



# **S-PLUS® 8**

# **Guide to Packages**

May 2007

Insightful Corporation  
Seattle, Washington

---

**Proprietary  
Notice**

Insightful Corporation owns both this software program and its documentation. Both the program and documentation are copyrighted with all rights reserved by Insightful Corporation.

The correct bibliographical reference for this document is as follows:

*S-PLUS<sup>®</sup> 8 Guide to Packages*, Insightful Corporation, Seattle, WA.

Printed in the United States.

**Copyright Notice**

Copyright © 1987-2007, Insightful Corporation. All rights reserved.

Insightful Corporation  
1700 Westlake Avenue N, Suite 500  
Seattle, WA 98109-3044  
USA

**Trademarks**

Insightful, Insightful Corporation, the Insightful logo and tagline "the Knowledge to Act," Insightful Miner, S+, S-PLUS, S+FinMetrics, S+EnvironmentalStats, S+SeqTrial, S+SpatialStats, S+Wavelets, S+ArrayAnalyzer, S-PLUS Graphlets, Graphlet, Trellis, and Trellis Graphics are either trademarks or registered trademarks of Insightful Corporation in the United States and/or other countries. Intel and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Microsoft, Windows, MS-DOS and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Sun, Java and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States or other countries. UNIX is a registered trademark of The Open Group. Linux is a registered trademark of Linus Torvalds in the United States and other countries.

# ACKNOWLEDGMENTS

The S-PLUS package system is adapted from the package system used in R. Many of the functions and scripts used in the S-PLUS package system were originally written by contributors to R, and these functions are distributed separately from S-PLUS under the GPL license.

We acknowledge the R core group:

Douglas Bates, John Chambers, Peter Dalgaard, Robert Gentleman, Kurt Hornik, Stefano Iacus, Ross Ihaka, Friedrich Leisch, Thomas Lumley, Martin Maechler, Duncan Murdoch, Paul Murrell, Martyn Plummer, Brian Ripley, Duncan Temple Lang, Luke Tierney, and Simon Urbanek

for their work in developing R and its package system.

We also acknowledge the R developer and user community that has made the packages concept a huge success. For more information on R and its contributors, go to <http://www.r-project.org>.

# S-PLUS BOOKS

The S-PLUS<sup>®</sup> documentation includes books to address your focus and knowledge level. Review the following table to help you choose the S-PLUS book that meets your needs. These books are available in PDF format in the following locations:

- In your S-PLUS installation directory (**SHOME\help** on Windows, **SHOME/doc** on UNIX/Linux).
- In the S-PLUS Workbench, from the **Help ► S-PLUS Manuals** menu item.
- In Microsoft<sup>®</sup> Windows<sup>®</sup>, in the S-PLUS GUI, from the **Help ► Online Manuals** menu item.

*S-PLUS documentation.*

<b>Information you need if you...</b>	<b>See the...</b>
Are new to the S language and the S-PLUS GUI, and you want an introduction to importing data, producing simple graphs, applying statistical models, and viewing data in Microsoft Excel <sup>®</sup> .	<i>Getting Started Guide</i>
Are a system administrator or a licensed user and you need guidance licensing your copy of S-PLUS and/or any S-PLUS module.	<i>S-PLUS licensing Web site <a href="http://keys.insightful.com/">keys.insightful.com/</a></i>
Are a new S-PLUS user and need how to use S-PLUS, primarily through the GUI.	<i>User's Guide</i>
Are familiar with the S language and S-PLUS, and you want to use the S-PLUS plug-in, or customization, of the Eclipse Integrated Development Environment (IDE).	<i>S-PLUS Workbench User's Guide</i>
Have used the S language and S-PLUS, and you want to know how to write, debug, and program functions from the <b>Commands</b> window.	<i>Programmer's Guide</i>

*S-PLUS documentation. (Continued)*

<b>Information you need if you...</b>	<b>See the...</b>
Are familiar with the S language and S-PLUS, and you want to extend its functionality in your own application or within S-PLUS.	<i>Application Developer's Guide</i>
Are familiar with the S language and S-PLUS, and you are looking for information about creating or editing graphics, either from a <b>Commands</b> window or the Windows GUI, or using S-PLUS-supported graphics devices.	<i>Guide to Graphics</i>
Are familiar with the S language and S-PLUS, and you want to use the Big Data library to import and manipulate very large data sets.	<i>Big Data User's Guide</i>
Want to download or create S-PLUS packages for submission to the Comprehensive S Archival Network (CSAN) site, and need to know the steps.	<i>Guide to Packages</i>
Are looking for categorized information about individual S-PLUS functions.	<i>Function Guide</i>
If you are familiar with the S language and S-PLUS, and you need a reference for the range of statistical modelling and analysis techniques in S-PLUS. Volume 1 includes information on specifying models in S-PLUS, on probability, on estimation and inference, on regression and smoothing, and on analysis of variance.	<i>Guide to Statistics, Vol. 1</i>
If you are familiar with the S language and S-PLUS, and you need a reference for the range of statistical modelling and analysis techniques in S-PLUS. Volume 2 includes information on multivariate techniques, time series analysis, survival analysis, resampling techniques, and mathematical computing in S-PLUS.	<i>Guide to Statistics, Vol. 2</i>



# CONTENTS

---

<b>S-PLUS Books</b>	iv
<b>Chapter 1 Guide to Packages</b>	<b>1</b>
<b>Overview of S-PLUS® Packages</b>	3
<b>"Quick Start" to Packages</b>	6
<b>Required Tools for Creating Packages</b>	10
<b>Package Details</b>	13
<b>Example: Creating an S-PLUS Package</b>	16
<b>How to Submit a Package to CSAN</b>	22
<b>Components of a Source S-PLUS Package</b>	23
<b>How to Convert a Package from R to S-PLUS</b>	29
<b>Differences between R and S</b>	38
<b>Index</b>	<b>49</b>

## *Contents*

# GUIDE TO PACKAGES

# 1

---

<b>Overview of S-PLUS® Packages</b>	<b>3</b>
Package Types	4
S-PLUS Package Structure	4
Location of User-Installed S-PLUS Packages	5
<b>"Quick Start" to Packages</b>	<b>6</b>
Installing the pkgutils Library Section	6
Finding Packages on CSAN	7
Downloading Packages from CSAN	8
Installing and Loading a Package	8
Creating a Package	9
Submitting a Package	9
<b>Required Tools for Creating Packages</b>	<b>10</b>
Windows	10
UNIX/Linux	12
<b>Package Details</b>	<b>13</b>
Installing the pkgutils Library Section	13
Browsing Packages	13
Example: Downloading and Installing the rpart Package	14
<b>Example: Creating an S-PLUS Package</b>	<b>16</b>
Soundex Example	16
Building, Checking, and Installing the Package	19
<b>How to Submit a Package to CSAN</b>	<b>22</b>
<b>Components of a Source S-PLUS Package</b>	<b>23</b>
DESCRIPTION File	24
data Directory	25
R Directory	25
man Directory	26
src Directory	28

<b>How to Convert a Package from R to S-PLUS</b>	<b>29</b>
Getting An R Source Package	29
Creating an S-PLUS Package from an R Package	30
Build Scripts	31
Differences Between S-PLUS and R Packages	32
Porting Tools	34
Trouble-Shooting Porting R Packages to S-PLUS Packages	36
Missing C/FORTRAN Functions	37
<b>Differences between R and S</b>	<b>38</b>
How to Set the Parser	38
T and F	38
assign Attribute in model.matrix	38
R's model.frame Adds predvars attribute	39
How to Get the Same Random Number Streams in S-PLUS and R	41
Mixing timeSeries and matrix Classes Causes Errors	43
by="mon" in timeSequence Interpreted Differently	43
Log Scale for x- and y-axis Specified Differently	43
lwd in abline() Behaves Differently	44
warn.conflicts not in S-PLUS	44
First Argument to .Call() Can Be Unquoted in R But Must be Quoted in S-PLUS	44
Porting C Code Using Rinternals.h and Rdefines.h	45
Namespaces	45
factanal()	45
nls() and plot.formula()	46
pairs() and dotchart() Functions	46
Wilcoxon Distribution	46
Using Underscore ("_") for Assignment	47

## **OVERVIEW OF S-PLUS® PACKAGES**

An S-PLUS® 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 S-PLUS users. These packages offer you and other S-PLUS users a mechanism to distribute user-generated functions quickly. You can download and install S-PLUS packages from an Insightful-maintained Web site, or you can create S-PLUS packages that you can submit for potential distribution.

The S-PLUS 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 Insightful 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 Archival Network (CSAN).

Insightful hosts the CSAN site at **[csan.insightful.com](http://csan.insightful.com)** to facilitate S-PLUS package distribution. This Web site serves as a centralized repository for S-PLUS 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 S-PLUS package system. The code is distributed separately from S-PLUS under the GPL license.

**Package Types** Packages on CSAN are available as either Windows<sup>®</sup> *binary* or *source*. By default, the functions for downloading and installing packages from CSAN use binary packages, while UNIX/Linux uses source packages.

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 10). 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.

**S-PLUS 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 the following:

- The directory, with the same name as the package.
- 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**: Any S language functions as ASCII files.
  - **src**: C, C++ or FORTRAN source code as ASCII files. (Optional.)
  - **inst**: This is copied, recursively, into the main package directory when the package is compiled.
  - **DESCRIPTION**: A text file containing information about the package (see the section DESCRIPTION File on page 24). This is the only file required in a package.

Note that a data-only package would only have the **data** and **man** directories, whereas an S-code-only package would only have the **R** and **man** directories. Similarly, a package might have an **inst** directory, containing only pdfs.

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 S-PLUS, 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 S-PLUS Packages

Starting with S-PLUS 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\Insightful\splus80\_WIN386\library**

Windows Vista:

**C:\Users\username\AppData\Local\Insightful\  
splus80\_WIN386\library**

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

**\$HOME/MySwork/splus80\_LINUX/library**

Note the S-PLUS version and platform designation are included in these default package locations for all platforms. This allows packages for multiple platforms and S-PLUS 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.

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 library sections ("packages") the user installs are separate from those installed with the S-PLUS product. This design simplifies creating the same S-PLUS environment on another computer: after you install S-PLUS on the new computer, simply copy over the local directory.

## "QUICK START" TO PACKAGES

### Installing the pkgutils Library Section

Before you can do any work with packages in S-PLUS, 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 S-PLUS distribution.

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

```
./INSTALL.PKGUTILS
```

in the top level directory of S-PLUS to download and install pkgutils in **library/pkgutils** under the top level directory of S-PLUS. The same individual should install S-PLUS and the pkgutils library section, because you need to have the appropriate permissions to install in the S-PLUS 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\Insightful\splus80_WIN386\library
```

Windows Vista:

```
C:\Users\username\AppData\Local\Insightful\  
splus80_WIN386\library
```

To install pkgutils in Windows, in the S-PLUS **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 S-PLUS 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 9.

## Finding Packages on CSAN

Use the following functions in S-PLUS to help you discover available packages on the CSAN site.

**Table 1.1:** *Package browsing functions.*

S-PLUS function	Description
<code>available.packages</code>	Use <code>available.packages</code> to determine which S-PLUS packages are available for download from the CSAN site.
<code>new.packages</code>	Use <code>new.packages</code> to discover any S-PLUS packages on CSAN that you have not yet installed.

## Downloading Packages from CSAN

Use the following functions in S-PLUS 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 13).

**Table 1.2:** *Package downloading functions.*

S-PLUS 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.

## 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 S-PLUS, 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 S-PLUS session with the `library()` command. If package `xyzzzy` was installed under the standard package location of your S-PLUS installation, then you need only to enter

```
library(xyzzzy)
```

to load the package into your current S-PLUS 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 `xyzzzy`, you must type

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

or

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

## **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 an S-PLUS 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.

## **Submitting a Package**

To share your S-PLUS package with the S-PLUS user community, send it to Insightful for posting on the CSAN site. Insightful 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 Insightful creates and posts the Windows binary. These S-PLUS packages are then posted to CSAN by Insightful and become available for download.

## REQUIRED TOOLS FOR CREATING PACKAGES

Downloading or installing S-PLUS<sup>®</sup> 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. Similar to the pkgutils library, you must be connected to the Internet to download the tools you need to edit and compile packages.

Note that all tools discussed in this section are available for free download and installation.

### Windows

S-PLUS 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://csan.insightful.com>**

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 *source* packages. You do not need any additional software if you are only installing binary packages for Windows from CSAN using S-PLUS 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.

### 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 S-PLUS, 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 S-PLUS 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.

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

S-PLUS supports FORTRAN code compiled with Visual Fortran<sup>®</sup>. At this time, there is no free version of Visual Fortran available.

<b>Note about Perl and Visual Studio Compiler Installations</b>
---

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 <b>vcvar32.bat</b> .
---

**Access to Windows Tools** For the S-PLUS 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**

for Perl, HTML Help Workshop, and the Visual Studio compiler, respectively. If you installed a Windows compiler, confirm that it is present. Click **OK**.

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
```

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

```
cl /help
```

5. In addition, 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<sup>®</sup>/Linux<sup>®</sup> platforms requires these additional software tools, in addition to S-PLUS:

- **Perl** - This must be in your path, so that typing

```
perl --version
```

displays information about your Perl version (which must be 5.8 or higher), and does not produce a "Command not found" error.

- **Compilers for C and/or FORTRAN** - This is required if your package includes source code.

## PACKAGE DETAILS

### Installing the pkgutils Library Section

Before you download or install S-PLUS packages, you must download and install the S-PLUS pkgutils library section. See the section Installing the pkgutils Library Section on page 6 for more details.

### Browsing Packages

Using your Web browser, you can browse for available packages on the CSAN site at **csan.insightful.com**. From your browser, you can download an S-PLUS package and save the package archive on your local machine. You can use scripts from a shell, or functions within S-PLUS to install the package archive.

Alternatively, you can get a listing of the packages on CSAN using the function `available.packages` within S-PLUS. You can then download and install the packages from within S-PLUS. 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 an S-PLUS 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 AN S-PLUS 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 10.

The easiest way to create a package is to use the `package.skeleton` function in S-PLUS. 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 S-PLUS Package on page 23.

### 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 S-PLUS, load the `pkgutils` library:

```
library(pkgutils)
```

2. In S-PLUS, 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 are new with 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 S-PLUS 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.

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 S-PLUS 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

S-PLUS includes utilities that you can run on the package (listed in Table 1.3), and you can run them from a command shell on Windows and UNIX/Linux. As noted in the section Windows on page 10, using these scripts requires additional software components. The scripts also require that the pkgutils library be installed.

**Table 1.3:** *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 <b>Commands</b> window.
REMOVE	Removes the package from the system. You can also use <code>remove.packages()</code> from a <b>Commands</b> window.

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

```
Splus CMD utility.name options ...
```

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

```
Splus 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 S-PLUS 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 (e.g., **soundex\_1.0\_WIN386.zip**). That package archive can be installed only on the same platform (e.g., Windows or Linux) that it was created on.

## Checking the Package

Before distributing a package archive to others, the `check` utility should be run on the package. This performs a number of checks on the package to verify that it contains all the necessary components in the appropriate format, and that the package can be built and installed.

The `check` utility attempts to run all the examples from the help files in the **man** directory. If the package includes a **tests** directory, any files in that directory with the file extension **.t** are run by the `do.test()` function; 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 S-PLUS, 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 S-PLUS, 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 S-PLUS, 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 S-PLUS package with other users within your department, company, or university. Just send them the package archive, and have them install it with the `INSTALL` script or the `install.packages` function (setting `repos=NULL`).

If you want to share your package with the entire S-PLUS community, you can submit your package for inclusion in the Comprehensive S 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`**

Once you have uploaded your file, send a message to

**`packages@insightful.com`**

stating the name of the package archive you submitted.

Before submitting a package for inclusion in CSAN, it should pass the check utility. 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 will not be posted to CSAN.

Insightful will review your submitted package, run the check utility, and create a Windows binary archive. If everything passes, the package is posted to the CSAN site. Any problems with the package are sent to the package submitter.

## COMPONENTS OF A SOURCE S-PLUS PACKAGE

The `package.skeleton` function automates some of the setup for a new source S-PLUS 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 S-PLUS 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. **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 **xyzy.txt** results in creation of a data frame called **xyzy** in the package. See the `installFromDataFiles` help files for more information.

For example, in the section Soundex Example on page 16, 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", "Engebretson", "O'Brien",  
  "Opnian", "van Lind", "Zita", "Zitzmeinn")
```

## R Directory

The **R** subdirectory contains ASCII dumps of all the functions included in your S-PLUS 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 S-PLUS, 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 an S-PLUS 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 S-PLUS objects, using `\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}
```

## 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 S-PLUS when the package is attached with the `library` command.

## HOW TO CONVERT A PACKAGE FROM R TO S-PLUS

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

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

### 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 S-PLUS:

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<sup>®</sup>/Linux<sup>®</sup> 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 an S-PLUS 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 10, you will likely need to install the `tar` utility.

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

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

can be used to specify code for conditional execution in R or S-PLUS.

3. The section Differences Between S-PLUS and R Packages on page 32 and section Trouble-Shooting Porting R Packages to S-PLUS Packages on page 36 indicate some areas to look out for.
4. Update the **DESCRIPTION** file to indicate changes made, porting to S-PLUS, 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 S-PLUS, you may want to submit the package to the S-PLUS CSAN site. See section Submitting a Package on page 9 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 S-PLUS, 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 1.4:** *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 S-PLUS help files from <b>.Rd</b> source files into the <b>.Data</b> directory of a specified destination directory.
SINSTALL	Installs S-PLUS code or data objects from source files into the <b>.Data</b> directory of a specified destination directory.
DATAINSTALL	Install S-PLUS code or data objects from source files into the <b>.Data</b> 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 S-PLUS and R Packages

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

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

- C and FORTRAN code is automatically loaded in S-PLUS packages when the package is attached with the library function. This is done through the S-PLUS 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.
- S-PLUS package system uses only the current S-PLUS 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 an S-PLUS 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 S-PLUS.
- S-PLUS does not have NAMESPACES so any references to them in a package needs to be modified to work in S-PLUS.
- The first argument to `.Call` must be a a string in S-PLUS. R allows the first argument to `.Call` to be a variable.
- The default storage mode for a numeric value with no decimal place in S-PLUS is an integer, while in R it is a double. For example, in S-PLUS:

```
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 S-PLUS:

```
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", meaning 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 S-PLUS mode.

- The first component in the return list from the `integrate` function is named "integral" in S-PLUS 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 S-PLUS is the `unresolvedGlobalReferences` function introduced in S-PLUS 8.0. This function looks for undefined functions and data in S-PLUS 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 S-PLUS, 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 S-PLUS 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 S-PLUS. To make a portable `classCenter` function that would run in both S-PLUS 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 to S-PLUS Packages

R and S-PLUS 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 S-PLUS are listed here.

## Scoping Rules

S-PLUS 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 S-PLUS, 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 34.

## **Missing C/ FORTRAN Functions**

S-PLUS 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 S-PLUS.
- R has its exponential random number generator available for calling from C, but S-PLUS does not. The uniform and gaussian random number generators are available in S-PLUS.

## DIFFERENCES BETWEEN R AND S

There are many differences between R and S-PLUS; functions with the same names may have different arguments, defaults may be set differently, and other significant differences may exist. This chapter discusses some of the important distinctions between R and S. If there are functions you see in R that you would like to see implemented in S-PLUS, contact Technical Support at [support@insightful.com](mailto:support@insightful.com).

### How to Set the Parser

In R, object names may include underscores (“\_”), and the underscore is not used as an assignment operator; in S-PLUS, an underscore means assignment. You can set the default to use either the R or S parser with `set.parse.mode()` or `parse()`. For example:

```
set.parse.mode(mode="R"/"Splus"/0/1, ...)
```

or

```
parse(mode="R"/"Splus"/0/1, ...)
```

Use either `Splus` or `0` to parse in S-PLUS mode, and `R` or `1` to parse in R mode. The default mode is given by `getenv("S_PARSER_MODE")`. To see the current parse mode without changing it, enter

```
set.parse.mode(mode=-1).
```

You can optionally set the parser to parse in R mode to read in R functions and then set it back to S-PLUS mode to automatically convert object names. For example:

```
prev.parse.mode <- set.parse.mode("R")
source("my_fun.R") # assume this creates my_fun
set.parse.mode(prev.parse.mode)
dump("my_fun", "my_fun.q")
```

### T and F

In R, `T` and `F` are not reserved words (`TRUE` and `FALSE` are) and are global variables. S-PLUS reserves `T`, `F`, `TRUE`, and `FALSE`. It is strongly recommended that you not use `T` and `F` as variable names.

### assign Attribute in model.matrix

This was noticed while comparing code in R vs. S-PLUS:

```
d<-data.frame(f1=factor(rep(c("One","Two"),c(6,6))),
  levels=c("One","Two")),
  f2=factor(rep(c("I","II","III"),4),
  levels=c("I","II","III")), x=101:112, y=log(1:12))
```

```
f<-function(form, data){
  z<-model.frame(form, data=data)
  zz<-model.matrix(attr(z, "terms"), z)
  invisible(dput(attr(zz, "assign")))
}
> f(y~., data=d)
```

In S-PLUS, we get a list:

```
named("(Intercept)" = 1, "f1" = 2, "f2" = c(3, 4), "x" = 5)
```

and in R we get an integer vector:

```
c(0, 1, 2, 2, 3)
```

Here are the results for various formulae:

```
Splus> f(y~f2+x-1, data=d), named("f2" = c(1,2,3), "x" = 4)
R> f(y~f2+x-1, data=d)
c(1, 1, 1, 2)
```

and

```
Splus> f(y~(f2+f1)^2, data=d)
named("(Intercept)" = 1
, "f2" = c(2, 3)
, "f1" = 4
, "f2:f1" = c(5, 6)
)
R> f(y~(f2+f1)^2, data=d)
c(0, 1, 1, 2, 3, 3)
```

and

```
Splus> f(y~f1*f2-1, data=d)
named("f1" = c(1, 2), "f2" = c(3, 4), "f1:f2" = c(5, 6)
R> f(y~f1*f2-1, data=d)
c(1, 1, 2, 2, 3, 3)
```

They both seem to encode the same information but in different ways.

## **R's model.frame Adds predvars attribute**

`model.frame()` in R returns information that `model.frame()` in S-PLUS does not. In particular, the `terms` attribute of the result has an attribute, `predvars`, that is not in the S-PLUS output. It is like the `variables` attribute, but includes information that is needed to make predictions from new datasets. `model.frame.default()` calls `makepredictcall()` to create this attribute and `help(makepredictcall)` provides documentation on its use.

It is important to know about this when porting R code that deals with modelling functions, and you might want to use a similar scheme to make predictions involving `poly()` or `ns()` valid.

The following is an example comparing `predvars` and `variables` in the output of R's `model.frame()`:

```
> library("splines")
> d<-data.frame(y=log(1:12),
               x=sqrt(1:12),
               letter=factor(rep(c("a","b"),c(6,6))),
               roman=factor(rep(c("I","II","III"),4)))
```

For "ordinary" terms in the model, `predvars` and `variables` are identical:

```
> attr(attr(model.frame(log(y)~sqrt(x+1),data=d), "terms"),
      "variables")
> list(log(y), sqrt(x + 1))
> attr(attr(model.frame(log(y)~sqrt(x+1),data=d), "terms"),
      "predvars")
> list(log(y), sqrt(x + 1))
```

But if your model has a term like `bs(x)` or `poly(x)` where there are other implied arguments whose default values depend on `x`, then `predvars` contains the calls you would need to use this with new data:

```
> attr(attr(model.frame(y~bs(x),d), "terms"),
      "variables")
> list(y, bs(x))
> attr(attr(model.frame(y~bs(x),d), "terms"),
      "predvars")
> list(y, bs(x, degree = 3, knots = numeric(0),
      Boundary.knots = c(1,
      3.46410161513775), intercept = FALSE))
```

The arguments that `makepredictcall()` adds to `bs()` come from `bs(d$x)`:

```
> bs(d$x)
           1           2           3
 [1,] 0.0000000 0.0000000 0.0000000
 ...
 [12,] 0.0000000 0.0000000 1.0000000
> attr("degree")
 [1] 3
> attr("knots")
 numeric(0)
```

```

> attr("Boundary.knots")
[1] 1.000000 3.464102
> attr("intercept")
[1] FALSE
> attr("class")
[1] "bs"      "basis"

```

and `trace(bs)` shows how `predict()` uses the data in `predvars`:

```

> trace(bs)
> fit<-lm(y~bs(x), data=d)
trace: bs(x)
> predict(fit, newdata=data.frame(x=c(2,3)))
trace: bs(x, degree = 3, knots = numeric(0),
Boundary.knots = c(1, 3.46410161513775), intercept =
FALSE)
      1      2
1.391730 2.191604

```

Plotting the results shows that R outputs the correct prediction, and S-PLUS does not (the predictions from new data are drawn as triangles, and they lie on the fitted data in R but not in S-PLUS):

```

> plot(d$x, fitted(fit))
> points(c(2,3), predict(fit,
  newdata=data.frame(x=c(2,3))), pch=2, cex=2)
trace: bs(x, degree = 3, knots = numeric(0),
Boundary.knots = c(1, 3.46410161513775), intercept =
FALSE)

```

In S-PLUS, the trace shows just `bs(x)`, with no extra arguments.

## How to Get the Same Random Number Streams in S-PLUS and R

The two seeds are the Tausworthe and congruence long integers, respectively. A one-to-one mapping to the `.Random.seed[1:12]` in S-PLUS is possible, but note this generator is *not* exactly the same as that in recent versions of S-PLUS:

```

# Input is an R random number seed for kinds="Super-Duper",
# which is Marsaglia's famous Super-Duper from the 1970s. It
# has a period of about 4.6*10^18 for most initial seeds. The
# seed is two integers (all values allowed for the first
# seed: the second must be odd). Output is a guess at the
# corresponding S+ seed
seed.RtoS _ function(seed){
  seed1 _ seed %% 2^16
  seed2 _ seed / seed1
#

```

```
f <- function(k){
  if(k<0) k _ k + 2^32
  result _ rep(0,6)
  for(i in 1:6){
    result[i] _ k %% 64
    k _ k %% 64
  }
  result
}
as.integer(c(f(seed[2]), f(seed[1])))
}
# In R:
RNGkind("Super")
set.seed(0)
.Random.seed # gives 402 -835792825 1280795613
runif(5) # 0.6404036 0.5927313 0.4129687 0.1877294 0.2679058
#
seed.RtoS(c(-835792825, 1280795613))
# In S
seed.RtoS(c(-835792825, 1280795613)) # 29 15 54 21 12 1 7
1 45 11 14 3 set.seed( seed.RtoS(c(-835792825, 1280795613)))
runif(5) # That matches R's first 5 numbers
#
# The streams seem to correspond for the first 1,111,032
numbers at least:
# In R:
RNGkind("Super")
set.seed(0)
temp = runif(1e3); runif(4)
temp = runif(1e4); runif(4)
temp = runif(1e6); runif(4)
temp = runif(1e7); runif(4)
temp = runif(1e8); runif(4) # this one failed
# [1] 0.4258198 0.3218063 0.2310835 0.1941062
# [1] 0.8936687 0.8485062 0.5086048 0.2790297
# [1] 0.5507552 0.7823178 0.5666004 0.5956264
# [1] 0.4331028 0.9279524 0.2352643 0.8160968
#
# In S+
set.seed(seed.RtoS(c(-835792825, 1280795613)))
temp = runif(1e3); runif(4)
temp = runif(1e4); runif(4)
temp = runif(1e6); runif(4)
temp = runif(1e7); runif(4)
# The results from S+ match those from R
```

**Mixing  
timeSeries and  
matrix Classes  
Causes Errors**

Objects that use old classes such as `timeSeries` and `matrix` try to inherit from both. In this example, `Y.ts` tries to inherit from both:

```
x = c(1.0622, 1.0785, 1.0797, 1.0862, 1.1556, 1.1674,  
      1.1365, 1.1155, 1.1267, 1.1714, 1.1710, 1.2298)  
y = c(1.5414, 1.5121, 1.4761, 1.4582, 1.3840, 1.3525,  
      1.3821, 1.3963, 1.3634, 1.3221, 1.3130, 1.3128)  
Y.ts = ts(cbind(USDEUR = x, USDCAD = y))      plot(Y.ts)  
Error in plot: Lengths of x and y must match
```

`tsplot(Y.ts)` does work as a possible workaround.

**by="mon" in  
timeSequence  
Interpreted  
Differently**

In the R version of `timeSequence()`, `by="mon"` gets partially matched with "month" so the sequence is by month:

```
> timeSequence(from="1/1/2001", to="1/1/2002", by="mon")  
[1] "Zurich"  
[1] [2001-01-01] [2001-02-01] [2001-03-01] [2001-04-01]  
[2001-05-01] [2001-06-01] [2001-07-01]  
[8] [2001-08-01] [2001-09-01] [2001-10-01] [2001-11-01]  
[2001-12-01] [2002-01-01]
```

In the S-PLUS version of `timeSequence()`, `by="mon"` indicates the sequence should be by Mondays:

```
> timeSequence(from=timeDate("1/1/2001"),  
               to=timeDate("1/1/2002"), by="mon")  
from: 01/01/2001  
to:    01/01/2002  
by:    +1mon  
[1] 01/01/2001 01/08/2001 01/15/2001 ...      12/31/2001
```

**Log Scale for  
x- and y-axis  
Specified  
Differently**

In R, the `xlog` value in `par()` indicates whether the current plot has a log x-axis. In S-PLUS, the `xaxt` value in `par()` is "1" if it is a log axis. Similarly, R has `ylog`, and S-PLUS has `yaxt`.

The following type of code makes the output the same:

```
if (is.R()){  
  log <- par("xlog")  
}  
else {  
  log <- (par("xaxt")=="1")  
}
```

**lwd in abline()** In R, the following works with the `lwd` ignored:  
**Behaves** `> plot(1:10)`  
**Differently** `> abline(v=5, lwd=NULL)`

In S-PLUS, it fails due to the `lwd=NULL`:

```
> plot(1:10)
> abline(v=5, lwd=NULL)
Error: Non-numeric vector
Error in segments
```

The workaround is to have code of the form:

```
if (!is.null(lwd)){
  abline(v = at, lwd = lwd)
}
else {
  abline(v = at)
}
```

**warn.conflicts not in S-PLUS** The `library()` function in R has a `warn.conflicts` argument that is not in the S-PLUS `library()` function. This causes the following example code in the `fBasics` package to fail:

```
> library(Design, warn.conflicts = FALSE)
> library(Hmisc, warn.conflicts = FALSE)
```

**First Argument to .Call() Can Be Unquoted in R But Must Be Quoted in S-PLUS** In R, the symbol name in the `.Call()` can be specified as unquoted:

```
armaSS <- function(y, mod)
{
  ### next call changes objects a, P, Pn so beware!
  .Call(R_ARIMA_Like, y, mod$phi, mod$theta,
        mod$Delta, mod$a, mod$P, mod$Pn, as.integer(0), TRUE)
}
```

In S-PLUS, the symbol name must be a string, so the first argument is quoted:

```
armaSS <- function(y, mod)
{
  ### next call changes objects a, P, Pn so beware!
  .Call("R_ARIMA_Like", y, mod$phi, mod$theta, mod$Delta,
        mod$a, mod$P, mod$Pn, as.integer(0), TRUE)
}
```

## **Porting C Code Using Rinternals.h and Rdefines.h**

Some C code within R packages is written to manipulate internal R objects, such as SEXP structures. This code typically includes the files **Rinternals.h** or **Rdefines.h**, and references structures and macros defined in these files. To make it easier to port C code from R to S-PLUS, the files **Rinternals.h** and **Rdefines.h** have been created in the S-PLUS **include** directory. These files contain macro definitions that map some common R functions and macros to corresponding ones supported by S-PLUS.

Most of the time, even when using these include files, warnings and errors are generated when compiling the C code. One common problem is that R encodes integer values as C type `int`, whereas S-PLUS uses C type `long`. These are generally the same on 32-bit architectures, but it is still important to examine warnings when one type is used instead of the other. For example, the R `INTEGER(x)` macro returns an `int*`, whereas S-PLUS returns a `long*`, and compiling the code

```
int* x = INTEGER(y)
```

could give the warning:

```
warning: assignment from incompatible pointer type
```

There are many R functions that simply do not have any meaning in S-PLUS. For some such functions, **Rinternals.h** includes a macro definition to change the function call to an undefined variable, so that it is easier for the user to understand the problem. For example, using the R macro `CHAR` produces an error:

```
'unsupported_R_entry_CHAR' undeclared
```

The R `CHAR` macro is used to create a string object, which doesn't exist in S-PLUS, so an error such as this indicates that the programmer needs to rewrite this part of the code.

## **Namespaces**

S-PLUS does not support name spaces, so the value of either `package::name` or `package:::name` is just `name`.

## **factanal()**

The `factanal()` function has a `covmat` argument in R and `covlist` argument in S-PLUS. To get code that works in both, specify just `"cov"` as the argument name:

```
Harman74.FA <- factanal(factors = 1, cov = Harman74.cor)
```

## **nls() and plot.formula()**

The `nls()` function has a `subset` argument in R and not in S-PLUS. So R code of the form:

```
fm1 <- nls(circumference ~ SSlogis(age, Asym, xmid, scal),
           data = Orange, subset = Tree == 3)
```

needs to use subscripting of the `data` argument instead:

```
fm1 <- nls(circumference ~ SSlogis(age, Asym, xmid, scal),
           data = Orange[Orange$Tree == 3,])
```

The same is true for `plot.formula()`:

```
plot(circumference ~ age, data = Orange, subset = Tree == 3,
     xlab = "Tree age (days since 1968/12/31)",
     ylab = "Tree circumference (mm)", las = 1,
     main = "Orange tree data and fitted model
           (Tree 3 only)")
```

## **pairs() and dotchart() Functions**

The `pairs()` and `dotchart()` functions in R have a `"main"` argument. The same functions in S-PLUS do not.

## **Wilcoxon Distribution**

R's `help(dwilcox)` defines the Wilcoxon distribution as follows:

Let  $x$  and  $y$  be two random, independent samples of size  $m$  and  $n$ , respectively. Then the Wilcoxon rank sum statistic is the number of all pairs  $(x[i], y[j])$  for which  $y[j]$  is not greater than  $x[i]$ . This statistic takes values between 0 and  $m * n$ , and its mean and variance are  $m * n / 2$  and  $m * n * (m + n + 1) / 12$ , respectively.

S-PLUS defines it differently:

If data consist of two random samples, a sample  $x$  of size  $m$ , and a sample  $y$  (independent of sample  $x$ ) of size  $n$ , then the Wilcoxon rank sum statistic is the sum of the ranks of  $x$  in the combined sample  $c(x, y)$ . This statistic can then be used for a non-parametric test of location shift between the parent populations.

The Wilcoxon rank sum statistic takes on values between  $m*(m+1)/2$  and  $m*(m+2*n+1)/2$ .

This means that `dwilcox`, `pwilcox`, and others give different results in R and S-PLUS.

**Using  
Underscore  
("\_") for  
Assignment**

Never use the underscore for assignment, since the parser used functions differently between R and S. In R, object names may include underscores ("\_"), and the underscore is not used as an assignment operator; in S, the underscore is an assignment operator.

Use back quotes (``my_data``) around names with underscores in them.



# INDEX

---

## Symbols

.Call 33  
.gz 9  
.zip 9

## A

ASCII 33  
available.packages 7, 13, 24, 29

## B

binary 4, 8  
build 19, 20, 23, 31

## C

check 19, 20, 22, 30  
classCenter 35  
compilers 12  
compiling code  
  scripts 31  
converting  
  from R to S-PLUS 29  
CRAN 29  
CSAN 3, 7, 9, 14, 22, 32

## D

data 4, 23  
DATAINSTALL 32  
DESCRIPTION 4, 13, 22, 23, 30  
  example 24

Dialect 30  
download.packages 8, 14, 30  
dump 18, 33

## E

example  
  Soundex 16

## H

help files  
  creating 26  
HELPINSTALL 32  
HTML Help Workshop  
  required software 10

## I

inst 4  
INSTALL 19, 21, 31  
install.packages 8, 14, 15, 21  
integrate 34

## L

lapply 36  
library 9, 19  
location  
  installing to another 9

## M

man 4, 18, 23, 26, 30

mapply 35

## **N**

new.packages 7

## **O**

optim 36

options 13

## **P**

package.skeleton 9, 16, 17, 23

package.skelton 27

packageDescription 24

Perl 4, 12

## **R**

R 4, 23, 25, 30

README 23

## **S**

S.dll 5, 33

s.dll 28

S.so 5, 28, 33

scripts

    compiling code 31

share

    packages 9, 22

SHLIB 32

SINSTALL 32

Soundex example 16

source 8, 13, 29

src 4, 23, 28

src2bin 32

## **T**

tar 30

tar.gz 11

## **U**

unresolvedGlobalReferences 34

## **V**

vcvars32.bat 12

Visual C++ compiler 12

## **W**

win.binary 13, 29

Windows

    required software 10