1. Preface

GCAL is a program which processes textual input to produce output in a layout determined by control information (‘markup’) within the text. It contains facilities for splitting up the text into pages and laying out text on a page, including text justification, page and section numbering, indexing, headlines, footlines, footnotes and so on.

The remainder of this document is a detailed description of the GCAL program which is the heart of the GCAL system. It is *not* a suitable introduction for the novice. In practice, GCAL is rarely used directly at this level. Instead, the basic facilities are packaged into ‘macros’ which combine a number of basic operations into one, higher level facility. For example, it is usual to define a macro to perform the ‘start of new section’ operation; this is a packaged instruction which combines the operations of terminating the previous section, incrementing the section number, checking that there is enough space on the current page to start a new section, outputting the new section heading in the required format with sufficient space before and after it, and creating an index and table of contents entry for the section. A collection of standard macros of this nature is available which provides a simplified means of accessing GCAL so that most of the details of setting up the layout are handled automatically. It is therefore suitable for inexperienced users, and having started in this way, it is not difficult to extend or alter the standard layouts, since many of the parameters can be changed. In the last resort, the files containing the control information that define the macros may be copied by users and then modified to their own requirements.

Users new to GCAL should start by reading the document entitled *An Introduction to GCAL*, which describes the basic facilities available by using standard macro library. There is also a more detailed description of the macros, entitled *GCAL’s Standard Macros*, for those who wish to understand how to modify the standard layouts. This is held in the file `SPEC.GCAL.MACROS`, lineprinter copies of which may be obtained using the `PRINTOUT` command (`LISTOUT` should not be used). Those with no previous experience of text processing are strongly advised to become familiar with the standard system before tackling the rest of this document.

Some major changes were made to GCAL in mid-1985, making it incompatible with previous versions, and there have since been several extensions. This document applies to versions numbered 136 and above. |

2. How to Approach GCAL

The basic facilities of the GCAL program which are described below operate in terms of lines and pages and spacing. They should be viewed as building blocks for constructing higher-level facilities that relate to the logical structure of the text that is being processed, using GCAL macros. The control strings that appear with the input text itself (commonly known as the *markup*) should (with rare exceptions) *never* contain explicit references to actual fonts, dimensions and positioning of characters. Instead they should give an indication of the logical operation that is required at that point in the text. This is known as *generic markup*.

Chapters and sections are obvious structural features, but it is easy to overlook the smaller structures in a text. For example, if different parts of the text are to appear with different indents or different line depths, then macros called, say **smallindent**, **bigindent**, **deeplines** and **shallowlines** should be used instead of including specific indent and line depth values at many points.

The larger the document, the more important it is to distinguish between the logical structures and the way in which they are to be displayed on the page. All layout details should appear at the head of the text, or, better still, in a separate file or macro library. If care is taken to ensure that no device-specific information appears within the input text itself, then not only is it an easy matter to change the layout, but it is also much easier to change from one output device to another. This document, for example, can be formatted for a lineprinter, laser printer, dot matrix or daisy-wheel printer simply by using a different header file. A further benefit of separating the logical structure from the definition of the representation is that it ensures consistency throughout the document.

3. Obsolescence

Many features have been added to GCAL since it was first implemented, and as experience in using it has grown, it has become clear that some of the early ideas have not worked out. There are a number of features that are retained for compatibility, but whose use is not recommended. They are given only brief descriptions in a separate section of the specification.

4. Running GCAL

GCAL is implemented on a number of different computers, and the way of running it varies from system to system. For this reason the instructions for actually running the program are issued separately from this specification. In all implementations, however, the same input and output files are used. The file `SPEC.GCAL.USING` contains details of running both the GCAL and GTYPE programs under Phoenix/MVS.

The the text to be processed is read from a sequential file which is normally associated with the keyword `FROM`. Other files can be accessed by means of the **include** and **library** directives, details of which are given later.

When creating output for any device other than a lineprinter, GCAL needs access to a library of font definitions. This is normally set up automatically by the standard calling mechanism. The format of font definition files is given in appendix B of this specification.

The formatted output from GCAL is written to the file associated with the keyword `TO`, for which there is a default in most systems. The file associated with the keyword `ASIDE` is optional; it is used for a secondary output to which out-of-line captions or references may be written. There is no pagination of this output. Finally, there is a file to which error messages and other comments are sent. This is usually the terminal by default.

A parameter string, usually set up by the keyword `OPT`, is used to supply special options to GCAL, mainly for debugging purposes. It consists of single letters, optionally separated by commas. The following are defined:

`D`	output debugging information after an abnormal end
`P`	output debugging information at the end of the run
`S`	output statistics about the number of lines processed at the end of the run
`T`	output the internal tree structure at the end of the run
`W`	output all warnings (by default, warnings are suppressed after a certain number)

The S and W options may sometimes be of interest to ordinary users.

5. Input Text Format

The input to GCAL consists of intermixed *copy* (the text that is actually to be printed) and *markup* (the instructions for printing it). There are two kinds of markup: *directives*, which occupy a complete input line at each appearance, and *flags*, which are mixed in with the copy. The input must consist entirely of printing characters and spaces. Any control characters encountered are faulted. The maximum length of any input line is 300 characters.

Normally the input starts with a collection of macro and flag definitions and other parameter settings which between them define the required style of layout, and this is followed by the main input text. Such a separation is often carried out by keeping the heading material and main text in separate files, but as far as the GCAL program is concerned, it is dealing with a single, sequential input file. A *directive line* is any input line beginning with one of the two *directive flags*, which are particular characters or sequences of characters defined by the user. There are, however, default settings: a single full stop for the normal directive flag, and the sequence ‘`$.`’ for the basic directive flag, which overrides a macro call. Flags and character handling are described in detail below. The directive flag is recognized only when it occurs at the very beginning of an input line (leading spaces are *not* permitted).

Directives give instructions to GCAL about the layout required for subsequent text. All characters between the directive flag and the end of the line are processed as part of the directive, and do not contribute to the output text. Any input line that does not begin with a directive flag is treated as a text line, and the characters it contains normally appear as part of the output. Various transformations can be applied to text lines as described below.

6. Output Text Format

Output from GCAL may either be a normal sequential file which can be printed on a lineprinter using any normal utility program, or a special format file containing control information for variable spacing devices. Such a file can be used to drive a particular device via an auxiliary program such as GTYPE, whose specification exists as a separate document. An introduction to its use is available in the file `SPEC.GTYPE.INTRO`, while the full specification may be found in the file `SPEC.GTYPE.FULL`. Other auxiliary programs are sometimes used for driving particular output devices.

The two output formats produced by GCAL are called *plain* and *fancy*, and the program runs either in *plain mode* or *fancy mode*. In plain mode (the default), output is in normal sequential format, suitable for sending direct to the lineprinter if required. Certain directives concerned with character and line spacing cause a warning message and are then ignored in plain mode (details appear below).

When the special features of fancy output devices, such as variable spacing and multiple fonts, are required, GCAL must be run in fancy mode, and it requires access to a library of *font definitions* which specify the widths of characters and their spacing. The output from such a run cannot be sent directly to a printing device. It is stored on disc in a coded, device-independent form called GCODE, and a subsidiary program is used to translate this into the particular control sequences required by individual devices. In many cases this is the GTYPE program, which is also capable of converting a GCODE file for proofing on a fixed character-width device such as a lineprinter or terminal, although naturally such features as variable spacing cannot be very accurately represented. Thus the possible data routes are as shown in figure 1 below.

```
     font library     raw input    style library
           |              |              |
           |              V              |
           |         -----------         |
 ------------------->|  GCAL   |<---------
 |                   -----------
 |                        |
 |         -------------------------------
 |         |                             |
 |       fancy                         plain
 |      output                         output
 |         |                             |
 |         V                             |
 |   -------------                       |
 |   |GCODE saved|                       |
 |   |  on disc  |                       |
 |   -------------                       |
 |         |                             |
 |         V                             |
 |     --------------------------        |
 ----->| GTYPE or other program |        |
       --------------------------        |
                  |                      |
   -----------------------------------   |
   |              |        |         |   |
   V              V        V         V   V
fancy printing devices   screen   lineprinter
```

Figure 1: GCAL Data Routes

7. Syntax of Directives

GCAL directives consist of a name followed, optionally, by one or more arguments. The directive name may either be the name of a GCAL built-in directive, or the name of a *macro directive*, whose definition must appear earlier in the input. In practice, a number of *macro libraries* have been created, and in many cases the macro user need not be aware of the details of macro definitions.

Directives are only recognized on lines beginning with a directive flag. There are two such flags: the ordinary directive flag defaults to a full stop, and causes GCAL to search the list of defined macros, followed by the list of built-in directives; the basic directive flag defaults to ‘`$.`’ and causes GCAL to search the built-in directive list only. There may be no spaces between the directive flag and the directive name. Some examples of directive lines are

```
.copy
.indent 6
.myfunction arg1 arg2
$.footnote
```

There is no artificial limit to the length of directive names, which consist of a sequence of letters and digits, starting with a letter. Upper and lower case letters are treated as synonymous. Arguments may be separated by arbitrary numbers of spaces and/or by commas. At least one space is required between a directive name and the first argument in cases where concatenation would otherwise occur, for example, when the name is followed by an unsigned number.

Lines containing only a directive flag or a directive flag followed by a space (i.e. a null directive) are ignored. This allows comments to be included in the source.

8. Expressions

The arguments for directives are GCAL expressions, of which there are four types:

(a) string
(b) numeric
(c) dimension
(d) logical

8.1 String Expressions

A string expression consists of any string of characters enclosed in single or double quotation marks. A quotation mark of the kind being used to delimit the string may be represented in the string by doubling. For example

```
"ABC"   "quick brown"   ’I can’’t’
```

With the advent of printers containing typographic character sets which distinguish between opening and closing quotes, it has become common to use the character sequence consisting of two single quotes to represent a double closing quotation mark. It is therefore inadvisable to use single quotes for representing the null string, and in general, the use of double quotes for GCAL strings is to be preferred.

There is a third way of delimiting strings, which uses the *quote flag*. (See below for a details of GCAL flags.) This flag is by default set to the character string ‘`$""`’, but can be changed by the user. It is intended for use in macros, where argument strings may contain both kinds of normal quote. For example

```
.macro chapter "title"
.set chapname $""~~1$""
...
.endm
```

The quote flag may not itself appear inside a string of any sort.

Strings may be of any length, including zero (the null string). In addition, a string may be followed by an explicit length in round brackets, for example

```
"123"(4)   "abcd"(1)
```

and a starting position may follow the length:

```
"123456"(2,3)
```

is equivalent to

```
"34"
```

Note that the position of the first character in the string is numbered 1. These features are of use when the contents of the string are created by inserting a variable, for example

```
"~~title"(20)
```

If the length is greater than the actual length of the string, spaces are added. The length (but not its comma) may be omitted if just a starting position is required. In this case, the length is taken to be the number of characters between the starting position and the end of the string. For example,

```
"19 September 1984"(,4)
```

is equivalent to

```
"September 1984"
```

The rule that every string must be enclosed in quotes is relaxed in the case of macro calls (*not* macro definitions). If a string argument is expected, and the next character in the input line is not a single or double quote (or the quote flag), then the rest of the input line is taken as the string, and any subsequent arguments take their default values. For example,

```
.subsection Numeric Expressions
```

8.2 Numeric Expressions

A numeric expression consists of a series of decimal numbers connected by the arithmetic operators '`+`' '`-`' '`/`' and '`*`', round brackets being available to force the order of evaluation in the normal way. For example,

```
2+2   (2+2)*3   (4-6)/7
```

All arithmetic is carried out with integers, and remainders from division are lost. Most expressions are actually constructed by the insertion of numeric variables into the directive line. For example,

```
.set A ~~B*2
```

sets the variable A to twice the value in variable B. (The character sequence '`~~`' is a GCAL flag that is used to insert the value of a variable – for details see under 'Inserts' below). The logical operators '`~`' (not), '`&`' (and) and '`|`' (or) may be used to perform bitwise operations on numeric values.

In many directives the full stop character may appear in numeric arguments in place of a number, when its meaning is 'the current value'. Thus, for example,

```
.indent .+2
```

means 'increase the existing indent by two'.

The length of a string may be included in a numeric expression by means of the monadic operator '`length`'. Thus, for example,

```
.set slen length "abcd"
```

sets the variable **slen** to the value 4.

8.3 Dimension Expressions

Dimension expressions are similar to numeric expressions, except that the numbers involved may contain decimal points. The result of the expression is interpreted as a dimension in printer's *points* (see *Fancy Mode Directives* below). For example

```
.linedepth .+3.6
.indent 12.5 + 14.0
.space 4 + 2.0
```

GCAL works to three decimal places when computing dimension expressions. A number without a decimal point in a dimension expression is sometimes multiplied by a factor to convert it to a value in points. This depends on the context in which the expression appears:

(a) If the expression is an argument for a directive that can appear in both plain and fancy modes (such as **space**, **indent**, **footlength**, etc.) then integer values are multiplied by the current exact space width or line depth, depending on whether a horizontal or vertical dimension is required. Thus, for example,

```
.space 4 + 1.0
.indent 3 + 3.0
```

means 'leave vertical space of depth four lines plus one point, and indent by three exact spaces plus 3 points'.

(b) In all other contexts, an integer in a dimension expression is treated as a number of points. Thus, for example,

```
.linedepth 4 + 1.0
```

sets a line depth of 5 points. (The **linedepth** directive is only allowed in fancy mode.) In particular, this rule applies to dimension expressions used to set variables or as arguments to macros.

8.4 Logical Expressions

A diadic logical expression consists of pairs of string or numeric expressions connected by the comparison operators '`<`' (less than), '`<=`' (less than or equals), '`=`' (equals), '`~=`' (not equals), '`>`' (greater than) and '`>=`' (greater than or equals); the pairs themselves are connected by '`&`' (and) or '`|`' (or). The monadic operators '`~`' (not), '`set`', '`macro`', '`odd`', '`even`' and '`fit`' are also available. The first of these negates a logical expression, the second and third test whether a variable or macro has been defined, and the final three test a numeric expression and return 'true' in the following circumstances:

`odd`	if the value is an odd number
`even`	if the value is an even number
`fit`	if the vertical space represented by the argument fits on the current page

The operators '`set`' and '`macro`' can be applied only to user variables and macros, respectively. Each must be followed by a variable or macro name as appropriate (without quotes or the insert flag). Once set, a user variable cannot become unset, but macros can be deleted by means of the directive **cancelmacro**.

The constants '`true`' and '`false`' are also available in logical expressions, which are primarily used as arguments for the **if** and **unless** directives. Some examples of this are:

```
.if odd ~%page
.if fit 6
.if set maindepth
.unless macro subsubsection
.if "~%date"(3,4) = "May"
.if ~%contigpending
.unless ~%usedonpage=0
.if ((~%pagelength=0)&(~%footlength~=0))|~~specialcase
```

8.5 Operator Precedence

In GCAL expressions, the precedence of the operators is as follows:

```
highest:  odd even fit set macro length ~ - (unary)
   |      * /
   |      + - (binary)
   V      <= < = ~= > >=
lowest:   & |
```

When operators of equal precedence occur, evaluation is from left to right.

8.6 Named Operators
All the arithmetic and other operators have alternative representations in the form of alphabetic names. This makes it possible to use the character sequences such as '>=' as flags to represent special printing characters not found on the keyboard. It is strongly recommended that the writers of macro packages use the name forms in preference to the symbols in order to avoid conflicts. The names are

```
plus minus times div
not and or
lt le eq ne gt ge
```

9. Default Operation

The input for a GCAL run may consist entirely of text to be formatted, though this is nowadays extremely rare. Normally there are at least such requirements as headings or displayed lines, which require extra information to be inserted into the input, often by making use of a style definition from a library of such definitions. In the simplest case, however, all the defaults of the system are used.

The default mode is *plain* (suitable for lineprinter or other fixed pitch device). The boundaries between the input text lines are not significant, and all multiple spaces are reduced to a single space. GCAL produces output lines that contain as many words as possible in the default width, which is 65 characters (suitable for printing double columned on a lineprinter). Any word that contains a hyphen (minus) character may be split at that point; otherwise words are kept intact. For example, the input

```
The quick
brown
fox jumps     over the lazy
dog   and  now is the
time  for all
good men to come to the aid of the party.
```

produces the output

The quick brown fox jumps over the lazy dog and now is the time for all good men to come to the aid of the party.

A new paragraph is started whenever one or more blank input lines are encountered, or when the text on an input line is indented. A single blank line is used to separate output paragraphs, however they are indicated in the input, and paragraph beginnings are not indented by default. Note, however, that many of the standard style definitions in the macro library make changes to the default method of indicating a new paragraph. A different amount of blank space may be used, or indenting may be used instead of blank space.

If a new paragraph is required within two lines of the bottom of a page, a skip to the top of the next page occurs. Thus the first two lines of a paragraph are always on the same page. For example, the input

```
This is the   first
paragraph; it is rather short.
  This is the second, longer paragraph, which was
introduced by indenting.

This is the third paragraph, introduced
by two blank lines.
```

produces the output

This is the first paragraph; it is rather short.
  This is the second, longer paragraph, which was introduced by indenting.
  This is the third paragraph, introduced by two blank lines.

The default number of lines on a page is 60, with no headlines or footlines. (However, most style definitions set up a default footline containing the page number.)

10. Fill and Copy Modes

GCAL processes text lines in one of two modes: *fill mode* or *copy mode*. In fill mode the text is assumed to be natural language and output lines are filled with as many words as possible. In copy mode, GCAL copies input lines to the output without any change of format. However, tab expansion and variable insertion are carried out. At the start of a run, GCAL is in fill mode. The directive **copy** switches to copy mode, while **fill** reverts to fill mode. The system variable **incopy** is 'true' in copy mode, and 'false' in fill mode.

11. Flags

In order to provoke more complicated processing of the text than has already been described, the input must contain GCAL control sequences. There are two different kinds of control sequence: *directives* and *flags*. Directives have been introduced above; they always occupy a whole input line. There are around 80 different directives, with many different uses.

Flags differ from directives in that they are recognized *anywhere in an input line*. For example, if all or part of a line is to be underlined, it should be preceded and followed by the *underline flag*, which is a single underline character. Thus the input

```
Sometimes _words are underlined_, sometimes
they are not.
```

produces the output

Sometimes <u>words are underlined</u>, sometimes they are not.

GCAL uses flags in the input text to trigger certain actions, such as the recognition of a directive line or the insertion of the contents of a variable into a line.

A flag consists of any given sequence of printing characters, but not beginning with a letter or digit. Upper and lower case letters in flags are treated as synonymous. If one flag is a substring of another, then in cases of ambiguity GCAL uses the longer string. At the start of a run, the following defaults are set for GCAL system flags (detailed descriptions are given later):

`~~`	insert
`~%`	system insert
`~+`	incrementing insert
`~!`	bare insert
`#`	exact space
`_`	underline
`.`	directive
`$.`	basic directive
`$$`	null
`$+`	superscript
`$-`	subscript
`$>`	non-splittable space
`$#`	non-stretchable space
`$<`	thin space
`$<>`	extra-stretchy space
`$""`	quote
`$~`	discretionary hyphen
`$!`	end of sentence
`$%`	uppercase
`$=`	special character
`$a`	absolute tab
`$b`	bold
`$c`	centering tab

`$d`	down
`$e`	ending tab
`$f`	font
`$fg`	font group
`$g`	graphic rendition reset
`$gf`	group font
`$h`	high water mark
`$i`	indent tab
`$j`	concatenation (join)
`$l`	line level
`$o`	overprint
`$pb`	stop emboldening
`$pe`	stop emphasizing
`$pcl`	stop capital letters
`$pll`	stop little letters
`$pu`	stop underlining
`$sb`	start emboldening
`$se`	start emphasizing
`$scl`	start capital letters
`$sll`	start little letters
`$su`	start underlining
`$t`	tab
`$u`	up
`$w`	width

The standard style definitions define additional flags, all of which normally start with the character ‘`$`’. Special action is required if it is necessary to include any of the following sequences as part of the actual text:

(1) the character ‘.’ at the beginning of an input line;

(2) the characters ‘#’, ‘~’, ‘`$`’ and ‘_’ (underline).

In addition, the hyphen character is special in that it is assumed to be a valid place to split a line. Printers with typographic character sets may have special joined-up forms for certain pairs of characters such as ‘f’ and ‘i’; these are known as *ligatures*, and are specified in the font definitions for the device. Ligaturing can be prevented by placing the null flag (‘`$$`’) between the characters. Apart from the compounds of ‘f’, it is common to specify two minus signs to represent an en-dash, and three to represent an em-dash, if these characters are available in the font. More details about GCAL’s treatment of ligatures is given later.

A number of GCAL flags take optional numerical arguments. To make the input more readable, space characters following any flag sequence in the input which ends in a letter or digit are, by default, *not copied to the output*. They serve to terminate the flag, but are otherwise ignored. For example in

```
The $it italic $rm fox jumps over the $bf bold $rm dog.
```

the spaces following ‘`$it`’, ‘`$rm`’ and ‘`$bf`’ are ignored by GCAL. However, this rule can be disabled by means of the directive **noignoreflagspace**. Once this has been obeyed, all spaces in the input are significant; if an explicit terminator for a flag is required, the null flag can be used. The default state of affairs can be re-instated by means of the **ignoreflagspace** directive. All the examples in this document assume that the default state prevails.

The directive flag is recognized only at the start of a line. GCAL scans most lines for flags when they are input. Certain lines, such as headlines, footlines and macro texts, however, are not scanned until they are used, while directives which define or cancel flag strings are never themselves scanned for flags.

The user may change any GCAL flag to any character string not including spaces or formatting characters, and not beginning with a letter or digit. However, at any time no two flags may have the same definition. The **flag** directive is used to define a system flag string for any flag except the directive flags. Its syntax is

`.flag` *name flag-string*

where *name* is the name of the flag to be defined. The permitted names are

```
atab            bareinsert      bold            concatenation
ctab            dhyphen         down            equals
etab            exactspace      font            fontgroup
gfont           gpop            highwater       incinsert
insert          itab            level           nosplit
nostretch       null            overprint       quote
sentence        startbold       startcapitals   startemphasize
startlowercase  startunderline  stopbold        stopcapitals
stopemphasize   stoplowercase   stopunderline   subscript
superscript     sysinsert       tab             underline
up              uppercase       width           xstretch
```

Thus, for example, to change the insert flag to be two question marks instead of two tilde characters,

```
.flag insert ??
```

would be used. Note that the string of flag characters is *not* enclosed in quotes. (This allows quotes to begin a flag string.) The directive

`.dflag` *flag flag*

sets the character strings representing the directive flag and the basic directive flag, which are only recognized at the start of a line. If these flags are undefined, the rest of the input is treated as text.

Input lines which contain the flag-defining directives are *not* normally scanned for flags and inserts before processing. Thus attempts to define the same system flag twice are successful. There is, however, one such directive whose input line *is* scanned before processing. This is **sflag** – it behaves exactly as **flag**, the only difference being in the scanning of its input line for flags. This directive is provided so that, for example, a macro can be defined to create a flag whose character string is given as an argument to the macro. It should be used with due care.

12. User Flags

The built-in flags of GCAL, like the built-in directives, provide facilities at a fairly low level. In the case of directives, higher-level operations can be built up by using macros. A similar facility for flags is provided by *user flags*, though arguments are not available.

The action of a user flag is defined by a string of characters which are interpreted in the place of the flag whenever it is encountered in the input. For example, ‘`[`’ could be defined as the string ‘`$U9`’ and ‘`<`’ as ‘`$U4.5`’, giving two different kinds of superscript. User flags can also be used to simplify the input for diacriticals. For example, the directive

```
.flag .. "$O $U6 .$W3 .$W3 $D6"
```

defines ‘`..`’ as a flag which produces an umlaut over the preceding character, assuming its width was 6 points. If the flag definition is omitted from the input, a source containing such sequences can be processed in plain mode and still produce readable output for proofing purposes. The user flag facility can be used to make several flag sequences synonymous, and in particular to disable flags by converting them to the null flag.

Sometimes it is useful to be able to define ‘brackets’ consisting of the same flag string. For example, suppose that a text contained many occurrences of individual words that had to be set in a special font. The number of keystrokes could be minimized by choosing a single special character to precede and follow each special word. The GCAL **flag** directive allows a flag to be defined with *two* replacement strings. These are used alternately when the flag is encountered. Thus the above example could be done by

```
.flag | "$f34" "$f"
```

whereupon the words could be ‘bracketed’ with vertical bars.

As well as defining user flag sequences, the **flag** directive is used for changing the character strings used for the built-in flags, as described in section 11 above. Input lines containing this directive are not themselves scanned for flags before processing. If such scanning is required, the

sflag directive (see above) should be used instead.

13. Unsetting Flags

A flag-setting directive with no flag-string argument causes the relevant flag to be undefined. Thus to cancel, say, the underline flag,

```
.flag underline
```

can be used. There is also a directive

.cancelflag *flag*

which cancels a particular system or user flag sequence, whatever flag it represents. The argument to this directive is a sequence of flag characters, not a flag name:

```
.cancelflag ??
```

cancelflag with no argument cancels all flags except the two directive flags.

14. Disabling Flags

It is sometimes useful to be able to disable the recognition of flag sequences temporarily, for example, when including fragments of a computer program in a document. The two directives **flags** and **noflags** enable and disable flag recognition, with the important exception of the directive flags, respectively. This facility acts at a low level when GCAL is processing input text. It is a global facility and is not related to the GCAL environment. Following a call to **noflags**, all ordinary text input lines and the contents of any obeyed macros are affected. It is therefore possible to obey the same macro with and without flag interpretation, as follows:

```
.macro demo
Use $bf flags $rm inside a macro
.endm
.noflags
.demo
.flags
.demo
```

Note that it is not possible to make use of arguments inside a macro if flags are not being interpreted.

At the start of processing head and foot lines, GCAL is set to interpret flags, independently of what the state for the main text is. The **noflags** directive can be used if necessary, but the external state is restored at the end of head or foot processing.

The string representing the directive flag can be changed using **dflag** (see above), and in particular it can be undefined, in which case the rest of the input is treated as text.

15. Fancy Mode Directives

Certain directives whose use is restricted to fancy mode are described in this section. With the exception of **fancy** itself, they are all ignored in plain mode (but a single warning message is given if any are encountered). Two of these directives, **fancy** and **bindfont**, normally appear only at the start of the input.

In plain mode, the units of length used for specifying horizontal and vertical distances are the (fixed) character width and the current line spacing respectively. In fancy mode the units used are *points*. The standard Anglo-American definition of the printers' unit called 'point' was established as exactly 0.013837 inches in 1886 (continental European countries still use a slightly larger point). This gives approximately 72.27 points per inch, with only a very small error. However, many devices for which GCAL is used have resolutions which are more easily converted to 'big points' of which there are exactly 72 in an inch. Consequently GCAL works by default in 'big points'.

Fancy mode is requested by the presence of the **fancy** directive, which must appear at the start of the input, except possibly for macro definitions and **set**, **include** or **library** directives. The standard styles in the macro library set up fancy mode if it is appropriate, so it is not normal to

find this directive in user input. The format of the **fancy** directive is

.fancy [*units*] ["*comment*"]

The first optional argument specifies the units of length, and must be one of the following words:

```
bigpoints realpoints
```

The second optional argument is a string. If present, it is added to the standard comment line giving time, date and GCAL version that appears at the head of GCODE files written by GCAL. Programs such as GTYPE display this text. The standard style definitions place the name of the style in this comment.

The line depth is set to 12 points when the **fancy** directive is obeyed; it can be changed subsequently by means of the directive

.linedepth *dimension*

where *dimension* is the new depth in points. Up to three decimal places are allowed, for example

```
.linedepth 13.5
```

In fill mode, a new line of output is forced by **linedepth**, any remaining material in the fill buffer being output at the old line depth. The vertical position of a line is the position of the bottom of the characters, not counting the 'descenders' of characters such as 'g' or 'p'. The line depth is used *after* a line has been printed, to determine how far to move down before printing the next line. Thus any white space created by means of a large line depth appears below the relevant lines.

15.1 Device Resolution

The horizontal and vertical resolutions of a device are the smallest distances that can be specified for character separation. They vary from very coarse (0.1, 0.125 inches for a lineprinter) to very fine (less than 0.001 inches for phototypesetters). The GTYPE output program does the best it can with the resolution available, but rounding effects can be eliminated by selecting character widths and line depths that can be converted exactly. For example, if the output device is to be a Diablo daisy-wheel printer, whose resolutions are 1/120 and 1/48 inches, then horizontal and vertical absolute distances should be expressed in multiples of 0.6 and 1.5 points respectively.

To guard against slips, there is a facility to make GCAL round dimensions to specified resolutions. The two directives

.minwidth *dimension*
.mindepth *dimension*

allow different resolutions to be set for horizontal and vertical distances. The default settings are both 0.001 points, the smallest dimension that GCAL can deal with. The standard style definitions make appropriate use of **minwidth** and **mindepth** .

15.2 Binding Fonts

Once fancy mode has been selected, there must be at least one occurrence of the **bindfont** directive before any material can be formatted. This creates an association between a GCAL font number (in the range 0–254) and a set of character and space widths held in a *font library*. The format of font libraries is described in appendix B. A font number may only be bound once, and it remains bound for the rest of the GCAL run. It is normal practice to bind all the fonts required at the start of the input, normally in a style definition file. The syntax of **bindfont** is as follows

.bindfont *number string* [*magnification*] [default]

If the word 'default' appears as the last argument, the font binding only takes place if the font has not previously been bound. This feature is used in the standard style definitions; it allows the user to bind some non-standard fonts and then call the style definition to bind the rest.

Of the two mandatory arguments to **bindfont**, the number is that of the GCAL font, while the string identifies the data in the font library. It normally consists of two parts, separated by a solidus character. The first part conventionally identifies a file, and the second a font within the file. For example

```
.bindfont 3 "diablo/elite"
```

which selects the font called ‘elite’ from a file called ‘`diablo`’. The third argument to **bindfont** is an optional magnification factor to be applied to the font. This is specified as an integer which is 1000 times the magnification required. Thus, to use a font at 1.2 times the size defined in the font library, a directive such as

```
.bindfont 45 "xdevice/roman10" 1200
```

would be used. For devices with a fixed set of fonts the font library usually contains absolute size information. However, for devices that can magnify their fonts arbitrarily the size information may be relative. For example, the font library member for the Adobe Type Library, used in PostScript printers, defines the sizes for a ‘one-point’ font. Magnification is therefore always required, and to use a 12-point font the correct directive would be

```
.bindfont 52 "atl/Times-Roman" 12000
```

To guard against mistakes, GCAL generates an error if a magnification of more than a million is requested.

The successful use of magnification is of course dependent on the availability of the facility on the final printing device. On certain laser printers, arbitrary magnifications are possible, on others only particular sizes are available, while on some there is no magnification facility at all. If the same character font is bound more than once (using different GCAL font numbers), the font file is only scanned for the first binding.

When binding extra fonts for use in conjunction with the standard style definitions, users are recommended to use font numbers greater than 50. There is then no possibility of a conflict with numbers already in use, or future extentions. The actual fonts used in all the standard styles are documented in *GCAL’s Standard Macros*.

15.3 Spacing Units in Fancy Mode

For compatibility with plain mode, directives such as **linelength**, **pagelength** and **tabset**, which specify horizontal and vertical distances, *and which are available in both modes*, work by default in terms of characters and lines in both modes. Because character widths are variable in fancy mode, the width of an exact space is used as the unit. This in turn depends on the font which is current at the time.

Although this is normally convenient, there are circumstances where the use of absolute units of length is easier. In fancy mode, it is possible to specify dimensions in *points* simply by including a decimal point in the value. Thus, for example,

```
.indent 6     means ‘indent 6 spaces’
.indent 6.0   means ‘indent 6 points’
```

The values contained in the system variables **linelength**, **pagelength**, etc. are always in absolute units, and are inserted with a decimal point in fancy mode. Thus if it necessary to preserve such a value and restore it later, a sequence such as

```
.set save ~%linelength
.linelength . - 6
<intervening input>
.linelength ~~save
```

can be used, which works in both plain and fancy modes. (In plain mode ‘`save`’ is a number variable, while in fancy mode it is a dimension variable.) The use of absolute dimensions should be confined to style definition and other header files, so as to keep the main input as device-independent as possible.

16. Variables

Two classes of variable are available in GCAL: *user* variables (often just called ‘variables’) and *system* variables. The contents of the former are controlled by the **set** directive, whereas system variables are updated within the GCAL program.

16.1 User Variables

The user of GCAL may define four sorts of variable; string, numeric, dimension and boolean. Variable names consist of any sequence of letters and digits, beginning with a letter. Upper case and lower case letters are synonymous. The style definition files usually used with GCAL make use of variables to control many of their actions. The existence of a user variable can be tested by the ‘`set`’ operator, as in

```
.if set vname
```

Once a user variable has been set, it can never become unset.

A *string* value may be assigned to a variable by the directive

```
.set name "string"
```

Single or double quotes may be used (though double quotes are recommended), and a quote of the type being used to delimit the string may be represented in the string by doubling. The quote flag (‘`$""`’) may also be used. The graphic characteristics of the string (font, bold, underline, and so on) are retained. An explicit length may be given in brackets following the string. Thus

```
.set A "123"(4)
.set B "abcd"(1)
```

sets A to ‘123 ’ and B to ‘a’. If the length is greater than the string length, spaces are used as padding (as in the first example above). If the length is greater than the maximum line length, an error occurs. A starting position may also be given following the length.

```
.set A "123456"(2,3)
```

sets A to ‘34’. The first character in the string has position 1.

A *numeric* value may be assigned to a variable by the directive

.set name *number*

where *number* is an arithmetic expression that evaluates to an integer. The character ‘.’ may be used in this expression to represent the previous value of the variable. Thus, after obeying

```
.set xyz 123
.set xyz . - 23
```

the value in the variable `xyz` would be 100.

A *dimension* value may be assigned to a variable by the directive

.set name *dimension*

where *dimension* is a dimension expression, that is, an arithmetic expression containing values with decimal points other than in pairs separated by the division operator. The character ‘.’ may be used in this expression to represent the previous value of the variable. For example

```
.set npindent 13.5
.set dsdepth (~%linedepth * 2)/3
```

A *boolean* value may also be assigned to a variable by the **set** directive. For example

```
.set a true
.set cond ~~b >= 6
```

If such a variable is inserted into a line, the strings ‘true’ or ‘false’ are substituted as appropriate.

16.2 System Variables

In addition to user variables, a number of pre-defined system variables exist, whose values the user cannot change directly. These have alphabetic names, but are distinct from any user variables with the same names. Thus the directive

```
.set page 23
```

sets the *user* variable **page**, not the system variable. System variables currently defined are shown in table 1 below. Those whose type is shown as ‘dimension’ are inserted with a decimal point in fancy mode.

contigpending	boolean	pending contiguous lines
date	string	date, e.g. ‘5 November 1990’
day	numeric	day of the month
exactspacewidth	dimension	exact space width
fancy	boolean	fancy mode
font	numeric	current font
fontgroup	numeric	current font group
fontsize	dimension	size of current font
footlength	dimension	foot length
headlength	dimension	head length
hpageoffset	dimension	horizontal page offset
inaside	boolean	in aside section
incontig	boolean	in contiguous section
incopy	boolean	in copy section
indent	dimension	indent
infootnote	boolean	in footnote
justify	boolean	right-justification is on
lastspace	dimension	previous white space
leftonpage	dimension	space left on page
linedepth	dimension	line depth
linelength	dimension	line length
linenumber	numeric	input line number
linespacing	numeric	line spacing
mindepth	dimension	vertical device resolution
minwidth	dimsneion	horizontal device resolution
minpar	numeric	minimum paragraph on page
month	numeric	month
page	numeric	current page number
pagecount	numeric	number of pages processed
pagelength	dimension	page length
parindent	dimension	paragraph indent
parspace	dimension	paragraph spacing
sentencespace	numeric	sentence space (thin spaces)
spacewidth	dimension	space width
textlength	dimension	length of text page
thinspacewidth	dimension	thin space width
time	string	time, e.g. ‘11.15’
used	dimension	amount of text in block
usedonpage	dimension	amount of text on page
version	numeric	GCAL version number
vpageoffset	dimension	vertical page offset
year	numeric	year

Table 1: GCAL System Variables

The variables **leftonpage** and **usedonpage** are only incremented when material is irrevocably committed to the page. Thus, while GCAL is in the act of reading the lines of a footnote or other contiguous section, they do not change. However, the variable **used**, which is the same as **usedonpage** at the outer level, is set to zero at the start of such sections and incremented as they are read. On returning to the outer level, **used** is reset to the same value as **usedonpage**. A call to the **newline** directive should appear before a directive which uses any of the variables **leftonpage**, **usedonpage** or **used**, to force out any left over part lines. This applies in copy mode as well as in fill mode, since a copied line is not output until GCAL is sure it is not followed by **nosep**.

The variable **lastspace** contains zero immediately after processing a text line; after any operation that generates vertical white space, it contains the current amount of space. During the processing of a contiguous section it relates to the lines in that section; afterwards it is reset if the section is not in fact printed inline.

The system variable **page** is incremented when a new page is begun. A precise definition is that **page** contains the number of the page to which the *end* of the previous text input line belongs. It remains constant throughout the processing of an individual input line. If a part line, into which the value of **page** has been inserted, is carried over from one page to the next, the page number that was current when the line was read (i.e. the previous page number) appears in the line.

The system variables **fontsize**, **spacewidth**, **exactspacewidth** and **thinspacewidth** contain the appropriate values for the currently selected font. The size includes any magnification that was specified when the font was bound.

The date and time are obtained from the operating system when GCAL is entered, and the values of all the associated system variables do not change during a run.

16.3 Inserts

The contents of a *user* variable may be inserted into a line by the use of one of the insert flags. The most useful flag for inserting text is the *bare insert* flag, which defaults to the string ‘`~!`’. Thus the sequence

```
.set a "abcd"
The string is ~!a.
```

causes the characters ‘abcd’ to be substituted for ‘`~!a`’. When characters from a string variable are inserted using this flag, the font used is the font which is current at the time of insertion. Thus the same string can be inserted time and time again, in any number of different fonts. Numerical variables are always inserted using the current insert-time font. Another insert flag, called simply the ‘insert flag’ (for historical reasons) retains the font which was current at the time the string variable was set. Its default character string is ‘~~’.

If either insert flag is followed by a digit, it specifies the insertion of a macro argument (see section on macros below). If the insert flag is not followed by a letter or digit, the following character is not interpreted. This may be used to insert the insert flag itself into text lines; it can also be used to insert hyphen characters that are not available for line splitting, full stops that are not to be treated as ending sentences, and any characters that might otherwise be taken as part of a flag sequence.

When the contents of a variable are being inserted, if the inserted characters are to be followed in the output by a letter or a digit *or a space*, the name of the variable must be terminated by the *null flag*. (Spaces in the input which follow flag sequences ending in letters or digits are ignored by default.) The default null flag is ‘`$$`’; it acts as a terminator in flag sequences but is otherwise ignored. For example, suppose the variable ‘`name`’ contains a name |
that is to be inserted into a sentence; the input should be |

```
This sentence contains ~!name$$ in the middle.      |
```

Alternatively, the **noignoreflagspace** directive can be used to cause GCAL not to ignore spaces |
following flag sequences that end in letters or digits. The null flag is still necessary to terminate a |
variable name if a letter or digit follows immediately. |

When a user variable containing a string is inserted into a line, the graphic characteristics (underlining, font and so on) which were in force when the variable was set are reproduced if the insert flag ‘~~’ is used. If the line into which insertion occurs is being underlined, uppercased, lowercased, or marked as emboldened at the point of insertion, these characteristics are added to the inserted characters if not already present. If it is desired that the graphic characteristics of a string insert be re-evaluated in the current environment, the *bare insert flag* (default ‘`~!`’) must be used. For a bare insert, all information in the variable concerning underlining, emboldening and font selection is ignored. However, the position of characters above or below the line (i.e. whether they are subscripts or superscripts) is still preserved. Numeric and boolean variables are always inserted using the current graphic characteristics. In no cases are inserted characters re-scanned for flag occurrences.

An insert flag which causes automatic incrementing of numeric user variables whenever they are inserted is also provided. The default for this flag is ‘`~+`’. The variable is incremented *before* its value is inserted into the line. For example, the sequence

```
.set a 123
The value is ~+a.
```

causes the string ‘124’ to be inserted for ‘~+a’. Note that the incrementing insert flag does not apply to variables that contain dimensional values. If used for such variables (or for string or boolean variables) it acts as the ordinary insert flag. |

The contents of a *system* variable may be inserted into a line by the use of the system insert flag. The default system insert flag is the string ‘~%’. Thus, if

```
~%date
```

is encountered in the input, the current date is inserted. The values of system variables are always inserted using the current graphic rendition. The insert flags apply both to text lines and to directive lines, except for those directives which define (or cancel) flag sequences. Thus directives are able to make use of the contents of variables.

16.4 Format of Numeric Inserts

The format in which a numeric (integer) variable is inserted into lines may be specified by the directive

`.format` *name format*

for user variables, and by

`.format` *%name format*

for system variables. The default format is simply a decimal digit string in arabic numerals, preceded by ‘-’ if the number is negative. The following formats may be specified:

`<null>`	resets default format
`A`	upper case alphabetic
`a`	lower case alphabetic
`R`	upper case Roman numerals
`r`	lower case Roman numerals
`F`	fixed point (for dimensions)
`f`	same as `F`

`<null>` means that no format is specified, for example

```
.format somename
```

and does not mean that the word ‘null’ is used. An attempt to set a format for a string, boolean or dimension variable is faulted. |

The alphabetic format applies to numbers in the range 1–26 only. If a number outside this range is inserted in alphabetic format, the default arabic representation is used. The fixed point format is used by GCAL for system variables such as **linelength**, and for dimension variables, | which are actually stored in millipoints. It is not normally of use at the user level, but is documented here for completeness.

Note that explicit arithmetic cannot be performed with variables whose format has been set to anything other than the default. If, for example, the directives

```
.set A 10
.format A R
```

have been obeyed, a directive involving arithmetic, such as

```
.set B ~~A + 1
```

would be expanded into

```
.set B X + 1
```

and would therefore generate an error message, since ‘X + 1’ is not a legal arithmetic expression. The incrementing insert flag, however, is unaffected by the format of a variable, as is the use of ‘.’ to specify the current value of the subject variable in a **set** directive.

A common requirement is to print page numbers in Roman numerals and also to vary the footline according as the page number is even or odd. The easiest way to achieve this is to leave the **%page** variable with the default format, and copy its value into a Roman format variable for printing. For example,

```
. Set format for copypage – must define it first
.set copypage 0
.format copypage r
.foot
.set copypage ~%page
.if odd ~%page
$e ~!copypage
.else
~!copypage
.fi
.endf
```

16.5 Indirect User Variables

It is sometimes convenient to be able to insert the value of a variable whose name varies under different circumstances. This can be done by using an *indirect variable*. A **set** directive of the form

`.set` *name* @"*string*"

creates such a variable; whenever it inserted, the effect is to insert the contents of the user variable whose name it holds. There is an error if no such variable exists. The final variable may be of any type, and any of the three user insertion flags may be used. Thus, for example,

```
.set xyz 24
.set example @"xyz"
~~example$$ ~+example
```

produces the output ‘24 25’.

16.6 Counting

GCAL variables can be used to count sections, paragraphs, equations, etc., thus removing the need to re-number following an insertion or deletion. The source of the standard macro library contains many examples of the use of GCAL variables for counting and other purposes. The following example shows a two level numbering scheme for sections and paragraphs using explicit incrementation:

```
.set S 1
.set P 1
~~S.~~P#Part 1.1
- - -
.set P .+1
~~S.~~P#Part 1.2
- - -
```

If a numeric variable is inserted using the incrementing insert flag (default ‘~+’), its value is automatically incremented before insertion. This could have been used instead of explicit incrementing directives in the example above, thus:

```
.set S 0
.set P 0
~+S.~+P#Part 1.1
- - -
~~S.~+P#Part 1.2
```

When the format of a variable has been set to Roman numerals or alphabetic, the incrementing insert flag still works correctly.

16.7 References

References to sections, paragraphs, equations, etc. can be made symbolic by the use of GCAL variables. Thus, for example, if the variable E were being used to count equations, one might have

```
.display
ps - qr = 4/(ab)$e(~+E)
.endd
.set eqn1 ~~E
- - -
- - - in equation ~~eqn1$$ - - -
```

Forward reference handling is not provided, however.

17. Macros

GCAL macros are a means of giving names to particular sequences of input lines. The sequence can then be used as often as needed at any point in the input, simply by quoting its name as a directive name. In other words, the user is given the facility of constructing directives out of the basic built-in ones. A macro which provides the initial text of a section could, for example, be defined as follows:

```
.macro section
.newline
.ensure 6
.unless ~%usedonpage = 0
.space 2
.fi
.endm
```

and called by starting an input line with '`.section`'. When GCAL encounters the **macro** directive, subsequent lines, up to the terminator **endm**, are not processed, but are stored up until the macro is called.

A given macro definition can be deleted at any time by means of the **cancelmacro** directive, which takes a single macro name as its argument. A new definition may then be given to take its place. The existence of a given macro can be tested by the '`macro`' operator in an expression. For example

```
.unless macro somename
.macro somename
<supply a definition>
.endm
.fi
```

Macros may be called as often as necessary. The definitions normally occur at the start of the input, usually in the form of separate macro libraries, but the only requirement is that macros are defined before they are used. In the example above, a macro was used as a shorthand for a fixed piece of input text. More flexibility can be obtained by the use of macros with arguments. By this means, portions of the substituted text may be varied from call to call.

The use of macros for high-level concepts such as start-of-section, start-of-chapter, etc. is highly recommended. Not only does it save typing, it also means that the format of these items can easily be changed as only the macro definition is involved. It also ensures uniform treatment throughout the document. The standard library frequently referred to in this document is a set of macros which provide commonly required facilities in styles suitable for a number of different output devices. Much use is made of variables and user flags to make it easy to change some aspects of the styles without having to alter the macros themselves. Nevertheless, the source of these macros provides examples for users who wish to create their own special styles.

The full syntax of the **macro** directive is

```
.macro name   defarg1[,]   defarg2[,]   ...   defargn
<lines>
.endm
```

which defines a macro with *n* arguments. Optional commas may appear between the prototype arguments. The terminator of the macro definition (**endm**) may be specified using the ordinary directive flag or the basic directive flag. The argument values given may be numbers, dimensions, boolean values or strings (enclosed in quotes), and they specify default values for the arguments,

used when the macro is called with fewer arguments than are present in its definition. Because of the common use of two single quotes in succession to mean a closing double quote on typographic devices, it is recommended that double quotes be used for strings in macro definitions, especially if the null string is involved.

In the body of the macro, the values of the arguments are referenced using one of the insert flags described in section 16.3 above. The flag is followed by the relevant argument number. For example,

```
.macro simple "text"
~!1
.endm
```

is a macro that simply inserts its argument whenever it is called.

Macros may be defined and called from within other macros, but there is no concept of restricted scope, as in block-structured programming languages. All macros are 'global' in this sense. Recursive calls are permitted, though this should be undertaken with due care. Lines read as part of a macro definition are not processed for tabs and inserts in the normal way at definition time, but only when they are subsequently recalled. The line containing the **macro** directive is, however, processed in the normal way before it is decoded.

Macros need not in general have names distinct from the built-in directives; however, the names of organizational directives such as **if** and **unless** may not be used. The standard styles make use of this feature to provide 'value-added' forms of some of the basic GCAL directives, for example, a **footnote** directive that maintains a count.

A macro is called in the same way as a directive, by writing its name following the directive flag at the start of a line. If a macro has been defined with the same name as a built-in directive, the macro takes precedence; the basic directive flag (default '`$.`') can be used to force recognition of the built-in directive. For example, the input line

```
.indent
```

is a call to the macro **indent**, if it exists, otherwise it is a call to the built-in directive **indent**. However,

```
$.indent
```

is always a call to the built-in directive.

When a macro is called with arguments, these must be separated by spaces or commas. Quotes must be used for strings, except for a string which is the last argument in a call. In this case, if the quotes are omitted, the rest of the input line is taken as the string. Leading ordinary spaces are removed from the string, but others (e.g. exact spaces) are not. Successive commas indicate omitted arguments, as does a premature end of line. In these cases the default values are used. The presence of too many arguments causes an error. The arguments may be accessed in the body of the macro by using the insert flag followed by the number of the argument. Numeric arguments may be incremented by means of the incrementing insert flag within the body of the macro.

For historical reasons, if a macro argument is defined as a dimension expression, but the macro is actually called with a string value for that argument, an attempt is made to convert the characters of the string into a dimension. This feature should not be relied upon.

A macro call should be thought of as a shorthand notation for the lines which form the macro body. Although there is no concept of block structure or limited scope, the insertion of macro arguments can only occur inside a macro body.

If a macro contains directives which cause input lines to be remembered for later use (e.g. the **head** directive), then it is not possible to cause insertion of the macro arguments into those lines, as they are not scanned for flags while the macro is current. For example, if

```
.macro setup "Heading"
.head
$C ~~1
.endh
.endm
```

is obeyed, then the macro call

```
.setup "Chapter 1"
```

does *not* have the (presumably) intended effect. It is equivalent to

```
.head
$C ~~1
.endh
```

which provokes an error message when the head lines are processed at the top of the next page. The correct way to achieve what was intended is to use a variable, thus:

```
.head
$c ~~heading
.endh
.set heading "Chapter 1"
```

18. Conditional Directives

A sequence of input lines

```
.if logical expression
- - -
.else
- - -
.fi
```

may be used to skip parts of the input. The directive **unless** is also available, and the separators **elif** (else-if) and **elul** (else-unless) may be used to form multi-way branches, for example

```
.if ~~A > 6
- - -
.elul "~~B" = "XYZ"
- - -
.else
- - -
.fi
```

A particular form of logical expression is

```
fit dimension
```

which tests the space left on the current page. The dimension can be specified as a number of lines or, by including a decimal point, as a number of points. The most common use of **fit** is to select whether to output blank lines or start a new page. For example

```
.if fit 6
.space 2
.else
.newpage
.fi
```

This ensures that at least the next four lines of output appear on the same page, preceded by two blank lines if not at the top of a page. When lines are skipped as a result of **if** or **unless** they are not processed for tabs or other flags. The **if** and **unless** directives may be nested to any depth.

19. Justification

By default the GCAL program does not right-justify lines of text that have been filled in fill mode. (However, several standard styles select justification.) The directive **justify** requests right-justification. In plain mode, justification is done by inserting additional space characters into the line, between words. This does not normally result in typographically pleasing output. In fancy mode the increment by which spaces are expanded to achieve justification is specified by the **minwidth** directive. The default value is 0.001 points, the smallest dimension recognized by GCAL. The directive **nojustify** suppresses right-justification of filled lines. The system variable **justify** is '`true`' when justification is turned on.

Justification of individual lines can be achieved by means of the directive

```
.newline justify
```

This works in fill mode and in copy mode.

20. Paragraphs

In fill mode, GCAL starts a new paragraph when a line beginning with a space character, or a blank line, is encountered in the input, unless the directive **noautopar** has been obeyed. In this case spaces at the start of lines are not treated specially, and blank lines are totally ignored. The default action can be restored by obeying **autopar**. There is also a directive to force the start of a new paragraph, independent of the auto-paragraph setting:

```
.par [justify]
```

If **justify** is specified, the last line of the previous paragraph (if not yet output) is right-justified. The action taken by GCAL at the start of a paragraph is controlled by three directives:

```
.parindent dimension
```

This specifies the amount of indentation at the start of paragraphs as a number of exact spaces or, in fancy mode, as a dimension in points. The default is zero, though the standard style definition files change this, depending on the output device. **parindent** is essentially an automatic way of setting a temporary indent (see below) at the start of each paragraph. It overrides any prior temporary indent setting, and can itself be overridden in turn by a subsequent temporary indent setting. When a non-zero (permanent) indent is set (see below) the paragraph indent is taken relative to it, and may therefore be negative.

```
.parspace dimension
```

This specifies the minimum amount of blank space that is to precede each paragraph, unless it starts at the top of a page. The default is one blank line, but many standard style definitions change this. See *GCAL's Standard Macros* for details. When a new paragraph is begun, GCAL checks to see if the last output to the page was blank space. If it was, and if it was greater than or equal to the **parspace** parameter, no action is taken. Otherwise a suitable amount of space is output to make the total up to this amount.

When the auto-paragraph feature is being used (see above), **parindent** and **parspace** directives must *not* be placed immediately after blank lines if they are to affect the next paragraph. The appearance of a blank line (when auto-paragraphing is on) causes paragraph spacing and indenting for the next paragraph to be set up at that point, before any subsequent lines are read. Hence the appearance of one of these directives after a blank line does not affect the paragraph that has effectively already begun.

```
.minpar number
```

This specifies the minimum *number of lines* of a paragraph which may appear on a page. It is *not* a dimension. A new page is forced if a new paragraph is begun within this many lines of the bottom of a page. The default value is two.

21. Hyphenation

GCAL contains some primitive facilities for hyphenation, for use when splitting lines in fill mode. Various levels of hyphenation may be requested by the directive

```
.hyphen level
```

The default hyphenation level is 1, and the various levels work as follows:

0	discretionary hyphens only
1	split at hyphenation indicators only
2	) reserved
3	) reserved
4	) reserved
5	Italian hyphenation

A discretionary hyphen is indicated by the discretionary hyphen flag, '$~'. It indicates a position at which it is permitted to split the word and insert a hyphen. If the word is not split at such a point, no hyphen is inserted. Discretionary hyphens are always recognized. The 'normal' hyphen indicator is the minus character. Except in hyphen mode zero, it is treated as a possible line-splitting point. However, if the hyphenation character appears in the input preceded by the insert flag, it is treated as a normal character, and the line is never split at that point. A word may contain any number of discretionary and/or ordinary hyphens, for example

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

The last type of hyphenation (Italian hyphenation) splits words at any point, underlining the last character on the first line and the first character on the second. To make it reasonable, only words longer than five letters are split. In a way this is a return to the ways of medieval scribes, who split words in much more cavalier fashion than is the current practice.

In some fonts, on some devices, the hyphenation character has a different character code to that generated by the 'minus' character on the keyboard. GCAL automatically arranges that hyphens are converted to the appropriate character where necessary. If, however, the insert flag is used to insert a minus character that is not eligible for hyphenation, no conversion is performed.

Many typographic fonts also contain two other kinds of short horizontal line, the en-dash and the em-dash. These are normally set up as ligatures, associated with input strings consisting of two and three minus characters respectively.

22. Spaces in Fill Mode

In fill mode, there are two independent attributes which pertain to a normal space between words.

(a) Whether it may be stretched to achieve right-justification;

(b) Whether the line may be split at that point.

There are thus four possible kinds of normal space, which are specified to GCAL as follows:

(1) An *ordinary* space (normally just called a space) is specified by a space character in the input. Such a space is both stretchable (attribute (a) above) and splittable (attribute (b) above). In fancy mode it has a minimum width, and is stretched in units specified by the **minwidth** directive. Not all space characters appearing in the input are treated as part of the text however. In particular, spaces following a flag sequence which ends in a letter or digit are ignored, unless **noignoreflagspace** has been obeyed. Also, all space characters preceding any form of tab are ignored. |

(2) An *exact* space is specified by a flag in the input whose default value is '#' (the printer's sign for 'space'). Such a space is neither stretchable nor splittable, and it behaves exactly as though it were a printing character.

(3) A *non-splitting* space is specified by a flag in the input whose default value is '$>'. Such a space may be stretched to achieve right-justification, but the line may not be split where it appears.

(4) A *non-stretching* space is specified by a flag in the input whose default value is '$#'. Such a space may not be stretched, but the line may be split at the point where it appears.

Whenever a line is split, *all* spaces at the splitting point with the splittable attribute (i.e. those of types (1) and (4)) are removed from the output.

In plain mode, the widths of all four kinds of space are the same, namely, one character width. In fancy mode the widths of ordinary and exact spaces are specified separately for each font. The non-splitting space takes the width of an ordinary space, while the non-stretching space takes the width of an exact space. However, it is possible to change these widths for individual spaces by means of the width flag, '$w', which is described in full in the section entitled *Overprints and Accents* below. The following flag sequences set the width of the preceding character, whether it be space or some printing character, to the values shown:

$w#	width of an exact space
$w>	width of an ordinary space
$w<	width of a thin space

(Thin spaces are described shortly.) Thus it is possible to set up user flags for all 12 possible kinds of space. System flags are provided only for those that are felt to be in common use.

For the purposes of underlining in fill mode (see below), exact spaces and non-stretching spaces are treated as 'characters', while ordinary spaces and non-splitting spaces are treated as 'spaces'. This also applies to the compression of multiple spaces, which GCAL does by default in fill mode. A succession of stretchable spaces (space characters or '$>'s) is reduced to a single space on input by ignoring all but the first of them. This action can be disabled by the directive **noautospace**. In this case space compression does not occur. The directive **autospace** can be used to reset the default state.

There are also two kinds of *special space* in GCAL. The flag '$<' indicates a *thin space*. The width is font-dependent, but is normally about one sixth of the normal space, where the device permits this. There is also an *extra-stretchy space*, indicated by the flag '$<>'. This behaves like a normal space except when it appears in an output line that is being stretched to achieve right-justification. If such a line contains any extra-stretchy spaces, *all* the stretch is distributed equally between them, the normal spaces remaining unaltered. A typical use is to push some text to the right at the end of a paragraph:

```
...  This is the text of the paragraph. We will
put the final word flush right at the$<>end.
.nl justify
```

Note that it is necessary to include the **newline** directive with the **justify** option, to force the last line of the paragraph to be right-justified. The initial width of an extra-stretchy space can be set using the '$w' flag as described above. The width of a thin space can also be so set, but then it is no longer a thin space.

23. Spaces in Copy Mode

In copy mode all spaces are treated as data characters. The only difference between the six representations (space, '#', '$>', '$#', '$<' and '$<>') is the default width used. Leading non-exact spaces in copied lines are never underlined.

24. Sentence Endings

In fill mode, GCAL automatically recognizes the end of a sentence if a full stop, question mark or exclamation mark is followed by an ordinary space or the end of the input line. The number of *thin* spaces which are to follow the end of a sentence may be set by the directive

.sentencespace *number*

If, however, this space is less than the normal space width, the latter is used instead. The width of a thin space is a font-dependent quantity. The default value for the sentencespace parameter is one, which gives even spacing for justified lines in both plain and fancy modes. This is in accordance with the recommendations of several authorities on typography.

The provision of a fixed amount of space after each sentence only takes place when **autospace** is on (the default). If **noautospace** has been specified, no special action occurs at the ends of sentences.

For those who insist on additional space at the ends of sentences there is an end of sentence flag (default value '$!') which may be used to indicate explicitly where end of sentence processing is to occur. This flag causes the *previous* character in the input line to be treated as a sentence ending character. An example of its use is

```
(See below for details.)$! Furthermore, ...
```

A full stop, question mark or exclamation mark in the input is *not* treated as a sentence ending character if it is preceded by the insert flag.

25. Line Lengths and Indents

People generally think of line lengths and indents as numbers of characters, rather than as actual lengths, particularly when working with fixed space devices. Using a character width as a unit also has the merit of being device independent. To allow for the possibility of variable width characters, the directives

```
.linelength dimension [nobreak]
.indent dimension [nobreak]
```

are defined to work in units of the width of the exact space character unless the dimension contains a decimal point, indicating an absolute dimension in points. Obeying **linelength** or **indent** causes a line break in the output, unless **nobreak** is specified. When **nobreak** is given, the only action is to alter the value of the relevant parameter. This means that there may be some text which precedes the directive in the input, but which is output at the new line length or indent.

The default line length of 65 characters (plain mode) or 390 points (fancy mode) is established at the start of processing or on obeying the **fancy** directive. Most standard style definitions change this, however. The default indent is zero. The system variables **linelength** and **indent** hold the current values of these parameters. In fancy mode they are inserted with a decimal point. | Variables can be used to preserve their values in plain or fancy mode, for example: |

```
.set linelengthsave ~%linelength                                     |
.linelength <new value>                                              |
<lines of input>                                                     |
.linelength ~~linelengthsave                                         |
```

In plain mode the integral value causes a numeric variable to be created, while in fancy mode a | dimension variable is used, but this need not concern the user. The current indent setting may be | temporarily over-ridden by the directive

```
.tempindent dimension [m] [nobreak]
```

This specifies an indent that is to apply to the next *m* output lines only. The default for *m* is one. It is useful for single-line headings between paragraphs, and for placing text in the margin of indented paragraphs. If *dimension* is an arithmetic expression containing the character '.' this is taken as the value of the current permanent indent, not the previous temporary indent. **tempindent** causes a line break unless **nobreak** is specified. The facility for automatically indenting paragraphs interacts with **tempindent** (see **parindent** above).

An analagous facility to **tempindent** is

```
.templinelength dimension [m] [nobreak]
```

It is useful for leaving space for diagrams, photographs, etc. Both the temporary line length and the temporary indent are reset (i.e. cancelled) whenever there is a change from copy to fill mode or *vice versa*, and whenever a nested environment is entered, for example, at the start of a footnote.

26. Joining Lines Together

Normally, when successive lines of input are being joined together in order to fill lines of output, a single space is inserted between them (except at the end of a sentence, when more than one space may be inserted). There are two facilities for changing this action, which act at different levels.

The concatenation flag (default value '`$j`', for 'join') takes effect when an input line is being scanned for flags, and it causes the next line of input to be joined on *as though it were an extension of the current input line*. This applies both in fill and in copy modes. Thus if the next line begins with the directive flag, it is *not* recognized, as logically it is treated as being part of the previous line. The concatenation flag cannot therefore be used immediately prior to a macro call. However, it may legally appear at the end of the last line of a macro body, as the effect of calling a macro is to substitute the lines forming the body for the line containing the call.

A higher level facility for joining lines is the directive **nosep**. This specifies that the next line of *text* (i.e. non-directive) input is to be joined to previous line (if any) with no separator. This facility works in both fill and copy modes.

To *prevent* successive lines of input being joined together in fill mode, the directive

```
.newline [justify]
```

may be used. This forces the start of a new line of output. It does *not* generate a blank line (see **space**). If **justify** is specified, the previous part line (if any) is right-justified. This facility works in copy mode as well as in fill mode, causing the previous copied line to be right-justified. If **newline** is frequently required in fill mode, the use of copy mode should be considered as an alternative.

27. Copied Line Specification

In copy mode, the linelength as set by **linelength** and **templinelength** is used only when processing centering and ending tabs (see below). Copied lines are always output with their length unaltered. However, if the length of a copied line exceeds the linelength, a warning message is output. The current indent, as set by **indent** or **tempindent**, is used to indent lines in copy mode. Both **tempindent** and **templinelength** are reset when entering or leaving copy mode.

Although in copy mode lines of input are not normally joined together, each line is not output until GCAL is sure that it is not followed by **nosep**. Therefore it is sometimes necessary to make use of **newline** to ensure that all previous lines of input have been disposed of.

28. Graphic Rendition

'Graphic rendition' refers to the way characters are printed or displayed on the final output device. GCAL supports four graphic rendition characteristics: underlining, emboldening, upper-casing and lower-casing.

28.1 Underlining

Underlined characters may be specified by means of the underline flag, which defaults to a single underline character. Whenever it is encountered, the underlining state is reversed. For example

```
not underlined _underlined_ not underlined
```

produces

not underlined <u>underlined</u> not underlined

There are also two flags for explicitly turning underlining on and off, the start underline and the stop underline flag. The default strings are '`$su`' and '`$pu`'.

In copy mode, the underline switch is only thing that determines which characters are underlined, except that leading non-exact spaces are never underlined. In fill mode, however, additional logic is required because of the creation of extra spaces caused by line filling and end of sentence processing. Two underline modes are available, controlled by the directive

```
.umode number
```

The possible values of n are:

0	all characters are underlined (default)
1	all character except spaces are underlined

In underline mode 0, the rule for underlining spaces is that all groups of stretchable spaces that are preceded and followed by an underlined character are themselves underlined. Non-stretchable spaces ('#' and '$#') are treated as characters for this purpose. Thus

```
_abcd_   #   _xyz_
```

produces

<u>abcd</u>   <u>xyz</u>

but

```
_abcd_      _xyz_
```

produces

```
abcd xyz
```

28.2 Emboldened Characters
For devices that can embolden *any* character (for example, by striking it twice), emboldened characters may be specified by means of the bold flag (default ‘`$b`’), which reverses the emboldening state in a similar manner to the way that underlining works. There are also two explicit flags to start and stop emboldening, with default values ‘`$sb`’ and ‘`$pb`’ respectively. For devices with typographic fonts, a separate bold font must be used instead of emboldening, which is not usually supported.

28.3 Uppercasing
The forcing of letters to upper case may be specified by means of the uppercase flag (default ‘`$%`’), which reverses the uppercasing state in a similar manner to the way in which underlining works. In addition, there are explicit ‘start upper-casing’ and ‘stop upper-casing’ flags, ‘`$scl`’ and ‘`$pcl`’. The mnemonic ‘cl’ stands for ‘capital letters’. Uppercasing is useful for dealing with headings, particularly when macros are being used.

28.4 Lowercasing
The forcing of letters to lower case may be specified by means of the explicit ‘start lower-casing’ and ‘stop lower-casing’ flags, ‘`$sll`’ and ‘`$pll`’. The mnemonic ‘ll’ stands for ‘little letters’. There is no flag to reverse the lower-casing state.

28.5 Stacking of Graphic Rendition
Whenever the graphic rendition is changed, the old state is remembered on a stack, and can be retrieved by use of the flag ‘`$g`’. The stack is 12 entries deep, and is ‘forgetful’ in the sense that, if more than 12 states are pushed onto it, the earliest are forgotten. There is a separate stack for each GCAL environment (see below). The main use of this stack is in flags for special characters, to enable them to restore the external state.

29. Tabs

Tabbed fields in the input are delimited by tab flags. The tabs are of three kinds:

(a) relative tabs
(b) absolute tabs
(c) special tabs

Relative tabs are similar in concept to the tabs on a typewriter, where each tabbed field is aligned according to a pre-set sequence of tab stops. The default relative tab flag is ‘`$t`’. The tab positions are specified by

```
.tabset a b c ...
```

This sets tab positions either relative to the start of the line, or relative to the previous position if the arithmetic expression contains a full stop character. Thus

```
.tabset 8 12 .+4 .+3
```

specifies tabs at columns 8, 12, 16 and 19. Tab positions may also be specified as left, right or centre aligned by following the tab position value by one of the letters ‘L’, ‘R’ or ‘C’ (the default being ‘L’). Right and centre aligned positions are specified by their rightmost or central character. The arguments for **tabset** represent a number of exact spaces unless decimal points are used to specify absolute dimensions in points. For historical reasons there is a difference in the way the values are treated, depending on the form of the *first* argument to **tabset**:

(a) If the expression defining the first tab contains no decimal points, then it is taken as representing a *column* number. The left most column of the page is numbered one.

(b) If the expression *does* contain one or more decimal points, it is taken as representing an absolute *offset* to the tab position. The lefthand edge of the page is offset zero. There is an exception to this rule if all the numbers with decimal points occur in pairs, and each pair is separated by a division sign. That is, the division of two absolute dimensions yields a relative dimension.

(c) An expression containing a mixture of numbers with and without decimal points is treated as type (b), with the integers being multiplied by the current exact space width to convert them to absolute dimensions.

For example, if the tab settings were

```
.tabset 10R 13C 17
```

then the input line

```
$t AB $t CDE $t XYZ
```

would be formatted to

```
        10  13  17
         |   |   |
        AB CDE  XYZ
```

Absolute tabs are independent of the relative tab settings. The absolute tab flag must always be followed by a dimension specifying the character or absolute position to which it refers, and optionally by ‘L’, ‘R’ or ‘C’ for a left, right, or centred tab. The default absolute tab flag is ‘`$a`’, and so the input line

```
abcd$a12 ef
```

would expand so that ‘ef’ appeared in columns 12–13. If an absolute tab is to be followed by a digit, a decimal point, or one of the letters ‘L’, ‘R’ or ‘C’, a space character or the null flag must be used as a separator.

Three special tab flags allow the specification of centred text, text aligned at the end of the output line, and text aligned at the current indentation. The default flags are ‘`$c`’, ‘`$e`’ and ‘`$i`’ respectively. For example, to centre equations with numbers at the right hand side, the input might be

```
.copy
$it $c pv = rc$e$rm(1)
$c watts = volts * amps$e(2)
```

The resulting output is

pv = *rc* (1)
watts = volts * amps (2)

The indent aligning flag (default ‘`$i`’) is useful when setting material in the left hand margin of indented text, particularly when variable width characters are involved. For example, the input

```
.indent 5
.tempindent 0
(1) $i Only the writer who is either very inexperienced
or singularly proof against experience
will let the beauties
of a word or phrase tempt him into displaying it
where it is conspicuously out of place.
```

produces

(1) Only the writer who is either very inexperienced or singularly proof against experience will let the beauties of a word or phrase tempt him into displaying it where it is conspicuously out of place.

All space characters in the input preceding a tab are ignored, and in fill mode, only those space characters that follow the first non-space character following the rightmost tab character in an input line are eligible for compression or line splitting.

It is important to realize that the tab operations in GCAL operate on *input* lines. Of themselves they do not prevent succeeding input lines being joined together in order to fill up output lines, though in fill mode a line containing a tab always causes a line break before it. For example, the input

```
$c Heading
This line follows the heading
```

produces the output

Heading This line follows the heading

which is probably not what is wanted. A **newline** or **space** directive should follow the heading, or it should appear in copy mode. The sensible uses of tabs in fill mode are very limited.

30. Input Line Processing Details

The use of backspace and other forms of over-printing in GCAL input is no longer supported. Most lines are re-scanned for GCAL flags and tabs immediately after they are read. However, lines which form part of a macro body (introduced by the **macro** directive) or which are specified as headlines or footlines (introduced by **head** or **foot**) are not re-scanned until the time they are used. Directive lines which define flag sequences are never re-scanned, but are used as they are (except in the case of **sflag**, which is provided specifically to break this rule).

When re-scanning does take place it proceeds from left to right and inserts are filled in as they are encountered. The text of inserts is not itself scanned for flags. Tab flags are replaced by a special indicator character, but are not expanded at this stage. Other flags such as underline cause the following characters to be suitably modified. If the line ends with the concatenation flag (‘`$j`’), the next input line is appended and flag processing continues.

Finally, tab expansion is carried out for each tab in turn, starting from the left. The field between the tab and the next tab (or end of line) is located, and the tab is replaced by a space of zero width, which is then expanded until one of the following occurs:

(a) For a relative tab:

  (1) The leftmost character of the field begins at an L-tab position.

  (2) The rightmost character of the field ends at an R-tab position.

  (3) The central character of the field lies at a C-tab position.

  and the width of the inserted space is greater than or equal to the device resolution (see **minwidth**) except at the start of a line.

(b) For an absolute tab:

  (1) The leftmost character of the field begins at the position specified by an absolute L-tab.

  (2) The rightmost character of the field ends at the position specified by an absolute R-tab.

  (3) The central character of the field lies at the position specified by an absolute C-tab.

(c) For a special tab:

  (1) For ‘`$c`’, the field is centred between the linelength (or templinelength) and the current indent (or tempindent).

  (2) For ‘`$e`’, the rightmost character of the field is in the last character position in the line.

  (3) For ‘`$i`’, the leftmost character of the field is at the start of the current indent (*not* tempindent).

The determination of the centre of fields takes into account the effects of variable width characters where relevant. Attempts to move fields to the left cause errors. The ‘`$e`’ flag must be the last tab on any input line. In fancy mode, the unit of tab expansion is specified by the **minwidth** directive. Lines which are being skipped as a result of one of the conditional directives are not processed for flags or tabs.

If a tab is used in the argument to a macro, it gets expanded when the directive line is processed prior to obeying the macro. However, the fact that the character is a tab is not forgotten, and when the argument is inserted into another line, the tab is re-expanded.

31. Footnotes

The directive

```
.footnote [n]
```

causes the output from processing all subsequent input lines until

```
.endf
```

to be gathered together and printed at the foot of the current page. In fill mode it does not cause a line break in the current text. It can also be used in copy mode, but not from within a contiguous section (see below). Different groups of footnotes may be specified by use of the group number *n*, which defaults to one. If the line length, line spacing or indent settings are changed while processing footnotes, the changed settings are remembered from footnote to footnote, and do not affect normal text. Each group of footnotes has an independent set of these parameters, which can be given at the start of processing by means of a **footnote** directive containing no text lines; this does *not* produce a null footnote on the first page.

Footnotes are separated from the body of the text on a page by a series of lines which are specified by the **fnsep** directive. Different groups of footnotes are also so separated. The separator lines are terminated by **endf**. There is no default, but macro packages normally create one. **fnsep** is treated as a conditional footnote, and it is processed in exactly the same way as other footnotes, but using the current environment.

The input lines comprising footnotes are not saved, but are processed immediately. The resulting *output* is saved until the end of the page is reached. Footnotes are never split over more than one page; a new page is started if a footnote which is too long for the current page is read. Any partially filled main text line is also carried over to the new page. Hence references to footnotes should *immediately precede* the **footnote** directive in the input. The directive **fninline** requests that footnotes not be printed at the bottoms of pages (as just described), but in-line where they fall in the input. This format is commonly used in printer’s copy, and is forced if galley proofs are being produced by specifying a page length of zero.

GCAL’s footnote placement algorithm is not sophisticated, and it can be defeated by a very high density of footnotes. The footnote text is read and accepted for the current page if its length together with one extra line (for the line in which it is referenced) will fit on the current page. Once accepted it cannot be withdrawn. If a subsequent footnote is referred to from the same output line, and if there is no room for it on the current page, it and the output line are moved on to the next page, leaving the previous footnote on the wrong page.

31.1 Saved Footnotes

Sometimes there is a requirement not to print footnotes at the bottom of a page, but instead to save them up till some later point in the document, such as the end of a section or chapter. Any given footnote group can be designated as a saved group by adding the word ‘save’ to a footnote directive. When this is done, the group number is mandatory. For example

```
.footnote 6 save
.endf
```

causes all subsequent footnotes in group six to be saved. It is not possible to undo this option. The saved notes are inserted either at the end of the document, or when the directive **insertnotes** is encountered. They may span many pages, but an individual note is never split over a page boundary. Like **footnote**, **insertnotes** takes an optional numeric argument specifying the footnote group number. This defaults to one.

By the use of several footnote groups, it is possible to have traditional footnotes, section notes, chapter notes and end notes all in the same document. Note, however, that the text of saved notes occupies main store, and GCAL may run out if too many are present.

32. Contiguous Sections

GCAL contains a pair of directives, **contiguous** and **endc**, which ensure that the input material which lies between them appears contiguously, that is, all on one page in the output. The full syntax is

```
.contiguous [n] [inline | bottom]
<input lines>
.endc
```

This facility is normally used for displayed material. When a contiguous section has been read which does not fit on the current page, it is saved until the next page is started. However, a new page is *not* automatically forced, and hence further material may appear on the original page, preceding the contiguous section in the output. When this happens, there is no automatic line break on the original page. However, when the contiguous section *does* fit, there *is* a line break. In other words, the text of the contiguous section itself always starts on a new line.

By default, held-over contiguous sections are printed at the top of the succeeding page. If there is more than one, as many as will fit are printed, the remainder being held over yet again. If the word '`bottom`' is used with the **contiguous** directive, and the section is held over, it is printed at the bottom of the subsequent page. Only one such section is ever printed on one page.

Holding over a contiguous section while continuing on the original page may be acceptable, and indeed desirable, when fairly large figures are being handled as contiguous sections. However, small displays are normally required to appear in sequence with the text. This is achieved by adding the word **inline** to the **contiguous** directive, for example

```
.contiguous inline
.copy
    a*a  + b*b   =   c*c
.fill
.endc
```

If such a section does not fit on the current page, a new page is automatically started, and there is always a line break. The two options, '`inline`' and '`bottom`', are mutually exclusive.

Once a contiguous section has been saved for the next page, any further contiguous sections encountered on the current page are also saved, whatever their size, in the default case. This effect can be avoided by specifying different *group numbers* for contiguous sections, for example

```
.contiguous 1
<text>
.endc
.contiguous 4
<text>
.endc
.contiguous 1
<text>
.endc
```

If the first section is too big for the current page, but the second one is not, the latter is not held over, because it has a different group number. However, the third section is held over, whatever its size.

Contiguous group numbers provide a means of keeping particular sequences of contiguous sections in order. For example, a document containing both figures and tables and equations displayed inline could use three different contiguous groups. (Macros would of course be used hide all this detail.)

When a contiguous section is being processed, the current page is not being altered, and in particular, the system variables **usedonpage** and **leftonpage** do not change. However, the system variable **used**, which is set to zero at the start of a contiguous section, *does* change as the section is processed. The system variable **lastspace** contains a value appropriate to the lines in the contiguous section (starting with a value of zero). If the contiguous section is not output immediately, **lastspace** is reset afterwards to the value it had at the start.

Certain directives are forbidden in contiguous sections. Most are obvious errors (e.g. **newpage**); one which is perhaps less obvious is **footnote**. A GCAL system variable called

incontig is set 'true' during the processing of contiguous sections, and **contigpending** is set 'true' or 'false' according as contiguous material is being held over for the next page or not. When a new page is forced by **ensure**, and this results in the output of pending contiguous sections, **ensure** is obeyed again after they have been output.

At the start of a contiguous section, a copy is made of the current environment (see 'GCAL Environments' below). At the end of the section the previous environment is restored. Contiguous sections may be nested inside other contiguous sections. In this case only the outermost group number and **inline** option are relevant, and the copying and restoring of the environment only occurs at the outermost level. Contiguous sections may also appear in headlines, footlines, footnotes and **aside** text, but have no effect, as these constructions are always contiguous anyway.

33. Page Control

GCAL output normally appears at the left hand side of physical output pages. The directive

.pageoffset *dimension*

can be used to offset to the right. Since this offset applies to the whole of the logical page produced by GCAL, it affects both copied and filled lines. **pageoffset** can also be called with two arguments to specify a vertical and a horizontal offset:

.pageoffset *dimv dimh*

If a vertical offset only is required, a second argument of zero must be given. The vertical page offset does not count as part of the page length (see figure below); thus, for example, if it is specified as 2 for lineprinter output, the page length should be set to 58. Users are exhorted not to use vertical page offsets on lineprinter output except for very good reasons, as it can lead to a lot of paper wastage.

Certain devices, (e.g. laser printers) can print outside the normal margins. For such devices it makes sense to use negative page offsets to achieve special effects, and therefore GCAL permits the arguments to **pageoffset** to be negative. However, because GCAL permits spaces to occur in the middle of expressions, if the second argument to **pageoffset** begins with a minus sign, it must be separated from the first argument by a comma, or a dummy zero must be inserted.

```
.pageoffset 12.0 -12.0    is mis-interpreted as zero
.pageoffset 12.0 0-12.0   works as expected
```

GCAL counts pages as it processes them. The directive

.page *number*

specifies the number of the current page, which is initially set to one, and which is incremented at the end of each page. The system variable **page** holds this value. The total length of a page is specified by the directive

.pagelength *dimension*

A value of zero may be used to obtain galley proof style output. In this case, footnotes always appear in-line (see **fninline**), as do contiguous sections. Galley-style output in fancy mode contains restart points every sixty lines or so, which enable the GTYPE program to start or resume its output part-way through the document. The default page length of 60 lines or 720 points is established at the start of processing, though many standard style definitions change this. Subsequent changes in line depth do not change the page length, measured as an actual length. The lengths of the areas set aside for headlines and footlines are specified by the directives

.headlength *dimension*
.footlength *dimension*

The defaults are both zero (though the standard style definitions change this). Footlines should not be confused with footnotes, which are treated as part of the text on a page. The layout of a physical output page is therefore as in figure 2 below. If the footlength is changed in the middle of a page, it affects the current page. However, changes to headlength do not take effect until the start of the following page.

```
       horizontal
    |<--- page --->|<---linelength--->|
    |    offset    |                  |
    -----------------------------------
            |              |
            |              |
          vertical page offset
            |              |
            V              V
    -----------------------------------
    |       |      |                  |
    |       |      |                  |
    |   headlength |   (headlines)    |
    |       |      |                  |
    |       V      |                  |
    |       ---------------------------
    |              |//////////////////|
    |              |//////////////////|
    |              |//(text of page)//|
pagelength         |//////////////////|
    |              |//////////////////|
    |              |//////////////////|
    |              |///(footnotes)////|
    |       ---------------------------
    |       |      |                  |
    |       |      |                  |
    |   footlength |   (footlines)    |
    |       |      |                  |
    V       V      |                  |
    -----------------------------------
```

Figure 2: Page Layout (not to scale)

GCAL automatically divides up the text into pages, fitting as much on to each page as it can. In addition, the directive

```
.newpage [justify]
```

forces subsequent output to be at the start of a new page. If **justify** is specified, the previous part line (if any) is right-justified. If GCAL is already at the start of a page, **justify** has no effect. In order to generate a blank page it is necessary to specify **space** between two successive **newpage** directives.

33.1 Vertical Spacing

When producing plain output, double-spacing is sometimes required. The directive

`.linespacing` *number*

sets the printed spacing of output lines in units of one line. The default value is one, which gives single spacing. Note that in fancy mode, much more control over the line spacing is possible by using the **linedepth** directive. As this directive is essentially device-specific, it should not itself be used within the source text. Instead, macros with names such as **doublespace** and **singlespace** should be used, as in the standard macro library. These can be defined to use **linespacing** for plain output, and the more flexible **linedepth** for fancy output. A directive whose sole function is to generate vertical space is

`.space [`*dimension*`]`

If the argument is omitted, one blank line is output. If the end of the page is reached, no further space is output. In fill mode **space** forces a new line of output. If the value of the argument is negative, it is treated as zero, and the only effect of **space** is to force a line break. There is a specialized form of **space** for use at the start of contiguous sections. This is the directive **cspace** (conditional space). It behaves as **space** except when it appears at the top of a page, in which

case no space is output. It is intended for use at the start of contiguous sections; if the section is held over and printed at the top of the next page, the space is not output. The test happens at output time, which is what makes it different to a normal **space** enclosed by a conditional test on **usedonpage**.

The system variable **lastspace** is set to contain the sum of the values of the spacing generated by successive **space**, **cspace** and **par** directives, until the next line of text is output, when it is set to zero. This makes it convenient to use constructions such as

```
.space 12.0 - ~%lastspace
```

which outputs 12 points of space if it follows a line of text, and a smaller amount if it follows a previous spacing directive.

The basic facilities of GCAL make it easy to ensure that paragraphs do not begin too near the bottoms of pages. For more complicated items there are no built-in concepts in GCAL, but instead there are more basic facilities which allow users to construct, for example, beginning-of-section handling rules to their own requirements. There are two ways of testing the amount of space left on a page. The simpler directive is

`.ensure` *dimension*

which does nothing if there is at least that much space on the current page, but otherwise forces a new page to be started. Footlines and footnotes that have already been processed are taken into account when computing the space left. **ensure** does not cause a line break; it does make allowance for for a partly filled line which has not yet been output, but it is normally good practice to use **newline** first. If **ensure** causes pending contiguous sections to be output, **ensure** is obeyed again afterwards. Examples of the use of **ensure** can be found in the standard macro library.

33.2 Headlines and Footlines

An arbitrary number of lines may be specified for output at the heads and feet of pages. These may include both directive and text lines, although some directives (e.g. **newpage**) are forbidden. The insert mechanism may be used to include page numbers and the date and time (and any other variables) if required. Special tabs may be used for centring and right aligning. Headlines and footlines are never filled or split, independent of whether GCAL is in copy mode or fill mode.

A line specified as a headline or footline is not processed for flags until it is used. At each use the original line is re-processed. The directive **head** causes the following lines of input up to **endh** to be remembered and processed at the head of each page. The output appears in an area whose size is specified by **headlength** (see above). If too much output for the headline area is generated as a result of processing the lines given in the **head** directive, only the top part is output; if too little is generated, blank space is added at the end.

At the start of processing headlines or footlines, GCAL is set to interpret flags. The **noflags** |
directive may appear in heads and feet, but at the end of processing, the external state is restored. |
The processing of headlines takes place when it is determined that there is further output to be produced on the page. This is not necessarily immediately following the end of the previous page. For example, if a sequence such as

```
.newpage
.headlength 2
.head
<some text>
.endh
First line of new page
```

is encountered, the top of page processing does not occur until ‘First line of new page’ has been read, and so the new headlength and headlines are used. The following are the precise conditions that cause GCAL to proceed with headline processing:

(a) If one or more contiguous sections are held over from the previous page;

(b) If any output text is encountered, whether or not it is within a contiguous section or footnote;

(c) If the **space** directive is obeyed – this means that the directive

```
.space 0
```

can be used to force headline processing without any other effect.

Note that a **space** directive that forces the end of a page does *not* carry over to the top of the next page, and that unless any output text is actually generated, a **footnote**/**endf** sequence does not immediately trigger headline processing.

The directive **foot** is similar to **head**, but is terminated by **endf**. It specifies text that is to be output in the footline area of each page, starting with the current page. Only the bottom lines of the text are output if too many are generated, while preceding blank space is inserted if there are too few. For example

```
.foot
GCAL $c [~%page] $e ~%DATE
.endf
```

has been used for the footlines in this document. There are no default footlines provided by the GCAL program, but the standard macro library does set one up. The default typographic and layout parameters for headlines and footlines are those in force when the **head** or **foot** directive is obeyed. However, they may be changed within the **head** and **foot** sequences, and are remembered from call to call (see ‘GCAL Environments’ below).

Different headlines and footlines for left and right hand pages may be produced by testing the page number in the headline or footline sections (see **if**, **odd** and **even**), and ‘running’ headlines and footlines can be produced by using variables to hold the text strings involved.

34. Marginal Notes

The directive

`.margin` *dimension*

specifies a margin position which should be greater than the maximum line length specification in the source. This is used as a position for the output of marginal notes, which are specified by the directive

`.note` *text*

Marginal notes may appear in footnotes and other contiguous sections as well as in the main output. Each note is associated with the text in which it was generated, and is output in the margin of the next available relevant output line; for example, a note in a contiguous section which gets pushed over to the next page is pushed over with it.

35. Indexing

The directive

`.index` *text*

causes the text following the directive name to be written to the `INDEX` file. This data is intended for post processing by a sorting program. For example

```
An early pioneer was Charles Babbage,
.index Babbage, Charles ~%page
who invented the differential engine.
```

36. Subscripts and Superscripts

Subscripts and superscripts are available only in fancy mode. The default depth for subscripts, and height for superscripts, can be set by

`.subdepth` *dimension*
`.superheight` *dimension*

The dimensions are always in points. In the absence of these directives, both values default to

four points. Five GCAL flags are provided for raising and lowering the characters on a line.

The subscript flag (default ‘`$-`’) causes the following characters to be lowered by an amount previously specified by the **subdepth** directive; alternatively, an explicit depth may be given by using the ‘down’ flag (default ‘`$d`’) followed by a number, which specifies the depth in points. If the ‘down’ flag is followed by an asterisk, the number which follows is taken as a fraction of the current font’s ‘design size’, which is the number held in the system variable **fontsize**. Thus, if a 12-point font is in use

`$d6`	means ‘go down 6 points’
`$d6.0`	also means ‘go down 6 points’
`$d*0.5`	means ‘go down by half the font size’, which in this case is also 6 points

Characters remain lowered until the end of the input line, or until raised by another flag.

The superscript flag (default ‘`$+`’) is the converse of the subscript flag, and its distance can be set by the **superheight** directive, which itself has the same default as **subdepth**. There is also an ‘up’ flag (default ‘`$u`’), which takes a numeric argument giving the height, either as an asbolute distance, or as a fraction of the current font’s size, as for the ‘down’ flag. If no argument is given for ‘`$u`’ or ‘`$d`’, they behave as ‘`$+`’ and ‘`$-`’ respectively. Note that ‘`$+`’ and ‘`$-`’ are only inverse operations if **subdepth** and **superheight** specify the same distance.

The level flag (default ‘`$L`’) can be used to restore characters to the base level of the line following subscripting or superscripting. Thus, to represent ‘A squared’, for example, the input might be

```
A$+2$L
```

Sub-subscripts and super-superscripts are allowed in any combination. As well as their traditional use, the subscript and superscript flags are useful for positioning diacritical characters, as described in the next section.

37. Overprints and Accents

Three GCAL flags are available to provide overprinting facilities. The overprint flag (default ‘`$o`’) causes the character(s) which follow it to overprint the preceding character; the character widths, fonts and vertical position on the line may differ.

When overprinting characters of differing widths, it is useful to be able to return the current pointer to a known position. The high water mark flag (‘`$h`’) permits this. It must always appear in pairs in a line: the first occurrence marks a ‘high water mark’ position, while the second moves the current pointer back to that position. The movement may be to the left or to the right. For example, if a sequence such as

```
A$O.
```

occurs, the ‘A’ is overprinted with a full stop, but the current pointer is moved by the width of the stop, which is typically much narrower than the letter. If

```
A$h$O.$h
```

is used instead, then any characters that follow are printed in the correct position following the ‘A’.

In fancy mode only, the width flag (default ‘`$w`’) can be used to set a width for an individual character, the character which *precedes* it. This character may be a printing character or any kind of space. The ‘width’ of a character in this sense is the amount by which the current position on the line is advanced after printing the character. However, it is not possible to change the width of any printing character (as specified in the font library), so the required effect of ‘`$w`’ is implemented by generating an exact space with an appropriate compensating width, either positive or negative. The flag is followed by a number specifying the width in points; the default is zero, which is precisely equivalent to ‘`$o`’. The number may or may not contain a decimal point – either way, it is interpreted as a number of printers’ points. For example, supposing the normal character width were 6, then

```
formula$W3 e
```

might be used to output the letter ‘e’ close to the letter ‘a’. Another use of ‘$o’ and ‘$w’ is in the production of accented characters. For example, suppose a letter ‘A’ with a dot above it is required, and suppose that the widths of ‘A’ and a full stop are 6 and 3.6 points respectively. What is needed is

```
A$W1.2 $U7.5 .$W4.8 $L
```

An A with two dots above it could be produced by

```
A$O $U7.5 .$W3 .$W3 $L
```

It is usually most convenient to set up commonly used sequences as user flags. Note that some laser printers treat accented letters as independent characters.

The width flag may be used with negative widths, to move the current point backwards after printing a letter. Spaces with negative widths are also permitted. If the argument is preceded by an asterisk, the number is taken as a multiplier to be applied to the width of the preceding character, rather than an absolute dimension in points. For example

`$w6`	means ‘width 6 points’
`$w6.0`	means ‘width 6 points’
`$w-6`	means ‘width -6 points’
`$w*0.5`	means ‘width half of what it was’

Negative fractions are permitted. The fractional width facilities make it easier to construct flags that work in fonts of different sizes.

The width flag can also be used to access the widths of the space characters in the current font.

```
$w#    specifies the width of an exact space
$w>    specifies the width of a normal space
$w<    specifies the width of a thin space
```

These widths can be applied to spaces or to printing characters. Finally, ‘$w’ can access the width of any printing character in the font if it is followed by a single quote and the character. Thus

```
A$w’e
```

outputs the letter ‘A’, but the current point is moved by the width of the letter ‘e’.

38. Ligatures

Printers with typographic character sets may have special forms of joined-up characters for sequences such as ‘ff’, ‘fi’ and ‘fl’. These are known as *ligatures*. GCAL supports ligatures as attributes of individual fonts, and they are defined in the font tables (see appendix B). The relevant character sequences are automatically converted by GCAL. To prevent an individual sequence from being treated as a ligature, the null flag can be used, for example

```
shelf$$ful
```

There is also a directive **noligatures** which suppresses the use of ligatures entirely. The default state can be restored by the use of **ligatures**.

In many typographic fonts the en-dash and em-dash characters are set up as ligatures for two and three successive minus characters respectively. When GCAL is outputting index lines, comment lines, or error messages, the text is ‘de-ligatured’.

39. Font Changes

In fancy mode, up to 255 different character fonts may be used, numbered 0–254. The current font is specified by the directive

`.font` *number*

or by the font flag (default ‘$f’) followed by a font number. Each time the font is changed explicitly, the previous font is stacked on a ‘forgetful’ stack of depth 12, and can be retrieved by

the use of the font flag without a font number. This is useful when defining flags for accessing special characters in special fonts. There is a separate stack for each GCAL environment. The default font is font zero. Thus the input lines

```
.font 3
ABD$f2 DEF $f
XYZ
```

specify that ABC and XYZ are in font three, while DEF is in font two. Since particular fonts are device-specific, the use of ‘$f’ should be confined to the definition of generic font-changing flags, such as ‘$rm’, etc. as set up in the standard styles. It should not normally appear in the main input.

It is sometimes useful to deal with fonts in groups. For example, different groups each containing a roman, italic and bold font, but at different point sizes, might be used for different parts of a document. It is convenient to be able to define the same flags (e.g. ‘$rm’, ‘$it’) to switch fonts, whichever point size is in use. GCAL provides an indirect font selection mechanism via font groups. The directive

`.fontgroup` *number f0 f1 ...*

defines a group with the given number, containing the given fonts. There is a current font group, selected by a call to **fontgroup** with only a single argument (the group number). There is also a flag, ‘$fg’, which can be used to select a new current font group. Font groups are stacked, like fonts, and a previous group can be retrieved by the use of the font group flag or the **fontgroup** directive without a group number. There is no default font group. A font from within the current group is specified by the directive

`.gfont` *number*

or by the group font directive, ‘$gf’. Thus, for example,

```
.fontgroup 4 8 99 123 6 0
.fontgroup 4
.gfont 1
ABC $gf3 DEF $f
```

specifies that ‘ABC’ is in font 99 and ‘DEF’ in font 6. Fonts selected via the group mechanism behave in exactly the same way as fonts selected directly. Any one font may belong to any number of different groups.

All font changes in the copy persist from line to line; a font, once selected, remains current until it is changed. However, any font changes that occur as the result of flags in a directive line are local to that line only.

Note that a change of font group does *not* automatically change the current font (which might not have been selected via a font group).

40. Additional Input Datasets

Two directives are provided for including additional input datasets at specified points in the main input. The directive

`.include` *string*

treats the string as the name of a file whose contents replace the directive line. Under Phoenix, the string must be a ddname which has been set up by the DD argument to the GCAL command, or has been set up prior to calling GCAL. For example

```
GCAL %H! DD=EXTRA=.MYFILE
<start of main input>
. insert EXTRA at this point
.INCLUDE "EXTRA"
<rest of GCAL input>
!
```

The alternative directive,

```
.library string
```

treats the string as the name of a member of a library of input texts. Under Phoenix this must be the name of a member of the partitioned dataset with ddname `LIBRARY`. The Phoenix command provides a keyword for setting this up. For example,

```
GCAL %H! LIBRARY .MYGLIB
<start of main input>
. insert LIBRARY member MYMACS at this point
.LIBRARY "MYMACS"
<rest of input>
!
```

The Phoenix command `GCAL` sets up the standard library (`PUBLIC.GCAL.STYLE`) in addition to any dataset given on the command line.

The directives **include** and **library** may appear inside macros as well as in the main input. The effect is to read the dataset each time the macro is obeyed. The included lines are treated as part of the macro, and hence may contain references to macro arguments.

41. GCAL Environments

The word 'environment' in GCAL is used to refer to the set of parameters which control the program's behaviour. The set contains those parameters whose values may be specified by the following directives:

```
autopar      autospace    copy           emphasize
fill         fninline     font           fontgroup
hyphen       justify      indent         linedepth
linelength   linespacing  margin         minpar
parindent    parspace     sentencespace  subdepth
superheight  tabset       tempindent     templinelength
umode
```

together with the underlining, emboldening, and upper- or lower-casing states. Separate environments are maintained for the main text, each footnote group, headlines, footlines, and aside sections. A copy of the current environment is taken at the start of each contiguous section, and the previous environment is restored at the end. This prevents anomalous effects when contiguous sections are held over until the start of the next page. (Strictly, this applies only to top-level contiguous sections in the main text. **contiguous** and **endc** directives which are nested in other contiguous constructions are ignored.)

At the start of a run, only the main environment exists; when one of the others is first required, a copy of the current environment is taken. This is then preserved for subsequent re-use *except* in the case of headlines, footlines and contiguous sections, where a new copy is taken each time a **head**, **foot**, or **contiguous** directive is obeyed.

On changing from one environment to an inner environment (e.g. obeying a **footnote** directive), the temporary indent and linelength values are reset. On returning to the outer environment they are restored to their previous values.

42. Specialized Device Control

Two directives are provided in fancy mode for inserting short, arbitrary control strings into the GCODE output. These are intended for passing device-specific information to the program that interprets the GCODE and drives the output device. The two directives are

```
.request "string"
.control "string"
```

The strings are inserted into the GCODE bracketed by the 'application comment' and 'device control' control sequences respectively. Neither of these directives should ever be used directly. They should only appear inside macros designed to set up GCAL to produce output for particular devices. Because they interact with the program that interprets the GCODE, such macros cannot be written without knowledge of its actions.

The argument for **request** is intended to contain commands to be obeyed by the program interpreting the GCODE. It is conventional to start the string with the name of the output format for which the command is relevant. For example

```
.request "fx80: pagelength 13"
```

Such commands can then be ignored when output is being sent to some other destination, such as a screen. The argument for **control** is intended to contain a control string to be passed directly to the printing device. Obviously, this is highly device-specific.

A third directive, **longcontrol**, is provided for including larger sections of device-specific text in a GCAL file. This is appropriate, for example, for including a diagram coded in PostScript into a file that is to be printed on an Apple LaserWriter. All input lines between **longcontrol** and **endc** are treated as though they were the arguments to a succession of **control** directives. GCAL's flag scanning mechanisms are still in operation, but can be disabled by the use of **noflags** if necessary.

Note that GCAL does *not* automatically add a 'newline' character to each line of text processed in this way. Newlines can be included (as they can for **control**) by means of the escape sequence '`\N`'. However, in the case of PostScript text it is usually easier to begin each line with a space. The standard GCAL A4ATL style contains a **picture** macro which uses this mechanism.

43. Miscellaneous

A number of directives that do not fit naturally into any of the other sections are described here.

```
.closefontlib
```

When the **bindfont** directive has been obeyed, the file containing the font information is left open, so that a subsequent **bindfont** can continue reading it and look for the new font. More | details of this process are given in Appendix B. The **closefontlib** directive forces the font file to | be closed. It is useful on some systems that only permit a small number of files to be open simultaneously.

```
.emphasize [dimension] ["string"]
```

This directive causes each line of output, other than head or foot lines, which contains any input text from subsequent lines of input, to be emphasized by the characters in the string. These are output starting at the given position, which must be greater than the line length. The directive **noemphasize** cancels the effect of **emphasize**. There are also two flags, '`$se`' and '`$pe`', to start and stop emphasizing, but it is not possible to use them to set up the emphasis string or position. There is no default string or column for emphasizing. Therefore, the first time that **emphasize** is used, both arguments should be given. Thereafter, **emphasize** with no arguments can be used to turn emphasizing on, using the previously established values, or **emphasize** can be used with only one argument, to change the position without changing the string. It is common practice to include a sequence such as

```
.emphasize 460.0 "|"
.noemphasize
```

at the head of a document, or in a setting-up macro. This sets the emphasizing parameters without actually generating any emphasized text.

```
.comment text
```

This directive causes the text to be output to the verification output. This is useful for generating reminders about special characters and figures that are to be inserted by hand.

```
.aside
```

This causes lines of text which follow, up to **enda**, to be sent to the `ASIDE` file instead of to the main output. Line filling and justification occur if requested, but no pagination is performed, that is, the output is all on one page as far as GCAL is concerned. **aside** does not cause a line break in the main output, but **enda** causes one in the aside output. While processing aside text, certain directives (e.g. **newpage**) are forbidden. Aside sections may be generated from within footnotes and contiguous sections, but not from heads and feet. The system variable **usedonpage** does not alter while aside output is being processed, but **used** does. In addition **lastspace** is maintained in accordance with the aside output, starting from zero for each aside section. It is reset following **enda**.

`.resetlinenumber` *number*

This directive resets the line number kept by GCAL and used in error messages to identify the point of error. It is useful when several datasets are being concatenated as input, and normally appears (with argument zero) as the last line of macro library files. If the argument is omitted it defaults to one.

`.usebs`

This directive is only of use in plain mode. It causes GCAL to make use of the ‘backspace’ character for overprinting and underlining. By default, GCAL uses ‘carriage return’ for this purpose, effectively making more than one pass over any output line that requires it.

`.usecr`

This directive cancels the effect of **usebs**. It causes a return to the use of ‘carriage return’ for overprinting and underlining in plain mode.

`.goto` *dimension*

This directive causes GCAL to move to a given absolute vertical position on the page. The dimension is an offset from the top of the page. It is only permitted to move downwards on the page by this means. Like its counterpart in programming languages, **goto** is considered harmful by some people, and it should certainly only be used with great care.

`.up` *dimension*

This directive is available only in fancy mode. It causes GCAL to move back up the page, and acts like **space** with a negative argument (which is ignored). It can be used in open text or in |
footnotes or contiguous sections, but it is not possible to move back past the top of the current |
page, footnote or contiguous section (whichever is relevant). This directive should be used with |
caution.

44. Errors

There are about ninety different messages that can be output by GCAL. Some of these are merely comments which do not affect the return code. There are four types of error, which are, in order of seriousness:

(1) Warnings: These give a return code of 4, and include conditions such as a copied line longer than the current line length. If more than 10 warnings are given, subsequent ones are suppressed (with a message to say so).

(2) Errors: These give a return code of 8. If more than 20 errors occur, GCAL is abandoned.

(3) Serious errors: These are mostly failures to open mandatory datasets. They give a return code of 12, and cause GCAL to abort immediately. The absence of `INDEX` or `ASIDE` is not treated as a serious error.

(4) Disasters: These give a return code of 16 and cause GCAL to abort immediately. They include the case of insufficient store and certain internal errors.

Certain repetitive conditions, for example, ‘index dataset missing’ cause only a single message to be output, the first time they occur.

*   *   *

Appendix A: GCAL Directive List

Abbreviation ↓	*Name*	*Use*	*Section*
	aside	switch to **aside** output	(43)
	autopar	automatic paragraphing	(20)
	autospace	automatic space handling	(22)
	bindfont *n string mag*	bind font definition	(15.2)
	cancelflag	cancel a flag	(13)
	closefontlib	close font library file	(43)
	comment *text*	output a comment	(43)
cont	**contiguous**	force contiguity	(32)
	control *string*	output control string	(42)
	copy	enter copy mode	(10)
	cspace	conditional space	(33.1)
	dflag *flag flag*	set directive flag(s)	(11)
	elif *condition*	follows **if** or **unless**	(18)
	else *condition*	follows **if** or **unless**	(18)
	elul *condition*	follows **if** or **unless**	(18)
em	**emphasize** *n* [*string*]	emphasize following lines	(43)
	enda	end **aside** section	(43)
	endc	end contiguous section	(32)
		end longcontrol	(42)
	endf	end footnote	(31)
		end footlines	(33.2)
	endh	end headlines	(33.2)
fi	**endif**	end **if**	(18)
	endm	end macro definition	(17)
	ensure *d*	ensure space on page	(33.1)
	fancy [*units*] [*string*]	set fancy mode	(15)
	fill	enter fill mode	(10)
	flag *flag string string*	define user flag	(12)
	flag *name flag*	define system flag	(11)
	flags	enable flags	(14)
	fninline	footnotes inline	(31)
	fnsep	footnote separator	(31)
	font *n*	change font	(39)
	fontgroup *n f0 f1 ...*	define font group	(39)
	fontgroup *n*	select font group	(39)
	foot	define footlines	(33.2)
	footlength *d*	reserve space for footlines	(33)
fn	**footnote** [*n*[**save**]]	start footnote	(31)
	format *name format*	define number format	(16.4)
	gfont *n*	change font via group	(39)
	goto *d*	set absolute place on page	(43)
	head	define headlines	(33.2)
	headlength *d*	reserve space for headlines	(33)
	hyphen *level*	set hyphenation	(21)
	if *condition*	conditional	(18)
	ignoreflagspace	ignore spaces after flags	(11)
	include *string*	include dataset	(40)

in	**indent** *d* [**nobreak**]	set indent	(25)
ix	**index** *text*	index rest of line	(35)
	insertnotes [*n*]	insert save footnotes	
	justify	right-justify lines	(19)
	library *string*	include library dataset	(40)
	ligatures	allow ligatures	(38)
ld	**linedepth** *d*	define line depth	(15)
ll	**linelength** *d [nobreak]*	set line length	(25)
ls	**linespacing** *n*	set line spacing	(33.1)
	longcontrol	start many **control** lines	(42)
	macro *name arg1 .. argn*	define macro	(17)
	margin *d*	set margin for notes	(34)
	mindepth *d*	set vertical resolution	(15.1)
	minpar *n*	minimum paragraph lines	(20)
	minwidth *d*	set horizontal resolution	(15.1)
nl	**newline** [**justify**]	force new line	(26)
np	**newpage** [**justify**]	force new page	(33)
par	**newpar** [**justify**]	force new paragraph	(20)
	noautopar	no automatic paragraphs	(20)
	noautospace	no automatic space handling	(22)
nem	**noemphasize**	cancel emphasis	(43)
	noflags	disable flags	(14)
	noignoreflagspace	use spaces after flags	(11)
	nojustify	no right-justification	(19)
	noligatures	forbid ligatures	(38)
	nosep	join with no separator	(26)
	note *text*	output in margin	(34)
	page *n*	set page number	(33)
pl	**pagelength** *d*	set page length	(33)
	pageoffset *d* [*d*]	set page offsets	(33)
	par [**justify**]	new paragraph	(20)
	parindent *d*	paragraph indent	(20)
	parspace *d*	paragraph spacing	(20)
	request *string*	output application string	(42)
	resetlinenumber *n*	reset input line number	(43)
	sentencespace *n*	sentence spaces	(24)
	set *name value*	set variable	(16.1)
	set *name @ string*	set indirect variable	(16.5)
	sflag *flag string*	) as **flag**, but directive	(12)
	sflag *name string*	) is itself scanned for flags	(11)
sp	**space** *d*	leave space on page	(33.1)
	subdepth *d*	set subscript depth	(36)
	superheight *d*	set superscript height	(36)
	tabset *d1 d2 ...*	define tab settings	(29)
ti	**tempindent** *d* [*m*] [**nobreak**]	one line indent	(25)
tl	**templinelength** *d* [*m*] [**nobreak**]	temporary line length	(25)
	umode *n*	set underline mode	(28.1)
	unless *condition*	conditional	(18)
	up *d*	negative **space**	(43)
	usebs	use backspace (plain output)	(43)
	usecr	use carriage return	(43)

Appendix B: Format of Font Libraries

When GCAL is operating in fancy mode it 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 normally consists of several separate font files, one for each kind of output device supported. The format of an individual font file is described in this appendix.

GCAL 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. Devices differ a lot in this respect. For example, although daisy-wheel printers have only one (or sometimes two) sets of characters, the character spacing is not fixed, and can very easily be altered. Thus several different 'fonts' which differ only in their character spacing can be used. The program driving a daisy-wheel printer is required to set up the character spacing, so in this case, arbitrary values may be set in the GCAL font file, and these will be used by the printer driver.

On the other hand, devices such as the Apple LaserWriter contain fonts where the spacing of each character is specified individually. Although it is in theory possible to adjust these spacings, it is neither very easy nor recommended on typographic grounds. The program that converts the output of GCAL into PostScript for sending to a LaserWriter has no facilities for changing character widths, and does not even look at the font library. Its contents must therefore correspond with the spacings built into the printer, and should not be changed.

In MVS, font files are held as members of partitioned datasets, concatenated as necessary when calling GCAL. The system font libraries have a record format of VB, and any concatenated libraries must also use this format if the concatenation is to work properly.

Each font file can contain descriptions of several fonts. The data for any one font is contained between a line containing only an exclamation mark, and a line containing the word 'end'. Any characters occurring between fonts are ignored, and can be used for descriptive comments. For systems where it is possible using standard BCPL library calls to skip to any given byte in a sequential file, font library processing can be speeded up considerably by the use of an index at the start of a font file. This masquerades as an individual font definition with the name '`$$Index$$`'. It is ignored on systems such as MVS that cannot use it. Thus the general layout of a font file is

> *<introductory comments>*
> `!`
> *<data for index>*
> `end`
> *<more comments>*
> `!`
> *<data for first font>*
> `end`
> *<more comments>*
> `!`
> *<data for second font>*
> `end`
>
> *etc*

where the index data is optional. Whether GCAL (and for that matter, GTYPE) use the index or not depends on the implementation for a particular operating system. The two modes of operation are as follows:

(a) **Non-indexing**: On obeying the first **bindfont** directive, GCAL reads serially through the file to find the relevant font. For subsequent fonts, it carries on reading from where it left off, and only closes and re-opens the file if necessary. It is therefore more efficient to specify **bindfont** directives in the same order as the fonts appear in the font file if possible.

(b) **Indexing**: On obeying the first **bindfont** directive, GCAL reads serially through the file till it reaches the first font definition. If this is for a font named '`$$Index$$`', it reads index data in the format described below and saves it in store. It then uses this data for the current and all subsequent calls to **bindfont** (for the same file) to jump directly to the start of the required font. If the first font does not have the conventional index name, GCAL proceeds as in (a) above.

The GCAL directive **closefontlib** can be used to force the closure of an open font library file. When a font is bound more than once (usually at different magnifications), the character width information is taken from the original information in store, and the font file is not re-read.

The format for the special index 'font' is now described. Following the line containing the exclamation mark, there is a line starting with the word **font**, followed by the conventional index name in quotes. Each subsequent line, up to one starting with the word **end**, consists of a string in quotes, followed by a decimal number. The string must be a font name, and the decimal number is the byte offset in the file at which the data for that font begins. The strings should be in alphabetical order, though GCAL does not at present depend on this. For example

```
!
FONT "$$Index$$"
"Courier" 63920
"Courier-Bold" 64234
"Helvetica" 35730
"Symbol" 64557
"Times-Roman" 986
END
```

Each line of the data for an individual 'real' font begins with a keyword. Some keywords are optional, but the overall order of the data is fixed. The first line following the exclamation mark is of the form

```
font "name"
```

This identifies the font within the font file. Next there may be any number of **request** lines (including none). Their syntax is

```
request "string"
```

and their purpose is to pass information to the program that interprets the GCODE generated by GCAL. One such program is GTYPE, and its convention is that a request string starts with a device type name, terminated by a colon. The rest of the string is interpreted as a GTYPE command, with any appearances of the character '#' replaced by the font number. Here are two examples of **request** lines, taken from fonts for two different devices:

```
request "diablo: font # ecs1"
request "fx80: font # 0A 32"
```

The first requests that this font use the first extended character set, while the second requests an alternate character set whose descriptor value is 32.

Following the **request** lines, if any, there is a mandatory line defining the 'design size' of the font, using the keyword **dsize**. All subsequent dimensions are relative to this size, which is given in *points* and which may have a decimal fraction. Next there are three lines defining the space characteristics, only the first of which is mandatory. The keywords are **space**, **thinspace** and **exactspace**. The only relative dimension that need be given is **space**, since **exactspace** defaults to this, and all the other widths default to **exactspace** unless specified separately.

A simple font for a fixed-pitch device need use no more than the keywords already described. Here is an example for a Diablo printer, used in 'typewriter mode':

```
!
font      "elite-tt"
dsize     12.00
space      0.50
end
```

Following the definition of the space widths, an optional **hyphen** keyword may appear. It takes a single argument, which may be a character number or a character value in quotes, and it defines which character in the font prints as a hyhpen. The default value is character number 45, the minus character in the Ascii encoding.

The remaining information is all optional, and there are three keywords that can appear: **ligature**, **kern** and **width**. The first two can be intermixed in any combination, but all occurrences of **width** must be together, and must immediately precede **end**. The format of the **ligature** lines is

```
ligature char1 char2 char3 [char4 char5]*
```

where *char1* is the first character of a pair, to be replaced by *char3* if followed by *char2*, and by *char4* if followed by *char5*, and so on. The characters may either be given as literals, in quotes, or as a character number in the font (normally Ascii encoding). For example,

```
ligature "f" "l" 130 "i" 129 "f" 128
```

specifies that 'f' followed by 'l' is to be replaced by character number 130, 'f' followed by 'i' is to be replaced by character number 129, and so on. Care must be taken if any ligatures involve the minus character. GCAL converts minus characters in the source into the character specified by the **hyphen** entry in the font before it searches for ligatures. Therefore such ligatures should be specified in terms of the hyphen character and not the minus sign. For example

```
hyphen 147
ligature 147 147 176
```

It is also possible to specify that, if two particular characters occur in sequence, then the last of them is *not* to be considered as the first character of a subsequent ligature. This is done by the use of an asterisk after the two letters. For example

```
ligature "f" "f" *
```

is used for the Adobe Times-Roman font to prevent the sequences '`ffi`' and '`ffl`' being set as a single 'f' followed by a ligatured character, because this looks bad.

In a similar way to **ligature**, the **kern** keyword is followed by a 'base' character, then an arbitrary number of character-dimension pairs. For example,

```
kern "Z" "i" 0.024
```

specifies that whenever 'Z' is followed by 'i', they are to be moved apart by 0.024 times the design size of the font. Negative numbers can be used to specify that characters should be moved closer together. The kerning information is not at present used by GCAL. It is hoped to make use of it in a future release.

The **width** lines, which must be the last information for an individual font, can be specified in several different formats.

```
width string  dimension
```

specifies that all the characters in the string have the same size. The dimension is a fraction of the design size, and may have up to three decimal places.

```
width string dimension dimension ...
```

specifies individual widths for each character in the string. When more than one dimension is given, there must be exactly as many as characters in the string.

```
width char-number dimension
```

specifies the width of a single character, while

```
width char-number–char-number dimension
width char-number-char-number dimension dimension ...
```

specify the widths of a range of characters. Some examples of the use of **width** are

```
width "!()" 0.312
width 91 0.432
width 48-57 0.5
width 93-94 0.456 0.456
```

As many **width** lines as necessary may be used, and the different formats can be freely intermixed. However, it is advisable to choose formats which lend themselves to proof-reading for the particular font in question.

Here is an example of a font definition for a Diablo printer that specifies a narrower width for some characters, and spaces that are thinner than the characters:

```
font       "elite"
dsize     12.00
space      0.40
thinspace  0.10
exactspace 0.50
width      ".,;:!’|" 0.30
end
```

The final example is a font for an Apple LaserWriter:

```
!
FONT "Times-Roman"
DSIZE 1
SPACE 0.360
THINSPACE 0.1
EXACTSPACE 0.600
HYPHEN 45
LIGATURE "f" "l" 175 "i" 174
LIGATURE "f" "f" *
LIGATURE "-" "-" 177
LIGATURE 177 "-" 208
LIGATURE "’" "’" 186
LIGATURE "‘" "‘" 170
WIDTH 0-7     0.722 0.722 0.722 0.722 0.722 0.722 0.667 0.611
WIDTH 8-15    0.611 0.611 0.611 0.333 0.333 0.333 0.333 0.722
WIDTH 16-23   0.722 0.722 0.722 0.722 0.722 0.556 0.722 0.722
WIDTH 24-31   0.722 0.722 0.722 0.611 0.250 0.250 0.250 0.250
WIDTH 32-39   0.250 0.333 0.408 0.500 0.500 0.833 0.778 0.333
WIDTH 40-47   0.333 0.333 0.500 0.564 0.250 0.333 0.250 0.278
WIDTH 48-55   0.500 0.500 0.500 0.500 0.500 0.500 0.500 0.500
WIDTH 56-63   0.500 0.500 0.278 0.278 0.564 0.564 0.564 0.444
WIDTH 64-71   0.921 0.722 0.667 0.667 0.722 0.611 0.556 0.722
WIDTH 72-79   0.722 0.333 0.389 0.722 0.611 0.889 0.722 0.722
WIDTH 80-87   0.556 0.722 0.667 0.556 0.611 0.722 0.722 0.944
WIDTH 88-95   0.722 0.722 0.611 0.333 0.278 0.333 0.469 0.500
WIDTH 96-103  0.333 0.444 0.500 0.444 0.500 0.444 0.333 0.500
WIDTH 104-111 0.500 0.278 0.278 0.500 0.278 0.778 0.500 0.500
WIDTH 112-119 0.500 0.500 0.333 0.389 0.278 0.500 0.500 0.722
WIDTH 120-127 0.500 0.500 0.444 0.480 0.200 0.480 0.541 0.250
WIDTH 128-135 0.444 0.444 0.444 0.444 0.444 0.444 0.444 0.444
WIDTH 136-143 0.444 0.444 0.444 0.278 0.278 0.278 0.278 0.500
WIDTH 144-151 0.500 0.500 0.500 0.500 0.500 0.389 0.500 0.500
WIDTH 152-159 0.500 0.500 0.500 0.444 0.250 0.250 0.250 0.250
WIDTH 160-167 0.250 0.333 0.500 0.500 0.167 0.500 0.500 0.500
WIDTH 168-175 0.500 0.180 0.444 0.500 0.333 0.333 0.556 0.556
WIDTH 176-183 0.250 0.500 0.500 0.500 0.250 0.250 0.453 0.350
WIDTH 184-191 0.333 0.444 0.444 0.500 1.000 1.000 0.250 0.444
WIDTH 192-199 0.250 0.333 0.333 0.333 0.333 0.333 0.333 0.333
WIDTH 200-207 0.333 0.250 0.333 0.333 0.250 0.333 0.333 0.333
WIDTH 208-215 1.000 0.250 0.250 0.250 0.250 0.250 0.250 0.250
WIDTH 216-223 0.250 0.250 0.250 0.250 0.250 0.250 0.250 0.250
WIDTH 224-231 0.250 0.889 0.250 0.276 0.250 0.250 0.250 0.250
WIDTH 232-239 0.611 0.722 0.889 0.310 0.250 0.250 0.250 0.250
WIDTH 240-247 0.250 0.667 0.250 0.250 0.250 0.278 0.250 0.250
WIDTH 248-255 0.278 0.500 0.722 0.500 0.250 0.250 0.250 0.250
END
```

Appendix C: Obsolete Facilities

A number of directives that were used in previous versions of GCAL are still accepted by the current program, though their use is not recommended, and they may be withdrawn at some future date. Originally there was a separate directive for setting the character string for each system flag, before the generalization of the **flag** directive. Those that remain from before this change are

```
atab         bareinsert    bflag          cflag
ctab         downup        eosflag        etab
exactspace   fflag         incinsert      insert
itab         nosplitspace  nostretchspace null
oflag        sysinsert     subsuper       tab
ucflag       uflag
```

GCAL also used to reset the underline state, the emboldening state, the uppercasing state and the font at the end of each input line. The directive **oldgcal** can be used to request this treatment to be re-instated. It is not guaranteed to work with the current macro libraries, and should be used only as a means of processing old-style GCAL input.

* * *

Appendix D: GCODE

GCAL produces two kinds of output: plain and fancy. The former is intended for lineprinters and similar fixed character width devices and is an ordinary sequential file; the latter is in a device-independent code called GCODE. This is interpreted by GTYPE and other suitable programs to produce output for driving particular devices.

1. General Form of GCODE

GCODE is a character stream code; record boundaries are irrelevant and ignored. It is recommended that there be no more than 72 characters per line (record), 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.

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.

Following a formfeed or pseudo-formfeed, GCAL always outputs the current line depth, whether or not it has changed. Hence is is always possible to restart the processing of a GCODE file from such a point.

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. This is the ‘current point’ at the start of a new page. All movements are relative to the current point.

3. Control Sequence Names

Two kinds of control sequence are used in GCODE, and the character strings involved are given names to facilitate discussion. The fixed-length control sequences consist of a backslash character followed by one other character. Two of these sequences (`DCS` and `APC`) mark the beginning of instances of the second kind of control sequence, which is of varying length. The fixed-length control sequences are listed in table 1 below. Note that, as record boundaries are not significant, it is possible for a newline to appear between the backslash and the following character.

mnemonic	*value*	*name*
`NUL`	`\P`	Pseudo-formfeed
`FF`	`\F`	Formfeed
`DCS`	`\D`	Device control string
`NL`	`\N`	Newline
`BS`	`\B`	Backspace
`APC`	`\A`	Application comment
	`\\`	Printing backslash

Table 1: Fixed-length control sequences

A number of different control sequences are followed by an argument which is a decimal number, possibly containing a decimal point. The argument is terminated by one of a number of special characters which determine the identity of the control sequence. These characters are named in table 2 below. Thus, for example, the control sequence ‘`LPR 23.7`’ is encoded as ‘`\23.7<`’.

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 ‘3’. This is the only use of backspace (or any other non-printing character) in GCODE and it enables concatenations of GCODE files to be processed correctly. 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. GCAL puts the text ‘`GCODE file written at <time> on <date> by GCAL(<version>) <comment>`’ between the GCODE version number and the first backslash, where ‘`<comment>`’ is the text given as a comment on the GCAL **fancy** directive.

5. Control Sequences Without Arguments

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

`NUL`	ignore (pseudo-formfeed)
`FF`	begin a new page
`NL`	begin a new line
`BS`	backspace one character

A pseudo-formfeed is output by GCAL every 50 lines when in ‘galley’ mode, to enable the file to be broken up for processing. The depth of line must be set before the use of `NL`. Between `NL` and the start of the next line, control sequences such as `FF` or `GDPR` 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 (`BS`, `RPR`, `LPR`, `UPR` and `DPR`). The control sequence `BS` always follows a printing character, and it specifies a leftwards movement equal to that character’s width.

6. Variable Length Control Sequences

An escape character (‘`\`’) followed by a decimal number, possibly including a decimal point, is used to introduce a number of different variable-length sequences of the form

```
\  P  T
```

where `P` is the sequence of decimal digits possibly including a decimal point, representing an unsigned parameter value. The control sequence is terminated by one of a number of terminators, `T`, which identify the control function required. Note that the spaces shown here are purely for readability, and do not actually appear in the GCODE text.

The control sequences are as follows:

```
\ P SSU
```

This sequence, if present, appears at the start of a GCODE dataset, following the identification

mnemonic	*value*	*function*
`BFT`	`=`	bind font
`FDL`	`"`	font name delimiter
`VSI`	`!`	vertical spacing
`SSU`	`#`	set space units
`RPR`	`>`	horiz position right
`LPR`	`<`	horiz position left
`DPR`	`$`	down position relative
`UPR`	`%`	up position relative
`SGR`	`_`	set graphic rendition
`FNT`	`\`	set font
`CHR`	`/`	output numbered char
`GDPR`	`)`	global DPR
`GUPR`	`(`	global UPR

Table 2: Special character names

and before the first formfeed. It specifies the units of size to be used throughout the dataset by the following values of P:

1	‘big’ points (72 to the inch)
2	‘real’ points (approximately 72.27 to the inch)

‘Big’ points are suitable for daisywheel and dot-matrix printers, and other devices which have character widths that can be converted exactly, while ‘real’ points are appropriate for photo-typesetters where font tables already exist in ‘real’ points. If this control sequence is absent, the size values should be assumed to be ‘big’ points.

    \ P RPR

This specifies a relative horizontal movement to the right of distance P. It is used for tabs and spaces.

    \ P LPR

This specifies a relative horizontal movement to the left of distance P.

    \ P DPR

This specifies a relative vertical movement down the page, of distance P, and local to the current line. It is used for subscript/superscript handling.

    \ P UPR

This specifies a relative vertical movement up the page, of distance P, and local to the current line. It is used for subscript/superscript handling.

    \ P GDPR

This specifies a global relative vertical movement down the page, of distance P. It is used for page filling and the GCAL **space** directive, and appears only between lines.

    \ P GUPR

This specifies a global relative vertical movement up the page, of distance P. It is used for the GCAL **up** directive, and appears only between lines.

    \ P VSI

This specifies the vertical spacing increment, that is, the vertical distance between successive lines on the page.

    \ P SGR

This specifies the ‘graphic rendition’ for succeeding characters. The following values are used:

0	neither underlined nor bold
1	bold, but not underlined
2	underlined, but not bold
3	bold and underlined

Whenever SGR is used, the entire graphic rendition must be re-specified. The ‘bold’ graphic rendition must be interpreted as ‘embolden the current font’. It must *not* be interpreted as ‘switch to a related bold font’, because the character widths of the bold font will be different to the original. This facility is not in fact intended for use on devices with typographic fonts; rather, it is for daisy-wheels and the like, where any character can be ‘emboldened’ by striking it two or more times.

\ P BFT M FDL *characters* FDL

This specifies a font binding for font number P (a value in the range 0–254). M is an argument which specifies the magnification used for the font. It consists of decimal digits only and is 1000 times the actual magnification specified. The character string identifies the font. By convention, it is normally split into two parts, separated by ‘/’. The first identifies a file of font definitions, and the second a particular font within the file, for example

    \23=1000"diablo/elite-es"

The way in which the two halves are related to actual names in the filing system is implementation-dependent, though normally upper and lower case letters should be synonymous. Any given font must not be bound more than once, and it must be bound before any of its characters are output.

    \ P FNT

Selects font number P for succeeding printing characters.

    \ P CHR

This specifies that character number P is to be output from the current font. It is used for characters that are not in the normal printing set. GCAL font encodings are based on the Ascii character set (see Appendix B).

7. Device and Comment Strings

The control sequence

DCS *characters* DCS

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 GCAL directive **control**. If the backslash character is required as part of the string, it appears as ‘\\’. In fact, backslash is used as an escape for certain special characters, as follows:

\N	for newline
\S	for space
\\	for backslash

These are therefore encoded in the GCODE as

\\N	for newline
\\S	for space
\\\\	for backslash

The use of space characters is not forbidden, but it is not recommended, because of potential truncation problems.

APC *characters* APC

This string is used as a general mechanism for passing information to the GTYPE program. It is generated as a result of obeying the **request** directive in GCAL. By convention the text consists of a device name terminated by a colon, followed by a GTYPE command. For example

    \AFX80:MARGIN=24\A

The same rules as for DCS apply for the inclusion of backslashes or non-printing characters.

* * *