Installation on page 72. If you haven't created a GUI but would like to, the following sections provide some guidance for doing that.

COMPONENTS OF A SPOTFIRE S+ GUI

The components of a Spotfire S+ GUI are listed in Table 2.1. Typically all the components listed in Table 2.1 are included in a GUI implementation for a package. Usually a drop-down *menu* is added to the Spotfire S+ menu bar containing *menu items* for selecting various operations. One operation may be to open a custom dialog with *control properties* for specifying data and setting options prior to running the *menu function* associated with the dialog. While interacting with the dialog, the *callback function* manages the information in the control properties and can control which properties are enabled or disabled.

Table 2.1: *Components of a Spotfire S+ GUI*

Component	Description
Menu	A drop-down menu for containing menu items.
Menu Item	A selectable item in a menu which initiates an action such as opening a dialog or displaying a document.
Control Property	A GUI control contained on a dialog for user input. This information is passed to the calling function when OK or Apply is selected.
FunctionInfo Object	An object that aligns the control properties in a dialog with the arguments of the function call when OK or Apply is selected.
Menu function	The function called when OK or Apply is selected on the associated dialog.
Callback function	The function controlling dynamic information in the dialog control properties.

Menus and Menu Items

Menus are containers containing menu items, which are designed to execute actions, such as opening a dialog or displaying a document when they are selected. Typically, a package with a GUI adds a menu to the Spotfire S+ menu bar containing an assortment of menu items that initiate actions when they selected. Typical actions are:

- Open a dialog for running a function.
- Open a demo script window.
- Open the package help files.
- Open the user documentation.
- Open an HTML page.

You can view a list of all the MenuItem objects in a Spotfire S+ session with the following simple command:

```
> guiGetObjectNames("MenuItem")
...
[83] "SPlusMenuBar$File"           "SPlusMenuBar$File$New"
[85] "SPlusMenuBar$File$Open"      "SPlusMenuBar$File$Close"
[87] "SPlusMenuBar$File$Close_All" "SPlusMenuBar$File$Separator1"
...
```

There are many MenuItem objects: over 600 in a standard Spotfire S+ 8.x session. Of particular interest are the ones that begin with SPlusMenuBar. These are the MenuItem properties responsible for defining the menu bar in Spotfire S+. You can use them to locate your custom menu at a logical location in the Spotfire S+ menu bar.

The following example specifies an index number for placing a menu on the menu bar. The function guiGetPropertyValue returns the index in the menubar for a given MenuItem.

```
> guiGetPropertyValue("MenuItem",
                      Name = "SPlusMenuBar$File",
                      PropName = "Index")
[1] "1"

> guiGetPropertyValue("MenuItem",
                      Name = "SPlusMenuBar$Statistics",
                      PropName = "Index")
[1] "12"
```

Note the use of \$ in the property name for designating different *components* of the menu bar.

Control Properties

Control properties contained in a dialog mostly are designed to take input from the user and pass it to a function for execution. However, a few properties (for example Group or Page) are for organizing other control properties. Allowable control properties include text fields, drop-down lists, check boxes, sliders, radio button groups, and pages. To see a complete list of properties, type

```
> sort(guiGetPropertyOptions("Property", "DialogControl"))
[1] "Button" "Check Box" "Color List" "Combo Box"
[5] "Float" "Float Auto" "Float Range" "Float Slider"
...
```

Creating a custom dialog amounts to assembling a set of calls to `guiCreate` with appropriate arguments to define each control and set defaults which are shown when the dialog is displayed. Use the documented examples and some experimentation to get started.

FunctionInfo Object

The `FunctionInfo` object aligns the control properties on a dialog with the arguments of the function to be called when the user clicks **OK** or **Apply**. Also, this object determines the arrangement of the properties on the dialog, the function called when the user clicks **OK** or **Apply**, and the callback function controlling dynamic information in the properties of the dialog.

The menu Function

When the user clicks **OK** or **Apply**, the menu function is called on the associated dialog. When it is created, the `FunctionInfo` object requires the menu function name.

The callback Function

The `callback` function allows dynamic control of the information in the properties of the dialog associated with the menu function. For example, with the callback function you can:

- Define content for properties when the dialog is first opened and initialized.
- Enable and disable properties, depending on options selected.
- Fill a drop-down list with values based on another selection (for example, column names of a selected data frame).

Some of the useful functions for managing property content are listed in Table 2.2.

Table 2.2: *Useful functions for managing property content in a dialog.*

Function Name	Description
cbIsInitDialogMessage	Returns TRUE if the user opens the dialog.
cbIsOkMessage	Returns TRUE if the user clicks OK in the dialog.
cbGetActiveProp	Returns a character string containing the name of the active property (that is, the property taking input).
cbGetCurrValue	Returns the value of a given property.
cbSetCurrValue	Sets the value of a given property.
cbSetEnableFlag	Enables (TRUE) or disables (FALSE) a property specified by name.
cbSetOptionList	Sets the values of a drop-down or scrollable option list.

GUI EXAMPLE

The best way to learn GUI programming in Spotfire S+ is to use it in an example. The following example demonstrates a simple but practical dialog to fit a linear model. The steps include:

1. Creating a menu item to the main Spotfire S+ menu bar.
2. Creating the function that the dialog called when the user clicks **OK** or **Apply**.
3. Creating a dialog to take user input for selecting data and setting options.
4. Creating the `FunctionInfo` object aligning the dialog with the function to be called and the callback function.
5. Creating a callback function for managing control property content dynamically.
6. Loading the GUI at startup by creating the `.First.lib` function (which adds the menu to the menu bar and loads the properties at the time the library is attached to the session).
7. Removing the menu at shutdown by creating the `.Last.lib` function (which removes the menu structure when the library is detached from the session).

Creating the Menu

Create a menu structure by creating a `Menu` property, a container for menu items, and a `MenuItem` property, which specifies the function to be called. In the example, the menu is created in an argumentless function that is called from the `.First.lib` function when the library section is attached to a Spotfire S+ session.

In the function below, locate the **Statistics** menu and add the new menu just after it. (If you do not provide a location for the menu, it is placed at the end of the menu bar.)

```
loadLmFitMenu <- function(){
  statMenuLoc <-
    guiGetPropertyValue("MenuItem",
                        Name = paste(guiGetMenuBar(),
                                     "Statistics", sep="$"),
                        PropName = "Index")
```

```
guiCreate("MenuItem",
          Name = "SPlusMenuBar$MyGUI",
          Type = "Menu",
          Action = "None",
          MenuItemText = "My GUI",
          StatusBarText = "My menu",
          Index = as.numeric(statMenuLoc) + 1,
          OverWrite = F,
          EnableMenuItem = T)

guiCreate("MenuItem",
          Name="SPlusMenuBar$MyGUI$lMFitExample",
          Type="MenuItem",
          Action="Function",
          Command="menuLmFit",
          MenuItemText="Linear Model")
invisible()
}
```

Comments:

- In the function above, the `statMenuLoc` is the index location of the **Statistics** menu on the Spotfire S+ menu bar. That is computed, so you can place the custom menu immediately following it.
- The first menu property is of `Type = Menu` located at `Index=as.numeric(statMenuLoc) + 1` (that is, just following the **Statistics** menu.)
- The name of the **Menu** property is `SPlusMenuBar$MyGUI`, which embeds it in the Spotfire S+ menu bar.
- `Action = "None"` implies that `SPlusMenuBar$MyGUI` is a container only and initiates no action.
- The second menu property of `Type="MenuItem"` with `Action="Function"` opens the dialog associated with the function specified by `Command="menuLmFit"` when it is selected from the menu bar.
- The `MenuItemText` for both menu properties corresponds to the string displayed in the menu bar. The `StatusBarText` is displayed in the Spotfire S+ status bar when mouse focus is on the menu item.

Creating the Calling Function

Creating the calling function goes hand-in-hand with creating the dialog. The arguments passed to the function being called must correspond one-to-one with control properties on the dialog taking input. The one-to-one correspondence is specified by the **FuntionInfo** object discussed below.

The calling function name begins with **“menu”** to indicate it is called from a menu.

```
menuLmFit<-function(x,
                    y,
                    data,
                    method = "qr",
                    removeNA = T,
                    saveAs = "last.lmFit"){

# Make x and y a formula
x <- unlist(unpaste(x, sep = ","))
form <- formula(parse(text = paste(y, paste(x,
                                     collapse = " + "),
                                     sep = " ~ "))

# Fit model conditional on removal of NAs
if(removeNA)
    fit <- lm(form, data = get(data), method = method,
              na.action = na.exclude)
else fit <- lm(form, data = get(data), method = method)
# Save the result
assign(saveAs, fit, where = 1)
# Return the result invisibly
invisible(fit)
}
```

Comments:

- The function is relatively simple, with only six arguments.
- `x` represents independent variables and `y` represents the dependent variable. The call to `unlist(unpaste(...))` converts a single string with comma delimiter separating values into a vector of strings, one for each `x` variable.
- `method` is an optional fitting method defaulting to “qr” the same as for the `lm` function.

- `removeNA` is a logical indicating whether NAs should be removed before attempting to execute the function.
- `saveAs` specifies the name of a Spotfire S+ object for saving the result.

Creating the Dialog

Creating a dialog is a matter of creating the properties contained in the dialog. After the `FunctionInfo` object is created specifying the properties its associated dialog contains and the calling and callback functions, the dialog is constructed from its component properties.

Data Selection

```
### Data Selection ###
guiCreate("Property",
          Name="lmFitExampleDataSet",
          DialogPrompt="Data Set:",
          Type = "Normal",
          DialogControl="Combo Box",
          DefaultValue = "",
          UseQuotes = T)

guiCreate("Property",
          Name="lmFitExampleY",
          DialogPrompt="Dependent:",
          Type = "Normal",
          DialogControl="List Box",
          DefaultValue = "",
          UseQuotes=T)

guiCreate("Property",
          Name="lmFitExampleX",
          DialogPrompt="Independent:",
          Type = "Normal",
          DialogControl="Multi-select List Box",
          DefaultValue="",
          UseQuotes = T)

guiCreate("Property",
          Name="lmFitExampleDataGroup",
          Type = "Group",
          DialogPrompt = "Data Selection",
          PropertyList = paste(c("lmFitExampleDataSet",
```

```

"lmFitExampleY",
"lmFitExampleX"),
collapse = ", ") )

```

Comments:

- The first four properties provide controls for data selection: the data set, the dependent variable, the independent variable, and a group structure to contain them.
- The Data Set property uses a Combo Box control, which allows the user either to select from a list or type the name of the data set into combo text box if the data set does not appear in the drop-down list. The callback function fills the Combo Box with all data frames in position 1 of the search list of the working data directory. If a data frame is in one of the other search positions, it does not show up in the list. In that case, its value is a quoted string.
- The Dependent (variable) property uses a List Box control, which displays only the variables in the data frame. The text box containing the List Box is not editable, so the user can select only variables in the data frame. Its value is a quoted string.
- The Independent (variables) property uses a Multi-select List Box control, which displays only the variables in the data frame; the user can select multiple variables by pressing the CTRL key while selecting variables from the list. The Multi-select List Box text box is not editable, so the user can select only variables in the data frame. Its value is a quoted string.
- The Group property contains the other three properties in a visible frame to separate them from the rest of the properties on the dialog.

Options

```

### Options ###
guiCreate("Property",
          Name="lmFitExampleMethod",
          DialogPrompt="Method:",
          Type = "Normal",
          DialogControl="List Box",
          DefaultValue = "qr",

```

```
        OptionList = "qr, svd, chol",
        UseQuotes=T)

guiCreate("Property",
        Name="lmFitExampleRemoveNAs",
        DialogPrompt="Remove NAs",
        Type = "Normal",
        DialogControl="Check Box",
        DefaultValue = T,
        UseQuotes=F)

guiCreate("Property",
        Name="lmFitExampleOptionsGroup",
        Type = "Group",
        DialogPrompt = "Options",
        PropertyList = paste(c("lmFitExampleMethod",
                               "lmFitExampleRemoveNAs"),
                             collapse = ", ") )
```

Comments:

- The next three properties specify the fitting options: the fitting method, the handling of NAs, and a group structure to contain them.
- The Method property uses a List Box to restrict the choices to one of three: “qr”, “svd” or “chol”. The default value is “qr”. Its value is returned as a quoted string. Note that the OptionsList (the values displayed in the drop-down list) is a single string of comma-separated values. This is the required format for specifying options contained in a List Box.
- The RemoveNAs property is a Check Box indicating whether to remove NAs before fitting the model. Its value is an unquoted T if the box is checked; otherwise it is F.
- The Group property contains the other two properties in a visible frame to separate them from the rest of the properties on the dialog.

Saving the Fit

```
### SaveAs ###
guiCreate("Property",
        Name="lmFitExampleSaveAs",
        DialogPrompt="Save As",
```

```

DialogControl="String",
DefaultValue="last.lmFit",
UseQuotes = T)

guiCreate("Property",
    Name="lmFitExampleSaveFitGroup",
    Type = "Group",
    DialogPrompt = "Save Fit",
    PropertyList = paste("lmFitExampleSaveAs") )

```

Comments:

- The last two properties specify the saved object, including the actual name and a group structure to contain it.
- The SaveAs property uses a String text box where the user can type the name of the object for saving the result. It defaults to "last.lmFit" and returns a quoted string.
- The Group property contains the SaveAs property in a visible frame to separate it from the rest of the properties on the dialog.

Creating the FunctionInfo Object

The FunctionInfo object contains four critical pieces of information:

- The name of the calling function (Function), or the *menu* function.
- The name of the callback function (CallbackFunction).
- The list of properties (PropertyList) in the order to be displayed in the dialog.
- The list of properties arranged in the exact order of the arguments in the calling function (ArgumentList).

```

### FunctionInfo Object ###
guiCreate("FunctionInfo",
    Name = "menuLmFit",
    Function = "menuLmFit",
    CallbackFunction = "backLmFit",
    DialogHeader = "Fit Linear Model",
    StatusString = "Fits multi-variable linear model",
    PropertyList =
        paste(c("SPropInvisibleReturnObject",

```

```
      "1mFitExampleDataGroup",
      "1mFitExampleOptionsGroup",
      "1mFitExampleSaveFitGroup"),
      collapse = ", "),
ArgumentList <-
  paste(c("#0=SPropInvisibleReturnObject",
          "#1=1mFitExampleX",
          "#2=1mFitExampleY",
          "#3=1mFitExampleDataSet",
          "#4=1mFitExampleMethod",
          "#5=1mFitExampleRemoveNAs",
          "#6=1mFitExampleSaveAs"),
        collapse = ", ")
WriteArgNames = T)
```

Comments:

- The `FunctionInfo` object Name *must* match the name of the function.
- The `DialogHeader` string specifies the title of the dialog.
- The `WriteArgNames = T` argument indicates that the argument names are written into the function call, which makes deciphering the call resulting from **OK** or **Apply** much easier.
- The `SPropInvisibleReturnObject` property is in an internal property that preserves space for a return object.

Creating the Callback Function

The callback function dynamically controls property content. It is used for filling drop-down lists with dynamic content (for example, variable names), for enabling or disabling controls, and for changing defaults depending on user selection. Also, it can be used to pop up message boxes based on user selections.

With every user interaction with a dialog, some information changes. That information, and all of the information contained in the dialog, is available in the form of a data frame that is passed to the callback function associated with the dialog. The callback function is designed to take the property data frame as its only argument and return the property data frame when it completes. Any changed values are reflected in the dialog. The callback function is called with every interaction with the dialog.

The property data frame contains a row for each control property and the following columns:

- `message` - Message codes indicating, for example, dialog initialization, clicking of **OK** or **Apply**, dialog rollback.
- `value` - The value contained in the property.
- `enable` - T or F depending on whether the property is enabled.
- `optionList` - the values in a List Box or Combo Box.
- `prompt` - The controls label.

The callback function for our `lmFitExample` dialog is listed below. (Comments are displayed below the code listing.)

```
backLmFit <- function(data){
  initialmsg <- cbIsInitDialogMessage(data)
  activeprop <- cbGetActiveProp(data)
  ### actions based on initializing the dialog
  if(initialmsg){
    data <- cbSetOptionList(data, "lmFitExampleDataSet",
                           paste(objects(class = "data.frame"),
                                 collapse = ", "))
  }
  ### actions based on selecting the data set
  if(activeprop == "lmFitExampleDataSet"){
    if(exists(cbGetCurrValue(data,
                              "lmFitExampleDataSet"))){
      data <- cbSetOptionList(data, "lmFitExampleY",
                             paste(colIds(get(cbGetCurrValue(data,
                                                              "lmFitExampleDataSet"))),
                                   collapse = ","))
      data <- cbSetOptionList(data, "lmFitExampleX",
                             paste(colIds(get(cbGetCurrValue(data,
                                                              "lmFitExampleDataSet"))),
                                   collapse = ","))
    } else {
      guiDisplayMessageBox(paste(cbGetCurrValue(data,
                                                "lmFitExampleDataSet"),
                                "does not exist. Please enter another data set."),
                           button = c("Ok"),
                           icon = c("error"))
    }
  }
}
```

```
    }  
    data  
}
```

Comments:

- The function starts by saving two objects: `initialmsg` and `activeprop`. The `initialmsg` value is `T` if the dialog is being initialized (that is, just opening). `activeprop` contains the property name of the property being modified. You use both `initialmsg` and `activeprop` to create conditional expressions.
- The first conditional expression depends on whether the dialog is being initialized. If it is, then the `lmFitExampleDataSet` property is filled with the names of all data frames in search list position one. Note that those names must be in a single string with values separated by commas.
- The second conditional expression checks to see if the active property is `lmFitExampleDataSet`. If it is, and if the data frame exists, both `lmFitExampleY` and `lmFitExampleX` are filled with the column IDs of the data frame, so the user can select the dependent and independent variables for the regression. If a data frame does not exist with the provided name, a message box is displayed indicating that the user must provide another data set name.
- The property `data` frame is returned and the dialog is updated, depending on changes made. All of this happens with each interaction with the dialog, so it is important to write the callback function efficiently so responsiveness is not compromised.

Loading the GUI at Startup

When the library section is attached, typically you can load the custom GUI with a `.First.lib` function. Assuming the menu and callback functions are saved in the library, the menu is loaded first, and the properties are loaded second. The order is important: functions, then menu, and then properties. The functions must exist so the properties (the menu items and `FunctionInfo` objects) that refer to them can find them when they are loaded into the Spotfire S+ GUI.

If the GUI is built as part of a package, the `.First.lib` function must have a particular form. The package build process looks for a certain directory structure, pulls the GUI property file from that structure, and installs the properties in the **.Prefs** directory for the installed package. The functions defined in **swingui/R** are installed in the package **.Data** directory.

```
.First.lib <- function(library, section){
  loadLmFitMenu()
  package.path <- file.path(library, section)
  prop.path <- file.path(package.path,
                        ".Prefs", "package.prp")
  info.path <- file.path(package.path,
                        ".Prefs", "package.fni")
  guiLoadDefaultObjects("Property",
                        FileName = prop.path)
  guiLoadDefaultObjects("FunctionInfo",
                        FileName = info.path)
}
```

Comments:

- The menu loads first, followed by the controls properties, and then the `FunctionInfo` object.
- The paths to the properties and `FunctionInfo` objects are fixed, relative to the package name. The properties are always saved in a file named **package.prp** and stored in the **.Prefs** directory for the package. The `FunctionInfo` objects are always saved in a file named **package.fni** and stored in the **.Prefs** directory.

Removing the Menu at Shutdown

To finish the GUI specification, you need a `.Last.lib` function, which removes the new menu structure when the library is detached. (It does not make sense to leave the menu in place if the functions in the package are not available to the session.)

```
.Last.lib = function(library, section, .data, where){
  if (is.sgui.app() && interactive() &&
      is.element("SPlusMenuBar$MyGUI",
                guiGetObjectNames("MenuItem"))){
    guiRemove("MenuItem",
              Name="SPlusMenuBar$MyGUI") }
}
```

}

Comment:

- The function is written to test whether the **MyGUI** menu is loaded. If so, the function removes the menu.

The Resulting Dialog

Figure 2.1 displays the resulting dialog after the menu and all the properties are loaded into Spotfire S+.

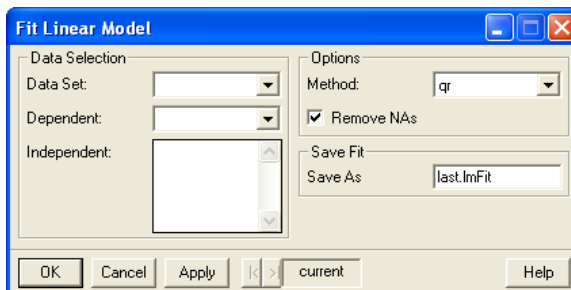


Figure 2.1: *The resulting dialog created in the previous sections.*

EXTENDING THE EXAMPLE

At this point, the implementation of `lmFitExample` is minimal, because it prints and plots nothing. Also, the call component of the resulting fit shows the model formula incorrectly. The example can be improved with a little more work: adding a second page to the dialog specifying output options, and fixing the saved call by constructing the formula. To accomplish these steps:

- Modify the menu function, `menuLmFit` (and therefore the `FunctionInfo` object) to take additional arguments.
- Add sufficient properties to specify the output via simple check boxes on a new page.

Additional Properties

Create two checkboxes: one for printing the summary of the fit, and one for plotting the diagnostics for the fit. (Group them for style.)

```
guiCreate("Property",
         Name="lmFitExamplePrintSummary",
         DialogPrompt="Print Summary",
         Type = "Normal",
         DialogControl="Check Box",
         DefaultValue = T,
         UseQuotes=F)

guiCreate("Property",
         Name="lmFitExamplePlotDiagnostics",
         DialogPrompt="Plot Diagnostics",
         Type = "Normal",
         DialogControl="Check Box",
         DefaultValue = T,
         UseQuotes=F)

guiCreate("Property",
         Name="lmFitExampleOutputGroup",
         Type = "Group",
         DialogPrompt = "Display Options",
         PropertyList = paste("lmFitExamplePrintSummary",
                             "lmFitExamplePlotDiagnostics",
                             collapse = ",") )
```

The additional properties do not need to be grouped, but grouping makes the second page more orderly.

Dialog Pages

Create the pages by creating new Page properties for each page of the dialog. The groups have been created already, so the pages are just containers for the groups.

```
guiCreate("Property",
          Name="lmFitExampleMainPage",
          Type = "Page",
          DialogPrompt = "Data/Methods",
          PropertyList = paste(c("lmFitExampleDataGroup",
                                "lmFitExampleOptionsGroup",
                                "lmFitExampleSaveFitGroup"),
                              collapse = ","))
```

```
guiCreate("Property",
          Name="lmFitExampleOutputPage",
          Type = "Page",
          DialogPrompt = "Output",
          PropertyList = "lmFitExampleOutputGroup")
```

Updating the Menu Function

The menu function has two additional arguments: `printSummary` and `plotDiagnostics`, which support the two new features and handle the formula more carefully so it appears properly in the saved call.

```
menuLmFit<-function(x,
                    y,
                    data,
                    method = "qr",
                    removeNA = T,
                    printSummary = T,
                    plotDiagnostics = T,
                    saveAs = "last.lmFit")
{
  # Make x and y a formula
  x <- unlist(unpaste(x, sep = ","))
  Args <- list(formula =
               formula(parse(text =
                             paste(y,
                                   paste(x, collapse = " + "),
```

```

        sep = " ~ ")@.Data,
    data = as.name(substitute(data)),
    method = method)

# fit model conditional on removal of NAs
if(removeNA)
  Args$na.action = as.name("na.exclude")
fit <- do.call("lm", Args)

# save result
assign(saveAs, fit, where = 1)

# print results
if(printSummary)
  print(summary(fit))

# plot diagnostics
if(plotDiagnostics)
  plot(fit)
# return result invisibly
invisible(fit)
}

```

Comments:

- The two additional arguments, `printSummary` and `plotDiagnostics`, are straightforward. They take logical values, and the code at the bottom of the function prints a summary of the fit or plots the fit if they are T.
- The greatest complication lies in handling the formula correctly. To do so requires constructing the arguments to `lm` as a named list, and then calling `lm` using the `do.call` function. The `do.call` function takes the name of the function as a character string and a list with the evaluated arguments.

**Updating the
FunctionInfo
Object**

Finally, update the `FunctionInfo` object to implement the Page grouping and the new function arguments.

```

guiCreate("FunctionInfo",
  Name = "menuLmFit",
  Function = "menuLmFit",

```

```
    CallbackFunction = "backLmFit",
    DialogHeader = "Fit Linear Model",
    StatusString = "Fits multi-variable linear model",
    PropertyList =
        paste(c("SPropInvisibleReturnObject",
                "lmFitExampleMainPage",
                "lmFitExampleOutputPage"),
              collapse = ", "),
    ArgumentList =
        paste(c("#0=SPropInvisibleReturnObject",
                "#1=lmFitExampleX",
                "#2=lmFitExampleY",
                "#3=lmFitExampleDataSet",
                "#4=lmFitExampleMethod",
                "#5=lmFitExampleRemoveNAs",
                "#6=lmFitExamplePrintSummary",
                "#7=lmFitExamplePlotDiagnostics",
                "#8=lmFitExampleSaveAs"),
              collapse = ", "),
    WriteArgNames = T
)
```

Comments:

- The PropertyList now contains only the Page grouping properties, because all the groups are contained in the pages.
- The two additional arguments are added to the ArgumentList just ahead of SaveAs.

The Resulting Dialog

The new dialog now has two pages: one labeled **Data/Methods** and the other **Output**.

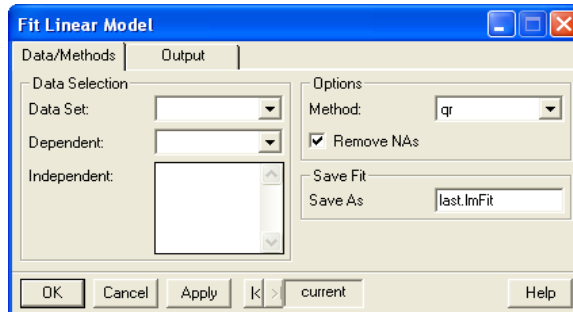


Figure 2.2: Page 1 of the extended dialog.

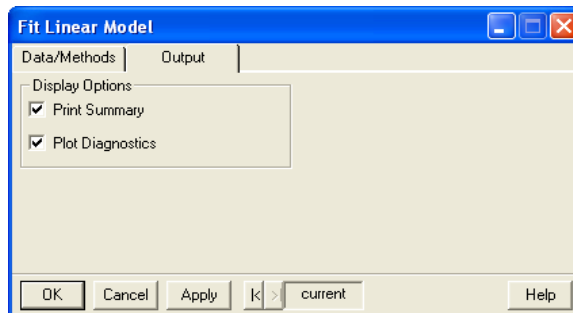


Figure 2.3: Page 2 of the extended dialog.

BUILDING THE GUI DURING INSTALLATION

To create the GUI properties automatically, along with the associated functions and objects for your package during build time, requires a particular code structure. It is not complicated, but it requires following some rules.

Basic Code Structure

The basic structure is depicted in the schematic displayed in Figure 2.4. A more detailed discussion appears below.

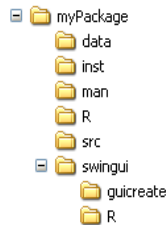


Figure 2.4: Directory for building a package with Spotfire S+ GUI included.

The steps for building a GUI for your package are as follows:

1. In the package directory, create the directory **swingui**, along with the other directories: **data**, **man**, and **R**.
2. Under **swingui**, create two subdirectories: **guicreate** and **R**.
3. In the **guicreate** directory, save a script file (not a function definition) defining all the dialog properties and FunctionInfo objects.

Note

Menu and MenuItem creation do not go in this directory. See step 4 for more detail.

4. In the **R** subdirectory of **swingui**, save all function definition files associated with the GUI. These function definitions include the *menu* functions called by the dialogs and any *callback* functions that manage dialog content during user actions.

5. In the package `.First.lib` function (in **myPackage/R**), load the properties and `FunctionInfo` objects with calls like the following:

```
package.path <- file.path(library, section)
prop.path <- file.path(package.path,
                        ".Prefs", "package.prp")
info.path <- file.path(package.path,
                       ".Prefs", "package.fni")
guiLoadDefaultObjects("Property",
                      FileName = prop.path)
guiLoadDefaultObjects("FunctionInfo",
                      FileName = info.path)
```

where `"packageName"` is a quoted string containing the package name. Alternatively, place the above calls in an argumentless function (for example, `loadProps`) saved in a file in **swingui/R**, and call it from `.First.lib`. In the example above, `packageName` is `lmFitExample`.

6. If any files must be installed with the package accessed during run time, place them in the **inst** subdirectory, contained in the package directory. Examples of these kinds of files include bitmaps needed by a `Picture List Box` or similar property that loads bitmaps when the dialog is opened.
7. In the `.Last.lib` function, add a conditional expression to remove the package menu from the Spotfire S+ menu bar when the package is detached. The code should look something like the following:

```
if (is.sgui.app() && interactive() &&
    is.element("SPlusMenuBar$packageName",
              guiGetObjectNames("MenuItem"))){
  guiRemove("MenuItem",
            Name="SPlusMenuBar$packageName" ) }
```

where `"packageName"` is the name used for saving the menu properties. In the example above, `packageName` is `MyGUI`.

INDEX

Symbols

.Call 42
.gz 12
.zip 12

A

ASCII 42
available.packages 9, 19, 31, 38

B

binary 4, 11
build 25, 26, 30, 40

C

Callback function component
 Spotfire S+ GUI 51
cbGetActiveProp 54
cbGetCurrValue 54
cbIsInitDialogMessage 54
cbIsOkMessage 54
cbSetCurrValue 54
cbSetEnableFlag 54
cbSetOptionList 54
check 25, 27, 29, 39
classCenter 44
Combo Box control 59
compiling code
 scripts 40
Control Property component
 Spotfire S+ GUI 51

CRAN 38
CSAN 3, 9, 12, 13, 20, 29, 35, 41

D

data 4, 30
DATAINSTALL 41
DESCRIPTION 5, 19, 29, 30, 39
 example 30
Dialect 39
do.call 69
download.packages 10, 20, 39
downloading packages
 using the GUI 11
dump 24, 42

E

example
 Soundex 22

F

FunctionInfo Object component
 Spotfire S+ GUI 51
FuntionInfo 57

G

Group 59
guiCreate 56
guiGetObjectNames 52
guiGetPropertyValue 52

H

- help files
 - creating 33
- HELPINSTALL 41
- HTML Help Workshop
 - required software 15
- http
 - //spotfire.tibco.com/csan 29, 41

I

- inst 5
- INSTALL 25, 28, 40
- install.packages 10, 20, 21, 28
- integrate 43

J

- Java code 35

L

- lapply 45
- Latex 35
- library 11, 25
- List Box control 59
- location
 - installing to another 11

M

- man 4, 24, 30, 32, 39
- mapply 44
- Menu component
 - Spotfire S+ GUI 51
- Menu function component
 - Spotfire S+ GUI 51
- Menu Item component
 - Spotfire S+ GUI 51
- MenuItem object 52
- method 57
- Multi-select List Box text box 59

N

- new.packages 9

O

- optim 45
- options 19

P

- package.skeleton 12, 22, 23, 30
- package.skelton 34
- packageDescription 31
- Page properties 68
- Perl 4
- plotDiagnostics 69
- printSummary 69

R

- R 4, 30, 32, 39
- README 30
- removeNA 58

S

- S.dll 5, 42
- s.dll 35
- S.so 5, 35, 42
- saveAs 58
- scripts
 - compiling code 40
- share
 - packages 12, 29
- SHLIB 41
- SINSTALL 41
- Soundex example 22
- source 11, 19, 38
- SPlusMenuBar 56
- SPropInvisibleReturnObject 62
- src 4, 30, 35
- src2bin 41
- statMenuLoc 56
- SWeave 35

T

tar 39
tar.gz 15

U

unresolvedGlobalReferences 43

V

vcvars32.bat 17
vignettes 35
Visual C++ compiler 16

W

win.binary 19, 38
Windows
 required software 13

