# 1 Introduction to S-mode

The S and Splus packages provide sophisticated statistical and graphical routines for manipulating data. The S-mode package provides useful routines for making the use of these packages much easier.

A bit of notation before we begin. I will refer to both the ‘new S’ package (as described in Becker, Chambers and Wilks, *The New S Language: A programming environment for data analysis and graphics*) and ‘Splus’ (an enhanced version of new S from Statsci) simply by “S”. The interface which is used to run S under Emacs (which this manual documents) will be referred to as “S-mode”, which should not be confused with the GNU Emacs major mode `S-mode` which is used for editing S source. Finally, the GNU Emacs Lisp source code in which this package is defined will be referred to as `S.el`.

For exclusively interactive users of S, S-mode provides a number of features to make life easier. There is an easy-to-use command history mechanism, including a quick prefix-search history. To reduce typing, command-line completion is provided for all S objects and “hot keys” are provided for common S function calls. Help files are easily accessible, and a paging mechanism is provided to view them. Finally, an incidental (but very useful) side-effect of S-mode is that a transcript of your session is kept for later saving or editing. No special knowledge of Emacs is necessary when using S interactively under S-mode.

For those that use S in the typical edit-test-revise cycle when programming S functions, S-mode provides for editing of S functions in Emacs edit buffers. Unlike the typical use of S where the editor is restarted every time an object is edited, S-mode uses the current Emacs session for editing. In practical terms, this means that you can edit more than one function at once, and that the S process is still available for use while editing. Error checking is performed on functions loaded back into S, and a mechanism to jump directly to the error is provided. S-mode also provides for maintaining text versions of your S functions in specified source directories.

## 1.1 Authors of and contributors to S-mode

S-mode is based on Olin Shivers’ excellent comint package (which comes with the S-mode distribution). The original version of S-mode was written by Doug Bates (`bates@stat.wisc.edu`) and Ed Kademan (`kademan@stat.wisc.edu`). Frank Ritter (`ritter@psy.cmu.edu`) then merged this version with his own S mode to form `S.el` version 2.1.

Version 2.1 of S.el was then updated and expanded by David Smith to form version 3.4. Most bugs have now been fixed (and several new ones introduced) and many new features have been added. Thanks must go to the beta testers for version 3.4:

- `S-eval-line-and-next-line` is based in an idea by Rod Ball.
- Thanks to Doug Bates for many useful suggestions.
- `comint-isearch` was written by Terry Glanfield (`tg.southern@rxuk.xerox.com`).
- Thanks to Martin Maechler for reporting and fixing bugs.
- Thanks to Frank Ritter for updates from the previous version, suggestions and invaluable comments on the manual.

- Thanks to Ken'ichi Shibayama for his excellent indenting code, and many comments and suggestions.
- Thans to Bob Stine for testing an early version of the Tek graphics support.

## 1.2 Getting the latest version of S-mode

The latest version of S-mode is always available for anonymous FTP from

```
attunga.stats.adelaide.edu.au
```

in the directory `pub/S-mode`. Check the `README` file first to see which files you need. S-mode is also available from the Emacs-Lisp archive on `archive.cis.ohio-state-edu` — retrieve

```
pub/gnu/emacs/elisp-archive/README
```

for information on the archive. The latest version is also available from Statlib by sending a blank message with subject “send index from S” to `statlib@stat.cmu.edu`, and following the directions from there.

## 1.3 Changes from version 2.1

For current users of S-mode, here are some of the incompatible changes and features new to version 3.3 of S-mode:

- Command-line completion of object names, and faster completion in other situations.
- ‘Hot Keys’ for the commonly-used functions `objects()`, `search()` and `attach()` and a facility to add your own hot keys with keyboard macros.
- Simultaneous multiple function editing, with integrated error-checking and parsing. Mnemonic names for edit buffers.
- Debugging features: facility for stepping through S code and evaluating portions of code with the output appearing as if the commands has been typed in manually.
- S can now be run from a different directory each session.
- A dedicated mode for viewing S help files. Individual help buffers are maintained for quick repeated access. Completion for help files without a corresponding object.
- Facility for maintaining organised backups of S source code.
- Indenting and formatting commands for editing S source code.
- Special handling of the S graphics facilities, including an experimental Tek graphics mode.
- Better handling of temporary files and buffers.
- Some keybindings have changed to conform to GNU guidelines.
- General code cleanups and optimizations.

## 1.4 How to read this manual

If S-mode has already been installed on your system, the next chapter has details on how to get started using S under S-mode.

If you need to install S-mode, read Appendix A [Installation], page 21, for details on what needs to be done before proceeding to the next chapter.

Appendix B [Customization], page 23, provides details of user variables you can change to customize S-mode to your taste, but it is recommended that you defer this section until you are more familiar with S-mode.

Don't forget that this manual is not the only source of information about S-mode. In particular, the mode-based online help (obtained by pressing `C-h m` when in the process buffer, edit buffer or help buffer) is quite useful. However the best source of information is, as always, experience — try it out!

# 2 Starting the S process

To start an S session, simply type *`M-x S RET`*, i.e. press `ESC`, then `x`, then capital `S` and then the `RETURN` key.

If the variable `S-ask-for-S-directory` has a non-`nil` value, you will be prompted with

```
From which directory?
```

with a default value chosen on the basis of the variable `S-directory`. The S program will be run from the directory you specify at this point, that is S will use the `.Data` subdirectory of this directory (if it exists.) Using this facility it is possible to have a number of distinct S directories for separate projects, etc. If the value of `S-ask-for-S-directory` is `nil`, the S program will be run from the directory specified by `S-directory` (which defaults to your home directory).

Next, if the value of `S-ask-about-display` is non-`nil` you will be presented with the prompt

```
Which X-display?
```

The value you enter here will be used as the value of the `DISPLAY` environment variable of the S process for use with the X windowing system. Unless this variable is set correctly, S commands such as `X11()` or `help.start()` will not work. Completion is provided on the basis of the variable `X-displays-list`.

Once these questions are answered (if they are asked at all) the program defined by `inferior-S-program` will be executed with arguments specified by `explicit-`*`S_program_name`*`-args`. You will be popped into a buffer with name '`*S*`' which will be used for interacting with the S process, and you can start entering commands.

# 3 Interacting with the S process

The primary function of the S-mode package is to provide an easy-to-use front end to the S interpreter. This is achieved by running the S process from within an Emacs buffer, so that the Emacs editing commands are available to correct mistakes in commands, etc. A sophisticated command history and recall mechanism is also provided, thanks to the `comint` package. Command-line completion of S objects and a number of 'hot keys' for commonly-used S commands are also provided for ease of typing.

## 3.1 Entering commands and fixing mistakes

Sending a command to the S process is as simple as typing it in and pressing the `RETURN` key:

- *`RET`* (`inferior-S-send-input`)
  Send the command on the current line to the S process.

If you make a typing error before pressing *`RET`* all the usual Emacs editing commands are available to correct it (see Section "Basic editing commands" in *The GNU Emacs Reference Manual*.) Once the command has been corrected you can press `RETURN` (even if the cursor is not at the end of the line) to send the corrected command to the S process.

S-mode provides two other commands which are useful for fixing mistakes:

- *`C-c C-w`* (`backward-kill-word`)
  Deletes the previous word (such as an object name) on the command line.
- *`C-c C-u`* (`comint-kill-input`)
  Deletes everything from the prompt to point. Use this to abandon a command you have not yet sent to the S process.

Finally, the beginning-of-line command (*`C-a`*) has been slightly redefined to leave you at the start of the current command instead:

- *`C-a`* (`comint-bol`)
  Move to the beginning of the line, and then skip forwards past the prompt, if any.

## 3.2 Completion of object names

In the process buffer, `TAB` only inserts a '`TAB`' character when the cursor is not following an object name. Otherwise, the `TAB` key is for completion:

- *`TAB`* (`S-complete-object-name`)
  Complete the S object name before point.

When the cursor is just after a partially-completed object name, pressing `TAB` provides completion in a similar fashion to `tcsh` except that completion is performed over S object names instead of file names. S-mode maintains a list of all objects known to S at any given time, which basically consists of all objects (functions and datasets) in every attached directory listed by the `search()` command along with the component objects of attached data frames (if your version of S supports them).

For example, consider the three functions (available in Splus version 3.0) called `binomplot()`, `binom.test()` and `binomial()`. Typing *`bin TAB`* after the S prompt will insert the characters '`om`', completing the longest prefix ('`binom`') which distinguishes these three commands. Pressing *`TAB`* once more provides a list of the three commands which have these prefix, allowing you to add more characters (say, '`.`') which specify the function you desire. After entering more characters pressing *`TAB`* yet again will complete the object name up to uniqueness, etc. If you just wish to see what completions exist without adding any extra characters, pass a prefix command to `S-complete-object-name` with *`C-u TAB`*.

S-mode automatically keeps track of any objects added or deleted to the system to make completion as accurate as possible. As long as the command that changed the search list matched `S-change-sp-regex`, when a directory or data frame is attached, the objects associated with it immediately become available for a completion; when detached completion is no longer available on those objects. Efficiency is gained by maintaining a cache of objects currently known to S; when a new object becomes available or is deleted, only one component of the cache corresponding to the associated directory is refreshed. The command *`M-x S-resynch`* forces the *entire* cache to be refreshed — use this command whenever S-mode gets confused about completion. One warning: S *never* automatically refreshes its idea of the components of attached data frames; if the names of the components of a data frame are modified during an S session, S-mode will not recognise any new components (or ignore any deleted components) until the command *`M-x S-resynch`* is issued.

S-mode also provides completion over the components of named lists accessed using the '`$`' notation, to any level of nested lists. Such information is never cached, and so is guaranteed to always be correct. This feature is particularly useful for checking what components of a list object exist while partway through entering a command: simply type the object name and '`$`' and press *`TAB`* to see the names of existing list components for that object.

## 3.3 Moving through the process buffer

Most of the time, the cursor spends most of its time at the bottom of the S process buffer, entering commands. Sometimes, however, we want to move back up through the buffer, to look at the output from previous commands for example.

Viewing the output of the command you have just entered is a common occurence and S-mode provides a number of facilities for doing this. Within the S process buffer, the variable `scroll-step` is set to 4 (you can redefine this using `inferior-S-mode-hook` if you wish - see Section B.2 [Hooks], page 26,) so that the cursor is usually near the bottom of the window. Longish command outputs may cause S to place the cursor at the middle of the window, however, making the first part of the output hidden above the top of the window. If this happens, you can use the command

- *`C-c C-v`* (`S-view-at-bottom`)
  Move to the end of the buffer, and place cursor on bottom line of window.

will make more of the last output visible. If the first part of the output is still obscured, use

- *`C-c C-r`* (`comint-show-output`)
  Moves cursor to the previous command line and and places it at the top of the window.

to view it. Finally, if you want to discard the last command output altogether, use

- *`C-c C-o`* (`S-kill-output`)
  Deletes everything from the last command to the current prompt.

to kill it.

If you want to view the output from more historic commands than the previous command, commands are also provided to move backwards and forwards through previously entered commands in the process buffer:

- *M-P* (`comint-msearch-input`)
  Moves point to the preceding command in the process buffer.
- *M-N* (`comint-psearch-input`)
  Moves point to the next command in the process buffer.
- *C-c C-b* (`comint-msearch-input-matching`)
  Prompts for a string and jump to the previous command you entered which matched that string.

When the cursor is not after the current prompt, the `RETURN` key has a slightly different behaviour than usual. Pressing *RET* on any line containing a command that you entered (i.e. a line beginning with a prompt) sends that command to the S process once again. This even works if the current line is a continuation line (i.e. the prompt is '`+ `' instead of '`> `') — in this case all the lines that form the multi-line command are concatenated together and the resulting command is sent to the S process (currently this is the only way to resubmit a multi-line command to the S process in one go.) If the current line does not begin with a prompt, an error is signalled. This feature, coupled with the command-based motion commands described above, could be used as a primitive history mechanism. S-mode provides a more sophisticated mechanism, however, which is described below.

## 3.4 Command History

S-mode provides easy-to-use facilities for re-executing or editing previous commands. An input history of the last few commands is maintained (by default the last 50 commands are stored, although this can be changed by setting the variable `input-ring-size` in `inferior-S-mode-hook`.) The simplest history commands simply select the next and previous commands in the input history:

- *M-p* (`comint-next-input`)
  Select the previous command in the input history.
- *M-n* (`comint-previous-input`)
  Select the next command in the input history.

For example, pressing *M-p* once will re-enter the last command into the process buffer after the prompt but does not send it to the S process, thus allowing editing or correction of the command before the S process sees it. Once corrections have been made, press *RET* to send the edited command to the S process.

If you have an idea which command you want from the history, the commands

- *M-s* (`comint-previous-similar-input`)
  Select the previous command in the history which matches the string typed so far.
- *M-S* (`comint-next-similar-input`)
  Select the next command in the history which matches the string typed so far.

may be more useful, as they only select commands starting with those characters already entered. For instance, if you wanted to re-execute the last `attach()` command, all you need to do is type *attach* and then *M-s* and *RET*.

Sometimes you want to re-execute a command that matches a particular string (a variable name for example) which does not appear at the start of the command. In this case

- *M-r* (`comint-isearch`)
  Interactively search backwards through the input history for a string.

may be useful. This command is very similar to `isearch-backward`, except that it operates on the input history instead of the buffer text. After typing *M-r*, commands which match the search string are displayed as you enter the string itself. If you entered some text before pressing *M-r* then only commands which begin with that text are considered as candidates, and the string is matched against the remaining part of the command. Use *C-r* to search further backwards and *C-s* to search forwards. *RET* sends the selected comand directly to the S process; use *ESC* if you wish to edit it first.

## 3.5 Hot keys for common commands

S-mode provides a number of commands for executing the commonly used functions. These commands below are basically information-gaining commands (such as `objects()` or `search()`) which tend to clutter up your transcript and for this reason some of the hot keys display their output in a temporary buffer instead of the process buffer by default. This behaviour is controlled by the variable `S-execute-in-process-buffer` which, if non-`nil`, means that these commands will produce their output in the process buffer instead. In any case, passing a prefix argument to the commands (with *C-u*) will reverse the meaning of `S-execute-in-process-buffer` for that command, i.e. the output will be displayed in the process buffer if it usually goes to a temporary buffer, and vice-versa. These are the hot keys that behave in this way:

- *C-c C-x* (`S-execute-objects`)
  Sends the `objects()` command to the S process. A prefix argument specifies the position on the search list (use a negative argument to toggle `S-execute-in-process-buffer` as well.) A quick way to see what objects are in your working directory.
- *C-c C-s* (`S-execute-search`)
  Sends the `search()` command to the S process.
- *C-c C-e* (`S-execute`)
  Prompt for an S expression, and evaluate it.

`S-execute` may seem pointless when you could just type the command in anyway, but it proves useful for 'spot' calculations which would otherwise clutter your transcript, or for evaluating an expression while partway through entering a command. You can also use this command to generate new hot keys using the Emacs keyboard macro facilities; see Section "Keyboard Macros" in *The GNU Emacs Reference Manual*.

The following hot keys do not use `S-execute-in-process-buffer` to decide where to display the output — they either always display in the process buffer or in a separate buffer, as indicated:

- *C-c C-a* (`S-execute-attach`)
  Prompts for a directory to attach to the S process with the `attach()` command. If a numeric prefix argument is given it is used as the position on the search list to attach the directory; otherwise the S default of 2 is used. The `attach()` command actually executed appears in the process buffer.

- *C-c C-l* (`S-load-file`)
  Prompts for a file to load into the S process using `source()`. If there is an error during loading, you can jump to the error in the file with *C-x `* (`S-parse-errors`). See Section 4.5 [Error Checking], page 15, for more details.
- *C-c C-h* (`S-display-help-on-object`)
  Pops up a help buffer for an S object or function. See See Chapter 5 [Help], page 16, for more details.
- *C-c C-q* (`S-quit`)
  Sends the `q()` command to the S process, and cleans up any temporary buffers (such as help buffers or edit buffers) you may have created along the way. Use this command when you have finished your S session instead of simply typing `q()` yourself, otherwise you will need to issue the command *M-x S-cleanup* command explicitly to make sure that all the files that need to be saved have been saved, and that all the temporary buffers have been killed.

## 3.6 Other commands provided by inferior-S-mode

The following commands are also provided in the process buffer:

- *C-c C-c* (`comint-interrupt-subjob`)
  Sends a Control-C signal to the S process. This has the effect of aborting the current command.
- *C-c C-z* (`S-abort`)
  Sends a STOP signal to the S process, killing it immediately. It's not a good idea to use this, in general: Neither `q()` nor `.Last` will be executed and device drivers will not finish cleanly. This command is provided as a safety to `comint-stop-subjob`, which is usually bound to *C-c C-z*. If you want to quit from S, use *C-c C-q* (`S-quit`) instead.
- *C-c C-d* (`S-dump-object-into-edit-buffer`)
  Prompts for an object to be edited in an edit buffer. See Chapter 4 [Editing], page 10.
- *C-c C-t* (`S-tek-mode-toggle`)
  Toggles Tek graphics mode. See Section 6.2 [tek4014], page 18, for more details.

# 4 Editing S functions

S-mode provides facilities for editing S objects within your Emacs session. Most editing is performed on S functions, although in theory you may edit datasets as well. Edit buffers are always associated with files, although you may choose to make these files temporary if you wish. Alternatively, you may make use of a simple yet powerful mechanism for maintaining backups of text representations of S functions. Error-checking is performed when S code is loaded into the S process.

## 4.1 The edit buffer

To edit an S object, type

- *`C-c C-d`* (`S-dump-object-into-edit-buffer`)
  Edit an S object in its own edit buffer.

from within the S process buffer (`*S*`). You will them be prompted for an object to edit: you may either type in the name of an existing object (for which completion is available using the *`TAB`* key,) or you may enter the name of a new object. A buffer will be created containing the text representation of the requested object or, if you entered the name of a non-existent object at the prompt and the variable `S-insert-function-templates` is non-`nil`, you will be presented with a template defined by `S-function-template` which defaults to a skeleton function construct.

You may then edit the function as required. The edit buffer generated by `S-dump-object-into-edit-buffer` is placed in the `S-mode` major mode which provides a number of commands to facilitate editing S source code. Commands are provided to intelligently indent S code, evaluate portions of S code and to move around S code constructs.

**Note:** when you dump a file with *`C-c C-d`*, S-mode first checks to see whether there already exists an edit buffer containing that object and, if so, pops you directly to that buffer. If not, S-mode next checks whether there is a file in the appropriate place with the appropriate name (See Section 4.3 [Source Files], page 13) and if so, reads in that file. You can use this facility to return to an object you were editing in a previous session (and which possibly was never loaded to the S session.) Finally, if both these tests fail, the S process is consulted and a `dump()` command issued. If you want to force S-mode to ask the S process for the object's definition (say, to reformat an unmodified buffer or to revert back to S's idea of the object's definition) pass a prefix argument to `S-dump-object-into-edit-buffer` by typing *`C-u C-c C-d`*.

### 4.1.1 Sending code to the S process

There are a wide range of commands for sending code to the S process. The primary command is for loading the current buffer (which usually contains a single function) into the S process:

- *`C-c C-l`* (`S-load-file`)
  Loads a file into the S process using `source()`.

After typing *`C-c C-l`* you will prompted for the name of the file to load into S; usually this is the current buffer's file which is the default value (selected by simply pressing *`RET`* at the prompt.) Your will be asked to save the buffer first if it has been modified (this happens automatically if the buffer was generated with *C-c C-d*.) If the buffer is not modified, S-mode assumed its contents are equivalent to S's value of the function and you will need to confirm that you want to load the file into S. The file will then be loaded and you will be returned to the S process. If any errors occur, S-mode will inform you of this fact: See Section 4.5 [Error Checking], page 15.

Other commands are also available for evaluating portions of code in the S process. You may choose whether both the commands and their output appear in the process buffer (as if you had typed in the commands yourself) or if the output alone is echoed. The behaviour is controlled by the variable `S-eval-visibly-p` whose default is `nil` (display output only.) Passing a prefix argument (*C-u*) to any of the following commands, however, reverses the meaning of `S-eval-visibly-p` for that command only — for example *C-u C-c C-j* echoes the current line of S-code in the S process buffer, followed by its output. This method of evaluation is an alternative to S's `source()` function when you want the input as well as the output to be displayed. (You can sort of do this with `source()` when the option `echo=T` is set, except that prompts do not get displayed. S-mode puts prompts in the right places.) The commands for evaluating code are:

- *C-c C-j* (`S-eval-line`)
  Send the line containing point to the S process.
- *C-c M-j* (`S-eval-line-and-go`)
  As above, but returns you to the S process buffer as well.
- *C-c C-f* or *ESC C-x* (`S-eval-function`)
  Send the S function containing point to the S process.
- *C-c M-f* (`S-eval-function-and-go`)
  As above, but returns you to the S process buffer as well.
- *C-c C-r* (`S-eval-region`)
  Send the text between point and mark to the S process.
- *C-c M-r* (`S-eval-region-and-go`)
  As above, but returns you to the S process buffer as well.
- *C-c C-b* (`S-eval-buffer`)
  Send the contents of the edit buffer to the S process.
- *C-c M-b* (`S-eval-buffer-and-go`)
  As above, but returns you to the S process buffer as well.
- *C-c C-n* (`S-eval-line-and-next-line`)
  Sends the current line to the S process, echoing it in the process buffer, and moves point to the next line. Useful when debugging for stepping through your code.

It should be stressed once again that these `S-eval-` commands should only be used for evaluating small portions of code for debugging purposes, or for generating transcripts from source files. When editing S functions, *C-c C-l* is the command to use to update the function's value. In particular, `S-eval-buffer` is now largely obsolete.

One final command is provided for spot-evaluations of S code:

*C-c C-e* (`S-execute-in-tb`)
Prompt for an S expression and evaluate it. Displays result in a temporary buffer.

This is useful for quick calculations, etc.

### 4.1.2 Indenting and formatting S code

S-mode now provides a sophisticated mechanism for indenting S source code (thanks to Ken'ichi Shibayama.) Compound statements (delimited by '`{`' and '`}`') are indented relative to their enclosing block. In addition, the braces have been electrified to automatically indent to the correct position when inserted, and optionally insert a newline at the appropriate place as well. Lines which continue an incomplete expression are indented relative to the first line of the expression. Function definitions, `if` statements, calls to `expression()` and loop constructs are all recognised and indented appropriately. User variables are provided to control the amount if indentation in each case, and there are also a number of predefined indentation styles to choose from. See Section B.1.3 [Indentation variables], page 24.

Comments are also handled specially by S-mode, using an idea borrowed from the Emacs-Lisp indentation style. Comments beginning with '`###`' are aligned to the beginning of the line. Comments beginning with '`##`' are aligned to the current level of indentation for the block containing the comment. Finally, comments beginning with '`#`' are aligned to a column on the right (the 40th column by default, but this value is controlled by the variable `comment-column`,) or just after the expression on the line containing the comment if it extends beyond the indentation column.

The indentation commands provided by S-mode are:

- *`TAB`* (`S-indent-command`)
  Indents the current line as S code. If a prefix argument is given, all following lines which are part of the same (compound) expression are indented by the same amount (but relative indents are preserved).
- *`ESC C-q`* (`S-indent-exp`)
  Indents each line in the S (compound) expression which follows point. Very useful for beautifying your S code.
- *`{`* and *`}`* (`S-electric-brace`)
  The braces automatically indent to the correct position when typed.
- *`M-;`* (`indent-for-comment`)
  Indents a comment line appropriately, or inserts an empty (single-'`#`') comment.
- *`M-x S-set-style`*
  Set the formatting style in this buffer to be one of the predefined styles (`GNU`, `BSD`, `K&R` and `C++` by default). This command causes all of the formatting variables to be buffer-local.

### 4.1.3 Commands for motion, completion and more

A number of commands are provided to move across function definitions in the edit buffer:

- *`ESC C-e`* (`S-beginning-of-function`)
  Moves point to the beginning of the function containing point.
- *`ESC C-a`* (`S-end-of-function`)
  Moves point to the end of the function containing point.
- *`ESC C-h`* (`S-mark-function`)
  Places point at the beginning of the S function containing point, and mark at the end.

Don't forget the usual Emacs commands for moving over balanced expressions and parentheses: See Section "Lists and Sexps" in *The GNU Emacs Reference Manual*.

Completion is also available in the edit buffer:

- *ESC TAB* (`S-complete-object-name`)
  Completes the S object name before point.

Note however that completion is only provided over globally known S objects (such as system functions) — it will *not* work for arguments to functions or other variables local to the function you are editing.

Finally, two commands are provided for returning to the S process buffer:

- *C-c C-z* (`S-switch-to-end-of-S`)
  Returns you to the S process buffer, placing point at the end of the buffer.
- *C-c C-y* (`S-switch-to-S`)
  Also returns to to the S process buffer, but leaves point where it is.

In addition some commands available in the process buffer are also available in the edit buffer. You can still read help files with *C-c C-h*, edit another function with *C-c C-d* and of course *C-c C-l* can be used to load a source file into S. See Section 3.6 [Other inferior-S-mode commands], page 9, for more details on these commands.

## 4.2 Modification flags in edit buffers

Within S-mode edit buffers, the modification flag has a slightly different meaning than it usually does. In general, S-mode tries to set the modification flag whenever the contents of the edit buffer differ from S's idea of the objects value, and clears the flag whenever the edit buffer has been successfully loaded into the S process. Thus you will be warned whenever you attempt to kill a buffer which represents an edited (i.e. different) version of the existing S object.

Edit buffers are marked as temporary buffers within S-mode, and will be killed whenever `S-quit` or `S-cleanup` are called. If the modification flag is set, however, you will be warned before that buffer is killed.

## 4.3 Maintaining S source files

Every edit buffer in S-mode is associated with a *dump file* on disk. Dump files are created whenever you type *C-c C-d* (`S-dump-object-into-edit-buffer`), and may either be deleted after use, or kept as a backup file or as a means of keeping several versions of an S function.

`S-keep-dump-files` [User Option]

> If this has a non-`nil` value, then dump files are never deleted. Otherwise dump files are silently deleted after each use, unless an error occurs.

When `S-keep-dump-files` is `nil`, the dump file is deleted immediately after it is read into the edit buffer. This is so that you can kill the edit buffer at any time without leaving the dump file behind. When loading the file back into S with *C-c C-l*, the dump file is again written out to disk and loaded into S. If the load is successful, the file is again deleted. If there is an error in your function, however, the file is retained so that you may edit the file at any time to correct the error.

When `S-keep-dump-files` is non-`nil`, dump files are never deleted. Thus you can maintain a complete text record of the functions you have editied within S-mode. Backup files are always kept, and so by using the Emacs numbered backup facility — see Section "Single or Numbered Backups" in *The Gnu Emacs Reference Manual*, you can keep a historic record of function definitions. As long as a dump file exists in the appropriate place for a particular object, editing that object with *`C-c C-d`* finds that file for editing (unless a prefix argument is given) — the S process is not consulted. Thus you can keep comments *outside* the function definition as a means of documentation that does not clutter the S object itself. Another useful feature is that you may format the code in any fashion you please without S re-indenting the code every time you edit it. These features are particularly useful for project-based work.

Dump buffers are always autosaved, regardless of the value of `S-keep-dump-files`.

When an object is dumped to a file, S-mode adds some comment lines to the end of the file, such as

```
# Local Variables:
# mode:S
# S-temp-buffer-p:t
# End:
```

These are included to ensure that whenever you next edit the file, it is in the correct mode for editing S source, and that the associated buffer is marked as a temporary buffer. If `S-keep-dump-files` is `nil` and you wish to keep the file associated with the edit buffer, remove the line

```
# S-temp-buffer-p:t
```

and *save* the buffer. The buffer will still be marked as temporary, however, and so deleted when you quit from S. You can change this by using *`M-x set-variable`* to set the value of `S-temp-buffer-p` to `nil`.

## 4.4 Names and locations of dump files

Every dump file should be given a unique file name, usually the dumped object name with some additions.

`S-dump-filename-template` [User Option]
: Template for filenames of dumped objects. `%s` is replaced by the object name.

By default, dump file names are the user name, followed by '`.`' and the object and ending with '`.S`'. Thus if user `joe` dumps the object `myfun` the dump file will have name `joe.myfun.S`. The username part is included to avoid clashes when dumping into a publicly-writable directory, such as `/tmp`; you may wish to remove this part if you are dumping into a directory owned by you.

You may also specify the directory in which dump files are written:

`S-source-directory` [User Option]
: Directory name (ending in a slash) where S dump files are to be written.

By default, dump files are always written to `/tmp`, which is fine when `S-keep-dump-files` is `nil`. If you are keeping dump files, then you will probably want to keep them somewhere in your home directory, say `~/S-source`. This could be achieved by including the following line in your `.emacs` file:

```
(setq S-source-directory (expand-file-name "~/S-source/"))
```

If you would prefer to keep your dump files in separate directories depending on the value of some variable, S-mode provides a facility for this also.

`S-source-directory-generator` [User Option]

Variable whose value is a function which, when called with no arguments, will return a directory name (ending in '`/`') into which dump files will be written. `nil` means use the value of `S-source-directory`.

If the directory generated by this function does not exist but can be created, you will be asked whether you wish to create the directory. If you do not or the directory cannot be created, the value of `S-source-directory` is used.

One application of `S-source-directory-generator` is to keep dump files in some subdirectory of the current S directory:

```
(setq S-source-directory-generator
      '(lambda ()
         (expand-file-name
          (concat
           (directory-file-name S-directory)
           "/Src/"))))
```

This is useful if you keep your dump files and you often edit objects with the same name in different directories. Alternatively, if you often change your S working directory during an S session, you may like to keep dump files in some subdirectory of the directory pointed to by the first element of the current search list. This way you can edit objects of the same name in different directories during the one S session:

```
(setq S-source-directory-generator
    '(lambda ()
       (file-name-as-directory
        (expand-file-name (concat
                           (car S-search-list)
                           "/.Src")))))
```

## 4.5 Detecting and correcting errors

After loading a file into the S process with *C-c C-l*, S-mode will report whether the load was successful. If it was not (i.e. there was some sort of error in your code) you can return to the file from the S process buffer with *C-x `* (`S-parse-errors`). You will be returned to the offending file (loading it into a buffer if necessary) with point at the line S reported as containing the error. You may then correct the error, and reload the file. Note that the corresponding S object will not be changed until the file has been successfully loaded; it is for this reason that temporary files containing errors are never deleted.

Sometimes the error is not caused by a syntax error (loading a non-existent file for example.) In this case typing *C-x `* will simply display a buffer containing S's error message. You can force this behaviour (and avoid jumping to the file when there *is* a syntax error) by passing a prefix argument to `S-parse-errors` with *C-u C-x `*.

# 5 Reading help files in S-mode

S-mode provides an easy-to-use facility for reading S help files from within Emacs. From within the S process buffer or any S-mode edit buffer, typing *C-c C-h* (`S-display-help-on-object`) will prompt you for the name of an object for which you would like documentation. Completion is provided over all objects which have help files.

If the requested object has documentation, you will be popped into a buffer (named `*help(`*obj-name*`)*`) containing the help file. This buffer is placed in a special 'S Help' mode which disables the usual editing commands but which provides a number of keys for paging through the help file:

Help commands:

- *?* (`S-describe-help-mode`)
  Pops up a help buffer with a list of the commands available in S help mode.
- *h* (`S-display-help-on-object`)
  Pop up a help buffer for a different object

Paging commands:

- *b* or *DEL* (`scroll-down`)
  Move one page backwards through the help file.
- *SPC* (`scroll-up`)
  Move one page forwards through the help file.
- `>` (`beginning-of-buffer`) and `<` (`end-of-buffer`)
  Move to the beginning and end of the help file, respectively.

Section-based motion commands:

- *n* (`S-skip-to-next-section`) and *p* (`S-skip-to-previous-section`)
  Move to the next and previous section header in the help file, respectively. A section header consists of a number of capitalised words, followed by a colon.

In addition, the *s* key followed by one of the following letters will jump to a particular section in the help file:

| | |
|---|---|
| '`a`' | ARGUMENTS: |
| '`b`' | BACKGROUND: |
| '`B`' | BUGS: |
| '`d`' | DETAILS: |
| '`D`' | DESCRIPTION: |
| '`e`' | EXAMPLES: |
| '`n`' | NOTE: |
| '`o`' | OPTIONAL ARGUMENTS: |
| '`r`' | REQUIRED ARGUMENTS: |
| '`R`' | REFERENCES: |
| '`s`' | SIDE EFFECTS: |

| | |
|---|---|
| ‘`s`’ | SEE ALSO: |
| ‘`u`’ | USAGE: |
| ‘`v`’ | VALUE: |
| ‘`<`’ | Jumps to beginning of file |
| ‘`>`’ | Jumps to end of file |
| ‘`?`’ | Pops up a help buffer with a list of the defined section motion keys. |

Miscellaneous:

- *`r`* (`S-eval-region`)
  Send the contents of the current region to the S process. Useful for running examples in help files.
- *`/`* (`isearch-forward`)
  Same as *`C-s`*.

Quit commands:

- *`q`* (`S-switch-to-end-of-S`)
  Returns to the S process buffer in another window, leaving the help window visible.
- *`x`* (`S-kill-buffer-and-go`)
  Return to the S process, killing this help buffer.

In addition, all of the S-mode commands available in the edit buffers are also available in S help mode (See Section 4.1 [Edit buffer], page 10). Of course, the usual (non-editing) Emacs commands are available, and for convenience the digits and ‘`-`’ act as prefix arguments.

If a help buffer already exists for an object for which help is requested, that buffer is popped to immediately; the S process is not consulted at all. If the contents of the help file have changed, you either need to kill the help buffer first, or pass a prefix argument (with *`C-u`*) to `S-display-help-on-object`.

Help buffers are marked as temporary buffers in S-mode, and are deleted when `S-quit` or `S-cleanup` are called.

# 6 Using graphics with S-mode

One of the main features of the `S` package is its ability to generate high-resolution graphics plots, and S-mode provides a number of features for dealing with such plots.

## 6.1 Using S-mode with the `printer()` driver

This is the simplest (and least desirable) method of using graphics within S-mode. S's `printer()` device driver produces crude character based plots which can be contained within the S process buffer itself. To start using character graphics, issue the S command

```
printer(width=79)
```

(the `width=79` argument prevents Emacs line-wrapping at column 80 on an 80-column terminal. Use a different value for a terminal with a different number of columns.) Plotting commands do not generate graphics immediately, but are stored until the `show()` command is issued, which displays the current figure.

## 6.2 Using S-mode with the `tek4014()` driver

When using `S` from the shell with the `tek4014()` driver active, control-codes are sent to your terminal to generate high-resolution graphics plots. The terminal recognizes these control-codes as graphics commands and duly generates the appropriate plots. When running `S` from Emacs, however, the control-codes are not treated specially by Emacs and simply appear as "garbage" in your `S` process buffer.

One way around this is to find out what tty you are using (by using the Unix `tty` command *outside* of Emacs) and using the `file=` argument to the `tek4014()` function to divert the graphics control codes directly to the terminal. For example, if the `tty` command returned `/dev/ttyp1` then the S command

```
tek4014(file="/dev/ttyp1")
```

will send graphics to your terminal. You may even use some other terminal which you are logged on to to have the graphics appear on another terminal. There are some problems with this method, however: depending on your terminal you may need to switch into graphics mode before issuing the plotting command, and if you are displaying the graphics on the same terminal as your Emacs session, you will need to switch back to text mode afterwards. Issuing commands while in graphics mode presents its own problems, because control-codes issued by Emacs interfere with the display.

S-mode attempts to automate this procedure by detecting output from `S` commands which looks like Tek graphics control-codes and sends those control-codes directly to the terminal. This behaviour is enabled by setting the variable `S-tek-mode` to any non-`nil` value (which may be achieved by using the function `S-tek-mode-toggle`, bound to *`C-c C-t`* by default. Tek mode is designed to work with Tek terminals which use the same screen to share graphics an text and also with terminals which provide separate screens. In the former case (tested on a Vis603 terminal) the variable `S-tek-pause-for-graphics` should be set to `t`; in the latter case (tested using `xterm`'s Tek emulation facilities) `S-tek-pause-for-graphics` should be set to `nil`.

This mode depends on being able to work out where the graphics finish and normal (text) output starts. In the easiest case, it finishes with your prompt and S-mode has no trouble detecting that. Sometimes plotting functions also display some text afterwards, and provided the function finishes and your prompt is displayed *at the start* of a line this is no problem either, but make sure any such function you write finishes any text with a newline. Functions like

```
badfun <- function() { plot(1:10) cat("Hello") invisible() }
```

will break the graphics detector. Other functions, such as `gam(obj,ask=T)` present a menu after the plot and wait for input (and so your prompt isn't displayed). The variable `S-tek-possible-graph-prompts` is a regular expression used to detect any alternative prompt used in this case.

When the graphics display has completed, press any key to return to your Emacs display. This mode also works with the `ask=T` option to `tek4014()`, however any single key is now the appropriate response to the '`GO?`' prompt.

Unexpected redisplays of the Emacs screen (such as caused by `display-time` or garbage collection) can possibly send garbage to your graphics display, but unfortunately there seems to way to prevent this.

If you have a very simple prompt, it may by chance appear in the graphics output which could possibly cause problems; if this occurs you will be given a warning. It is advisable to choose a prompt with at least two characters. If your prompt changes during the S session, be sure to tell the Tek graphics detector with `M-x S-tek-get-simple-prompt`.

When `S-tek-mode` is enabled, S-mode will make your Emacs process unusable while waiting for the first output from a function (so it can determine whether or not it's graphics output). You may be stuck for a long time when executing a time-consuming function that produces no output. If this becomes a problem, use *`C-c C-t`* to turn Tek mode on just before pressing *`RET`* to issue a plotting command, and turn Tek mode off again after the plotting command has completed.

### 6.2.1 Warning

Tek mode is really an experimental feature of S-mode, and has only been tested on one system, and even then not particularly thoroughly. If it works for you, well and good, but don't be surprised if it takes some tinkering before it produces any results on your system. See Chapter 7 [Bugs], page 20, for a few of the things that can go wrong.

## 6.3 Using S-mode with windowing devices

Of course, the ideal way to use graphics with S-mode is to use a windowing system. Under X windows, this requires that the DISPLAY environment variable is appropriately set, which may not always be the case within your Emacs process. S-mode provides a facility for setting the value of DISPLAY before the S process is started if the variable `S-ask-about-display` is non-`nil`. See Appendix B [Customization], page 23, for details of this variable, and see Chapter 2 [Starting Up], page 4, for information on how to set the value of DISPLAY when beginning an S session.

## 7 Known bugs in S-mode

- Commands like `S-display-help-on-object` and list completion cannot be used while the user is entering a multi-line command. The only real fix in this situation is to use another S process.
- The `S-eval-` commands can leave point in the S process buffer in the wrong place when point is at the same position as the last process output. This proves difficult to fix, in general, as we need to consider all *windows* with `window-point` at the right place.
- It's possible to clear the modification flag (say, by saving the buffer) with the edit buffer not having been loaded into S.
- Backup files can sometimes be left behind, even when `S-keep-dump-files` is `nil`.
- Passing an incomplete S expression to `S-execute` causes S-mode to hang.
- Completing over lists indexed with '`$`' destroys the value of `.Last.value`
- The function-based commands don't always work as expected on functions whose body is not a parenthesised or compound expression, and don't even recognise anonymous functions (i.e. functions not assigned to any variable).
- Multi-line commands could be handled better by the command history mechanism.
- There's a zillion things wrong with Tek-mode:
  - Any graphics output that does not come directly after the command is not detected.
  - Graphics output that does not end with some text (either the prompt or something which matches `S-tek-possible-graph-prompts`) causes S-mode to hang.
  - Spurious junk gets sent to the graphics display whenever Emacs updates its display — `display-time` (which updates the mode line) and garbage collection (which puts a message in the echo area) are the main culprits. If only there were a way to stop Emacs from redisplaying for a time . . .
  - Interaction with the plot (via the crosshair cursor) is not possible.
  - `S-tek-mode` should really be a minor mode.

  Let's face it, Tek mode is flaky. It really needs a major overhaul by someone who really knows about Tek control codes. It needs to be written using sentinels to detect the start and end of graphics streams, and an efficient method for swapping between text and graphics modes, including support for terminals with separate graphics and text screens. Anyone who wants to have a go at it is more than welcome!

Until the end of August 1992, please send bug reports to `dsmith@stats.adelaide.edu.au`. After this date, mail to that address will not be answered for some time; please contact Frank Ritter (`Frank_Ritter@SHAMO.SOAR.CS.CMU.EDU`) or any of the other authors then (please `CC:` to `dsmith@stats` as well though – you never know your luck!) Comments, suggestions, words of praise and large cash donations are also more than welcome.

# Appendix A Installing S-mode on your system

The following section details those steps necessary to get S-mode running on your system.

First of all, you need to create a directory (say, `~/elisp`) to place the Emacs-Lisp files. Copy `S.el`, `S-tek.el`, `comint.el`, `comint-isearch.el` and `comint-extra.el` to that directory, and add the lines

```
(setq load-path (cons (expand-file-name "~/elisp") load-path))
(autoload 'S "S" "Run an inferior S process" t)
(autoload 's-mode "S" "Mode for editing S source" t)
```

to your `.emacs` file.

This will be enough to get S-mode running on most systems — see Chapter 2 [Starting Up], page 4, for details on starting S-mode. If it does not work, see Section A.1 [System dependent], page 21, for other variables you may need to change. See Appendix B [Customization], page 23, for other variables you may wish to set in your `.emacs` file, but it is suggested you defer this section until you are more familiar with S-mode.

It is recommended that the `.el` files all be byte-compiled with *`M-x byte-compile-file`* for efficiency.

## A.1 Other variables you may need to change

If you run the S program (from the shell) with a command other than '`Splus`' you will need to set the variable `inferior-S-program` to the name of the appropriate program by including a line such as

```
(setq inferior-S-program "S+")
```

in your `.emacs` file (substituting '`S+`' for the name of your S program.)

If you need to call this program with any arguments, the variable you need to set is dependent on the value of `inferior-S-program`; for example if it is `"Splus"`, set the variable `inferior-Splus-args` to a string of arguments to the `Splus` program. If `inferior-S-program` has some other value, substitute the `Splus` part of `inferior-Splus-args` with the appropriate program name. There aren't many instances where you need to call S with arguments, however: in particular do not call the S program with the '`-e`' command-line editor argument since S-mode provides this feature for you.

If you are running an older version of S, you may need to set the variable `S-version-running` to reflect this fact. The default is `"3.0"` which indicates the August '91 revision; any other value indicates an older version. This variable is effective only when S-mode is *loaded*; setting it during an S session has no effect.

If you are running Splus (the enhanced version of S from Statsci) you may also need to set the variable `S-plus` to `t`. If your value of `inferior-S-program` is `"S+"` or `Splus` this will not be necessary, however; `S-plus` defaults to `t` in this case.

Finally, if you use a non-standard prompt within S, you will need to set the variable `inferior-S-prompt` to a regular expression which will match both the primary prompt (`"> "` by default) and the continuing prompt (default of `"+ "`.) The default value of this variable matches S's default prompts. For example, if you use (`"$ "`) as your primary prompt (you have `options(prompt="$ ")` in your `.First` function), add the following line to your `.emacs`:

```
      (setq inferior-S-prompt "^\\(\\+\\|[^\\$]*\\$\\) *")
```

You will also need to set the variable `inferior-S-primary-prompt` to a regular expression which matches the primary prompt only. Do not anchor the regexp to the beginning of the line with '^'. Once again, the default value matches S's default prompt; in the example above the appropriate value would be `"[^\\$]*\\$ *"`.

Once these variables are set appropriately, S-mode should work on any system.

# Appendix B Customizing S-mode

S-mode can be easily customised to your taste simply by including the appropriate lines in your `.emacs` file. There are numerous variables which affect the behaviour of S-mode in certain situations which can be modified to your liking. Keybindings may be set or changed to your preferences, and for per-buffer customizations hooks are also available.

## B.1 Variables for customization

S-mode is easily customisable by means of setting variables in your `.emacs` file. In most cases, you can change defaults by including lines of the form

```
(setq variable-name value)
```

in your `.emacs`.

In what follows, variable names will be listed along with their descriptions and default values. Just substitute the variable name and the new value into the template above.

### B.1.1 Variables for starting S

`S-ask-for-S-directory` [User Option]

Default: `t`

If this variable has a non-`nil` value, then every time S-mode is run with *`M-x S`* you will be prompted for a directory to use as the working directory for your S session; this directory should have a `.Data` subdirectory. If the value of `S-ask-for-S-directory` is `nil`, the value of `S-directory` is used as the working directory.

`S-directory` [User Option]

Default: Your home directory

The working directory for your S session if `S-ask-for-S-directory` is `nil`, and the default when prompting for a directory if it is not. For example, you may wish to set this to the value of the current buffer's working directory before starting S by adding the following line to your `.emacs` file (See Section B.2 [Hooks], page 26)

```
(setq S-pre-run-hook
   '((lambda () (setq S-directory default-directory))))
```

`S-ask-about-display` [User Option]

Default: `nil`

If this variable has a non-`nil` value, then every time S-mode is run with *`M-x S`* you will be asked for a value for the `DISPLAY` environment variable to be used in the current S session. If this variable is not set correctly, S will not be able to create any windows under the X windowing environment. Completion is provided over the `X-displays-list` variable; the default is the current value of `DISPLAY`. This feature is useful is you often run S on a different display than that of the machine you are running S from. If `S-ask-about-display` is `nil`, the current value of `DISPLAY` is used.

`X-displays-list` [User Option]

Default: `'(":0.0")`

List of possible values for the `DISPLAY` environment variable, provided for completion when prompting for such a value.

### B.1.2 Variables for dump files

`S-insert-function-templates` [User Option]

Default: `t`

If this variable has a non-`nil` value, then dumping a non-existent object will result in the edit buffer containing a skeleton function definition, ready for editing.

`S-source-directory` [User Option]

Default: `"/tmp/"`

Directory name (ending in '`/`') in which dump files are placed. This should always be a writable directory.

`S-source-directory-generator` [User Option]

Default: `nil`

Alternative, dynamic method of specifying the directory for dump files.

`S-dump-filename-template` [User Option]

Default: *user_name*`.`*object_name*`.S`

Naming system to use for dumped object files. See Section 4.4 [Source Directories], page 14, for details of this and the previous two variables.

`S-keep-dump-files` [User Option]

Default: `nil`

Boolean flag signifying whether to keep dump files or to delete them after each use. See Section 4.3 [Source Files], page 13, for more details.

### B.1.3 Variables controlling indentation

`S-tab-always-indent` [User Option]

Default: `t`

If non-`nil`, then *`TAB`* in the edit buffer always indents the current line, regardless of the position of point in the line. Otherwise, indentation is only performed if point is in the lines indentation, and a tab character is inserted is point is after the first nonblank character.

`S-auto-newline` [User Option]

Default: `nil`

Non-`nil` means automatically newline before and after braces inserted in S code.

`S-indent-level` [User Option]

Default: 2

Extra indentation of S statement sub-block with respect to enclosing braces.

`S-brace-imaginary-offset` [User Option]

Default: 0

Extra indentation (over sub-block indentation) for block following an open brace which follows on the same line as a statement.

`S-brace-offset` [User Option]

Default: 0

Extra indentation for braces, compared with other text in same context.

`S-continued-statement-offset` [User Option]

Default: 0

Extra indent for lines not starting new statements.

`S-continued-brace-offset` [User Option]

Default: 0

Extra indent for substatements that start with open-braces. This is in addition to `S-continued-statement-offset`.

`S-arg-function-offset` [User Option]

Default: 2

Extra indent for arguments of function `foo` when it is called as the value of an argument to another function in `arg=foo(...)` form. If not number, the statements are indented at open-parenthesis following `foo`.

`S-expression-offset` [User Option]

Default: 4

Extra indent for internal substatements of the call to `expression()` specified in

```
obj <- expression(...)
```

form. If not a number, the statements are indented at open-parenthesis following '`expression`'.

In addition, a number of default styles are defined for you (in `S-style-alist`):

`S-default-style` [User Option]

Default: `GNU`

The default formatting style to use in edit buffers: See Section 4.1 [Edit buffer], page 10, for more details.

### B.1.4 Variables controlling interaction with the S process

`input-ring-size` [User Option]

Default: 50

Number of commands to store in the command history.

`S-execute-in-process-buffer` [User Option]

Default: `nil`

If this is `nil`, then the `S-execute-` commands (see Section 3.6 [Other inferior-S-mode commands], page 9) output to a temporary buffer. Otherwise, the output goes to the S process.

`S-eval-visibly-p` [User Option]

Default: `nil`

If non-`nil`, then the `S-eval-` commands (see Section 4.1 [Edit buffer], page 10) echo the S commands in the process buffer by default. In any case, passing a prefix argument to the eval command reverses the meaning of this variable.

## B.2 Customizing S-mode with hooks

S-mode provides five hooks, as follows:

`S-mode-hook` [Hook]

Called every time `S-mode` is run, i.e. every time an edit buffer is generated.

`S-pre-run-hook` [Hook]

Called before the S process is started with *`M-x S`*.

`S-mode-load-hook` [Hook]

Called just after the file `S.el` is loaded. Useful for setting up your keybindings, etc.

`inferior-S-mode-hook` [Hook]

Called just after the S process starts up, when the S process buffer is initialised.

`S-help-mode-hook` [Hook]

Called every time an S help buffer is generated.

## B.3 Changing the default S-mode keybindings

S-mode provides a separate keymap variable for the S process buffer, for edit buffers and for help buffers.

`inferior-S-mode-map` [Keymap]

Keymap used in the S process buffer. The bindings from `comint-mode-map` are automatically inherited.

`S-mode-map` [Keymap]

Keymap used within edit buffers.

`S-help-mode-map` [Keymap]

Keymap used within help buffers. In addition, `S-help-sec-map` is the keymap for the '`s`' prefix key. Keys defined in `S-help-sec-keys-alist` are automatically inserted into this keymap when S-mode is loaded.

# Concept Index

# Variable and command index

# Table of Contents