Specification of the SGCAL text formatter

by

Philip Hazel

New Museums Site
Pembroke Street
Cambridge CB2 3QH
United Kingdom

Edition 3.1
December 2004

Contents

1. Introduction

SGCAL is a text formatting program that is a direct descendent of the GCAL program that was originally written for an IBM mainframe around 1980. The current release should run on any system with a standard C compiler. However, the building apparatus that is supplied is aimed at Unix-like systems.

SGCAL's input takes the form of a text file that contains markup describing how the text is to be formatted. There are two forms of output:

- 'Plain' output is normal ASCII text, suitable for viewing online using a text editor or any other means of file display.
- 'Fancy' output encodes typeset pages in a format known as Gcode. This has to be further interpreted for display or printing.

An auxiliary program called **sgtops** is used to translate Gcode into PostScript. It is able to select specific pages and perform certain transformations on them.

Another auxiliary program called **sgpoint** is used to display Gcode output on a laptop screen for "slide" projection. A special style is provided to make it easy to define "pages" that are the correct size. The same input can be re-formatted as two-up pages for printing.

Plain output is obviously restricted to what can be represented as ASCII text. Within an SGCAL source file, alternative input can be processed, depending on whether the output is plain or fancy. For example, you can arrange that marked up strings of a certain type are displayed in italic in fancy output, but put in quotes in plain output.

Line graphics are available directly in SGCAL input, but at a low level. An auxiliary program called Aspic (distributed separately) can be called from within SGCAL to process a high level graphics description language into the low level operations that SGCAL can interpret. This is useful only for fancy output.

As it processes an input file, SGCAL can be requested to output index and table-of-content information to an auxiliary file.

SGCAL makes a single pass over the input text. It cannot therefore handle forward references directly. However, it contains a mechanism for remembering the values of certain variable settings, and re-using them on a second pass. A script called **sgcal-fr** is provided for running SCCAL two (or sometimes three) times, in order to resolve forward references.

As well as **sgcal**, **sgtops**, and **sgpoint**, two further programs are provided as part of the SGCAL distribution:

- **sgbuildhy** is a program for building the indexed hyphenation dictionary that is used by SGCAL.
- **sghytest** is a program for testing hyphenation.

The rest of this document is divided into a number of parts.

Part I contains an introduction to SGCAL text processing. It explains the standard style that is provided in the SGCAL library, and introduces most of the basic facilities. Part II contains a complete specification of SGCAL, but with little introductory material, and scant motivation for the various facilities. Part III contains specifications of the auxiliary programs **sgtops**, **sgpoint**, **sgbuildhy**, and **sghytest**.

This document was constructed from a number of separate documents in September, 2003, when the SGCAL source code was arranged into a source distribution that could be built using the conventional 'configure', 'make', 'make install' method. At that time, the documentation had not been touched for about ten years. It was brought up-to-date as regards the current facilities and specification, but there was no serious overall re-editing, because of lack of time. This explains the variations in style, and the lack of an index.

Part I

Introduction to text processing with SGCAL

2. Basic SGCAL Concepts

2.1 What is text processing?

The terms *word processing* and *text processing* are often confused. The former normally refers to a system or program that permits the user to lay out text on a screen more or less in the form in which it is subsequently printed. The latter is a more involved process in which the text of a document (often called the *copy*) is entered into a computer system together with additional information as to how it should be laid out and printed. A word processor or text editor can be used to do this. The extra information mixed in with the copy is called the *markup*, and it has the same function as the marks formerly added to a paper manuscript by a copy editor before sending it to be typeset, in the days before 'manuscripts' were electronic. The mixture of copy and markup is read by a text processing program, which formats the copy as requested and generates instructions for the device on which it is to be printed or displayed.

While the added complication imposed by a text processor may not always be appropriate for short documents, for longer ones there are several advantages. The device on which the input is prepared can be very much simpler than the ultimate printing device; a normal workstation can be used to prepare text for the most sophisticated phototypesetter, for example. Another advantage is that it is easy to format the same input text in different ways or for different output devices, provided care is taken in the marking up. Text processors also offer facilities such as automatic chapter, section and footnote numbering, floating inserts, creation of indexes and so on.

2.2 Specific and generic markup

Many text processors allow the user to include very specific instructions in the markup, for example *'leave 12 points of white space and indent the next line by 24 points'*, which might be used at the start of a paragraph. (A *point* is a traditional unit of length used in the printing industry; it is approximately 1/72 of an inch.) Including such *specific markup* in an input file is not a good idea because if there is any need to change it for some reason, every occurrence in the file must be sought out and changed.

The alternative is to use *generic markup*, which indicates the logical structure of the document without specifying how this structure is to be represented on the page. For example, the start of each paragraph is indicated by a markup instruction *'start of paragraph'*, and the start of a chapter by *'start next chapter with title such-and-such'*.

Of course, the text processor has ultimately to be told what *'start of paragraph'* actually means in layout terms. This is done by a series of definitions that can either appear at the start of the input, or in a separate file which is referenced from within the main input. Such a set of definitions specifies a *document style*. It is then easy to alter the layout parameters if the need arises, and, what is perhaps more important, the style is guaranteed to be consistent throughout the document.

2.3 Coding the markup in SGCAL

Because the input to a text processor is a conventional file of characters that can be typed on a keyboard, there has to be some way of distinguishing which characters are copy and which are markup. In SGCAL there are two distinct forms of markup encoding:

- *Directives* are major instructions which always occupy an entire input line by themselves. The line begins with a dot, followed by the name of the directive and possibly other information. Some examples of SGCAL directives are

  ```
  .display
  .section Coding the markup in SGCAL
  .index directives
  ```

  Directive names are normally in lower case (small letters) – in SGCAL, upper case and lower case letters are considered as distinct.

- *Flags* are the other form of markup; they normally appear mixed up with the copy, and normally consist of a character that is neither a digit nor a letter, possibly followed by other characters. Upper and lower case letters are distinct in flags as well as in directives, so, for example, `'$f'` and `'$F'` are two completely different flags.

  Flags are used to encode instructions that apply to the immediately surrounding text (for example, to change font) or to cause the insertion of a character that is not available on the normal keyboard. Some examples of SGCAL flags are

`$it{`	to change to italic text
`#`	to insert an 'exact space', of fixed width
`$alpha`	to insert a Greek alpha

When input is being prepared for SGCAL it is important that the copy and the markup not be confused. The typist must take special action if any line begins with a dot (which is not very likely) or if any of the special characters that begin a flag appears in the copy. The set of such characters can be changed by SGCAL directives, but the default set that is used with the standard styles is

`~`	tilde
`_`	underline
`#`	sharp sign, or 'hash'
`$`	dollar
`}`	closing curly bracket

If it is necessary to include one of these characters as part of the copy, it should be preceded by a 'commercial at' character (@). If you really want to print an 'at' character, you have to double it. Thus:

for `~` type `@~`
for `_` type `@_`
for `#` type `@#`
for `$` type `@$`
for `}` type `@}`
for `@` type `@@`

The flag sequence '`@`' is called the *quote flag*, and it can also be used to insert a dot at the start of a line, should this ever be necessary.

As well as these flag characters, the '`-`' character has special significance in that it is treated as a possible hyphenation point. To prevent hyphenation in an individual instance, '`@-`' can be used.

2.4 Special characters

There is normally a distinction between opening and closing quotation marks in typographic fonts, the normal computer 'quote' character producing a closing single quote. For an opening single quote the character called 'grave accent' in the ASCII character set is used.

Double opening and closing quotes are obtained by typing two successive 'grave accents' or 'quote' characters respectively. Here is an example of some copy that uses this convention:

```
He said `I shall write to ``The Times'' tonight'.
```

Typographic fonts may also distinguish between a hyphen, an en-dash and an em-dash, which are all different lengths of short horizontal line. In SGCAL input, a single '`-`' character is treated as a hyphen, while en-dashes and em-dashes are entered as two and three successive '`-`' characters respectively. For example,

```
An en-dash is used for ranges, such as
19--42, and a spaced en-dash is used -- as
here -- to set off parenthetical comments.
The use of an em-dash---without
spaces---for this purpose is going out of
fashion in the UK.
```

2.5 SGCAL's standard macros

The SGCAL program implements a number of basic text processing facilities, including a number of particular directives and flags. It also provides means by which these basic facilities can be combined into higher level operations. A directive which is built from more primitive operations is known as a *macro directive*, and a collection of macro definitions is called a *macro library*. Additional flags can also be defined.

SGCAL is intended to be used in conjunction with a macro library, and the use of 'raw' SGCAL without any macro directives is exceptional. A standard library containing definitions for a standard document style is part of the distribution. The directives and flags that are described in the following sections include many that are in fact part of the standard library rather than 'raw' SGCAL.

3. SGCAL Input File Structure

An SGCAL input file is divided into two parts, the first of which is normally only a few lines long. This selects the style for the document and possibly makes some changes to the standard options. The remainder of the input file consists of the marked up copy, as described in what follows.

3.1 Selecting a standard style

The first line of an SGCAL file normally consists of a **library** directive specifying the name of the style, enclosed in quotes. For example:

```
.library "a4ps"
```

selects the style that is designed for PostScript output on A4-sized paper. A style can alternatively be specified as a parameter on the command that invokes SGCAL. This single line is all that is needed in many cases.

Certain features of some styles can be varied by setting parameters before using the **library** directive. For example, for the `a4ps` and `a5ps` styles, the size of type and the typeface family can be specified in this way. No other directives should normally appear before **library**.

3.2 Special character flags

A number of standard flags are provided to give access to certain special characters that are not in the normal ASCII character set. These standard flags, and the characters they represent, are as follows:

`-->`	→	right arrow
`<--`	←	left arrow
`<->`	↔	two-headed arrow
`($)`	£	pound sterling
`($E)`	€	Euro
`(c)`	©	copyright sign
`(TM)`	™	trademark
`$'`	′	feet (or minutes)
`$.`	•	bullet

An example of the use of these special characters is as follows:

```
.library "a4ps"
6$' of pipe cost ($)7.85.
```

SGCAL also contains support for letters from the Greek alphabet. However, these are not available by default, and it is necessary to obey the directive

```
.usegreek
```

to gain access to them. The names of the flags are '`$alpha`', '`$beta`', etc. for lower case letters, and '`$Alpha`', '`$Beta`', etc. for the capitals. Similarly, the directive **useaccents** defines a set of flags for printing accented characters. They have names like '`$eacute`'.

The more advanced user may wish at this point to include directives to vary the standard style, for example to indent all the displays, or change the amount of white space preceding each section. Some of the possibilities are described later in these introductory chapters. If this text is of any length, it may be convenient to keep it in a separate file which is inserted into the main file by means of the **include** directive, as for example

```
.include "header"
```

The **include** directive can also be used to put together a single document from a number of separate files.

3.3 An example of SGCAL input

This is an example of a complete input file for SGCAL which illustrates the general style. The markup items are described individually later in this document. The output produced from this file follows on the following page.

```
.library "a4ps"
.chapter An Example of SGCAL Output
This is an example of the output produced by SGCAL
when it   processes   the  input on the previous
page using the style definition ‘a4ps’.

.section A short story
Once upon a time there was a $it{beautiful}
princess who often used to go for long walks by
herself in the woods near her castle.

.subsection The plot thickens
One day, while she was out walking, she was
confronted by a fierce
.display
$c $bf{D R A G O N}
.endd
which was spitting fire and flame.

.subsection The dilemma
Fortunately the princess had grown up in the
electronic age, and knew all
about dragons and other monsters.
.footnote
In her palace she had a huge collection
of home computers.
.endf
Should she
.numberpars
Zap it with her laser cannon?
.nextp
Lure it to the
bottomless pit just around the next corner?
.nextp
Utter the magic spell given to her by the
Great Binary Wizard?
.endp

.subsection Conclusion
The dragon, seeing that it had met its match,
surrendered, and they both lived
happily ever after.
```

An SGCAL input file

4. An Example of SGCAL Output

This is an example of the output produced by SGCAL when it processes the input on the previous page using the style definition ‘a4ps’.

4.1 A short story

Once upon a time there was a *beautiful* princess who often used to go for long walks by herself in the woods near her castle.

4.1.1 The plot thickens

One day, while she was out walking, she was confronted by a fierce

D R A G O N

which was spitting fire and flame.

4.1.2 The dilemma

Fortunately the princess had grown up in the electronic age, and knew all about dragons and other monsters.[1] Should she

(1) Zap it with her laser cannon?

(2) Lure it to the bottomless pit just around the next corner?

(3) Utter the magic spell given to her by the Great Binary Wizard?

4.1.3 Conclusion

The dragon, seeing that it had met its match, surrendered, and they both lived happily ever after.

[1] In her palace she had a huge collection of home computers.

5. SGCAL Markup for Running Text

In a previous chapter the general form of SGCAL markup was described. In this chapter the particular ‘marks’ relevant to sections of running text are defined and explained.

5.1 Paragraphs

Most paragraphs of running text can be typed verbatim. The only times when markup is required are

- When one of the characters that SGCAL uses to introduce a flag is part of the copy;
- When a character not in the normal printing set is encountered;
- When a change of font or underlining state is required;
- When some change from the normal spacing or line splitting rules is wanted.

5.1.1 Characters that introduce flags

These are the characters ‘`$`’, ‘`#`’, ‘`~`’, ‘`_`’, ‘`}`’, and a full stop at the beginning of an input line; any occurrences that are part of the copy must be preceded by ‘`@`’, for example:

```
... the @$20,000 question ...
```

Hyphens in the copy are taken as possible line-splitting places unless preceded by ‘`@`’.

5.1.2 Font changes and underlining

The following standard flags are defined for changing font:

`$rm{`	to select the roman font
`$it{`	to select the italic font
`$sl{`	to select the slanted roman font
`$bf{`	to select the bold font
`$tt{`	to select the typewriter font
`$ss{`	to select the sanserif font
`$sc{`	to select the ‘small caps’ roman font

Whenever there is a change of font, the previous font is remembered on a stack, from where it can be recalled by means of a flag consisting of a single closing curly bracket. An example of a sentence that uses several fonts is

```
These words are $it{italic}, $bf{bold}, and roman.
```

Advanced users who make use of other fonts are recommended to create suitable mnemonic flags.

Sections of text to be underlined are bracketed by the *underline flag*, which consists of a single underline character. For example, the input

```
Here is an _underlined_ word.
```

produces as its output

Here is an underlined word.

SGCAL is very flexible in the way its flags are defined, and it is possible to change the meaning of any flag sequence. For example, if a document has been marked up with underlines, these can be changed to, say, italic *without changing the main input*, by re-defining the meaning of the flag sequence ‘_’. In general it is best to mark up for the most sophisticated form of output that is ever likely to be used for the document, as it is easier to re-define the flags for a simpler output than *vice versa*.

5.1.3 Formatting paragraphs

When SGCAL is formatting a paragraph it tries to fit as many words onto each output line as possible. The length of input lines is of no account, nor is the number of spaces between input words. For example, the input for the present paragraph could be as follows:

```
When SGCAL is formatting a paragraph
it tries  to  fit  as many words onto
each output line as possible.  The length of input
lines is of no account,
nor    is   the number of spaces between words.
For example, the input
for the present paragraph could be as follows:
```

Each line of a paragraph is stretched out to reach the right hand margin if SGCAL is operating with both left and right justification enabled. The stretching is done by including extra spaces between words. For output in which spaces are not significantly narrower than the printing characters (plain output), the result is not very pleasing. Right-justification can be turned on or off at any time by means of the **justify** directive. Right-justification is turned off by default for plain output styles.

It is occasionally necessary to tell SGCAL not to split a line at a particular place. The flag '`$>`' is used to specify a *non-splitting space*, for example

```
The author's name is A.N.$>Other.
```

Such a space can, however, be stretched if necessary. Another sort of space that behaves just like a printing character and neither stretches nor is a possible splitting point is called an *exact space*. It is indicated by the flag '#'.

A new paragraph is started whenever an empty input line or the **newpar** directive is encountered. SGCAL reads the entire text of a paragraph before splitting it up into lines. The first line of a paragraph is never placed at the bottom of a page, nor the last line at the top of a page, except in the case of single-line paragraphs.

When a paragraph does not start at the top of a page, an amount of vertical space is output above it such that the new paragraph is preceded by at least the amount specified by the **parspace** directive. Style definition files normally select a default value appropriate to the output format, but this can be changed any number of times. For example,

```
.parspace 36
```

specifies that, from now on, at least 36 points of space are to precede each paragraph.

A temporary indent is set for the first line of each paragraph. The size of this is controlled by the **parindent** directive, and again the style definition files select an appropriate default. It is also possible to override the indent for one individual paragraph by means of the **tempindent** directive (see below). Normal typographic convention is to use *either* blank space *or* an indent to signify a new paragraph, but not both.

If it is necessary to force a new line of output without treating it as a new paragraph, the **newline** directive can be used. This can optionally be followed by the word 'justify', which requests that the line just terminated be right-justified, that is, stretched out to end flush with the right-hand margin.

5.2 Chapters, sections, subsections and sub-subsections

Paragraphs of text are usually grouped into larger units, and SGCAL has provision for up to four different levels, not all of which need be used. For example, a novel may consist only of chapters, whereas a technical note may be divided only into sections. The start of each portion of the text is indicated by one of the following directives:

```
.chapter <title>
.section <title>
.subsection <title>
.subsubsection <title>
```

For example

```
.chapter SGCAL Markup for Running Text
.section Paragraphs
```

The way in which the titles are printed is determined by the macros and flags in the style definition file: For example, when formatting for ASCII output, chapter titles are printed in upper case, but on a laser printer they are printed in large type. In both cases they are centred.

If a chapter or section title starts with a double quote character, it is necessary to include the entire title in double quotes, and to double up any double quote characters that are in it. For example:

```
.section """Special"" features"
```

Titles can always be enclosed in double quotes, but they are mandatory only in this special case.

5.3 Indentation

Automatic indentation at the start of each paragraph is controlled by the **parindent** directive, as described above. Paragraph indents are relative to the overall indentation, which is set by the **indent** directive. For example,

```
.indent 6 em
```

specifies an indentation of six ems in the current font, as has been done for this text. An *em* is a traditional printer's unit of horizontal width, approximately the size of a capital 'M'. In SGCAL it is taken as the width of an exact space. Indents may also be specified in points, inches, or centimetres:

```
.indent 36
.indent 1 in
.indent 2.54 cm
```

The paragraph indent as set by **parindent** is taken relative to the overall indent, and when there is a positive indentation set, the paragraph indentation may be set to a negative value, giving the effect demonstrated here. Other effects can be obtained by using the *temporary indent* facilities, of which the paragraph indent is really just a special case.

It is sometimes necessary to specify indents relative to the current indent. This can be done by writing an arithmetic expression as part of the **indent** directive, and using the text '`~~sys.indent`' to represent the current indent. For example,

```
.indent ~~sys.indent + 24
```

This is an example of the use of the insert flag '`~~`' to insert the value of an SGCAL *variable* into the text. The name of this particular variable is **sys.indent**. Variables whose names begin with **sys.** are *system variables*, and their contents are automatically maintained by SGCAL. The contents of **sys.indent** are defined to be the current indentation value, expressed in points.

The directive

```
.tempindent <e1> <e2>
```

where <*e1*> and <*e2*> are arithmetic expressions, specifies an indentation of value <*e1*> which lasts for the next <*e2*> output lines. If <*e2*> is omitted, the temporary indent lasts for one line only. For example, the directive

```
.tempindent 4 em 2
```

has been used here. In effect, the directive

```
.tempindent ~~sys.indent + ~~sys.parindent
```

is automatically inserted at the start of every paragraph. A subsequent explicit **tempindent** directive can override this, making

```
.newpar
.tempindent <e1> <e2>
```

have a completely different effect to

```
.tempindent <e1> <e2>
.newpar
```

in which the **tempindent** directive is cancelled by the subsequent **newpar**.

It is often necessary to place text in the indent space at the start of paragraphs. This can be done by specifying a temporary indent of zero. Following the special text, the flag '`$i`' is used to restore the alignment. This is one of SGCAL's *tab flags*, more details of which are given in a later section; it tabs to the current overall indent. An example of its use is as follows:

```
.indent 6 em
.tempindent 0
(1)$i This is the start of a numbered paragraph.
```

5.3.1 Enumerated paragraphs

Three common cases of paragraph indentation have been 'wrapped up' into a macro command which allows paragraphs to be numbered, lettered, or 'bulleted' with an arbitrary character, respectively. The directive **numberpars** indicates the start of a sequence of 'numbered' paragraphs, indented relative to the current indent. Each subsequent paragraph must be marked by the directive **nextp**, and the sequence is ended with **endp**. If **numberpars** is used on its own, the paragraphs are numbered with ordinary arabic numerals which are placed in parentheses in the indent of the first line of each paragraph. For example, the input

```
.numberpars
This is the first numbered paragraph. If
it is long enough to require more than one line,
they are all indented by the same amount.
.nextp
The is the second numbered paragraph.
.endp
```

produces the following output:

(1) This is the first numbered paragraph. If it is long enough to require more than one line, they are all indented by the same amount.

(2) This is the second numbered paragraph.

Blank space is always left between the paragraphs, and there is additional space before the first and after the last. If roman numerals are required, one of the words 'ROMAN' or 'roman' can be added to the **numberpars** directive:

```
.numberpars roman
```

This gives lower case roman numerals; 'ROMAN' gives upper case ones. Similarly, to obtain lettered rather than numbered paragraphs, the **numberpars** directive is followed by 'alpha' or 'ALPHA'. Finally, if an arbitrary character sequence follows **numberpars** it is used to mark each paragraph. For example

```
.numberpars *
```

marks each paragraph with an asterisk. Calls to **numberpars** can be *nested*, causing 'inner' paragraphs to be further indented, as shown in this example:

```
$bf{Check List for Monday, 10th June}
.blank
.numberpars
Check that
.numberpars *
Luggage contents are correctly listed;
.nextp
The luggage is labelled.
.endp
.nextp
Include a first-class stamped postcard.
.nextp
$bf{Arrival at school:}
.numberpars roman
Children should arrive at School between
8.30 and 8.45 am.
.nextp
If the coach is not already outside the gates,
luggage should be deposited outside the Craft Room.
.endp
.nextp
The coach will depart at 9.00 am.
.endp
```

The output produced is as follows:

Check List for Monday, 10th June

(1) Check that

* Luggage contents are correctly listed;
* The luggage is labelled.

(2) Include a first-class stamped postcard.

(3) **Arrival at school:**

(i) Children should arrive at School between 8.30 and 8.45 am.

(ii) If the coach is not already outside the gates, luggage should be deposited outside the Craft Room.

(4) The coach will depart at 9.00 am.

Care must be taken to match each **numberpars** with an **endp**, as otherwise strange effects occur.

5.4 White space

There is a directive called **blank** which can be followed by a number, and which leaves that many units of vertical white space, unless SGCAL happens to be at the top of a page, in which case it has no effect. If no number is given, a single unit of space is left. The size of the unit depends on the style being used (and can be varied); it is normally about two-thirds of the depth of a line. There is a similar directive called **space** which leaves a fixed amount of white space, even at the top of a page, and whose meaning cannot be varied. Examples of these two directives are

```
.blank
.blank 3
.space 2 in
.space 5 cm
```

5.5 Doublespaced output

The directives **doublespace** and **singlespace** can be used to specify double and single spacing of the output, respectively. Single spacing is the default. For many of the standard styles, **doublespace** does not give literally double the spacing by default, as this often looks too deep.

5.6 Hyphenation

If a word contains a hyphen character, it is a candidate for splitting across a line break. There is also a *conditional hyphen* flag, '~'. Its occurrence in a word indicates an allowable splitting point. If the word is in fact split, a hyphen is inserted; if it is not, the word is closed up. Words may contain any number of hyphens or conditional hyphens, for example

```
...  photo~type~setter type-faces ...
```

To prevent a word from being split at a particular hyphen character, precede it with the quote flag '@'. For example:

```
Been there, done that, got the T@-shirt.
```

Words that contain neither real nor conditional hyphen characters may be split across linebreaks by means of SGCAL's hyphenation dictionary. Hyphenation occurs only when the line is sufficiently 'loose', that is, when the amount of space in the line is large relative to the number of space positions. To prevent automatic hyphenation, enclose the word in '`$nh{`' and '`}`'. For example:

```
Do not split $nh{hyphenation}.
```

5.7 Horizontal lines

Printers call straight lines (whether horizontal or vertical) 'rules'. SGCAL has flags for generating both kinds of rule, though vertical rules are not supported for plain output. For the simple case of a horizontal line right across the page, there is a directive called **rule**.

6. Notes, emphasis and indexing

6.1 Footnotes

Footnotes are placed in the input text at the point they are referenced, and SGCAL automatically supplies numbers for them. The numbers are reset at the start of each chapter. For fancy output only, you can request that the numbers be reset for each page, provided there are no more than nine footnotes per page, by including

```
.set perpagenotenumbers true
```

at the start of the input. The text of each footnote is enclosed between the directives **footnote** and **endf**, as in the following example:

```
Footnotes are normally printed at the bottom
of each page
.footnote
Like this.
.endf
and separated from the text by a short line.
```

Footnotes are normally printed at the bottom of each page[1] and separated from the text by a short line. The styles in which the reference numbers are printed, in the body of the text and at the start of the note, are specified by flags which can be changed by the user. Details are given below in section 8.2 (*Changing a standard style*).

6.2 Emphasis

A particular form of marginal annotation commonly found in manuals and other documents that undergo revision is the ‘emphasis bar’, which is a short vertical line printed in the margin to indicate where a document has been changed. SGCAL is set up so that any output lines containing sections of the text appearing between the directives **em** and **nem** are marked in this way. This paragraph is an example. For example,

```
.em
The line containing this text will be emphasized.
.nem
```

6.3 Indexing

The directive **index** causes the text that follows it on the same line, together with the page number, to be written to the index file. This must subsequently be sorted and consolidated by some means before being reprocessed by SGCAL.

[1] Like this.

7. Displayed Text

Display is a printer's word for material that is not part of the running text of a document, but instead is 'displayed' in some special fashion.

7.1 In-line displays

In SGCAL, displayed material that must be printed in sequence with the main text is enclosed between the two directives **display** and **endd**. When processing a display, SGCAL does not perform its normal line filling, but instead copies the input to the output, line for line. In addition it ensures that the display does not cross a page break.

At the start of a display the current line length, indent, and so on are the same as in the immediately preceding text, but any changes that are made inside the display are automatically cancelled when it is completed. The font is automatically changed to a fixed pitch font, but this can be changed by the use of '`$rm{`', '`$it{`', etc. At the end of a display the font reverts to what it was previously. Here is an example of the input to generate a display:

```
.display
  1 + 2 + 3 + 4 + 5 = 15
.endd
```

By default, SGCAL flags are still interpreted in displayed material, so changes of font can still be made and tabs (see below) can be used. However, it is sometimes useful to be able to disable this processing, for example, when including text from other sources. A common case is the inclusion of computer program fragments, which often make use of the special characters that introduce SGCAL flags. If the word 'asis' is added to the **display** directive, the displayed text is not scanned for embedded flags. However, SGCAL directive lines are still recognized – this is necessary, of course, in order that **endd** should terminate the display. Here is an example that displays a fragment of a BCPL program. Without the use of 'asis', the '`$`' and '`#`' characters would need to be preceded by '`@`'.

```
.display asis
LET hypot(a, b) = VALOF
$(
LET hh = a#*a + b#*b
RESULTIS realsqrt(hh)
$)
.endd
```

It is sometimes desirable to allow very long displays to flow over page breaks. To do this, add the word 'flow' to the **display** directive:

```
.display flow
<many lines of input>
.endd
```

The first six lines of such a display always appear on the same page. If both 'flow' and 'asis' are needed for a display, 'asis' must come second.

```
.display flow asis
<many lines of verbatim input>
.endd
```

Even though displayed material is copied line for line from input to output, it is possible to request that individual lines be justified, that is, stretched out to end flush with the right-hand margin, by increasing the sizes of the spaces in the line. The **newline** directive with the **justify** option achieves this.

7.2 Figures and tables

The form of display described above appears in the output at the same point as it appears in the input. For large amounts of display text it is sometimes more convenient to use a *floating display* which appears near but not necessarily at the point of input. Such displays are normally figures with captions, and SGCAL has two directives for processing them which are used as in the following example:

```
.figure This is the figure's title.
**************************************************
The text for the figure, normally quite a few lines.
**************************************************
.endfigure
```

The output from this example appears as figure 1 below.

```
**************************************************
The text for the figure, normally quite a few lines.
**************************************************
```

Figure 1: This is the figure's title.

Notice that SGCAL has supplied a figure number automatically. The reference to the figure was generated by including the following text *before* the figure:

```
... appears as figure ~~figurenumber below.
```

This is an example of the use of the insert flag '~~' to insert the value of an SGCAL *variable* into the text. The name of this particular variable is **figurenumber**.

The **endfigure** directive can be followed by a dimension. This specifies an amount of extra white space that is to be left below the figure title.

If there is room on the current page for the figure, it is output immediately. Otherwise it and any subsequent figures are held over to the top of the next page (even if the subsequent figures would in fact fit).

Another pair of SGCAL directives is **table** and **endtable**. They work in exactly the same way as figures, except that tables have their own sequence of numbers, held in the variable **tablenumber**. The fact that a figure is being held over to the next page does not force tables to be held over (and *vice versa*).

7.3 Subscripts and superscripts

Displayed material often requires the use of subscripts and superscripts. Three SGCAL flags are available for this purpose:

(1) '`$U`' is the up flag, which causes subsequent text on the line to be raised with respect to what went before;

(2) '`$D`' is the down flag, which causes subsequent text on the line to be lowered with respect to what went before;

(3) '`$L`' is the level flag, which causes subsequent text on the line to be printed at the normal line level.

The ability to print characters above and below the line is only available for fancy output. If one of the ASCII styles is being used, these flags have no effect. Here is a typical example of their use:

```
.display
H$D$$2$UO + SO$D$$3$L --> H$D$$2$USO$D$$4
.blank 2
e = mc$U$$2
.endd
```

The output is

$$H_2O + SO_3 \rightarrow H_2SO_4$$

$$e = mc^2$$

Note the use of the null flag '$$' to terminate these flags when a digit follows. This is necessary because if a number follows the up or the down flag, it is taken as the number of points to move up or down. If no number is present, the distance moved is one third of the current line depth.

7.4 Tabs

Several different kinds of tab are available in SGCAL. The *indent tab* has already been described above: it tabs to the current indentation position, and is represented by the flag '$i'.

A conventional tab operation is represented by the flag '$t' in the input. Tab positions are set by the **tabset** directive, which is followed by a list of column widths. Here is an example of some tabbing input:

```
.display
.tabset 7em 11em 7em 9em
12345678901234567890123456789012345678901234567890
X $t XX $t XXX $t XX $t X
.endd
```

The output that is produced is:

```
12345678901234567890123456789012345678901234567890
X      XX          XXX    XX       X
```

Any space characters in the input preceding or following the tab flag are ignored. The tabs in the example above are *left* tabs, because the portions of text are set with their left-hand edges at the tab positions. SGCAL also supports *right* tabs and *centred* tabs, and these are set in the **tabset** directive by typing the letters 'R' or 'C' following the tab position, respectively. (It is also permitted to type 'L' to indicate a left tab explicitly.) Here is an example showing all three kinds of tab:

```
.display
.tabset 7em 13em R 11em C 14em
12345678901234567890123456789012345678901234567890123456789012345678901234567890
X $t XX $t XXXX $t XXX $t XX
.endd
```

which produces the following output:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
X      XX       XXXX          XXX             XX
```

Finally, there are two special tabs that are useful in headings and footings, the *centring* tab and the *ending* tab, represented by the flags '$c' and '$e', respectively. Any text following the ending tab on the input line is moved to the right until it ends at the right-hand margin. Any text following the centring tab, up to the next tab if present, or to the end of line if not, is centred in the current line width. Thus the input

```
.display
the left $c the middle $e the right
.endd
```

produces the following output line:

7.5 Heads and feet

Headlines are lines that appear at the top of a page, above the main text, and similarly *footlines* appear at the bottom. They are used for page numbers and titles of various sorts. The standard arrangement in SGCAL is to have two footlines, the first being blank and the second containing a centred line number, and no headlines.

The directive **nofoot** can be used to suppress the footlines altogether. Alternatively, the size of the headline or footline areas can be changed by the **headdepth** and **footdepth** directives. Thus, for example,

```
.headdepth 3 ld
.footdepth 4 ld
```

specifies three headlines and four footlines instead of the default zero and two. The abbreviation ‘ld’ after a number stands for ‘line depths’. The material that appears in the headline area is defined between the directives **head** and **endhead**, for example

```
.head
The $c First $e Headline
.newline
Another $e Headline
.endhead
```

and similarly footlines are defined between **foot** and **endfoot**. There may be more than one occurrence of these directives in an SGCAL input file; at the start or end of a page the most recent definition is used. If there are not enough headlines to fill the headline area, the bottom is left blank; if there are not enough footlines to fill the footline area the top is left blank.

The most common use of headlines and footlines is for printing the page number, and this can be done in SGCAL by including the text ‘`~~sys.pagenumber`’ where the page number is required.

Another useful system variable is **sys.date**, which can be used to insert the date of processing into the text, as in the following example of footline definition:

```
.footdepth 3 ld
.foot
$c [~~sys.page] $e ~~sys.date
$bf{FIRST DRAFT}
.endfoot
```

It is possible, by using the more advanced features of SGCAL, to cause the page numbers to be printed in roman numerals, to specify different headlines and footlines for odd-numbered and even-numbered pages, and to create ‘running’ headlines and footlines.

8. Advanced Features

The facilities described in previous chapters are enough to cope with many text processing jobs. There is, however, a lot more to SGCAL. In this chapter some ways of changing the standard actions described above will be covered, as well as some additional features. New users are advised to become familiar with the preceding material before reading further.

8.1 Variables

The directives for handling chapters, sections, footnotes and so on make use of *variables* for counting. Many other features of the standard styles are controlled by variables. Additional variables can be defined and used for many purposes, some examples of which are given below. The **set** directive is used to define a variable and its contents, which may be a number or a string of characters. One simple use of a variable is to hold a long string that appears many times in the document, for example:

```
.set scal "supercalifragilisticexpialidocious"
```

This can then be inserted whenever required by simply typing

```
~~scal
```

wherever it is needed.

8.2 Changing a standard style

In this section some of the variables and flags which control the standard styles are described. The may all be changed by the user in order to achieve a different effect.

8.2.1 Numbering chapters and sections

The variables which contain the current chapter, section, subsection and sub-subsection numbers are called **chapter**, **section**, **subsection** and **subsubsection**. These can be used in several ways:

(1) If any of them is set to contain a negative number, the numbering of the relevant item is suppressed. Thus

```
.set chapter -1
```

switches off chapter numbering, while leaving section, subsection and sub-subsection numbering on. It is sometimes useful to turn chapter numbering off for a preface, then turn it on again (by setting the variable **chapter** to zero) just before the call to the **chapter** directive for chapter 1.

(2) A long document can be processed in parts by setting the appropriate chapter or section number at the head of each part.

(3) A current number can be saved in a different variable for later use in a reference, or output as part of an index entry.

8.3 Display indentation

Displays are automatically indented by an amount specified by the variable **displayindent**, which defaults to zero. Thus to indent all displays by 36 points, the following directive would be used:

```
.set displayindent 36
```

The indent is always reset on exit from a display.

8.3.1 White space

The amount of vertical white space surrounding headings can be controlled by changing the following variables:

`chapspaceb`	space after chapter headings
`sspacea`	space before section headings
`sspaceb`	space after section headings
`ssspacea`	space before subsection headings
`ssspaceb`	space after subsection headings
`sssspacea`	space before sub-subsection headings
`sssspaceb`	space after sub-subsection headings
`fnspace`	space between footnotes

8.3.2 Heading styles

The styles in which headings appear (large type, bold face, etc.) are controlled by flags which are placed before each heading text. They are:

`$chead{`	chapter heading
`$shead{`	section heading
`$sshead{`	subsection heading
`$ssshead{`	sub-subsection heading
`$fkt{`	footnote key in text
`$fkn{`	footnote key for note
`$ftitle{`	figure title
`$ttitle{`	table title

The heading texts are always terminated by a closing curly bracket. Before re-defining one of these flags, it is first necessary to cancel it using the **cancelflag** directive. Thus, for example, to arrange for section headings to be in sanserif type:

```
.cancelflag $shead{
.flag $shead{ "$ss{"
```

The variable **hnspace** contains a string to be inserted between the number of a chapter, section or subsection and its title. By default this string is a single space, but it can be changed as required.

8.4 Number formats

Page, chapter, section and subsection numbers are normally printed in arabic numerals. Roman numerals can be requested by means of the **format** directive, which is followed by the name of the relevant variable and one of the words ‘roman’ or ‘ROMAN’ for lower or upper case numerals, respectively. The page number is held in a *system variable* called **sys.pagenumber**, because it is incremented automatically inside the SGCAL program. For example,

```
.format chapter ROMAN
.format sys.pagenumber roman
```

would number chapters in uppercase roman numerals, and pages in lower case. For numbers in the range 1–26, an alphabetic format is also available, again in either case, specified by the letters ‘alpha’ or ‘ALPHA’ in a **format** directive.

8.5 Varying heads and feet

Two common requirements for headlines and footlines are the inclusion of the current chapter or section title, and varying the text on alternate pages. The first of these can easily be done by making use of the variables **chapname**, **sectname** and **ssectname**, which are automatically maintained by SGCAL. The second requires the use of SGCAL’s conditional directive, **if**. This is a powerful facility which can be used to test many conditions; the simple requirement here is to test whether the current page number is odd or even. Its use is best demonstrated by an example, which places the current chapter title on the left at the top of even-numbered (left hand) pages, and the current section title at the top of odd-numbered pages. In both cases the titles are forced to be in the sanserif typeface.

```
.head
.if even ~~sys.page
$ss{~~chapname}
.else
$e$ss{~~sectname}
.fi
.endhead
```

The two parts of the conditional construction are separated by **else**, and the whole thing is terminated by **fi**. Nested conditions are permitted.

When a new chapter or section is being started, SGCAL sets the name variable to the null string until the title has been output. The above example would therefore result in a null headline at the start of a chapter and whenever a section started at the top of a page (which is normally what is wanted).

8.6 Thin and wide spaces

The thin space flag, '$<', can be used to generate a small amount of horizontal space, with a width approximately one sixth of a normal space (except of course for ASCII output, which does not have variable spaces). There is also an 'extra-stretchy' space flag, '$<>', which has the width of an ordinary space, but which, if it appears in a line that is being right-justified, absorbs all the stretch, leaving the other spaces unaltered. This can be used to force text to the right hand side of the page at the end of a paragraphs, as follows:

```
This is the text of the paragraph.
The quick brown fox jumps over the lazy dog.
Use the extra-stretchy space at the end,$<>thus.
```

The output produced is:

This is the text of the paragraph. The quick brown fox jumps over the lazy dog. Use the extra-stretchy space at the end, thus.

Part II

Full specification of SGCAL

9. Command line interface

The **sgcal** command line has a number of options that fall into three categories:

- ‘Normal’ options that are used for the most common used of the program;
- Two special options that are concerned with handling forward references;
- Some additional options that allow alternate library files to be used.

9.1 A ‘normal’ command line

The syntax for the most common calls to SGCAL is as follows:

```
sgcal [-to <file>]
      [-style <style(s)>]
      [-index <file>]
      [-aside <file>]
      [-define <name>[=<value>] ...]
      [-id | -help]
      [-verbose]
      [[-from] <file(s)>]
```

The keywords **-style**, **-index**, **-aside**, **-define**, and **-verbose** may be abbreviated to their first letters, and **-o** is a synonym for **-to**.

Up to nine input file names, separated by spaces, can be given. They are read in order. If no names are given, the standard input is read. If no output file is given, there are two possibilities:

- If no input is given, the output is written to the standard output.
- Otherwise, the name of the first input file is used to construct the name of an output file. Any existing suffix is removed from the input file name, and then ‘`.sgout`’ is added.

The **-style** keyword can be used to supply the name of a standard style, for example

```
sgcal myfile -style online
```

Has the same effect as the line

```
.library "online"
```

at the beginning of the first input file. In fact, up to nine file names can be given with the **-style** keyword, enabling auxiliary style definition files also to be specified by this means. For example:

```
sgcal myfile -style a4ps psgreek
```

The **-define** keyword can be used to cause named SGCAL variables to be set at the start of processing. If no value is given for a variable, it is set to ‘true’. The keyword can be followed by up to nine variable settings, separated by spaces. If fewer than nine are given, and the name of an input file follows, the **-from** keyword is required to indicate that it is not another variable setting. An alternative is to specify the input file first, as in the example below.

Setting variables on the command line can be useful when parameterizing an input file so that its output depends on which variables are set. For example:

```
sgcal myfile -define local nofigures
```

The **-index** and **-aside** keywords both define additional output files. For details of the data that is written to these files, see the descriptions of the **index** and **aside** directives in chapter 21. Warnings are issued if either of the index or aside files are missing when they are required, but SGCAL continues processing.

The **-id** option causes SGCAL to output its version number and exit; the **-help** option causes it to output a summary of the command line options.

The **-verbose** option requests SGCAL to output, to the standard error file, comments of the form 'Page <*n*>' as it starts to process each page. If the verification output is displayed on a terminal (the default case) this makes it possible to monitor the progress of a long SGCAL run.

9.2 Handling forward references

If your input file contains forward references, you will need to run SGCAL twice so that they can be resolved. Any variable setting in the source that is referenced earlier than its definition must be set using the **rset** directive, instead of the ordinary **set**. (See sections 21.59 and 21.66 for the specification of these directives.)

For the first pass, SGCAL must be called with the **-rsetout** option, followed by the name of a scratch file. For example:

```
sgcal -rsetout /tmp/sgtemp myfile.sgcal
```

The values of the **rset** variables are written to the scratch file, and no errors are generated for references to unset variables.

For the second pass, SGCAL must be called with the **-rsetin** option, referencing the same scratch file. This reads the previously saved definitions at the start, so that the forward references can now be correctly substituted.

In some cases, inserting the value of a forward reference may change the page layout of the document, and this may affect the value of subsequent forward reference variables. The way to handle this is to supply both **-rsetin** and **-rsetout** for the second pass, and check for any changes.

All this housekeeping for forward references is handled for you if you call the **sgcal-fr** script instead of **sgcal**. Its parameters are the same as for a 'normal' SGCAL call; it handles all the **-rsetin** and **-rsetout** stuff automatically, using scratch files in **/var/tmp**. The **sgcal-fr** command runs SGCAL up to three times:

- If the scratch file is empty after the first pass, it implies there were no **rset** variables. A second pass is not needed, so the script exits.
- After the second pass, the new values of the **rset** variables are compared with the old; if they are the same, the script exits.
- Otherwise, a third pass is done. If there is a discrepancy this time in the variable values, **sgcal-fr** gives up.

sgcal-fr requires the input to be a file or files, so that they can be read multiple times. It does not work for the standard input.

9.3 Using alternate library files

Options are provided for changing the locations of various data files that SGCAL uses. The default location for these files is compiled into the binary. The default location is likely to be under **/usr/local**; below we show the default paths when the overall 'prefix' is just **/usr/local**. You only need these options if you want to use versions of the files that are different from those whose paths are built in to SGCAL.

- **-afmlib** specifies the directory in which AFM files may be found (**/usr/local/share/sgcal/AFM/**). These are files that specify the widths of characters in a font in an Adobe standard format. For the 36 standard PostScript fonts, SGCAL has character width data in its own format in its own 'font library'. For other fonts, you need to put an indirection into the SGCAL font library to make it search for an AFM file. See chapter 26 for details.
- **-hyphendata** specifies the file that contains SGCAL's indexed hyphenation dictionary (**/usr/local/share/sgcal/HyphenData**).
- **-enclib** specifies the directory in which font encodings for files whose width data is obtained from AFM files is found (**/usr/local/share/sgcal/Encoding**).

- **-fontlib** specifies the directory in which SGCAL's font information directory is found (**/usr/local/share/sgcal/**).
- **-library** specifies the directory in which SGCAL's library files can be found (**/usr/local/share/sgcal/**).

Except for **-hyphendata**, which names a single file, all these options name directories, and in fact they may list a number of directories, separated by spaces or colons. When looking for a particular file, each directory is searched in turn, from left to right.

9.4 Return codes from SGCAL

SGCAL issues the following return codes:

0	Success
4	Warnings only
8	Serious errors
12	More than 40 serious errors; run abandoned
16	Internal disaster; run abandoned

The multiples of four are a historical relic from the IBM mainframe days.

10. Overview of SGCAL processing

This chapter contains information about the general format of SGCAL input files and how they are processed. Details of particular directives and flag sequences are given in later chapters.

10.1 Input line format

Input to SGCAL consists of a mixture of text to be processed and *markup*, that is, additional information that tells SGCAL how the document is to be formatted. Input files are considered line by line. There are two kinds of line: *directive lines* and *text lines*. Directive lines are those that begin with the *directive flag*, which consists of a single full stop. For example,

```
.library "a4ps"
```

is a directive line. Such lines contain large scale instructions to SGCAL; they are used, for example, to start a new chapter, begin a footnote, or change the indentation.

All lines that are not directive lines (i.e. do not begin with a full stop) are text lines. They contain the text which is to be formatted, possibly interspersed with markup *flags*, whose form is described in section 10.4.

10.2 Standard styles

The SGCAL program itself provides a basic set of facilities for formatting paragraphs and pages. It does not contain directives for laying out ‘higher level’ objects such as sections and chapters. Such features are provided by a set of *standard styles* which exist the the SGCAL library. It is assumed in most of this document that a standard style is in use. However, users are free to create their own styles, possibly by modifying one of the standard ones, if the facilities provided are insufficient.

10.3 Macros

A macro is a sequence of SGCAL input lines that has been encapsulated and given a name. This name can then be used as if it were one of SGCAL’s built-in directives. For example, if a macro called **mymacro** has been defined to contain the lines

```
The quick brown fox jumps over the lazy dog.
Pack my box with five dozen liquor jugs.
```

then whenever SGCAL encounters an input line of the form

```
.mymacro
```

it behaves exactly as it would if this line were replaced by the two lines above. Macros can be defined with *arguments*, which can cause variations to be made to the text each time the macro is called. Macros are heavily used in the definitions of the standard styles. Details of the macro facility are given in section 21.42.

A macro can be given the same name as one of SGCAL’s basic directives, in which case it overrides the directive. However, the basic directive can still be accessed by following the leading dot with a percent sign. Thus, for example,

```
.newline
```

obeys a macro called **newline** if one exists; otherwise it obeys the basic **newline** directive. However,

```
.%newline
```

always obeys the basic **newline** directive.

The processing of macros (and included files – see section 21.32 and 21.37) happens at an early stage, and the main part of the SGCAL program processes a single sequential stream of input lines.

10.4 Flags

Flags are particular sequences of characters which are recognized in lines of input and which cause some special action, such as the insertion of the contents of a variable, or a change of font. For example, using the standard styles, the character string `$it{` causes subsequent text to printed in italic, until the character `}` is reached.

Most of the flag strings can be defined to suit the user's taste. The flags can be considered in three types, depending on the time at which they are recognized:

- The *join* flag is recognized only at the end of a line just read from an input file. It causes the next external line to be joined on as though it were part of the current line.
- The *insert* flag is recognized when scanning a line for inserted variables. This happens to both directive and text lines.
- All other flags are recognized in text lines only.

Details of particular flags are given in chapter 20 (*Basic flags*).

10.5 Case sensitivity

SGCAL input is case-sensitive. All names of directives and flags must be entered in the correct case. In practice, all the basic directives have lower case names, as do all the macros in the standard styles. The set of standard flag strings, however, makes use of both lower and upper case.

10.6 The setup section

The stream of input lines presented to SGCAL is in two parts: an initial *setup* section, followed by the main portion. The setup section must contain all the font bindings and the setting of page offsets (if required). These may *not* occur later in the input. The setup section is terminated by the occurrence of the first non-blank text line, or the directive **endsetup**. It consists, therefore, of any number of directive lines, possibly interspersed with blank lines.

10.7 Empty lines

Within the setup section, empty input lines are ignored. Within the main part of the text, empty input lines are converted to directive lines of the form

```
.newpar
```

This normally results in the start of a new paragraph, but the user may define a macro called **newpar** to specify different or additional action.

10.8 Tab characters in input

Tab characters in input lines are expanded into an appropriate number of space characters, assuming a tab stop every eight characters. This is done to accommodate text editors that insert tabs into files. Note that tab characters do *not* cause SGCAL to perform tabbing operations, which must be notated using the various tab flags (see sections 20.1, 20.6, 20.7, 20.12, 20.22, 20.40, and 21.70).

10.9 Processing of input lines

SGCAL processes its input sequentially, substituting the lines of a macro definition whenever a macro call is encountered. The sequence of events is:

(1) The next input line is obtained from the current input file or current macro, as appropriate.

(2) If input is from a file, the line is examined to see if it ends with the join flag (default string `+++`). If it does, the next line is read and joined on to it (the join flag being removed, of course). This operation is repeated if necessary. The maximum total input line length is 1024 characters. Longer input lines are arbitrarily split, and a warning message is output.

If input is from a macro, the join processing is not carried out. However, the join flag can be used in macro definitions, as it will be processed when the macro is first read in.

(3) The start of the line is examined to see whether it begins with the directive flag (a full stop), and it is thereby classified as a directive line or a text line. If it is a directive line in which the initial full stop is followed by a space character, no further processing is done on the line at all. This provides a means of inserting comments into SGCAL input files.

(4) If the line is a genuine directive line, it is scanned for insertions. An insertion is recognized when an instance of the insert flag (default string ~~) is immediately followed by a letter or a digit. The appropriate system or user variable is inserted into the line, as described in chapter 13 (*Variables*). To prevent recognition of the insert flag in a directive line, it can be preceded by the quote flag (whose default is the single character @).

After a directive line has been processed for insertions, the name of the directive is extracted. If the directive flag at the start of the line is followed by a percent character, the name that follows is taken as the name of a basic, built-in directive. Otherwise, if the name immediately follows the directive flag without an intervening percent character, a search is first made for a macro directive of that name, and only if that fails is the name interpreted as a basic directive name.

Because directive lines are scanned for insertions before the name of the directive is extracted, it is possible for an insertion to change the name of a directive. It is also possible for an insertion to cause a space to appear immediately after the directive flag, thus turning the directive line into a comment, which is then not processed further (though all insertions in the line are done before this check is applied).

If the directive is a macro directive, SGCAL arranges for the lines that make up the macro body to be processed next. Otherwise, if the directive is one of the built-in basic directives, the appropriate action is taken. Details of the individual basic directives are given in chapter 21.

Note that flags other than the insert and quote flags are never recognized in directive lines.

(5) When a macro directive is obeyed, the remainder of the line following the name is read as a series of *arguments* for the macro. Different macros have different numbers of arguments. A space character in the line separates different arguments, unless the text for a particular argument is enclosed in double quotes. However, when the final argument of the macro is reached, the entire remainder of the line is assigned to it, whether or not it is enclosed in quotes.

(6) If the line is a text line, it is firstly scanned for insertions, exactly as a directive line. Then it is re-scanned for occurrences of any other flag sequences (thus a flag sequence is recognized in inserted text). Finally, it is added to the buffer in which the current paragraph is being built.

The above description applies to the majority of input lines, but there are some directives which cause the lines which follow them to be processed in a different manner.

The **if** directive can be used to cause portions of the text to be skipped, and not included in the output. The lines in the skipped portion are not normally processed at all, but if a macro directive is encountered in these lines, it is expanded into its constituent lines (except in one special case, described below). This makes it possible to define macro pairs which include the conditional directives **if**, **else**, **elif**, and **fi** within their bodies.

The special case where a macro is *not* expanded while skipping lines is when the macro is already active. This makes it possible to write recursive macros, that is, macros which call themselves, either directly or indirectly.

The **aside**, **call**, and **longcontrol** directives cause a number of following lines to be processed specially. Inserts and macro expansions are applied to these lines, but no other processing is done.

10.10 Special characters

The *character* flag (see section 20.8) provides a means of entering up to 256 different text characters. The effect of printing any character on an output device is dependent on the device itself. The common characters (letters, digits, punctuation) normally follow the ASCII encoding, and are the same on most devices.

PostScript printers are special, in that their standard fonts contain characters which do not have a default encoding, and the user may specify which codes correspond to which characters, on a per-font basis, changing the default encoding if necessary.

SGCAL, in combination with **sgtops**, uses the standard PostScript encoding for those characters which do have standard codes. For text fonts that use the standard encoding, the following additional codes are defined:

0	Á	Aacute	1	Â	Acircumflex
2	Ä	Adieresis	3	À	Agrave
4	Å	Aring	5	Ã	Atilde
6	Ç	Ccedilla	7	É	Eacute
8	Ê	Ecircumflex	9	Ë	Edieresis
10	È	Egrave	11	Í	Iacute
12	Î	Icircumflex	13	Ï	Idieresis
14	Ì	Igrave	15	Ñ	Ntilde
16	Ó	Oacute	17	Ô	Ocircumflex
18	Ö	Odieresis	19	Ò	Ograve
20	Õ	Otilde	21	Š	Scaron
22	Ú	Uacute	23	Û	Ucircumflex
24	Ü	Udieresis	25	Ù	Ugrave
26	Ÿ	Ydieresis	27	Ž	Zcaron
28	Ý	Yacute	29	Đ	Eth
30	Þ	Thorn	31	™	trademark
128	á	aacute	129	â	acircumflex
130	ä	adieresis	131	à	agrave
132	å	aring	133	ã	atilde
134	ç	ccedilla	135	é	eacute
136	ê	ecircumflex	137	ë	edieresis
138	è	egrave	139	í	iacute
140	î	icircumflex	141	ï	idieresis
142	ì	igrave	143	ñ	ntilde
144	ó	oacute	145	ô	ocircumflex
146	ö	odieresis	147	ò	ograve
148	õ	otilde	149	š	scaron
150	ú	uacute	151	û	ucircumflex
152	ü	udieresis	153	ù	ugrave
154	ÿ	ydieresis	155	ž	zcaron
156	ý	yacute	157	ð	eth
158	þ	thorn	159	©	copyright
160	€	Euro			
209	¼	onequarter	210	½	onehalf
211	¾	threequarters	212	¦	brokenbar
213	¹	onesuperior	214	²	twosuperior
215	³	threesuperior	216	¬	logicalnot
217	±	plusminus	218	−	minus
219	÷	divide	220	×	multiply
221	°	degree	222	µ	mu
223	®	registered			

SGCAL and **sgtops** also make some additional encoding definitions for the ZapfDingbats PostScript font. The following are added:

0	❮	a205	1	❰	a206	2	❯	a85
3	❱	a86	4	❲	a87	5	❳	a88
6	❨	a89	7	❩	a90	8	❪	a91
9	❫	a92	10	❬	a93	11	❭	a94
12	❴	a95	13	❵	a96			

10.11 Paragraph processing

SGCAL collects the text of an entire paragraph in store before splitting it up into lines and allocating those lines to a page. The end of a paragraph is indicated by one of the directives **newline**, **newpar**, **newcolumn**, **newpage**, **space**, **cspace**, or **multicolumn**, or by a change in the line filling state, or by reaching the end of the input.

SGCAL does not normally generate paragraphs where the final line contains only a single word, unless the word is wider than 18 points.

When SGCAL is not filling lines, that is, when each line of input corresponds to one line of output, the behaviour is as if there were a **newline** directive immediately before each text line. (Note that this is not the same as a **newline** directive *after* each text line – it means that the **nosep** directive can be used when filling is disabled.)

There are two parameters which control the way in which paragraphs are allocated to pages; they are called *minparB* and *minparT*. When a paragraph is complete, SGCAL checks to see whether there is enough room on the page for the entire paragraph, or at least *minparB* lines. If not, it starts a new page.

If SGCAL can fit only part of the paragraph on the current page, it checks the number of lines that will be printed on the following page. If this is less than *minparT* lines, then one line less is printed at the bottom of the first page. If this would result in fewer than *minparB* lines appearing on the first page, then the entire paragraph is printed at the start of the second page.

The default values for *minparB* and *minparT* are both 2, and there is currently no way of changing them. This means that, provided paragraphs are longer than one line, neither ‘widow’ nor ‘orphan’ lines are generated. Vertical page stretching (see below) can often smooth out the appearance of pages where one line has been moved forward to improve the appearance of the following page.

The way paragraphs are handled means that certain variables are effectively updated only after a paragraph is complete. For example, the variable **sys.pagenumber** contains the number of the page on which the previous paragraph ended. If its contents are inserted into the middle of a paragraph, this may or may not be the number of the page on which it is printed. The variable **sys.usedonpage**, which measures how much of a page has been used, is similarly only updated at the end of a paragraph.

Certain directives are synchronized with the text in a paragraph. For example, the directives for changing the indentation and line length can be included in the middle of a paragraph, and they will take effect at the start of the next output line following the point at which they appear. Other directives are not so synchronised; changing the line depth in the middle of a paragraph, for example, affects the whole paragraph.

10.12 Tab processing

The processing of tabs (as specified by SGCAL flags, not by tab characters in the input) is delayed until SGCAL is splitting up a paragraph into lines. The tab positions are therefore relative to the output lines being generated. For indenting, absolute, centring, and line-ending tabs, SGCAL will start a new output line if necessary.

10.13 Page processing

SGCAL collects together all the lines for a page (more strictly, for a single column on a page) before outputting any of them. This allows it to stretch the page vertically by slightly increasing the line spacing, which improves the appearance of pages that are marginally shorter than the defined depth.

10.14 Galley-style output

SGCAL does not support a true galley mode because it is designed to do formatting on a page-by-page basis, and this does not fit will with the idea of a galley mode. However, an approximation can be achieved by setting the ‘galley’ option.

The output is still produced page by page, but conditional space at the tops of pages is not suppressed, and no formfeeds are generated at the start of plain output pages. Normally, SGCAL fills up plain output pages by generating appropriate amounts of white space. This is suppressed when the ‘galley’ option is set. In addition, footnotes are stored up and printed at the end of the final page.

The ‘galley’ option is controlled by the **enable** and **disable** directives (see sections 21.15 and 21.13). The ‘online’ standard style makes use of it.

10.15 Footnote processing

By default, an output line and all the footnotes associated with it are always printed on the same page. When there are many footnotes, or long footnotes, this can lead to unacceptable amounts of white space at the bottoms of pages. There is an option for requesting *split footnotes* which removes the constraint that a footnote must appear on the same page as the line which which it is associated. This option also permits individual footnotes of more than four lines to be split over more than one page.

The standard macro **splitfootnotes** (see chapter 16) is the normal way of controlling this option, though the underlying control is via the **enable** and **disable** directives (see sections 21.13 and 21.15).

The footnote splitting facilities are somewhat experimental. When a paragraph and its footnotes do not entirely fit on the current page, there are often many different ways of dividing up the text and the footnotes. The existing rules are not very sophisticated and may be altered in the light of experience. (But nothing has changed in the last ten years!)

11. Types of output and dimensions

SGCAL can produce two kinds of output: *plain* output is a straightforward text file which can be read using a text editor; GCODE output is a device-independent encoded form of output which must be processed by another program (e.g. **sgtops**) in order to view or print it. Certain facilities in SGCAL (e.g. sub/superscripts, multicolumning) are not available for plain output.

SGCAL works internally in millipoints (there are 72000 millipoints to an inch), whichever form of output is being generated. When plain output is being produced, the width of characters is assumed to be 6 points, and the depth of lines 12 points.

Unadorned dimensions specified in directives are always taken as points. Thus, for example, specifying

```
.linedepth 13
```

always sets a line depth of 13 points. Such dimensions are rounded to the resolution of the output device. In the case of plain output, this would be rounded to 12 points.

It is possible to specify units for dimensions in directives. The following are recognized:

`pt`	points
`pica`	picas – one pica is 12 points
`in`	inches
`cm`	centimetres
`em`	ems
`en`	ens
`ld`	linedepths

An em is the width of an ‘exact space’ in the current font, while an en is half an em, except in plain mode, when both are equal to 6 points. Here are some examples:

`.indent 20`	indent 20 points
`.indent 20 em`	indent 20 ems
`.space 5 pt`	space 5 points
`.space 5 ld`	space 5 line depths
`.space 1 in`	space one inch

The recognition of dimensions is done at a very low level in the expression decoder. Therefore a directive such as

```
.set var 5.5 cm
```

is permitted. The value placed in the variable **var** would be 155.905.

Dimensions are also used as arguments for certain flag strings. In these cases, the values given must always be in points, optionally with a fractional part.

12. The SGCAL environment

The set of parameters which control how SGCAL formats pages is known as the *environment*. There are three kinds of environment parameter:

- *Global* environment parameters are values which are not expected to change very often during processing, and on the whole they relate to the overall layout of pages. Examples of these are the page depth, the head and foot depths, and the page offsets.
- *Local* environment parameters are values which are changed from time to time as the document is processed. Examples of these are the current font, the current indent, and whether or not output lines are to be right-justified.
- *Temporary* environment parameters are those that cause a temporary change to the environent. Currently the only ones available are those that affect the indent and the line length.

Frequently it is necessary to change a value in the local environment and later to restore the previous value. SGCAL provides a stacking mechanism for this. There are both directives and flags to ‘push’ and ‘pop’ the contents of the local environment, and these are used heavily in the standard styles.

The components of the local environment are a number of switches and a number of values. Many (but not all) of the switches are controlled by the **enable** and **disable** directives. The remainder are controlled by individual directives or flags. The switches control the following:

forcing capital letters
emphasizing each output line
filling output lines by joining and splitting input lines
filling output pages by stretching the space between lines
the ‘galley’ option
interpreting flags in text lines
automatic hyphenation
forcing automatic hyphenation of all words
kerning of letter pairs
checking letter pairs for ligatures
underlining
joining of the next text line without a break (**nosep**)
splitting of footnotes between pages
using formfeed characters in plain output

The values in the local environment are

the number of the current font
the number of the current font group
the justification option (left, right, both or centre)
the current set of tab stops
the current indent
the current line length
the current line depth
the current paragraph looseness
the minimum number of paragraph lines at the top and bottom of a page
the paragraph indent
the paragraph space
the greyness or colour of text, rules, and filled shapes
the width of rules

These switches and values are all preserved over a **push/pop** operation. Other environmental parameters are not preserved.

Note that the temporary indent and line length (and associated counts) are *not* in the local environment.

13. Variables

SGCAL supports two kinds of variable: user variables, and system variables. User variables have names beginning with a letter and containing letters and digits. The standard styles make use of a number of user variables (see chapter 15). System variables have names beginning with `sys.`, for example, `sys.pagenumber`.

All variables contain characters strings which can be inserted into both text and directive lines by means of the insert flag (default string ~~). For example,

`~~myvar`	insert contents of `myvar`
`~~sys.time`	insert the current time

System variables are maintained by SGCAL and cannot be directly changed by the user. User variables can be set by the **set** directive. The value for a variable may be specified as a string, or it may be specified as an expression which is evaluated and then converted into a string representation. For example,

```
.set abc "02 + 2"
```

sets the value of the variable `abc` to ‘`02 + 2`’, but

```
.set xyz 02 + 2
```

sets the value of the variable `xyz` to ‘`4`’, because it evaluates its argument as an arithmetic expression.

User variables can have a format associated with them. The default format is simply to insert the variable’s character string as it is. The alternative formats are for Roman numerals and ‘letter’ numerals. If one of these formats is specified and the variable’s character string consists entirely of digits and is in the appropriate range, it is converted to roman numerals or a letter as appropriate, before each insertion. More details are given in section 21.25.

It is always possible to force the insertion mechanism to insert the basic character string for a user variable, even if its format is not the default. This is done by preceding its name with ‘`raw.`’ For example,

```
.set romannumber ~~raw.romannumber + 1
```

would be the way to increment a variable which is normally inserted in roman numerals.

14. Expressions

Expressions can occur as arguments to a number of directives (see chapter 21). Expressions are *not* recognized in any context in text lines. It is, however, always possible to achieve the effect of an expression in a place where one is not permitted by assigning the value of the expression to a variable (using the **set** directive), and then inserting that variable where the value of the expression is required.

The constituents of an expression are values and operators. Round brackets can be used for grouping in the normal way. The recognized types of value are:

strings	enclosed in double quotes
numbers	written in conventional notation
truth values	written as `true` or `false`

An explicit length may be given for a string by following it with a number in round brackets. The string is extended with space characters if necessary. A second number, separated by a comma, may also appear within the same brackets. This specifies a starting offset within the given string, counting from one. In the following example, all the strings are equivalent:

```
"abcd"(2,1)  "abcd"(2)  "cdab"(2,3)  "cdab"(,3)  "ab"
```

A string may be forced to upper case by preceding it by a circumflex character. Within a string, a double quote character is represented by doubling. Whenever a variable containing a double quote is inserted into a directive line between double quotes, its double quote is inserted twice.

For the purposes of logical operations, `false` is taken as a value of zero, and `true` as a value of one. However, tests assume that any non-zero value corresponds to `true`.

Numbers may be written with a decimal point and a fractional part. SGCAL works with fixed point numbers, to three decimal places, to make it straightforward to handle points and millipoints. However, those operators which do bit manipulation operate only on the integer part of their arguments, clearing any fractional part to zero.

The following unary operators are provided:

`set` <*name*>	test whether variable <*name*> is defined
`odd` <*number*>	test whether number is odd
`even` <*number*>	test whether number is even
`length` <*string*>	compute length of string in characters
`width` <*string*>	compute width of printed string
`+` <*number*>	unary plus operator
`-` <*number*>	unary minus operator
`~` <*number*>	unary bit negation operator
`!` <*number*>	unary logical negation

The **length** operator returns the number of characters in its argument, while the **width** operator returns a dimension (in points) which is the width that its argument would occupy if printed in the current font. In both cases, the string is taken literally – because expressions occur in directive lines, no flag processing (other than inserts) takes place in these strings. Because of this, these operators should be used with care.

The following binary operators, shown with their binding priorities, are provided:

`0`	`||`	logical ‘or’
`0`	`|`	logical ‘or’ (synonym)
`1`	`^`	logical exclusive ‘or’
`2`	`&&`	logical ‘and’
`2`	`&`	logical ‘and’ (synonym)
`3`	`!=`	not equal
`3`	`<>`	not equal (synonym)
`3`	`~=`	not equal (synonym)

3	==	equal
3	=	equal (synonym)
3	>=	greater than or equal
3	<=	less than or equal
3	>	greater than
3	<	less than
4	round	rounding operator
5	-	subtraction
5	+	addition
6	/	division
6	*	multiplication
6	%	remainder (modulo)

The rounding operator is used in the standard styles for rounding dimensions to the resolution of the output device. SGCAL does such rounding internally when, for example, a line depth is set, but it is useful externally when computing the values of variables that hold dimensions.

Operators of equal priority are evaluated from left to right. The comparison operators can be used between two numbers or two strings. The truth values are considered to be numbers for this purpose.

15. Standard styles

SGCAL is designed to be used with a *style definition* which sets up the basic parameters of the output layout and defines character sequences for the flags. A style is selected either by means of the **-style** option on the command line, or by means of the **library** directive. The following standard styles exist:

`a4ps`	for A4 page size on a PostScript output device
`a5ps`	for A5 page size on a PostScript output device
`printer`	'plain' output suitable for a lineprinter
`online`	'plain' output suitable for an online file
`sgpoint`	for full-screen slides for projection

Each of these styles sets up a suitable page size and font flags for the output device, and then defines a standard set of macros and flags. However, the set used for the **sgoint** style is somewhat different to those used for the other styles.

If the **library** directive is used to request a standard style, then (if relevant) certain variables can be set beforehand to alter the default typeface and line spacing. These variables are as follows (default values are shown in square brackets):

`typeface`	main typeface family ['Times']
`sanstypeface`	sanserif typeface family ['Helvetica']
`maintypesize`	size of the main typefaces [11 (A4) or 10 (A5)]
`fntypesize`	size of footnote typefaces [9]
`typespacing`	main line depth [typesize plus one point]
`fntypespacing`	footnote line depth [footnote type size]
`fnsuptypesize`	size of footnote superscripts [typesize times (7/11)]

For example,

```
.set typeface "Palatino"
.set typespacing 12
.library "a5ps"
```

Basic SGCAL directives can be intermixed with the macros set up for a standard style, but this must be done with care, as the standard styles make certain assumptions about their environment. For example, at the start of each chapter or section, the environment is reset to the 'top' level; a local change to the environment which is expected to persist into the next section may not do so.

Another example of a mixing of basic and macro directives that does not work is the explicit use of the **contiguous** directive to surround large textual items such as sections. The macros for starting sections have checks to ensure that they are *not* called inside contiguous sections, in order to diagnose terminating directives that have been accidentally omitted.

16. Standard macros

The following macro directives are available in all the standard styles except the **sgpoint** style, which has its own special set of macros.

16.1 Aspic and endspic

```
.aspic
<Aspic drawing instructions>
.endspic
```

If SGCAL is generating fancy output, the drawing instructions are passed to the Aspic program, which analyses the drawing and returns lines of input for SGCAL to process. If SGCAL is generating plain output, the text '<<picture omitted>>' is substituted.

These macros automatically include the drawing inside a display (see section 16.8 below) and ensure that any indentation is set to zero. Text in the drawing is set in roman type by default, but the usual flags can be used to change this.

16.2 At

```
.at <numeric-expression>
```

This macro generates a **space** directive with a suitable positive or negative argument so that the next line to be printed appears at the absolute depth on the page given by the argument to **at**. It is useful for laying out pages to a fixed specification.

16.3 Blank

```
.blank [<numeric-expression> [line[s]]]
```

This macro inserts conditional vertical blank space. The amount is calculated by multiplying the given numeric expression by half the current line depth, unless the word 'line' or 'lines' is present, in which case the whole line depth is used. The actual quantity of space output is sufficient to make the white space at the current point at least as deep as the calculated amount, except at the top of a page, where nothing at all is output (except in 'galley' mode). The default argument for **blank** is '1'. The <*numeric-expression*> should not contain any spaces; if it does, it should be enclosed in double quote characters, as otherwise the first space enountered terminates the first macro argument.

16.4 Box

```
.box <text>
```

The given text string is output enclosed in a rectangular box, provided the output medium is capable of supporting this (in plain text it is not). Otherwise the text is underlined. The string is never split over more than one line.

16.5 Chapter and chapenv

```
.chapter <title>
```

The **chapter** macro defines the start of a new chapter. The **chapenv** macro defines the environment in which the title is printed. The standard style definitions should be consulted for details.

16.6 Chapternotes

```
.chapternotes
```

This macro requests that all footnotes be saved up and printed at the end of each chapter, instead of at the end of each page.

16.7 Columns

```
.columns <integer-expression>
```

This macro sets the number of columns on a page, and adjusts the footnote linelength and the emphasis point appropriately. For further details, see the **multicolumn** basic directive in section 21.43.

16.8 Display and endd

```
.display [flow] [asis] [rm]
<text lines>
.endd
```

These macros are used to define displays – lines of text that are not filled and which are left justified. They are printed all together on one page, unless the ‘flow’ option is given, in which case a page break is permitted after the first few lines.

The ‘asis’ option disables the recognition of flags in the text lines that make up the display, though not the recognition of the directive flag.

At the start of a display, the font is set to the font number set in the variable `displayfont` (which defaults to the typewriter font) unless the ‘rm’ option is given, in which case it is set to the roman font.

The options are all optional, but if more than one is present, they must be in the order shown above.

At the start of a display, the current local environment (see chapter 12) is pushed onto the stack, and at the end it is restored. Therefore any changes that are made within the display do not propagate beyond it.

White space is automatically inserted at the beginning and end of displays. The amounts are contained in the variables **displaystartspace** and **displayendspace**, which can be altered if desired. The default values for both variables are half the normal line depth.

16.9 Displayenv

```
.displayenv <font number>
```

This macro should not be called directly by the user. It is called by **display** to set up the environment, and its default definition is

```
.macro displayenv ""
.%disable filling
.%justify left
.%indent ~~sys.indent + ~~displayindent
.%font ~~1
.%nosep
.endm
```

It can be deleted and re-defined by the user as necessary.

16.10 Doublespace

```
.doublespace
```

This macro sets the current line depth to 1.5 times the typespacing value, and forces a new line to be started.

16.11 Em and nem

```
.em
.nem
```

These two macros are shorthand for

```
.enable emphasis
.disable emphasis
```

respectively. See the description of **enable** and **disable** in sections 21.13 and 21.15.

16.12 Endnotes

```
.endnotes
```

This macro requests that all footnotes be saved up and printed at the end of the whole document, instead of on each page.

16.13 Figure and endfigure

```
.figure "<title>" [rm]
<text>
.endfigure
```

Figures differ from displays in that they are not constrained to appear inline in the output. If there is not enough room on the current page, a figure will be held over and printed at the top of the next page.

The title is printed below the figure, and it is automatically numbered. The number of the next figure is held in the variable **figurenumber**. If this value is set negative, figure numbering is suppressed. If the variable **figuretitle** is set to the value ‘false’, then figure titles are not printed.

16.14 Footnote and endf

```
.footnote
<text lines>
.endf
```

These macros are used to define footnotes. They automatically number the notes and arrange to output the numbers appropriately. The footnote numbers are reset at the start of each chapter, except in ‘galley’ mode and after **.endnotes** has been obeyed.

Normally, footnotes are output at the foot of the current page, and each footnote is complete on one page. The **splitfootnotes** macro can be used to vary this.

In addition, the **chapternotes** and **endnotes** macros can be used to request that footnotes be printed at the ends of chapters or at the end of the entire document, respectively.

By default, footnotes are numbered sequentially through chapters, or through the entire document if **endnotes** is used. However, for fancy output only, it is possible to specify that the footnotes on each page are separately numbered, starting from 1, by including

```
.set perpagenotenumbers true
```

at the nead of the document. This facility assumes that there are never more than nine footnotes per page, since it allocates space for just one digit for each number.

16.15 Footnoteenv

```
.footnoteenv
```

This macro should not be called directly. It is called by the **footnote** macro at the start of each footnote, to set up the footnote environment, and by default its definition is

```
.macro footnoteenv
.%linedepth ~~fntypesize
.%fontgroup 2
.%font 0
.%linelength ~~fnlinelength
.%indent ~~fnindent
.%justify both
.%enable filling
.endm
```

It can be deleted and re-defined by the user if a different environment is required.

16.16 Nofoot

```
.nofoot
```

This macro cancels any defined foot lines and sets the foot depth to zero.

16.17 Numberpars, nextp and endp

```
.numberpars [<type>]
<text>
.nextp
<text>
.endp
```

This set of macros provides for automatically numbered, indented paragraphs. If the type is unspecified, arabic numbering is used. The alternative types are

`roman`	lower case Roman numerals
`ROMAN`	upper case Roman numerals
`alpha`	lower case letters (a, b, c, etc.)
`ALPHA`	upper case letters

If the type is anything else, it is used as a string to mark the paragraphs. Thus it can be used to supply a ‘bullet’ if required. By default, the paragraph numbers are printed in round brackets. This can be altered by redefining the flag `$npbracket` which should be defined with two strings, one for before each number, and one for after. For example,

```
.cancelflag $npbracket
.flag $npbracket "[" "]"
```

arranges for square brackets to be used instead of round ones, while

```
.cancelflag $npbracket
.flag $npbracket "" "."
```

causes them to be followed by a full stop, with no preceding characters. This flag is used only when paragraphs are actually being numbered, in any printing format. It is not used if an arbitrary string is being used to mark the paragraphs.

White space is automatically inserted before each numbered paragraph, and after the last one. The amount is controlled by the **npspace** variable, whose contents default to half the line depth.

The **numberpars**, **nextp** and **endp** macro directives can be used in a nested fashion; that is, within one set of numered paragraphs, another may be enclosed.

16.18 Pagenumbers

```
.pagenumbers centre
.pagenumbers atedge
```

This macro sets up a foot depth to be twice the current line depth, and defines a single foot line containing the page number, either in the centre of the line, or at the lefthand or righthand edge, depending on whether the number is odd or even. The default is centred page numbers.

16.19 Rule

```
.rule
```

This macro causes a horizontal rule (that is, a straight line) to be drawn from the current indent to the current line length, followed by a call to the **blank** macro. Details of more complicated rules are given in sections 20.20, 20.38, and 20.43. SGCAL is also capable of drawing curved lines – see section 20.3.

16.20 Section and sectenv

```
.section <title>
```

The **section** macro defines the start of a new section. The **sectenv** macro defines the environment in which the title is printed. The standard style definitions should be consulted for details.

16.21 Singlespace

```
.singlespace
```

This macro sets the current line depth to the original typespacing value, and forces a new line to be started.

16.22 Splitfootnotes

```
.splitfootnotes on | off
```

This macro controls whether footnotes must appear *in toto* on the same page as their associated lines, or whether they may be separated from them, or split into more than one piece. ‘On’ implies that splitting is permitted, ‘off’ that it is not. The default is to keep footnotes and their lines together (i.e ‘off’). The macro may be used several times in a single source, to change the state for different parts of the document. The new state applies to the current paragraph and any subsequent paragraphs. (See section 10.15 for further information.)

16.23 Subsection and subsectenv

```
.subsection <title>
```

The **subsection** macro defines the start of a new subsection. The **subsectenv** macro defines the environment in which the title is printed. The standard style definitions should be consulted for details.

16.24 Subsubsection and subsubsectenv

```
.subsubsection <title>
```

The **subsubsection** macro defines the start of a new subsubsection. The **subsubsectenv** macro defines the environment in which the title is printed. The standard style definitions should be consulted for details.

16.25 Table and endtable

```
.table "<title>" [rm]
<text>
.endtable
```

Tables are essentially like figures, except that a separate sequence of numbers is maintained, and their title lines start with the word ‘table’ instead of the word ‘figure’.

The title is printed below the table, and it is automatically numbered. The number of the next title is held in the variable **tablenumber**. If this value is set negative, table numbering is suppressed. If the variable **tabletitle** is set to the value ‘false’, then table titles are not printed.

16.26 Useaccents and usegreek

```
.useaccents
.usegreek
```

These macros request the inclusion of sets of standard flags for accented characters, and for the Greek character set. Accented characters are requested by flags of the form `$aacute`, etc. for lower case, and `$Aacute`, etc. for upper case. Greek letters are requested by flags of the form `$alpha`, etc. for lower case, and `$Alpha`, etc. for upper case.

16.27 Usespecials

```
.usespecials
```

This macro requests the inclusion of flag definitions for all the special characters in the standard encoding for PostScript fonts, using the PostScript character names preceded by a dollar. For example, the upside-down exclamation mark is given the name **$exclamdown**. If this macro is called when generating plain output, all the flags cause a question mark to be printed.

17. PostScript-only macros

The following macros are available only in the PostScript-generating styles:

17.1 Landscape

```
.landscape
```

This macro requests that the current and subsequent pages of output be in landscape orientation. It does not of itself cause any change in the settings of the line length or page depth – it just arranges for the output pages to be 'turned round'. It is usually necessary, therefore, to make adjustments to the line length and page depth after calling **landscape**.

17.2 Picture, endpicture, and psinclude

```
.picture <space> [<x>] [<y>] [<mag>]
<PostScript description of picture>
.endpicture
```

This macro is used for including PostScript generated by other systems into an SGCAL document. The first argument is an expression specifying the total vertical amount of space required for the 'picture'. The second and third arguments are dimensions which specify the offset of the origin from the bottom lefthand corner of this space, and the final argument is the magnification. The last three arguments default to 0 0 1, and must be specified as single numbers; they may not be expressions.

The PostScript itself can be included in one of two ways. The **longcontrol** SGCAL directive (section 21.40) can be used to include lines of PostScript directly. Alternatively, the macro

```
.psinclude <file name>
```

can be used to generate a suitable directive for including the given file of PostScript. Pictures are normally inserted inside displays or figures.

17.3 Portrait

```
.portrait
```

This macro can be used to reset the output orientation to portrait, following the use of **landscape**. It applies to the current and any subsequent pages. It is usually necessary to make adjustments to the line length and page depth after calling **portrait**.

17.4 Transformfont

```
.transformfont <font> <slope> <vstretch>
```

This macro requests the application of a transformation matrix to the given font. It can be used to set up sloped or vertically stretched fonts. The transformation matrix that is applied to the font is, in PostScript notation,

```
[1 0 <slope> <vstretch> 0 0]
```

The arguments must be specified as single numbers; they may not be expressions. For example, to stretch font number 53 vertically by 25% while at the same time sloping it to the right by 15%, the directive

```
.transformfont 53 0.15 1.25
```

could be used. See section 21.3 for details of how to set up additional fonts.

18. Standard flag strings

The following basic flag strings are defined by the standard styles. Each string is followed by the name that is used for defining the flag. For details of what each flag does, see chapter 20 (*Basic flags*).

`$a`	abstab	`~~`	insert
`$B`	back	`+++`	join
`$bc`	bezier	`$L`	level
`$caps`	caps	`$M`	mark
`$c`	centretab	`$N`	nextfnumber
`$C`	centreheretab	`$nh`	nohyphen
`$=`	character	`$>`	nosplitspace
`$rgb`	colour	`$pop`	pop
`$D`	down	`$push`	push
`$nocaps`	endcaps	`@`	quote
`$E`	endheretab	`rl`	right-to-left
`$e`	endtab	`$sf`	shapefill
`$pu`	endunderline	`$s`	space
`~`	dhyphen	`$S`	splitspace
`$f`	font	`$sr`	srule
`$g`	fontgroup	`$su`	startunderline
`$ff`	forcefont	`$<>`	stretchspace
`$fh`	forcehyphen	`$t`	tab
`$F`	forward	`$<`	thinspace
`$hr`	hrule	`$U`	up
`-`	hyphen	`$vr`	vrule
`$i`	indenttab		

The following compound flag strings are defined by the standard styles:

`$$`	is defined as	`""`
`$fh{`	is defined as	`"$push$fh"`
`$nh{`	is defined as	`"$push$nh"`
`$rl{`	is defined as	`"$push$rl"`
`}`	is defined as	`"$pop"`
`#`	is defined as	`"$s"`
`$npbracket`	is defined as	`"("  ")"`
`_`	is defined as	`"$su"  "$pu"`

The closing curly bracket is a general terminator for changes of font and other changes to the local environment. The font-changing (and some other) flags cause the local environment to be pushed onto the stack, and this flag causes the previous values to be restored. The font changing flags are defined below, and details of the local environment can be found in chapter 12.

Note that a number of macros in the standard styles also cause restoration of the previous state of the local environment – for example, the start of a new section, or of a new numbered paragraph. Therefore it is best to keep font changes that are expressed using the standard flags entirely within such larger text items.

The flag consisting of two successive dollars is defined to have no effect. Its use is for terminating other flags that might otherwise be misinterpreted. For example, if an occurrence of the centring tab (`$c`) is immediately followed by the letters ‘aps’, it would be misinterpreted as an occurrence of the caps flag unless followed by two dollar signs.

The flag consisting of a single sharp sign (#) expands into a call to the space flag with no dimension following it. This causes an amount of space equal to the ‘exact space’ of the current font to be inserted into the line.

The flag consisting of an underscore character alternately switches underlining on and off, while the use of the flag called `$npbracket` is described in chapter 16.

The following compound, device-specific flags are defined by the standard styles in order to access frequently used special characters:

`---`	em-dash
`--`	en-dash
`‘‘`	double opening quote
`’’`	double closing quote
`-->`	right arrow
`<--`	left arrow
`<->`	double arrow
`($)`	pound sterling
`($E)`	Euro
`(c)`	copyright sign
`(TM)`	trademark sign
`$’`	minutes sign
`$.`	‘bullet’

The following font-changing flags are defined by the standard styles. The actual definitions of each flag are dependent on the output device; however, each always begins by pushing the current environment onto the stack before changing typeface. The terminating flag is therefore always ‘`}`’.

`$rm{`	roman
`$it{`	italic
`$sl{`	slanted
`$bf{`	bold
`$bi{`	bold italic
`$tt{`	typewriter
`$ss{`	sanserif
`$sc{`	small caps
`$sp{`	special chars font
`$erm{`	enlarged roman
`$crm{`	compressed roman

The small caps flag simply changes font; it does not of itself force subsequent input into capital letters. All except the last two of these flags work within the current fontgroup; that is, the size of the font is taken from the group. See section 21.20 for details of font groups. The last two flags select the fonts by absolute number.

The styles in which headings appear (large type, bold face, etc.) are controlled by flags which are placed before each heading text. They are:

`$chead{`	chapter heading
`$shead{`	section heading
`$sshead{`	subsection heading
`$ssshead{`	sub-subsection heading
`$fkt{`	footnote key in text
`$fkn{`	footnote key for note
`$ftitle{`	figure title
`$ttitle{`	table title

The heading texts are always terminated by a closing curly bracket. Before re-defining one of these flags, it is first necessary to cancel it using the **cancelflag** directive. Thus, for example, to arrange for section headings to be in sanserif type:

```
.cancelflag $shead{
.flag $shead{ "$ss{"
```

The variable **hnspace** contains a string to be inserted between the number of a chapter, section or subsection and its title. By default this string is a single space, but it can be changed as required.

19. Standard variables

The standard styles make use of a number of variables, both as parameters for varying what they do, and also for internal working. Those that control the typeface and type spacing have been described above. Others that are of most interest to the user are:

`chapname`	name of the current chapter
`chapstart`	'true' while processing **.chapter**
`chapter`	number of the current chapter
`contents`	set 'true' to generate contents information
`displayindent`	amount by which to indent displays; default 0
`figurenumber`	number of the next figure
`figuretitle`	if false, no figure titles
`footnote`	number of the previous footnote
`hndot`	dot to print after chapter number
`hnspace`	space after chapter/section titles
`npindent`	amount to indent numbered paragraphs
`perpagenotenumbers`	set 'true' for per-page footnote numbers
`rchapter`	set 'true' to start chapters on right-hand pages
`section`	number of the current section
`sectname`	name of the current section
`sectstart`	'true' while processing **.section**
`ssectname`	name of the current subsection
`ssectstart`	'true' while processing **.subsection**
`sssectname`	name of the current sub-subsection
`sssectstart`	'true' while processing **.subsubsection**
`style`	the name of the current style
`subsection`	number of the current subsection
`subsubsection`	number of the current sub-subsection
`tablenumber`	number of the next table
`tabletitle`	if false, no table titles

The numbering of chapters, sections, subsections, sub-subsections, figures, and tables can be suppressed by setting the variable holding the current number to a negative value. For example,

```
.set chapter -1
```

causes subsequent chapters not to be numbered. The numbering of footnotes normally continues throughout a chapter (or the whole document if **endnotes** is used). However, if

```
.set perpagenotenumbers true
```

is used, the numbers are reset for each page. This facility is available only for fancy output, and it assumes that there are no more than nine footnotes on each page.

Figure and table numbers refer to the next such item, so that it is easy to include references such as

```
in figure ~~figurenumber below
```

immediately before the definition of a figure.

The variables holding the names of chapters, sections, etc. can be used to generate running heads and feet, and the variable **chapstart** can be used to suppress or change a running head at the start of a chapter. Similarly, **sectstart** etc. can be used to do this if a section coincides with the top of a page.

The variable **hndot** is initialized to contain a single full stop. Its contents are printed after the chapter number at the start of a chapter.

The variable **hnspace** is initialized to contain a single space. Its contents are printed between chapter and section numbers and their titles. In the case of chapters, it follows **hndot**.

If the variable **contents** is set to ‘true’ (the default is ‘false’) then information about chapters and sections etc. is automatically output to the index file, for processing into a table of contents. The contents entries can be distinguished from other entries in the index file by the fact that each such entry contains ‘`$e`’ immediately before the page number.

The **style** variable can be used as follows to supply a default style if one is not given on the SGCAL command line:

```
.if !set style
.library "<default style>"
.fi
```

This variable is also used internally to prevent more than one style being set at once.

20. Basic flags

Basic flags are those whose actions are built in to SGCAL. In this chapter, details of the action for each such flag are given. The character sequence for each flag that is used in the standard styles is given in parentheses for each flag, preceded by the name used to define the flag when it is not the same as the descriptive name.

Flags are defined by the **flag** directive (see section 21.18). The standard styles contain definitions of a standard set of flag strings (see chapter 18). These are listed for each flag, and are used in the examples in this chapter.

A number of flags are followed by arguments, which are frequently dimensions (for example, the width of space to insert). SGCAL does not recognize expressions in text lines (which is where these flags are processed), but because it scans lines for variable insertions before it scans for other flags, it is possible to compute values for these arguments. For example, to draw a vertical rule that has a length of 10 times the current line depth, the following could be used:

```
.set rlength 10*~~sys.linedepth
$vr~~rlength
```

This facility can be used for any type of argument. It can even be used to insert the flag strings themselves.

20.1 Absolute tab (abstab, $a)

This flag must be followed by a dimension in points specifying an absolute horizontal position on the output line for the start of subsequent text, for example:

```
$a46this is 46 points from the left
```

If the flag is followed by an asterisk, then the number which follows is interpreted as a number of ems in the current font, for example

```
$a*20this is 20 ems from the left
```

If the requested position is to the left of the current position, a new line is started, unless this flag is encountered at the start of an indented line, in which case a leftwards movement is generated. Space characters in the input immediately before and immediately after this flag and its argument are ignored.

20.2 Back ($B)

This flag is used in conjunction with the *mark* flag; see section 20.26.

20.3 Draw Bezier curve (bezier, $bc)

This flag must be followed by six dimensions, separated by commas. Negative values are permitted. The dimensions are interpreted as points, unless preceded by an asterisk. For horizontal dimensions, an asterisk specifies a dimension in ems; for vertical dimensions an asterisk specifies a dimension in units of the current font size.

The dimensions are interpreted as three pairs of horizontal and vertical coordinates, relative to the current point on the output line. A Bezier curve is drawn from the current point to the position specified by the third pair, using the first and second pairs as the coordinates of the control points. For example, using the standard flag:

```
$bc10,10,40,10,50,0
```

The current position on the line is moved to the end of the curve. The width of line is controlled by the **rulewidth** directive, and the colour by **graphcolour** or **graphgrey** (or their synonyms **rulecolour** and **rulegrey**). This flag is intended mainly for use by programs generating line art as SGCAL input.

20.4 Force capitals (caps, $caps)

The caps flag switches on the environment option to force all subsequent text letters to upper case.

20.5 Do not force capitals (endcaps, $nocaps)

The endcaps flag switches off the environment option to force all subsequent text letters to upper case.

20.6 Centre tab (centretab, $c)

This flag causes text between it and the next tab flag (of any kind) or the end of the input line in which it occurs (whichever comes first) to be centred between the indent and the line length. For example:

```
$c this text is centred
```

If centring on the current line would require the insertion of a negative amount of space, because of previous text on the line, a new output line is automatically started. Space characters in the input immediately before and immediately after this flag are ignored.

20.7 Local centre tab (centreheretab, $C)

This flag causes text between it and the next tab flag (of any kind) or the end of the input line in which it occurs (whichever comes first) to be centred at the current point on the output line. Its effect is the same as inserting the appropriate amount of negative space at the current point, and it does not check for overprinting. It is typically preceded by another sort of tab or a sequence of spaces. Space characters in the input immediately before and immediately after this flag are ignored.

20.8 Character ($=)

The character flag is used to specify a text character by means of its ASCII code in decimal. For example,

```
$=185
```

specifies character number 185 in the current font. A character specified with the character flag is always treated as a text character; in particular

```
$=32
```

causes character 32 in the current font to be printed – it is not treated as a space character.

20.9 Colour ($rgb)

This flag changes the colour of subsequent text. It does not affect the colour of graphics (see **graphcolour**). The flag must be followed by three real numbers in the range 0.0–1.0, separated by commas. They specify the red, green, and blue components of the colour, respectively. For example,

```
$rgb0,0,0.9This text is almost full-strength blue.
```

If a digit follows, the null flag ($$) must be used to terminate the final number. To obtain grey text, use three identical numbers. Three zeroes gives black; three ones gives white. The current colour is kept in the environment, and so can be saved and restored.

20.10 Discretionary hyphen (dhyphen, ~)

This flag marks positions in words where a hyphen may be inserted if necessary. If no hyphen is required, nothing is printed. If a word contains one or more discretionary hyphens it is only ever hyphenated at those places; the automatic hyphenation rules are not used. See also the hyphen flag and chapter 23.

20.11 Down ($D)

The down flag is used to move the current printing point down within a line, typically for subscripts. It can be followed by an absolute number of points, or an asterisk and a factor which is multiplied by the current font size. If the following character is neither an asterisk nor a digit, the movement is one-third of the current font size. Thus,

`$D1.5`	moves down by 1.5 points
`$D*0.6`	moves down by 0.6 times the current font size
`$D`	moves down by one-third of the current font size

The down flag operates only within the current output line. It does not affect the vertical position of characters on subsequent lines.

20.12 End-of-line tab (endtab, $e)

This flag causes the text that follows it, up to the next tab flag (of any kind) or the end of the input line in which it appears (whichever comes first) to be output at the end of an output line. If necessary, because of the length of the text, a new output line is started for this purpose. The end-of-line tab is often used in conjunction with the centring tab. For example:

```
left text $c centre text $e right text
```

Space characters in the input immediately before and immediately after this flag are ignored.

20.13 Local right-aligning tab (endheretab, $E)

This flag causes the text that follows it, up to the next tab flag (of any kind) or the end of the input line in which it appears (whichever comes first) to be output such that it ends at the current point. Its effect is the same as inserting the appropriate amount of negative space at the current point, and it does not check for overprinting. It is typically preceded by another sort of tab or a sequence of spaces. Space characters in the input immediately before and immediately after this flag are ignored.

20.14 End underlining (endunderline, $pu)

This flag turns off the underlining switch in the current environment, thereby causing subsequent text not to be underlined.

20.15 Change font (font, $f)

The font flag must be followed by a font number in the range 0 to 99. It selects a font from the current *font group*. If the current font group is group zero, then the font which is selected is the one with the given absolute font number. Otherwise the definition of the font group is consulted and the given number is used as an index into the list of fonts which make up the group.

20.16 Change font group (fontgroup, $g)

This flag must be followed by the number of a defined font group, and it makes that group the current font group. It does not cause a change of font. See section 21.20 for further details of font groups.

20.17 Force output of font (forcefont, $ff)

When SGCAL is outputting in fancy mode (i.e. outputting GCODE) it normally writes a font change command only when it is about to output characters in the new font, thereby avoiding redundant font changes. This means that font changes almost always follow spacing commands in GCODE. For example, an input line of the form

```
aaa $it{bbb ccc} ddd
```

generates as output

aaa<*space*><*change font*>bbb<*space*>ccc<*space*><*change font*>ddd

although the size of the third space is that of the original font. In most cases the order of spacing and font-changing commands in the GCODE is immaterial, but there are some special applications where it does matter. To cater for these cases, a flag which causes a pending font change to be output is provided. The standard styles define the string `$ff` for this flag; an input line of the form

```
aaa $it{bbb ccc}$ff ddd
```

generates as output

aaa<*space*><*change font*>bbb<*space*>ccc<*change font*><*space*>ddd

If this facility is frequently required, users can define shorter flags of their own, or even re-define the closing curly bracket to include it.

20.18 Force hyphenation (forcehyphen, $fh)

SGCAL does not by default automatically hyphenate the last word of a paragraph, nor any word which contains capital letters. This flag can be used to request it to do so. The standard styles define the string `$fh` to set this option, and also the string `$fh{` to 'push' the environment and then set the option, so that `}` can be used to return to the *status quo*, as in this example:

```
this is the end of a $fh{paragraph}.
```

Details of hyphenation are given in chapter 23.

20.19 Forward ($F)

This flag is used in conjunction with the *mark* flag; see section 20.26.

20.20 Horizontal rule (hrule, $hr)

This flag must be followed by a dimension which specifies the length of horizontal rule to be drawn. The dimension is interpreted as a number of points, unless preceded by an asterisk, in which case it specifies a number of ems in the current font. The rule is drawn at the current base line level, and the current point is moved to the end of the rule. The dimension may be negative, which causes the rule to be drawn to the left. Double negatives are permitted, so a construction such as

```
$hr-~~somevar
```

where the variable **somevar** contains a negative number, work as expected.

For example, to draw a horizontal rule of length 1.5 inches at the current point,

```
$hr108
```

is used. The thickness and colour of the line are specified by the **rulewidth** and **graphcolour** or **graphgrey** directives.

If there is no dimension following the flag, no rule is drawn unless there is a line position mark in effect, in which case the rule is drawn to the current 'high water mark' of the line (i.e. the rightmost point ever reached). See section 20.26 for details of position marking.

20.21 Hyphen (-)

This flag marks positions in words where a hyphen is always inserted, and where the line may be split if necessary. It is recognized only if preceded and followed by a letter. In the standard styles, a single hyphen character is used for this flag.

If a word contains one or more hyphens it is only ever hyphenated at those places; the automatic hyphenation rules are not used. See also the discretionary hyphen flag (section 20.10).

20.22 Indent tab (indenttab, $i)

This flag causes the current point in the output line to be moved to the current indent. It is useful in conjunction with the **tempindent** directive for outputting material in the indent space. For example:

```
.indent 5em
.tempindent 0
XX $i This is indented 5 ems, with XX in the margin.
```

If necessary, that is, if the current point is already past the indent width, a new line is started. Space characters in the input immediately before and immediately after this flag are ignored.

20.23 Variable insertion (insert, ~~)

The insert flag is used for the insertion of variables and macro arguments in both text and directive lines.

20.24 Line joining (join, +++)

The line joining flag is recognized only at the ends of lines which are read from an input file. It causes the subsequent line to be joined on to the one in which it appears, to make a single long input line.

20.25 Level ($L)

The level flag causes the current point to be moved back to the base level of the line. It is typically used after the up, down, or vertical rule flags.

20.26 Position marking (mark, $M)

This flag is used in conjunction with the forward and back flags to achieve overprinting effects in lines. The mark flag causes the current horizontal position to be saved on a stack; the back flag causes an amount of (usually negative) space to be inserted into the line so that the current point returns to the marked horizontal position; the forward flag causes an amount of non-negative space to be inserted into the line to take the current point to the rightmost position reached since the last mark.

For example, using the standard definitions for these flags, the words ‘over’ and ‘print’ can be printed on top of each other by the following input:

```
$Mover$Bprint$F
```

It is important to include the final forward flag, even if it is clear that it will not result in any space being inserted, because between the mark and forward flags, all spaces are marked as non-splitting spaces.

All three flags should always appear in the same logical input line. The back flag may be used any number of times between a mark and its corresponding forward flag. Also, uses of these three flags may be nested for more complicated effects. The horizontal rule flag (section 20.20) behaves differently if it appears without a dimension between a mark and a forward flag.

20.27 Per-page footnote numbers (nextfnumber, $N)

For fancy output, SGCAL is capable of automatically generating footnote numbers that reset for each page, provided there are no more than nine footnotes on a page. At the point where you want to reference the next number in the text, and also at the point where the number appears in the footnote, this flag is used. Normally this is handled automatically by a macro for footnotes.

20.28 Disabling hyphenation (nohyphen, $nh)

This flag unsets the switch in the local environment that allows SGCAL to attempt to hyphenate words automatically. There is no flag for resetting the switch; this is normally done by restoring the previous environment or by using the **enable** directive (section 21.15). The standard styles define

`$nh` as the basic flag string, and `$nh{` as a flag which 'pushes' the local environment and then unsets the switch. The previous state can then be restored by means of the `}` flag, as in this example:

```
don't hyphenate $nh{hyphenation}
```

The word 'hyphenation' in this example will not be hyphenated. Details of hyphenation are given in chapter 23.

20.29 Non-splitting space (nosplitspace, $>)

This flag inserts a space into the current line which has the width of a normal space, and is stretchable, but which will not be recognized as a place at which the line may be split.

20.30 Environment restore (pop, $pop)

The pop flag causes the local environment to be restored from the top entry on the environment stack, provided that it is an 'anonymous' entry. For further details of the environment stack, see the description of the **pop** directive in section 21.56.

20.31 Save environment (push, $push)

The push flag causes the local environment to be saved on the environment stack. The entry is marked 'anonymous'. For further details of the environment stack, see the description of the **push** directive in section 21.56.

20.32 Output right-to-left (righttoleft, $rl)

This flag sets a switch in the environment that causes subsequent input to be reversed, and output in right-to-left order. The standard styles define `$rl` as the basic flag string, and `$rl{` as a flag that 'pushes' the local environment and then sets the switch. The previous state can be restored by means of the `}` flag, as in this example:

```
The next word is $rl{backwards}.
```

The output is 'The next word is sdrawkcab.'

There is a system variable called **sys.righttoleft**, which is 'true' if the right-to-left state is currently set in the environment.

The right-to-left state has an effect only when a line is about to be output. All processing prior to that is unaffected. The text between the setting of right-to-left and its unsetting, or the end of the line, or any tab, is output by reversing the order of the words, and within each word, reversing the order of the letters. Thus,

```
The quick brown fox jumps $rl{backwards over} the lazy dog.
```

comes out as

The quick brown fox jumps revo sdrawkcab the lazy dog.

Any kerning or ligatures that are set for the font take effect on the reversed text. For example, the output from `$rl{if}` in a roman font is the 'fi' ligature.

You need to take care when using right-to-left output. In particular, the following should be noted:

- Take care with underlining. It does work if it is turned on and off wholly within a backwards section, or if a backwards section is contained within an underlined section, but if it is not nested like this, there may be problems.
- Each tabbed field is independently reversed, and tabs still work from left to right.
- The justification setting is not changed automatically.
- Automatic hyphenation does not work. It is probably best to turn off automatic hyphenation.

20.33 Character quoting (quote, @)

The quote flag is a means of entering characters as text that would otherwise be interpreted as flags. The standard styles define the `@` character as the quote flag. For example, if a closing curly bracket is wanted in the text, it must be entered as `@}`. If a space character is preceded by the quote flag, it is treated as character 32 in the current font, and is not treated as a word separator.

To enter an `@` character itself as text, `@@` must be typed. When using a standard style, it is good practice always to use `@` in front of any occurrences of the characters #, $, and _ in the text, even though not all occurrences of `$` will be recognized as the start of a flag.

20.34 Space insertion (space, $s, see also #)

The space flag is used to insert given amounts of space into a text line. Such space is neither stretchable nor recognized as a point at which the line can be split, and is sometimes called ‘hard space’. If the flag is followed by a digit or a minus sign, then what follows must be a dimension specifying a positive or negative amount of space, in points. The dimension may include a decimal point and a fractional part. A negative amount of space moves the current point to the left. It is possible to move it beyond the left-hand margin by this means.

The space flag may alternatively be followed by an equals sign and another character, in which case the amount of space inserted is equal to the width of the given character in the current font. To insert a negative amount of space equal to the width of a given character, the space flag is followed by a minus sign, an equals sign, and then the character.

If the space flag is followed by an asterisk, this must be followed by a number, optionally preceded by a minus sign and containing a fractional part, and it signifies a multiple of the exact space width of the current font.

If the flag is followed by none of these alternatives, then the width used is precisely the exact space width for the font.

The standard styles define the string ‘`$s`’ as the space flag. Using this definition, the following examples show each of the possible kinds of space that may be inserted:

`$s`	the exact space width
`$s=x`	the width of the character ‘x’
`$s-=x`	minus the width of the character ‘x’
`$s1.2`	1.2 points
`$s-3.6`	-3.6 points
`$s*0.6`	0.6 times the exact space width
`$s*-1.3`	-1.3 times the exact space width

The standard styles also define the flag ‘#’ to expand to ‘`$s`’, which always gives an exact space even if it is followed by a digit, since searches for flag arguments do not go beyond the end of previous flag expansion strings.

20.35 Splittable non-stretchable space (splitspace, $S)

This flag operates exactly as the space flag, except that a line may be split at the point where it appears.

20.36 Extra-stretchy space (stretchspace, $<>)

This flag inserts a space into the current line which has the width of a normal space, and is stretchable, but which will not be recognized as a place at which the line may be split. It differs from the non-splitting space in the following way: if a line which is being stretched for justification contains any extra-stretchy spaces, then *all* the additional space is distributed between these spaces, the other stretchable spaces being left at their initial widths. In addition, if the final line of a paragraph contains at least one extra-stretchy space it is always fully justified, provided the appropriate justification option is set.

20.37 Filled shapes (shapefill, $sf)

A sequence of drawing instructions that is enclosed between two instances of this flag causes a filled shape to be drawn. No outlines are drawn; if an outlined shape is required, the outline must be separately specified. The colour of the shape is the current graphic colour (set by the **graphcolour** directive). It is not expected that this flag be used directly; it is provided as a facility that Aspic can use in the output it returns to SGCAL.

20.38 Sloping rule (srule, $sr)

This flag must be followed by two dimensions, separated by a comma, which specify the horizontal and vertical offsets of the rule to be drawn. Each dimension is interpreted as a number of points, unless preceded by asterisk, which causes the horizontal dimension to be taken as a number of ems, and the vertical one to be taken as multiplying the current font size.

The rule is drawn starting at the current base line level, at the current horizontal position, and the current point is moved to the end of the rule afterwards. The dimensions may be negative, which cause the rule to be drawn to the left (for the horizontal dimension), or downwards (for the vertical dimension). The thickness and colour of the line can be specified by the **rulewidth** and **graphcolour** or **graphgrey** directives.

20.39 Start underlining (startunderline, $su)

This flag causes the underlining switch in the current environment to be set. Subsequent text will be output underlined.

20.40 Tab ($t)

This flag causes the current point in the output line to be moved on to the next defined tab stop. It may also cause centre or end-aligning of the tab field; for details see the **tabset** directive in section 21.70. If there are no further tab stops to the right of the current position in the output line, an error is generated and the flag is ignored. Space characters in the input immediately before and immediately after this flag are ignored.

20.41 Thin space (thinspace, $<)

This flag inserts a small amount of non-splittable, non-stretchable space into the current line. The width is determined by the current font.

20.42 Up ($U)

This flag is the complement of the down flag, and causes the current point to be moved upwards within the current line.

20.43 Vertical rule (vrule, $vr)

This flag causes a vertical rule to be drawn at the current point. It must be followed by a dimension in points, which may be negative. Positive rules are drawn upwards. If the dimension is preceded by an asterisk, it is multiplied by the size of the current font.

The current point is moved vertically to the end of the rule. There is no horizontal motion. The level flag can be used to restore the current point to the base line. The thickness and colour of the line can be specified by the **rulewidth** and **graphcolour** or **graphgrey** directives.

21. Basic directives

This chapter contains descriptions of all the basic directives that are built into the SGCAL program. Most of those whose arguments are numeric allow expressions to be used as well as single numbers; exceptions are the directives dealing with fonts. The phrase ‘dimension expression’ is used to mean an expression which has a numeric value that is interpreted as a dimension, while ‘integer expression’ is used for an expression which evaluates to an integer value.

21.1 Aside

This directive causes all input lines that follow it, up to a line containing the directive **enda**, to be processed for inserts only, and then written to the file defined by the **-aside** keyword on the SGCAL command line. While searching for **enda**, any macros that are encountered are expanded. To avoid this expansion, the directive flag on such lines can be preceded by the quote flag.

21.2 Backspace

This directive specifies how the action of backspacing, for the purpose of overprinting characters, is to be represented in plain output. It must be followed by one of the following words:

- **backspace**: overprinted characters are output as ‘character, backspace’, with underlined characters characters as ‘_, backspace, character’.
- **cr**: a carriage return is used to separate two overprinting lines, with the overprinting characters (including underlines) on the first of them so that if the file is displayed on a screen, the overprints get wiped out.
- **crlf**: overprinted characters are output on a second line underneath the main line. Underlined characters have ‘`-`’ (a hyphen) printed below them.
- **none**: overprinted characters are not output at all.

The default setting in the SGCAL program is

```
.backspace backspace
```

However, the **online** style sets the option to

```
.backspace crlf
```

The **backspace** directive has no effect when SGCAL is generating fancy output.

21.3 Bindfont

This directive is used to specify which fonts are to be used by SGCAL, and its syntax is

```
.bindfont <n1> [<n2>]* "<font name>" <size1> [<size2>]*
```

where <*n1*>, <*n2*>, etc. are font numbers, in the range 0–99, and <*size1*>, <*size2*>, etc. are the corresponding sizes required, in points (with permitted fractional part).

The name of the font is in two parts, separated by a slash. The first names the font family, and the second the actual font itself. For PostScript output, the font family name is ‘atl’ (Adobe Type Library). The same font name may be used in several **bindfont** directives, but it is more efficient to specify all the fonts that require it together, as the font metric file is then read only once. For example,

```
.bindfont 0 5 8 "atl/Times-Roman" 10 14.5 17
```

The appearance of a **bindfont** directive in an SGCAL input file indicates that the output is to be GCODE. If no **bindfont** directives appear, output is as plain text.

The standard styles (apart from ‘printer’) bind appropriate sets of fonts, but the user is free to bind additional ones if required. It is suggested that user font numbers start at 51 and work upwards, to avoid clashes with the numbers used by the standard styles.

All **bindfont** directives must appear at the start of the input, preceding any text lines.

See the descriptions of **font** and **fontgroup** (sections 21.19 and 21.20) for further information on the use of fonts.

21.4 Call

This directive must be followed by the name of a command in the underlying operating system, possibly followed by one or more options settings. No quotes are used. For example:

```
.call aspic -sgcal -nv
```

This directive causes the lines that follow it, up to a line containing the directive **endcall**, to be read, processed for inserts, and written to a temporary file. While searching for **endcall**, any macro directives that are encountered are expanded. To avoid this expansion, the directive flag on such lines can be preceded by the quote flag.

The named command is then called with two additional arguments: the name of the file containing the copied lines, and the name of a second temporary file into which the result of processing the lines is to be written.

When the command returns, the output file is read and processed as SGCAL input. This facility enables parts of SGCAL input files to be processed by specialised pre-processors before being typeset. In particular, this is the mechanism by which the Aspic line-art program is called.

21.5 Cancelflag

This directive must be followed by a defined flag string. It causes it to become undefined. For example:

```
.cancelflag $rm{
```

This is necessary if a flag string is to be re-defined.

21.6 Cancelmacro

This directive must be followed by the name of a defined macro. It causes it to become undefined. If a macro is re-defined without an intervening use of **cancelmacro**, a warning message is output.

21.7 Colseparation

This directive must be followed by a dimension expression. It specifies the amount of horizontal separation between multiple columns. The default value is 12 points.

21.8 Comment

The rest of the input line following the **comment** directive is written to the standard error stream.

21.9 Contiguous

This directive specifies the start of a block of text which must be printed contiguously, that is, all on one page. The end of the block is indicated by the **endc** directive. SGCAL reads and processes all the lines in the block, and computes its depth. It is then associated with the current output line that is being processed.

When the time comes to allocate the current output line to a page, if the contiguous block also fits on the page, it is output following the current line. Otherwise it is held over and printed at the top of the next page. This may result in its appearing ‘out of line’ with the input.

There are three optional arguments that may be used with the **contiguous** directive. It may be followed by a number, to indicate its 'series'. When a contiguous block in a given series is encountered, it is always held over to the next page if there is already a block in that series being held over, even if it would fit on the current page. If no number is given, series 1 is used.

In the standard styles, tables and figures are implemented as contiguous blocks in different series. Thus the tables and figures in a document will always appear in the order in which they occur in the input, but if a table is being held over to the next page, it does not prevent a small figure from being printed on the current page.

Contiguous can also be followed by the word 'inline'. This ensures that the block is printed in sequence with the text in which it is embedded, a new page being started if necessary. This feature is used by the **display** macro in the standard styles.

Finally, **contiguous** can also be followed by the word 'novstretch'. This disables the vertical stretching of lines in the contiguous block, and means they will always be spaced by the current linedepth. Without it, the lines may end up further apart as a result of vertical stretching.

It is normally preferable to use one of the standard macros (**display**, **figure**, or **table**) rather than the **contiguous** built-in directive, since they provide additional features such as saving the environment and adding space at the top and bottom of the contiguous text.

21.10 Control

This directive must be followed by a string in quotes. It is output in the GCODE as 'printer control information'. When the ultimate destination is a PostScript printer, the string is taken as a line of raw PostScript to be included in the output at the relevant point.

When **control** appears at the start of a document, it may be intended as part of the 'setup' informtion for the document, or it may be intended to appear at the head of the first page. In the latter case, it should be preceded by **.endsetup** (see section 21.16).

The directive **longcontrol** (section 21.40) is more convenient than **control** when there are a number of lines of control information to be output.

21.11 Cset

This directive behaves like the **set** directive (section 21.66), except that it takes an additional argument expression, and the variable is set only if that expression evaluates to 'true'. For example,

```
.cset a (~~b > 6)  "string"
```

sets the variable 'a' to the value 'string' only if the contents of the variable 'b' are a string representing a number greater than 6. One special use is in setting a variable if it is not already set:

```
.cset b (!set b) <some expression>
```

The action of **cset** can be entirely duplicated using **if**, but it is more compact.

21.12 Cspace

This directive is like the **space** directive (section 21.68), except that it outputs the space only if the current position is not at the top of a page. (However, if the galley option is set, this top-of-page test is not applied.) In addition, the dimension given specifies a minimum amount of space which is required above the new current point. Thus **cspace** ensures, rather than outputs, a given amount of space.

If the previous item to be added to the page was a negative (upwards) amount of space, then **cspace** inserts no space.

Cspace can be used at the start of a contiguous block, in which case it causes space to be generated only if the block is printed other than at the top of a page (except in galley mode).

21.13 Disable

This directive switches off a number of optional processing features. It must be followed by one of the following words:

`emphasis`	emphasizing output lines
`filling`	filling of output lines by joining input
`flags`	interpretation of non-directive flags
`formfeed`	use of formfeed characters in plain output
`galley`	galley-style output
`hyphenation`	automatic hyphenation
`kerns`	processing of kerns
`ligatures`	processing of ligatures
`paradjust`	retrospective paragraph adjustment
`splitfoottext`	splitting of foot texts
`vstretch`	vertical stretching of pages

With the exception of the line emphasis, galley, and split foot text options, all these options are on by default. The state of the vertical stretching option is kept in the global environment; the rest are in the local environment and can therefore be preserved by **push** and **pop**.

The **flags** option affects only text lines; the insert flag is always interpreted in directive lines.

The **galley** option changes the way SGCAL works in a number of places. Details are given in section 10.14. It is intended mainly for use when preparing online documentation.

The **hyphenation** option can also be disabled by means of the **nohyphen** flag (section 20.28). Details of hyphenation are given in chapter 23.

The **paradjust** option allows SGCAL to re-consider a paragraph after it has split it up into lines according to how much text can fit on each line. If there is a very ‘loose’ line that is preceded by a very ‘tight’ line, and the ‘tight’ line ends with one or more short words, SGCAL tries moving some of the short words onto the next line to see if this gives a more balanced distribution of words and spaces. It is unlikely that you will ever want to turn this feature off; the facility is provided mainly for testing.

The option for splitting foot texts is normally controlled via the **splitfootnotes** macro (see chapter 16).

21.14 Emphasis

This directive must be followed by a list of horizontal positions in a line at which ‘emphasis bars’ are to be printed when emphasizing output lines is switched on. Typically one or more values slightly greater than the line length are given, but negative values are permitted, to generate emphasis bars to the left of the text. The standard styles set up suitable default positions.

21.15 Enable

This directive is the complement of **disable** and takes the same arguments.

21.16 Endsetup

This directive forces an end to the setup section of the SGCAL input, which otherwise terminates only when a non-empty text line is encountered. It is useful for forcing **control** directives to be treated as part of the first page instead of part of the setup.

21.17 Error

The rest of the input line is output to the standard error stream, preceded by the text ‘**Error’, and SGCAL reflects the current input position, as it does for internal errors. The return code is set as for an internal error.

21.18 Flag

This directive is used for defining the strings which are to be recognized as flags. It is followed by the string to be defined (not in quotes) and then either the name of an in-built flag, or by one or two strings in quotes which are to replace the flag whenever it is encountered. For example,

```
.flag ~~    insert
.flag $push push
.flag $it{  "$push$f2"
.flag _     "$su" "$pu"
```

The in-built flag names are given in chapter 20 (*Basic flags*). Flags may be re-defined, but this causes a warning message unless the **cancelflag** directive has been used first. Several different flag strings may have the same interpretation.

When a flag is defined with a replacement string, then that string is itself scanned for flags when it is inserted into a line. However, a flag instance cannot begin in such an inserted string and continue in the main part of the line.

For example, with the definitions above, when a line containing `$it{45}` is scanned, the string `$it{` is replaced by `$push$f2`. This is then re-scanned, and `$push` and `$f` are recognized as flags. The latter must be followed by a font number; in this case the number is 2. The fact that there is another digit (4) following in the input line does not cause the argument to become 24, because scanning of flags and their arguments stops at the end of a replacement string.

When a flag is defined with two replacement strings, they are used alternately. Thus, using the final example above, the first time an underscore is encountered, it is replaced by `$su`, the next time by `$pu`, the third time by `$su`, and so on.

21.19 Font

This directive must be followed by a number. It causes a new font to be selected. If the current font group is zero, the number is interpreted as an absolute font number; otherwise it is interpreted as in index into the list of fonts in the current group.

21.20 Fontgroup

This directive defines a font group, or specifies a change of the current font group. If it is followed by a single number, that must be the number of a defined font group, or zero, and it becomes the current font group.

Otherwise it must be followed by at least two numbers, with an equals sign after the first, which is the number of the font group being defined. The rest are a list of fonts in that group. The order is important, as fonts are selected from groups by indexing into this defined list.

Here is an example of the use of font groups, taken from one of the standard styles:

```
.fontgroup 1 =  0   1   2   3
.fontgroup 2 = 10  11  12  13
.flag $rm{    "$push$f0"
.flag $it{    "$push$f1"
.flag $sl{    "$push$f2"
.flag $bf{    "$push$f3"
```

The flag for italic text, for example, is defined so as to select font number one from the current group. When group 1 is current, it is actually absolute font number 1 that is selected, but when group 2 is current, it is font number 11. The standard PostScript styles make use of three font groups:

1	used for normal text
2	used for footnotes – smaller in size
3	used for chapter headings – larger in size

The nine standard fonts are defined in each of these groups.

Note that font group zero is special and is always available. It does not need to be, and indeed, cannot be, defined. Selecting a font when font group zero is current always selects by absolute font number.

21.21 Foot

This directive is used for defining foot lines. The lines between it and the directive **endfoot** are saved up and obeyed at the foot of each page. If the resulting text is too deep for the foot space, lines at its beginning are removed; if it is not deep enough, blank space is inserted at its beginning.

The processing of foot lines is exactly the same as for normal text lines. They may be filled and justified as required. See also **footenv** below.

21.22 Footdepth

This directive, which must be followed by a dimension expression, specifies the amount of space at the bottom of each page which is set aside for the printing of foot lines.

21.23 Footenv

This directive causes a copy of the current environment to be made. This is reinstated as the current environment whenever foot lines are to be processed. The foot lines may make changes to their environment, but these are abandoned at the end of foot line processing.

21.24 Foottext

This directive is the basic one that is used in the **footnote** macro. It causes the lines between it and the directive **endtext** to be saved up and (normally) printed at the bottom of the page on which the current output line is printed. Users should usually use the **footnote** macro, which handles things like change of type size and automatic numbering, rather than calling **foottext** directly.

The **savetexts** directive can be used to cause footnotes to be saved up for printing later in the document, for example, at the end of the chapter. When the galley option is on, footnotes are automatically saved until the end of the document.

21.25 Format

This directive specifies the format in which a user variable is to be printed. It is followed by the name of an existing variable (i.e. one that has previously been set), and one of the words

`roman`	lower case Roman numerals
`ROMAN`	upper case Roman numerals
`alpha`	lower case letter ‘numerals’
`ALPHA`	upper case letter ‘numerals’
`arabic`	arabic numerals
`indirect`	specifies an indirect variable

The Roman and ‘alphabetic’ numerals apply only if the contents of the variable are a digit string in a suitable format. For Roman numerals the string must represent a whole number in the range 1–9999, while for ‘alphabetic’ numerals the string must represent a whole number in the range 1–26.

If the contents of a variable are not suitable for insertion in the specified numerical format, they are inserted as a text string without modification. For example, if the following directives have been obeyed:

```
.set a 1991
.set b 5
.set c -43
.format a ROMAN
.format b alpha
.format c roman
```

Then the effects of inserting the three variables would be:

```
~~a     yields   MCMXCI
~~b     yields   e
~~c     yields   -43
```

When an variable whose format is ‘indirect’ is inserted into a line, the contents of the variable are taken as the name of a second variable, which is substituted for the original variable. It may have any format, and may itself be an indirect variable. To guard against infinite loops, indirections may be no more than 10 deep. If the variable named in an indirect variable does not exist, an error is generated.

21.26 Graphcolour

This directive defines the colour of graphics items – lines, curves, and filled shapes. It has no effect on text. It is followed by three real numbers that specify the red, green, and blue components of the colour, respectively. Their values must be between 0 and 1. For example,

```
.graphcolour 1 0 0.5
```

sets a lot of red and a little blue. Note that the only way of changing the colour of graphics is by a directive. There is no flag (as there is for text).

21.27 Graphgrey

This is a convenience directive sets a grey colour for graphics items. It takes a single real number argument, with a value between 0 and 1, and is equivalent to **graphcolour** with three arguments of the same value.

21.28 Head

This directive is used for defining head lines. The lines between it and the directive **endhead** are saved up and obeyed at the head of each page. If the resulting text is too deep for the head space, lines at its end are removed; if it is too shallow, blanks space is inserted at its end.

The processing of head lines is exactly the same as for normal text lines. They may be filled and justified as required. See also **headenv** below.

21.29 Headdepth

This directive, which must be followed by a dimension expression, specifies the amount of space at the top of each page which is set aside for the printing of head lines.

21.30 Headenv

This directive causes a copy of the current environment to be made. This is reinstated as the current environment whenever head lines are to be processed. The head lines may make changes to their environment, but these are abandoned at the end of head line processing.

21.31 If

This is SGCAL’s conditional directive, which must be followed by an expression which evaluates to a numerical value. A value of zero is taken as *false*; any other value is *true*.

If the value is *true*, the input lines between **if** and the next **elif**, **else**, or **fi** are obeyed, and thereafter any lines up to **fi** are skipped.

If the value is *false*, lines up to **else**, **elif**, or **fi** are skipped. If the terminator is **else**, then lines following it up to **fi** are obeyed; if it is **elif** then another condition must appear. This is tested as before.

While skipping lines, nested **if** directives are correctly handled, and macro directives are expanded, except when a macro is already active (to allow recursive macros to be written). This allows a

conditional section to begin in one macro and finish in another. However, no flags are processed in skipped lines.

As an example of the use of **if**, here are the first few lines of the source of this document:

```
.if !set style
.set typeface "Palatino"
.set typespacing 11
.library "a5ps"
.fi
.
.if ~~sys.fancy
.flag \ "$bf{" "}"
.else
.flag \ "_" "_"
.fi
```

If the style has not been set on the command line (which would have resulted in a value being placed in the variable **style**) then the ‘a5ps’ style is chosen, with a particular typeface and line spacing. Then the flag string consisting of a backslash character is defined; its definition depends on whether the document is being processed for lineprinter (‘plain’) output, or for some other device (’fancy’ output).

21.32 Include

This directive must be followed by a string in quotes. It is taken as the name of a file which is to be included in the input at the point where **include** appears. See also **library** (section 21.37).

21.33 Indent

This directive must be followed by a positive or negative dimension expression. It alters the indentation of the output. **Indent** does not of itself cause a line break in the output, and it is obeyed synchronously with the input text, taking effect from the next output line.

When an indent is set, it is possible to cause a line to start to the left of the indent by specifying a space with a negative width, or by using a tab whose position is less than the indent. The **tempindent** directive (section 21.71) can also be used to cause a given number of lines to start to the left of the current indent.

When positioned to the left of the current indent (by any of the above means), the *indent tab* flag can be used to move the current point to the current indent position (see section 20.22).

21.34 Index

The remainder of the line is taken as text to be output to the file defined by the **-index** keyword on the SGCAL command line. The text is remembered with the current output line, and when that line is allocated to a page, the relevant page number is added to the index text before it is written. If no index file is defined, a single warning message is output.

If an **index** directive appears before the first line of any text at the start of the file, the output to the index file does not contain a page number. This is useful for generating index entries of the form ‘disc, *see disk*’.

21.35 Inserttexts

When footnotes (see **foottext**) are not being printed at the bottom of each page, but are being saved up (see **savetexts**), this directive causes any that have been saved to be printed at the bottom of the current page.

21.36 Justify

This directive controls the formatting of output lines, and must be followed by one of the words

`left`	left justification (‘ragged right’)
`right`	right justification (‘ragged left’)
`both`	both left and right justification
`bothR`	both left and right justification
`centre`	centre justification – all lines are centred

The default setting is ‘both’. The difference between ‘both’ and ‘bothR’ is that, when the latter is selected, short lines at the ends of paragraphs are right-justified instead of left-justified.

Justification is independent of line filling. If line filling is disabled when justification is set to ‘both’ or ‘bothR’, every input line will be stretched to form an output line.

In modes other than ‘both’ or ‘bothR’, no stretching is done, whether or not line filling is in force (but see **looseness** below).

21.37 Library

This directive is similary to **include**, but it takes its argument (a string in quotes) as the name of a member of the standard library. The translation of the library name to an actual file name depends on the operating system under which SGCAL is running. The main use of **library** is for selecting a standard style. For example,

```
.library "a4ps"
```

should appear at the start of a document that is to be formatted for A4 pages on a PostScript device. (A style can also be selected by a keyword on the command line.)

21.38 Linedepth

This directive sets the current line depth to the value of its argument, which is a dimension expression. It does not cause a line break. If it occurs in the middle of a paragraph, the entire paragraph is formatted at the new depth. The program’s default is 12 points, but some standard styles change this.

21.39 Linelength

This directive sets the current line length to the value of its argument, which is a dimension expression. It does not cause a line break. If it occurs in the middle of a paragraph, the new length is applied to the next complete output line which follows this directive. The program’s default length is 390 points, but the standard styles may change this.

21.40 Longcontrol

This directive provides a shorthand for a succession of **control** directives (see section 21.10). Each line between it and **endlc** is treated as though it were enclosed in quotes and preceded by **control**, unless the line starts with a macro directive, in which case the macro is expanded. To avoid this expansion, the quote flag can be used.

Note that, as would be the case with a sequence of **control** directives, the insert flag is recognized in the data lines, unless switched off by the **disable** directive.

21.41 Looseness

When a paragraph is formatted, the width of each stretchable space is multiplied by a *looseness factor* before the paragraph is broken into lines. The default looseness factor is unity. This directive is provided to alter the current looseness, which is kept in the local environment and so can be preserved by **push** and **pop**. The argument is a numeric expression. Multiplication of space widths by the looseness factor is independent of the justification mode.

Occasionally, a value slightly greater than one can cause a paragraph to take up one more line than it otherwise would, which may help to improve the appearance of a page. Similarly, a value less than one can be used to make a ‘tight’ paragraph. Values outside the range of around 0.8 to 1.5 do not generally result in acceptable paragraphs.

21.42 Macro

This directive is used to define macros. It must be followed by a macro name and a number of prototypical arguments which are strings separated by spaces. If a string includes spaces, it must be enclosed in double quotes.

The lines between the **macro** and **endm** directives are read and saved up. Each time the macro is called these lines are obeyed as SGCAL input lines. Within the macro, the insert flag can be used immediately preceding a sequence of digits to insert a macro argument. Argument numbers start from one. As an example, here is the definition of the **blank** macro from the standard styles:

```
.macro blank 1 ""
.newline
.if "~~2" = "line" | "~~2" = "lines"
.cspace ~~1 ld
.else
.cspace (~~1*~~typespacing/2) round ~~sys.vresolution
.fi
.endm
```

When a macro is called, its arguments are given in the same format as when it is defined, except that the final argument need not be delimited by quotes, even if it does contain spaces. Macros can be called recursively, and macros can be defined within macros. If a macro is re-defined, a warning is issued.

Macros are expanded in input which is being skipped as a result of the **if** directive, and in portions of the input which are being collected for separate processing as a result of the directives **aside**, **call**, or **longcontrol**. However, in such circumstances, macros are never expanded recursively. That is, if a macro is found to be already active, it is only expanded again if it is encountered as a normal input line.

21.43 Multicolumn

When generating GCODE output, SGCAL is capable of producing more than one column on a page. This directive specifies the number of columns required. Its argument is an integer expression. When it is obeyed, it causes the current line length to be reset appropriately.

SGCAL cannot handle a change of number of columns in the middle of a page, except in the special case of going from one column to more than one column. In other cases, a new page is always forced.

When processing multiple columns, each column is treated as a ‘mini-page’ as far as the variables which contain such things as space used and space left are concerned. Contiguous block processing is also done on a per-column basis.

Users should normally use the **columns** macro instead of obeying this directive directly. Note that it is not available for lineprinter output.

21.44 Newcolumn

This directive forces subsequent output to be at the top of a new column. If the current column is the last one on the page, it has the same effect as **newpage**.

21.45 Newline

This directive forces a line break in the output. If it is followed by the word ‘justify’ then any previous part line that it causes to be output is justified both left and right. This can be useful for special effects, since the last line of a paragraph is not normally fully justified (even if ‘justify both’ is set). It is not necessary to use **newline** with the justify option when the previous line contains one or more extra-stretchy spaces, since their presence automatically causes the line to be justified.

21.46 Newpage

This directive forces subsequent output to be at the top of a new page. If called several times it does not generate blank pages. (If this is required, **space** should be used to generate some blank space on each page.)

When vertical stretching of pages is in force, pages are stretched only if they are fairly full. It is possible to force a page to be stretched in all circumstances by following the **newpage** directive with the word ‘vstretch’.

21.47 Newpar

This directive forces a line break in the output and then outputs an amount of vertical white space defined by the **parspace** directive. Note that a blank line in the input is equivalent to **newpar**.

21.48 Nosep

This directive causes the next text line to be joined onto the current paragraph being built without the insertion of any separating spaces. When line filling is enabled, it has the effect of suppressing the insertion of a space between one line and the next. When filling is not enabled, it provides a way of joining two input lines together.

Note that **nosep** affects the next *text* line processed. Intervening directive lines are not affected. It may appear at the end of a macro to cause a text line that follows the macro to be joined onto a text line within the macro. Its behaviour is thus different from that of the join flag.

21.49 Page

This directive sets the current page number, which is held in the system variable **sys.pagenumber** and automatically incremented whenever SGCAL starts a new page. Its argument is an integer expression.

21.50 Pagedepth

This directive sets the total depth of the output pages, including the space for head and foot lines. Its argument is a dimension expression.

21.51 Pagerequest

This directive is similar to the **request** directive (see below), except that the request is repeatedly obeyed at the top of each output page (including the current page).

21.52 Pagexoffset

This directive sets a horizontal offset for the placing of output pages on the paper. Its argument is a dimension expression. It must appear at the head of the input, before any text lines are encountered. By default, output normally starts one inch in from the left hand side of the page, so the horizontal page offset is relative to this. It may be positive or negative. Multiple occurrences of **pagexoffset** are cumulative.

If the **pagexoffset** directive is given with two arguments, they are taken as the horizontal offset dimensions for verso (left-hand) and recto (right-hand) pages, respectively. This feature operates for

fancy output only; for plain output the second argument is ignored and all pages have the same offset.

21.53 Pageyoffset

This directive sets a vertical offset for the placing of output pages on the paper. Its argument is a dimension expression. It must appear at the head of the input, before any text lines are encountered. By default, output normally starts one inch down from the top of the page, so the vertical page offset is relative to this. It may be positive (to move further down) or negative (to move up). Multiple occurrences of **pageyoffset** are cumulative.

21.54 Parindent

This directive sets the standard indentation to be used at the start of paragraphs. Its argument is a dimension expression More specifically, whenever **endpar** is obeyed, this value is set up automatically as a **tempindent** (see section 21.71) for one line. The default paragraph indent is zero.

21.55 Parspace

This directive specifies the amount of vertical space to appear between paragraphs. Its argument is a dimension expression. More specifically, this amount of space is output (conditionally) after an **endpar** directive has resulted in the outputting of a non-null paragraph. The program's default value is 12 points, but the standard styles change this.

21.56 Pop

This directive is used to restore a previous set of local environment values from the environment stack. There are three forms:

```
.pop
.pop <letter>
.pop *
```

The first reverts to the most recent set of values on the stack, provided that this is an anonymous set. The second form reverts to the set of values preceding the most recent set that is identified by the given letter. The final form reverts to the set of values at the bottom of the stack – that is, it goes all the way back to 'top level'. In the last two cases, a warning message is output if intermediate frames are discarded.

21.57 Push

This directive pushes the current values of the local environment onto the environment stack. There are two forms:

```
.push
.push <letter>
```

If **push** is followed by a letter, the pushed set of values is identified by that letter; otherwise it is anonymous. An identified set cannot be popped by a call to **pop** without the matching identifier, whereas a call to **pop** with an identifier skips over intervening anonymous sets. This provides a means of recovery from user errors in nesting pushes and pops.

21.58 Request

This directive must be followed by a string in quotes. It generates a request to the program which is to process the GCODE produced by SGCAL. By convention, the string starts with a word terminated by a colon which identifies which processing program is to take note of the request. At present, only 'PostScript:' is relevant; this enables various options to be passed directly to the **sgtops** program. The standard PostScript styles make use of this facility, but it should rarely be needed by ordinary users.

See also the **pagerequest** directive, which specifies requests that happen at the top of each output page.

21.59 Rset

This directive works like **set** in that it sets a value for a variable. However, if **rset** is used, SGCAL assumes that the variable is a reference to some other place in the document, which might be a forward reference. See section 9.2 for a discussion of forward reference handling. At the end of a run, any variables that were set using **rset** are output to the file defined by the **-rsetout** command line option. Once a variable has been set using **rset** it cannot subsequently be reset to a different value.

21.60 Resolution

This directive specifies the resolution of the output device, in points. It is followed by two dimension expressions giving the horizontal and vertical resolution respectively. The program's default is 6 points horizontally and 12 points vertically (suitable for plain output).

21.61 Rulecolour

This directive is a synonym for **graphcolour** and is provided to match the **rulegrey** directive, which has existed for a long time.

21.62 Ruledash

This directive sets parameters for the drawing of dashed lines. There are two forms. To specify that subsequent rules and curves are to be drawn dashed, two non-zero dimension expressions must be specified. The first gives the length of each dash, and the second the length of the gaps between dashes. For example,

```
@.ruledash 4.5 5.1
```

To specify that subsequent rules and curves are to be drawn solid, a single dimension expression whose value is zero must be specified.

21.63 Rulegrey

This directive is a synonym for **graphgrey**, which it predates. Originally it applied only rules, but now it applies to all graphics.

21.64 Rulewidth

This directive specifies the thickness of subsequent rules as a dimension expression. The default is 0.4 points.

21.65 Savetexts

This directive, which has no arguments, specifies that foot texts (footnotes) are to be saved up instead of being automatically printed at the bottom of each page. Instead, they are inserted when the **inserttexts** directive is encountered.

21.66 Set

This directive sets the values of user variables. It must be followed by a variable name and an expression. The expression is evaluated and the result is converted into the form of a string, which is then stored in the variable. For example:

```
.set displayindent 2 em
.set ps "PostScript"
.set oldindent ~~sys.indent
.set reference "~~chapter.~~section"
```

See also the **cset** directive in section 21.11.

21.67 Showhyphens

The remainder of the input line following the **showhyphens** directive is a list of words separated by spaces. Each is processed by SGCAL's automatic hyphenation routine, and the result is output to the verification file. This provides a means of checking whether SGCAL is capable of hyphenating particular words. The **sghytest** command provides another way of doing this.

21.68 Space

This directive outputs vertical white space; the amount is specified by its argument, which may be a positive or negative dimension expression. There is no default, and so a value must be given. If the space causes the current point to go off the end of the page, no further space is output at the start of the next page.

21.69 Stop

This directive causes SGCAL to stop processing without reading any further input.

21.70 Tabset

This directive is used for setting tab stops. It must be followed by a list of dimension expressions. By default, these are relative to one another. For example,

```
.tabset 30 10 10 10
```

sets tab positions 30, 40, 50, and 60 points in from the left hand margin. Repeated identical values can be specified by giving a repeat count and the letter 'x'. The above example could be changed to

```
.tabset 30 3x10
```

If a value is followed by the letter 'a', it is taken as an absolute position instead of relative to the previous stop.

The dimensions can also be followed by one of the letters 'l', 'c', or 'r', indicating a left-justified, centred, or right-justified tab stop respectively. The absence of a letter is equivalent to 'l'. For example:

```
.tabset 10em 15em r 60.5
```

Tab flags in the input are processed when paragraphs are being split up into output lines. If a tab is encountered and there are no further tab stops to the right of the current position in the output line, an error message is generated and the tab flag is ignored.

21.71 Tempindent

This directive sets a temporary indent which lasts for a given number of lines (default 1). For example,

```
.tempindent 24 3
```

sets a temporary indent of 24 points which lasts for the next three lines. The first argument is a dimension expression, and the second an integer expression.

Tempindent does not cause a line break. If encountered in the middle of a paragraph, the new indent applies to the next complete output line. Note that the **endpar** directive sets up new temporary indent parameters, but these can be overridden by **tempindent**. Unlike the permanent indent, the temporary indent value is not part of the local environment.

21.72 Templinelength

This directive sets up a temporary line length in similar way that **tempindent** sets up a temporary indent.

21.73 Textcolour

This directive sets the colour of subsequent text. It must be followed by three real numbers in the range 0 to 1, which specify the red, green, and blue components of the colour, respectively. The numbers must be separated by spaces or commas. The text colour can also be changed by the 'colour' flag.

21.74 Textgrey

This directive is followed by a single argument that sets a greylevel for subsequent text. It is a convenience directive, and is a synonym for **textcolour** with three identical arguments.

21.75 Warning

The rest of the input line is output to the verification output, preceded by '**Warning'. SGCAL then reflects the current input line as for an internal warning, and sets the return code appropriately.

22. System variables

This chapter contains a list of all the available system variables. Those that contain numbers cause strings without decimal points to be inserted when the fractional part is zero. Minus signs are used for negative numbers; nothing precedes a non-negative number.

Certain system variables such as **sys.usedonpage**, **sys.leftonpage**, and **sys.usedcontig**, are updated only when SGCAL is forced to start a new line by a directive such as **newline**, **newpar**, **newpage**, or **space**. Such a directive causes SGCAL to format the text it is holding in its paragraph buffer and allocate it to a page. Any reference to these system variables should always be preceded by a call to the **newline** directive (or any other directive that forces a line break).

`sys.caps`	true if upper case being forced
`sys.colseparation`	the column separation
`sys.columns`	the number of columns on the page
`sys.contigpending`	true if a contiguous block is being held over
`sys.contiguous`	true when reading contiguous block
`sys.date`	today's date
`sys.daynumber`	the day in the month
`sys.emphasize`	true if emphasizing
`sys.fancy`	true if output is in GCODE
`sys.filling`	true if line filling is enabled
`sys.font`	the current absolute font number
`sys.fontgroup`	the current font group
`sys.footdepth`	the current foot depth
`sys.foottext`	true if reading a foot text
`sys.galley`	true if in galley mode
`sys.graphred`	the red component of the graphics colour
`sys.graphgreen`	the green component of the graphics colour
`sys.graphblue`	the blue component of the graphics colour
`sys.headdepth`	the current head depth
`sys.hresolution`	the horizontal resolution
`sys.indent`	the current, non-temporary, indent
`sys.justify`	the current justify mode
`sys.lastcontigdepth`	the depth of the last contiguous block
`sys.leftonpage`	the space left on the current page
`sys.linedepth`	the current line depth
`sys.linelength`	the current, non-temporary, line length
`sys.looseness`	the current looseness
`sys.minparB`	the *minparB* parameter
`sys.minparT`	the *minparT* parameter
`sys.monthname`	the name of the current month
`sys.monthnumber`	the number of the current month
`sys.pagedepth`	the current page depth
`sys.pagenumber`	the current page number
`sys.pagexoffsetR`	the horizontal page offset for right-hand (recto) pages
`sys.pagexoffsetV`	the horizontal page offset for left-hand (verso) pages
`sys.pageyoffset`	the vertical page offset
`sys.parindent`	the paragraph indent
`sys.parspace`	the paragraph space value
`sys.returncode`	the current return code
`sys.righttoleft`	true if outputting right-to-left
`sys.rulewidth`	the width of the next rule
`sys.savetexts`	true if **savetexts** has been obeyed

`sys.tabcount`	the number of tab stops set
`sys.textred`	the red component of the text colour
`sys.textgreen`	the green component of the text colour
`sys.textblue`	the blue component of the text colour
`sys.time`	the current time of day
`sys.underline`	true if underlining
`sys.usedcontig`	the amount of space used in a contiguous section
`sys.usedonpage`	amount of space used on current page
`sys.vresolution`	the vertical resolution
`sys.year`	the current year

The value in **sys.usedcontig** excludes any conditional space that may exist at the start of the contiguous section (which will be omitted if the section ends up at the top of a page). Footnotes are a special kind of contiguous section, and **sys.usedcontig** can be used within them too.

The variable **sys.pagexoffset** also exists for backwards compatibility. It yields the same value as **sys.pagexoffsetV**.

23. Details of hyphenation

SGCAL attempts automatic hyphenation when it is splitting up a paragraph into lines and it comes across a line which is very *loose*, that is, if the amount of unused space left over at the end of the line is large in comparison with the total amount of white space within the line.

Hyphenation is independent of justification, and can occur on left-justified, right-justified, and centre-justified lines as well as on lines which are justified at both ends.

By default, SGCAL never hyphenates the last word of a paragraph, nor any word containing capital letters. A part word at the end of a paragraph does not usually look nice, and nor does a sentence starting with a hyphenated word; other words containing capitals may be acronyms or proper nouns which should not be hyphenated. However, the forcehyphen flag can be used to request hyphenation in these cases if required. For example,

```
The large $fh{ELEPHANT} was made of $fh{aluminium}.
```

The nohyphen flag can be used to suppress automatic hyphenation for particular words:

```
Do not $nh{hyphenate}!
```

The **disable** and **enable** directives can also be used to control this option. Disabling automatic hyphenation does not stop hyphenation at explicit or discretionary hyphens.

To prevent hyphenation at an explicit hyphen, it should be preceded by the quote flag. Note, however, that an explicit hyphen is in any case recognized as such only if it is preceded and followed by a letter.

Before attempting to hyphenate a word (which in this sense is any string of characters delimited by white space), SGCAL ‘cleans’ it by removing all non-letters at the beginning and at the end, and also the sequence ‘apostrophe s’ from the end if it is present.

SGCAL then attempts to ‘de-plural’ the word by means of the following rules:

- If the word is shorter than five characters, or does not end in ‘s’, de-pluralling fails.
- If the word ends in ‘es’, then if the ending is ‘shes’, ‘ches’, ‘sses’, or ‘oes’, remove ‘es’; otherwise, unless the ending is ‘ices’, ‘eses’, or ‘ies’ remove ‘s’.
- If the word does not end in ‘es’ then remove ‘s’ unless the word ends in ‘ss’, or ‘as’, ‘is’, ‘os’, ‘us’ or ‘ys’ not preceded by another vowel.

If de-pluralling succeeds (i.e. if something is removed by the de-pluralling algorithm) then the hyphenation dictionary is searched using the singular form of the word. If an entry is found, it is used; otherwise another search is tried using the original (plural) form of the word. If de-pluralling fails, then the dictionary is searched using only the original form of the word.

Hyphens are generated solely by reference to a hyphenation dictionary. They are not generated by any form of algorithm. It is guaranteed that only those hyphenations that appear in the dictionary can ever be generated. The current dictionary contains nearly 16,500 words.

The hyphenation dictionary is in principle just a file of hyphenated words, one to a line, in alphabetical order (excluding the hyphens from the sorting process). However, so that SGCAL can search it quickly for any given word, it is used with an index of the first four letters of words. This index is stored at the front of the file and copied into main store when SGCAL starts up. There is an auxiliary program called **sgbuildhy** that reads a simple list of hyphenated words and writes a copy of the list with the index on the front. See section 29.1 for details.

24. Miscellaneous

This chapter describes a few miscellaneous features of SGCAL that need mentioning, but are not covered elsewhere.

24.1 Kerning

Kerning refers to the adjustment of the space between individual pairs of letters, to obtain nicer looking output. Compare, for example, ‘AWFUL’ (kerned) and ‘AWFUL’ (unkerned). In the kerned version, the space between ‘A’ and ‘W’ has been narrowed. Kerning information is contained font metric tables, and the user need take no action to obtain its benefits.

24.2 Vertical spreading

SGCAL saves up an entire page in store before outputting any of it. If the page is reasonably full, it stretches it vertically by increasing all the line depths very slightly, so that the bottom line is exactly at the page depth. This gives much nicer looking pages. Vertical spreading can be disabled using the **disable** directive. It can also be independently disabled for individual contiguous sections using the ‘novstretch’ keyword on the **contiguous** directive.

24.3 Flag handling

SGCAL scans directive lines for the insert flag only. Other flags are recognized only in text lines, and what is more, text lines are processed for inserts before they are scanned for other flags. In effect, SGCAL recognizes the insert flag as a means of operating on the input lines, and the other flags as the main markup which controls the format of the output.

24.4 Rules and other lines

SGCAL has built-in support for rules (horizontal, vertical, and sloping lines), making it possible to set up boxed character strings entirely within SGCAL, and also to generate boxes for figures, etc. As the width and colour of the lines are controllable, the rule feature can be used for generating coloured background rectangles.

SGCAL also supports the drawing of curved lines in the form of Bezier curves, on output devices where this is possible (via PostScript). It is possible to write preprocessors for SGCAL that make it possible to include simple line art within SGCAL input files, and indeed one such preprocessor, called Aspic, has been implemented. It is distributed separately, because it can generate Encapsulated PostScript and Scalable Vector Graphics as well as input for SGCAL.

24.5 Widow and orphan lines

A ‘widow’ line is the final line of a paragraph that appears on its own at the top of a page. An ‘orphan’ line is the first line of a paragraph that appears as the last line at the bottom of a page. SGCAL avoids generating orphan lines and widow lines (except for paragraphs that consist of one line only).

24.6 Paragraph ends

SGCAL avoids putting a short word on a line by itself at the end of a paragraph. It also never automatically hyphenates the final word of a paragraph.

25. Format of level 4 GCODE

SGCAL produces level 4 GCODE (previous versions were used by SGCAL's predecessor, GCAL).

25.1 General format

GCODE is a character stream code; line and record boundaries are irrelevant and ignored. It is recommended that there be no more than 72 characters per line, to avoid potential problems when transfering GCODE files between different systems. The space character is not used in GCODE (except possibly in the GCODE heading text), in order to avoid trailing space truncation problems if GCODE files are copied using text-oriented utilities.

Logically, a GCODE stream consists of printing characters and control sequences, which may be of fixed or varying length, as defined in the following sections. However, this logical structure is encoded using printing characters only. This makes it easy to translate from one character code to another, and also avoids potential difficulties when transmitting GCODE files across networks.

One printing character, the backslash ('\') is chosen as an escape character for introducing control sequences. If a backslash character is required as data, it is encoded as the control sequence '\&'. The more obvious encoding of '\\' is avoided, as this gives rise to ambiguities which make it difficult to process the GCODE without parsing it sequentially.

There is a solitary exception to the use of printing characters. The very first character in a GCODE file is a backspace. This makes it possible to distinguish GCODE files automatically under normal circumstances, and it also makes concatenations of GCODE files detectable. (However, the current version of **sgtops** does not support concatenated GCODE files.)

25.2 Coordinate system

The GCODE coordinate system has its origin at a point one inch in from the left, and one inch down from the top of the page, by default. This is the 'current point' at the start of a new page; it is the left-hand end of the baseline for the top line on the page. Almost all movements are relative to the current point. There are control sequences to alter the 'page offset', and these have the effect of moving the origin.

25.3 Control sequences

Two kinds of control sequence are used in GCODE. The fixed-length control sequences consist of a backslash character followed by one other character.

The variable-length control sequences are followed by an argument which is a decimal number, possibly containing a decimal point and fractional part. The argument is terminated by one of a number of special characters which determine the identity of the control sequence.

As record boundaries are not significant, it is possible for a newline to appear between the backslash and the following character, or anywhere in the middle of a control sequence.

25.4 Introductory control sequence

The control sequence which appears at the start of a GCODE file consists of a backspace character followed by a digit identifying the version of GCODE being used. The character for the version described in this document is '4'. This is the only use of backspace (or any other non-printing character) in GCODE.

The GCODE proper starts with the first backslash character following the version number. Any other characters may appear before it, allowing identifying information to be included in the file. SGCAL puts the text

```
GCODE file written by SGCAL <version> (<style>)
   on <date> at <time>
```

between the GCODE version number and the first backslash.

25.5 Control sequences without arguments

The following control sequences are of fixed length and have no arguments:

`\&` include a backslash as a data character
`\f` start/end a filled shape
`\F` begin a new page
`\N` begin a new line
`\n` next footnote number
`\o` reset footnote number

The depth of line must be set before the use of `\N`, and between it and the start of the next line, control sequences such as `\F` (new page) or `\<`*n*`>)` (global move down) may appear. The start of the next line is indicated by the appearance of a printing character or a control sequence pertaining to an individual line (local move left, right, up, or down).

The `\f` sequence should always appears as a pair. Between the two occurences, only sequences that define lines or curves should normally appear. The shape that is defined by the path they define is filled with the current graphics colour. If any other type of item appears between instances of `\f`, the result is undefined.

The `\n` and `\o` sequences are used by SGCAL when it is configured to reset footnote numbers for each page. Instead of outputting an actual number, it outputs `\n`, and at the start of each page and the start of each footnote section, it outputs `\o` reset the footnote number. This approach is used because SGCAL does not know which page a footnote will end up on at the time it generates it. The scheme works only if there are never more than nine footnotes on a page.

25.6 Control sequences with arguments

An escape character (‘\’) followed by a decimal number, possibly including a decimal point, and possibly preceded by a minus sign, is used to introduce a number of different variable-length sequences of the form

`\<`*n*`><`*t*`>`

where <*n*> is the sequence of decimal digits etc. representing an argument value. When this value is interpreted as a dimension, its units are points. The control sequence is terminated by one of a number of characters, <*t*>, which identify the control function required.

Separate control sequences for moving up, down, left, and right are provided, because they are frequently used. The values of their arguments are always positive.

25.7 Control sequences before the first page

Certain control sequences may appear only at the start of a GCODE file, before the first printing page. This restriction makes it possible for a program that processes GCODE to skip pages very rapidly. These control sequences are:

`\<`*n*`>=<`*m*`>"<`*font name*`>"`

This specifies a font binding for font number <*n*> (a value in the range 0–99). The argument <*m*> specifies the size required for the font. It consists of decimal digits only and is in millipoints. By convention, the font name is normally split into two parts, separated by a slash. The first identifies a class of fonts, and the second a particular font within the group, for example

```
\22=15000"atl/Times-Roman"
```

Any given font must not be bound more than once. The bindings may appear in any order.

`\<`*n*`>H`
`\<`*n*`>V`

These sequences specify a horizontal and vertical page offset, respectively, for every page in the document. In effect, they move the default position of the origin of the coordinate system. Their arguments may be positive or negative.

`\<`*n*`>h`

This specifies a horizontal page offset for recto (right-hand) pages only. To specify different offsets for verso and recto pages, `\<`*n*`>H` should first be used to set the same value for both, then the value for recto pages can be changed using `\<`*n*`>h`.

25.8 Control sequences on pages

This section describes those control sequences that may only appear within the data for a given page. A page starts with the sequence `\F`, and this is always followed by two further sequences:

`\<`*n*`>P`

This specifies the *logical* page number for the page, as specified by the **page** command to SGCAL. This makes it easier for programs that process SGCAL to access pages by logical number, as well as by their absolute position in the GCODE file.

`\<`*n*`>C`

This specifies a *column offset* for subsequent output. At the start of a page the argument value is always zero, but if there is more than one column on the page there will be subsequent calls within the page specifying different values. This parameter is in effect a local page offset.

At the start of a page, there is always an explicit selection of a font, and an explicit setting of the vertical spacing increment. Underlining is assumed to be off. This makes it possible for programs that process the GCODE to skip pages simply by searching for instances of `\F` in the file.

The following control sequences are used within pages to control the printing of characters, rules, and Bezier curves:

`\<`*n*`>X`

This specifies a horizontal movement to the absolute x-coordinate given by the argument. The current y-coordinate is unaltered. This is the only non-relative movement specified in GCODE. It is used by SGCAL to move to positions for printing emphasis bars.

`\<`*n*`>>`

This specifies a relative horizontal movement to the right. Its argument is always positive. It is used for tabs and spaces within lines.

`\<`*n*`><`

This specifies a relative horizontal movement to the left. Its argument is always positive.

`\<`*n*`>$`

This specifies a relative vertical movement down the page, and local to the current line. It is used for subscript/superscript handling, and its argument is always positive.

`\<`*n*`>%`

This specifies a relative vertical movement up the page, and local to the current line. It is used for subscript/superscript handling, and its argument is always positive.

`\<`*n*`>)`

This specifies a global relative vertical movement down the page. It is used for page filling and the SGCAL **space** directive, and appears only between lines. Its argument is always positive.

`\<`*n*`>(`

This specifies a global relative vertical movement up the page. Its argument is always positive, and it appears only between lines. It is used for the SGCAL **space** directive when it has a negative argument, and also for re-positioning to the top of a new column in multi-column output.

`\<n>!`

This specifies the vertical spacing increment, that is, the vertical distance between successive lines on the page, which is the amount of downward movement that takes place when the `\N` sequence is obeyed. This sequence is always output before the first occurrence of `\N` on a new page.

`\<n>_`

This specifies whether succeeding characters are to be underlined or not. The argument is either 0 for no underlining, or 1 for underlining. Underlining is always considered to be off at the start of a new page.

`\<n>:`

Selects font number <*n*> for succeeding printing characters.

`\<n>/`

This specifies that character number <*n*> is to be output from the current font. It is used for characters that are not in the normal printing set. SGCAL font encodings are based on the Ascii character set.

`\<n>G`
`\<n>b`
`\<n>g`
`\<n>r`

These settings control the colour of subsequent text or graphics. `\G` sets all three colour components to the same value, that is, it sets black, white, or a shade of grey. The other three change the individual red, green, and blue colour components.

`\<n>T`

This specifies the thickness of subsequent rules and Bezier curves. SGCAL always outputs an explicit thickness for each rule and Bezier curve that it generates.

`\<n>I\<m>I`

This sequence specifies a dash pattern for the next rule or curve *only*. The first number gives the length of the dashes, and the second the length of the spaces.

`\<n>R`
`\<n>U`

This requests that a horizontal or vertical rule, respectively, be drawn. The argument <*n*>, which may be negative, specifies the length of the rule. For vertical rules, positive is upwards. The current point moves to the end of the rule.

`\<n>S\<m>S`

This sequence requests that a sloping rule be drawn. The arguments specify the horizonal and vertical dimensions of the rule, respectively, and may be negative. The current point moves to the end of the rule.

`\<x1>Q\<y1>Q\<x2>Q\<y2>Q\<x3>Q\<y3>Q`

This sequence requests that a Bezier curve be drawn from the current point to a point whose position relative to the current point is (<*x3*>,<*y3*>). The intermediate pairs of numbers give the coordinates of the Bezier control points, again relative to the current point. The thickness and greyness of the curve are controlled in the same way as for rules. The current point moves to the end of the curve.

25.9 Control and request strings

The control sequence

`\D<characters>\D`

represents a device control string, which is intended as an escape mechanism for controlling devices not handled by the existing facilities. It is generated as a result of obeying the SGCAL directive **control**. When **sgtops** is used to process GCODE, control strings are interpreted as inline PostScript.

The control sequence

\A<*characters*>\A

is used as a general mechanism for passing information to programs that interpret GCODE. It is generated as a result of obeying the **request** directive in SGCAL. By convention the text starts with a device name terminated by a colon. For example:

```
\APostScript:landscape\A
```

is an instruction to **sgtops** to output in landscape format. Processors that do not recognize the initial name should ignore the request sequence.

If the backslash character is required as part of a ‘control’ or ‘request’ string, it appears as ‘\&’. In fact, backslash is also used at a higher level as an escape for certain special characters within the string, as follows:

\N	newline
\S	space
\\	backslash

These are therefore encoded in the GCODE as

\&N	newline
\&S	space
\&\&	backslash

Control and request strings may appear both before the first page of text, and within the data for a page.

26. Font metric definitions

SGCAL reads in the widths of the characters in a font whenever the **bindfont** directive is obeyed. The width information is held in human-readable form in a *font library*, which consists of a number of separate font files.

There is a fairly common format for font metric information known as an AFM file; in particular this is used for many PostScript fonts. If an AFM file is available for a particular font, SGCAL can be instructed, by entries in its font file, to read the character width and kerning information from the AFM file. Otherwise this information must be provided in the SGCAL font file.

The name of each font file has to be derived from the font name which is given to the **bindfont** directive. The first part of the name (before the slash) is used as the name of a sub-directory in the SGCAL library in which to look for the font file proper.

In the SGCAL library there is a file called `FontTran` which consists of a number of lines of text, each containing a font name and the equivalent font library file name. This indirection is a result of history. An example of an entry in this file is:

```
Palatino-Roman              Palatin-rm
```

If a font is requested which is not in this configuration file, SGCAL takes the first 14 characters of its name. (See how old this code is!)

If a font file cannot be found for a given font, or if the data in the file does not correspond to the given font name, SGCAL generates an error message and exits with a serious error code.

SGCAL is designed to make use of the existing fonts in a printing device. The data in the font library must therefore correspond to the capabilities of the device if correct formatting is to be achieved.

26.1 Font file format

The format of an individual font file is now described. All dimensions are given in millipoints for a 1-point font. Thus, for example, if a character width is specified as 722, then that character in a 10-point version of the font would be 7.22 points wide. All font files start as follows:

```
FONT "<font name>"
REQUEST "<request information>"
SPACE <n>
THINSPACE <n>
EXACTSPACE <n>
HYPHEN <n>
LIGATURES <count>
<ligature data>
```

If the width and kerning information is to be provided inline, the rest of the font file takes the following form:

```
KERNS <count>
<kerning data>
WIDTHS
<256 widths>
```

Alternatively, if the width and kerning information is to be read from an AFM file, the rest of the font file takes this form:

```
ENCODING "<encoding file name>"
AFMFILE  "<AFM file name>"
```

After the initial identification line, the keywords may appear in any order, except that `ENCODING` must precede `AFMFILE`. The keywords are all optional, except that if `AFMFILE` is present, then `ENCODING` is mandatory, and `WIDTHS` and `KERNS` must not be present.

The purpose of the optional `REQUEST` that which follow the font identification is to pass information to programs that interpret the GCODE generated by SGCAL. At present there are no programs that make use of font request information.

The `SPACE` keyword has the effect of setting a value for all three space parameters; thus it normally comes first if either of the other two are specified. The 'ordinary' space is the width used to separate words in this font. It can, of course, be stretched to effect justification, and it is subject to SGCAL's 'looseness' parameter for an individual paragraph.

The `HYPHEN` keyword defines which character in the font is to be used as a hyphen character. The default is 45, the minus character in the ASCII encoding.

The `LIGATURES` keyword is followed by a number which specifies the number of lines of data which follows it. Each ligature data line consists of a specification of three characters. If the first two are encountered together in a word, they are replaced by the third. A character may be specified either as a number represending an ASCII encoding, or as a character preceded by a single double-quote character. For example,

```
LIGATURES 2
"f "i 174
"f "l 175
```

specifies that 'f' followed by 'i' is to be replaced by character number 174, and 'f' followed by 'l' is to be replaced by character number 175. The ligature definitions can be presented in any order, and any text after the first three items on the line is ignored.

26.2 Inline kerning and width data

Inline kerning data is specified in a similar way to ligatures. Each line consists of the specification of two characters followed by a dimension. A negative dimension indicates that the characters should be moved closer together, while a positive one indicates that they should be moved further apart. Any text following the third item on each line is ignored and may be used for comment. For example,

```
KERNS 3
"A "W -80
"f "’  55
"f 174 -18    f followed by fi
```

The `WIDTHS` keyword appears on a line by itself. It must be followed by 256 widths for the characters in the font. They may occupy as many lines as necessary.

Here is an example of the definition of a PostScript font with inline kerning and width data (shortened to save space).

```
FONT "Times-Roman"
REQUEST "preview: FONT # atl"
SPACE 312
THINSPACE 100
EXACTSPACE 600
HYPHEN 45
LIGATURES 2
"f "i 174
"f "l 175
KERNS 102
"’ "’ -74
"’ "s -55
<100 more lines of kerning data>
WIDTHS
```

```
722 722 722 722 722 722 667 611
<31 more lines of width data>
```

26.3 Kerning and widths from an AFM file

When the kerning and width data are to be read from an AFM file, the font file contains pointers to two other files: the AFM file itself, and a file that defines the encoding scheme. AFM files contain definitions of characters *by name*, and it is the encoding file that associates each character with a code number.

The following example is for a PostScript file for which an AFM file is available:

```
FONT "Times-Roman"
REQUEST "preview: FONT # atl"
SPACE 312
THINSPACE 100
EXACTSPACE 600
HYPHEN 45
LIGATURES 2
"f "i 174
"f "l 175
ENCODING "standard"
AFMFILE  "Times-rm"
```

SGCAL searches in an encoding library for the encoding file, and in an AFM library for the AFM file; the whereabouts of these libraries are built into the SGCAL binary, but can be overridden on the command line. The encoding file contains a list of 256 character names, with full stops for those that are undefined. The SGCAL standard encoding is as follows:

Aacute Acircumflex Adieresis Agrave Aring Atilde Ccedilla Eacute Ecircumflex Edieresis Egrave Iacute Icircumflex Idieresis Igrave Ntilde Oacute Ocircumflex Odieresis Ograve Otilde Scaron Uacute Ucircumflex Udieresis Ugrave Ydieresis Zcaron Yacute Eth Thorn trademark space exclam quotedbl numbersign dollar percent ampersand quoteright parenleft parenright asterisk plus comma hyphen period slash zero one two three four five six seven eight nine colon semicolon less equal greater question at A B C D E F G H I J K L M N O P Q R S T U V W X Y Z bracketleft backslash bracketright asciicircum underscore quoteleft a b c d e f g h i j k l m n o p q r s t u v w x y z braceleft bar braceright asciitilde . aacute acircumflex adieresis agrave aring atilde ccedilla eacute ecircumflex edieresis egrave iacute icircumflex idieresis igrave ntilde oacute ocircumflex odieresis ograve otilde scaron uacute ucircumflex udieresis ugrave ydieresis zcaron yacute eth thorn copyright Euro exclamdown cent sterling fraction yen florin section currency quotesingle quotedblleft guillemotleft guilsinglleft guilsinglright fi fl . endash dagger daggerdbl periodcentered . paragraph bullet quotesinglbase quotedblbase quotedblright guillemotright ellipsis perthousand . questiondown . grave acute circumflex tilde macron breve dotaccent dieresis . ring cedilla . hungarumlaut ogonek caron emdash onequarter onehalf threequarters brokenbar onesuperior twosuperior threesuperior logicalnot plusminus minus divide multiply degree mu registered . AE . ordfeminine . . . . Lslash Oslash OE ordmasculine . . . . . ae . . . dotlessi . . lslash oslash oe germandbls . . . .

This is the standard encoding used by Adobe fonts, with additional assigments for those characters not given an encoding by Adobe, as detailed in section 10.10 above.

Part III

Auxiliary programs

27. The sgtops command

The **sgtops** command is a program for converting GCODE output from SGCAL into PostScript. It is capable of selecting particular pages, making certain size reductions and magnifications, and arranging small pages appropriately on larger sheets.

```
sttops [-from] <file> [-to <file>] [-header <file>]
       [-pages <list>] [-format <name>] [-reverse]
       [-odd] [-even] [-noduplex] [-nocolour]
       [-pamphlet [1|2]] [-copies <n>] [-landscape]
       [-quiet] [-help]
```

If no destination file is specified, the output is written to a file whose name is constructed from the input file by replacing its extension, if any, with `.ps`. If neither input not output files are specified, input is read from the standard input and written to the standard output.

The **-header** option specifies a PostScript header file; you should not normally need to use this option. When it is omitted, the file **PShead** in the SGCAL library is used.

The keywords **-to**, **-pages**, **-format**, and **-pamphlet** may be abbreviated to **-o**, **-p**, **-f**, and **-pa**, respectively.

The list of pages to include consists of comma-separated items, each item consisting of a single number or a pair of numbers separated by a minus sign. The list should be in ascending order. The numbers refer to the count of pages in the GCODE file, that is, they are *physical* page numbers, for example,

```
sgtops myfile.sgout -p 1-4,7,18-22
```

The **-odd** and **-even** options cause only odd-numbered or even-numbered pages, from among those selected, to be processed.

The **-format** item specifies the format of the input file and how is it to be reduced or magnified, if required. This overrides any format specification that may be embedded in the GCODE. Allowed values for the format name are:

`a3`	input is formatted for A3 page size
`a4`	input is formatted for A4 page size
`a5`	input is formatted for A5 page size
`a6`	input is formatted for A6 page size
`a3toa4`	input is formatted for A3 page size; reduce it to A4
`a3toa5`	input is formatted for A3 page size; reduce it to A5
`a3toa6`	input is formatted for A3 page size; reduce it to A6
`a4toa3`	input is formatted for A4 page size; enlarge it to A3
`a4toa5`	input is formatted for A4 page size; reduce it to A5
`a4toa6`	input is formatted for A4 page size; reduce it to A6
`a5toa3`	input is formatted for A5 page size; enlarge it to A3
`a5toa4`	input is formatted for A5 page size; enlarge it to A4
`a5toa6`	input is formatted for A5 page size; reduce it to A6

A5 and A6 pages are printed two-up and four-up, respectively, on an A4 page, and similarly, when A3 paper is being used, multiple A4 and A5 page images are printed on each page.

The **-reverse** option causes the selected pages to be output in reverse order; this is really only of use for pages that are the same size as the paper, as otherwise the ordering of multiple smaller pages on larger paper will be strange.

The default is to generate PostScript that specifies duplex printing if the printer has that capability, in ‘tumble’ mode if **pamphlet** is used. This can be disabled by specifying **-noduplex**.

If **-nocolour** is specified, any colour specifications are converted to grey levels by averaging the sum of the red, green, and blue components. However, as most black-and-white printers can themselves turn colours into greylevels, this option will rarely be needed.

The **-pamphlet** option is useful for A5 and A6 pages printed on A4 paper, or for A4 and A5 pages printed on A3 paper. It causes the pages to be output in the correct order such that the resulting pages can be reproduced double-sided directly and then folded and bound. If **-pamphlet** is followed by the number 1, then only the first of every pair of full-size page images is output; if it is followed by 2, then the second of every pair is output. This makes it easy to print all the first sides, then put the output pages back into a non-duplex printer to print the second sides.

The **-pamphlet** option can be used in conjunction with the **-pages** option. It prints only those pages that are selected, but in the appropriate pamphlet configuration.

The **-copies** option does what its name implies; the keyword must be followed by a number, and it inserts in the PostScript a directive which causes multiple copies of each page to be printed.

The **-landscape** option causes pages to be printed in landscape (long side horizontal) instead of the default portrait (short side horizontal) orientation. This applies to the logical pages, not necessarily to the physical pages.

Finally, the **-quiet** option suppresses the comments that **sgtops** normally outputs.

27.1 Control and request sequences in GCODE

sgtops treats control sequences in the GCODE (generated by the SGCAL **control** directive) as in-line PostScript, and copies them directly to the output.

sgtops also recognizes a number of request sequences (generated by the SGCAL **request** directive). Those that are not recognized are ignored. The following requests are recognized:

```
PostScript: format <format>
```

If the **-format** keyword (see above) is not used on the command line, the value from this request is used.

```
PostScript: font <number>
```

This request forces **sgtops** to output a setting for the given font number in the PostScript. Normally, it outputs a font number only when it is about to output text in that font. However, if some external PostScript is about to be included, having the current font forcibly set first may be necessary.

```
PostScript: get <filename>
```

The named file is included verbatim in the output.

```
PostScript: landscape
PostScript: portrait
```

These requests tell **sgtops** the orientation that is to be used.

```
PostScript: modifyfont <number> <string>
```

This request causes a font to be modified. The <*string*> is PostScript that is applied to the font when it is defined. The most common use of this it for creating slanted fonts. For example, the **a4ps** standard style contains several SGCAL commands of this form:

```
.request "PostScript: modifyfont 2 +++
  font 2 get [1 0 0.25 1 0 0] +++
  makefont font 2 3 -1 roll put "
```

For the inclusion of images, **sgtops** recognizes:

```
jpeg: <num> <denom> <filename>
png: <red> <green> <blue> <filename>
```

which request the inclusion of the named JPEG or PNG file, respectively. For a JPEG file, the first two arguments specify a JPEG decompression ratio. For a PNG file, the first three arguments specify the colour of the background that is to be used if the image contains transparent pixels.

28. The sgpoint program and style

The **sgpoint** command is used to display full-screen images that are suitable for projection. In other words, it is a slide-show bolt-on for SGCAL. The command reads a Gcode file, and displays its contents in a window that is just larger than the physical screen size, so that the window decorations are not visible. If you are running a window manager with a virtual screen and can move to other parts of it using the mouse, you can 'escape' to see other windows while leaving **sgpoint**'s display still available. The **sgpoint** window is moveable if you can get to its moving handles.

sgpoint expects the output to have an appropriate page width and depth. Typically, it will have been formatted using the **sgpoint** style, which is described below. This style operates in three modes: a 'display' mode for showing slides, a 'handout' mode for printing handouts, two-up on A4 paper, and a 'table-of-contents' mode for making a list of the slides.

28.1 Building sgpoint

sgoint uses the GTK+ library, and also the JPEG and/or PNG libraries if you want to display JPEG or PNG images in slides. At present, JPEG and PNG are the only kinds of image that are supported.

To compile the SGCAL suite to include **sgpoint**, add **--enable-sgpoint** to the **./configure** command. To include the JPEG support as well, add **--enable-jpeg**, and similarly, for PNG, add **--enable-png**. The JPEG and PNG support, if so configured, also applies to **sgtops**. You may also need set CFLAGS and LFLAGS for **./configure** if the GTK+, JPEG, or PNG libraries and include files are not in the standard places.

28.2 Running sgpoint

If **sgpoint** is started without any arguments, it prompts in a dialogue box for a file name. Otherwise, the name of the file containing the Gcode can be given on the command line. There are are no other arguments or options. For example:

```
sgpoint myslides.sgout
```

The following keystrokes are recognized:

space Advance to next part of the current slide (if the slide contains **wait**s) or to the next slide if there are no more **wait**s.

G Puts up a dialogue box into which you can type a slide number. Press **Return** to go to that slide.

Q Quits **sgpoint**.

R Reloads the input file. This is useful for testing while creating a slide set.

→ Advance to the next slide, skipping any pending undisplayed parts.

← Go back to the previous slide.

The left mouse button has the same effect as **space** and the right mouse button has the same effect as a left arrow.

No other keys or buttons have any effect.

28.3 Control and request sequences in Gcode

sgpoint ignores control sequences in the Gcode – there should not normally be any.

sgpoint recognizes the following request sequences:

```
sgpoint: wait
```

This marks a ‘wait point’ in the slide. The display is halted until the space bar or the left mouse button is pressed. This feature makes it possible to ‘reveal’ a slide bit by bit, including overlaying parts of it.

```
sgpoint: background <red>,<green>,<blue>
```

This request sequence specifies a colour for the background to the slides. For example, the following SGCAL directive generates a request of this type.

```
.request "sgpoint: background 0.8,0.8,1.0"
```

There is a convenience macro called **background** in the **sgpoint** style (see below). Space is an acceptable alternative to comma as a separator.

```
jpeg: <num> <denom> <filename>
```

This sequence requests the display of the JPEG image in the given file with its top left hand corner at the current point. The image can be scaled by setting <*num*> and <*denom*>, but this scaling is limited by the JPEG library to 1/1, 1/2, 1/4, or 1/8. There is a convenence macro called **jpeg** in the **sgpoint** style. This should normally be used, because it adjusts the PostScript scale for handout output.

```
png: <red> <green> <blue> <filename>
```

This sequence requests the display of the PNG image in the given file, with its top left hand corner at the current point. The colour that is specified is used as the background colour for images that contain transparent pixels. There is a convenience macro called **png** in the **sgpoint** style. This should normally be used, because it adjusts the PostScript scale for handout output.

The numerical arguments for both JPEG and PNG display requests may be separated by commas instead of spaces.

28.4 The sgpoint style

When an input file is processed normally using the **sgoint** style, the output is formatted for screen display. The slide number is shown in the bottom right hand corner. The line length and page depth work nicely on my laptop screen, but may need adjusting for different screen sizes.

If the variable **toc** is defined while processing using the **sgpoint** style, SGCAL generates a table of contents that is written to the **aside** file. The normal output is also generated.

If the variable **handout** is defined while processing using the **sgpoint** style, SGCAL generates output with two slides per page, each enclosed in a box, for an A4 page size. The easiest way to do this is, for example:

```
sgcal myslides.sg -d handout
```

Note that **-d** should be at the end of the command because it can define more than one variable. Output produced by a run of this kind must be processed by **sgtops** in the usual way before printing. Centred page numbers are added as a footing; you can add to the footing by setting the variables **footleft**, **footcentre**, and **footright** if you wish. Alternatively, you can provide your own **foot** setting – copy the one from the style so as to preserve the numbering for slides.

You can set **typeface** and **sanstypeface** before including the style, in the same was as for the normal **a4ps** style. The standard flags are defined, and you can use **useaccents**, **usegreek**, and **usespecials** in the normal way.

You can set the variables **displaycolour** and **seccolour** to set the colour for text in displays and second-level bullet points (the default is black). If you want to do this only for slides, and not for handouts, use something like this:

```
.if !set handout
.set displaycolour "0.2,0.2,1.0"
.set seccolour "0.4,0.5,0"
.fi
```

The following macros are defined in the **sgpoint** style:

.a <*text*>
Outputs a top-level bullet point.

.ab <*text*>
Outputs a top-level bullet point, followed by a ‘blank’ of white space.

.as <*text*>
Outputs a top-level bullet point, followed by one linedepth of white space.

.aspic
The start of an in-line Aspic input section, for a line-art drawing. There is no need to put this inside a display – all that is handled automatically. This macro can be given, as an optional argument, the name of the Aspic command that is to be run (the default is ‘aspic’). This is useful for testing.

.at <*n*>
Moves to an absolute position on the slide, measured from the top.

.b <*text*>
Outputs a second-level bullet point.

.background <*red*>,<*green*>,<*blue*>
Sets a background colour for slides; ignored for handouts. Spaces can be used instead of commas as a separators.

.bb <*text*>
Outputs a second-level bullet point, followed by a ‘blank’ of white space.

.blank <*n*>
Include vertical white space, approximately half a line depth times <*n*>.

.box <*text*>
Displays the text inside a rectangular box.

.bs <*text*>
Outputs a second-level bullet point, followed by one linedepth of white space.

.display <*args*>
Much the same as in other SGCAL styles, except that the only available arguments are `asis` and `rm`.

.endd
Ends a display.

.enddb
Ends a display, followed by a ‘blank’ of white space.

.endds
Ends a display, followed by one linedepth of white space.

.endspic
Ends an Aspic definition.

.jpeg <*file*> <*indent*> <*depth*> <*num*> <*denom*>
Include an image from a JPEG file. The indent defaults to zero, and the depth to `10ld`. The <*num*> and <*denom*> are the JPEG decompression scaling, defaulting to 1/1. The only valid values are 1/1, 1/2, 1/4, and 1/8. The current point is not moved automatically; you have to set the depth explicitly.

.png <*file*> <*indent*> <*depth*> <*background colour*>
Include an image from a PNG file. The indent defaults to zero, and the depth to `10ld`. The background colour is used only if the PNG image contains transparent pixels. It should be specified as three comma-separated numbers. The current point is not moved automatically; you have to set the depth explicitly.

.rule
Draws a horizontal rule.

.s
A shorthand for **.space** with an argument of `1ld`.

.slide <*title*>
Start a new slide; the title may be empty (the default).

.toc <*title*>
If a slide has no title, you can specify a string to be used in the table of contents via this macro.

.wait
Inserts a ‘wait point’ into the slide. Note that wait points can also be specified inside Aspic graphic definitions.

29. The sgbuildhy and sghytest commands

The **sgbuildhy** and **sghytest** commands are auxiliary programs that build and test SGCAL's hyphenation dictionary, respectively.

29.1 The sgbuildhy command

```
sgbuildhy <source file> <destination file> [<max index size>]
```

This command builds an indexed hyphenation dictionary. The source file that is supplied with SGCAL is in the **src** directory, and is called **hyphenlist**. It is an alphabetically ordered list of words, one per line, each containing a hyphen at all its potential hyphenation points. For example, here is a short extract:

```
eye-tooth
eye-wash
eye-wit-ness
fab-ri-cation
fab-ri-cator
fabu-lous
fabu-lously
```

You must ensure that the words are in alphabetical order, and that there are no trailing spaces; otherwise, the resulting dictionary will not work properly.

Note that SGCAL 'de-plurals' words before searching the dictionary. See the description in chapter 23 for the rules that are used. However, for some plural forms you have to include both versions. For example:

```
bat-teries
bat-tery
```

If you include a word only it its plural form, the singluar will not be hyphenated. An example of this is

```
bed-clothes
```

The **sgbuildhy** command creates a destination file that is a copy of the source file, preceded by an index that specifies points in the file where words with certain four leading characters start. Here is part of a typical index:

```
hate47724
hawt47817
hazi47913
heal48154
```

This means that the words starting with 'hate' begin at offset 47724 in the file, and so on. The very first line of the file contains the number of index entries.

By default, the maximum size of the index is 2048 entries, but this can be changed by giving a third argument to the command. In practice, the index is likely to be smaller than the maximum, because duplicates are removed. The currently distributed list contains approximately 16,000 words. Thus there should be an index point roughly every eight words, but there are many sequences of more than eight words with the same initial four letters.

29.2 The sghytest command

```
sghytest <dictionary> [<input file> [<output file>]]
```

The **sghytest** command is used to test new hyphenation dictionaries that have been built by **sgbuildhy**. The first, mandatory, argument is the name of the dictionary to be tested. Input and output files can be specified; if they are not, the standard input and output are used.

The input file is split up into words (there may be any number on a line). The output shows the hyphenation points for the words, one per line.

The best way to test a new hyphenation dictionary is to make a copy of the source file, remove all the hyphens, and run it through **sghytest**. The result should be identical to the original source file. The most common reason why it may not be is that the original is not in alphabetical order.