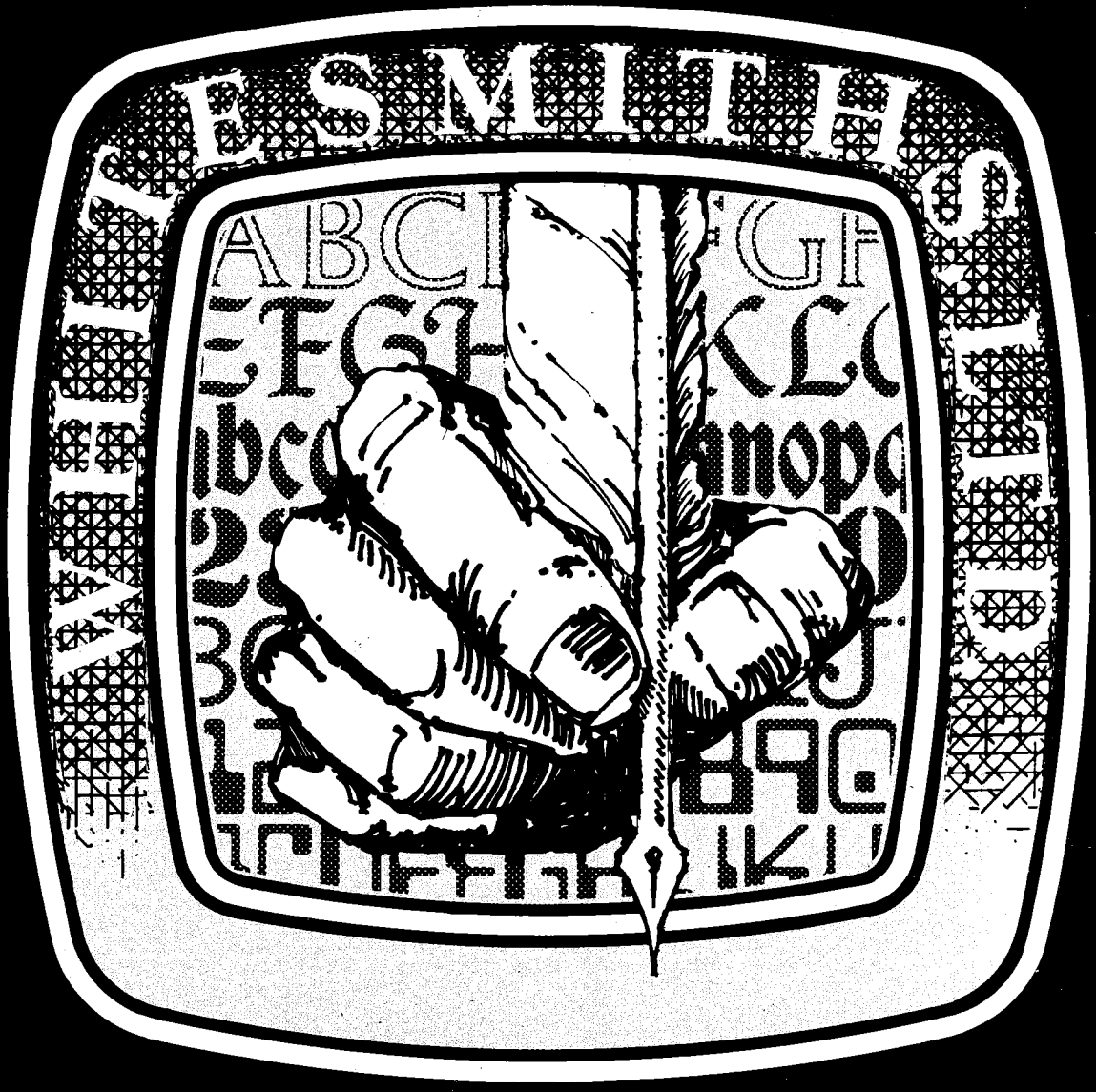
Whitesmiths, Ltd. Ctext Users' Manual



Whitesmiths, Ltd.

CTEXT USERS' MANUAL

Date: April 1984

The C language was developed at Bell Laboratories by Dennis Ritchie; Whitesmiths, Ltd. has endeavored to remain as faithful as possible to his language specification. The external specifications of the Idris operating system, and of most of its utilities, are based heavily on those of UNIX, which was also developed at Bell Laboratories by Dennis Ritchie and Ken Thompson. Whitesmiths, Ltd. gratefully acknowledges the parentage of many of the concepts we have commercialized, and we thank Western Electric Co. for waiving patent licensing fees for use of the UNIX protection mechanism.

The successful implementation of Whitesmiths' compilers, operating systems, and utilities, however, is entirely the work of our programming staff and allied consultants.

CTEXT USERS' MANUAL

SECTIONS

SCOPE

This manual describes the ctext document formatting system. Section I documents how ctext appears to the user and to the programmer. Beginning users should refer to the essay titled **Intro** at the start of the section; a detailed specification of the macro sets, called **Macros**, follows this introduction. This section also contains a full description of the text formatter itself, called simply **Ctext**, meant for those who must maintain or extend the macros or device support supplied with the standard package.

Section II consists of "manual pages" for the programs that compose the ctext system; these pages explain how to run each program, and what role it plays in the system. Section III, aimed exclusively at programmers interested in new device support, contains specifications for a small set of system interface functions sometimes needed by the existing ctext device drivers. These descriptions are portable, and are intended to guide the implementation of new drivers, and the porting of existing ones to new environments.

Finally, a set of appendices documents various internal details of the ctext program that may be of interest to sophisticated users or programmers. In particular, all ordinary diagnostic messages output by ctext are described here.

TABLE OF CONTENTS

I. Ctext Reference Manual

II. Ctext Component Programs

III. Ctext Device Interface Functions

I. Ctext Reference Manual

NAME

Intro - an introduction to ctext

FUNCTION

ctext formats text. You use ctext by creating one or more input files containing textual material, like paragraphs, headings, or tables. By reading these input files, ctext creates an output file that contains the same material, formatted to have a more pleasing or regular appearance. ctext can perform a variety of services, from simple ones like maintaining even margins for paragraphed input, to more complex ones like building attractive tables.

Just as important, ctext permits you to regularize and simplify the process of getting your document to look a certain way. First of all, instead of typing in exactly what you want the document to look like (an error-prone process that is tedious to duplicate), you give commands that tell ctext what effect you want. Not only are these commands easier to use; you can change what a command does by giving it a new definition. So you can change the appearance of your document without changing anything in the document itself. To provide still more flexibility, ctext will automatically make use of whatever special features are available on the terminal, printer or other device on which your document will appear. You name this output device; the formatter does the rest, optimizing its output for the device without needing any changes in your input file.

1. RUNNING CTEXT

You will probably run ctext by using a command file (like a shell script under Idris/UNIX, or a ".COM" file under VMS). Two of the command files shipped with ctext are likely to be of interest. One, named **cth19**, will format text for a VT52 video terminal (like the Heathkit H19); the other, named **ctdbl**, will format text for generic printers, or for Diablos. Which of these you use depends on what your output device will be. Keep in mind, if you have doubts on the subject, that ctdbl will do something decent for just about any device. The simplest way to run either script is just to name it, followed by the name of your input file, as in:

```
% ctdbl test.doc
```

The output from ctext will then appear directly on your terminal or printer. Both command files accept a number of "flags", or options, to change some aspect of how they work. These flags are given before the name of your input file. For instance, to get ctdbl to use horizontal incremental spacing in formatting for a Diablo, you could type:

```
% ctdbl -hi test.doc
```

Full descriptions of **cth19** and **ctdbl** appear in Section II of this manual. There you will find a listing of all the possible flags. Also, at the start of Section II, a manual page called **Programs** summarizes all of the separate programs that make up the ctext system, and how they fit together.

2. INPUT FILE CONVENTIONS

What does an input file look like? If you know how to create and edit textfiles under the operating system where ctext is running, you know most of what you need to. Very little is special about a ctext input file; the simplest kind, in fact, contains nothing but the text to be output. Specifically, an input file contains zero or more lines of text, each ending with a newline character. (You may type a carriage return or a line feed to end a line of text; we will use the term newline to refer to whichever character you use.) Text can be broken into lines in any way you want. It can have spaces embedded in it in any pattern you like. Some things, like tab characters (the ASCII character HT), or lines that start with whitespace (tab characters or spaces), or blank lines (two consecutive newlines), do have a special meaning. But, as you'll see, they mean to ctext just about exactly what they're likely to mean to you.

2.1. Using Fill Mode

Suppose, for instance, you wanted to format two paragraphs of text. Your input file could look exactly like this:

```
It is a truth universally acknowledged,
that a single man
in possession of a good fortune,
must be in want of a wife.

However little known
the feelings or views of such a man may be
on his first entering a neighborhood,
this truth is so well fixed
in the minds of the surrounding
families,
that he is considered
as the rightful property
of some one or other of their daughters.
```

What ctext does with this input is determined by several internal settings. By default, ctext fills the lines of text it outputs from one or more input lines, by adding words to the output line until adding the next word available would make the line wider than a specified line length. At that point, the line being constructed is output, and ctext begins a new one. This way of creating output lines is called fill mode, and is most useful for running text like these paragraphs. Also by default, ctext will pass through blank lines from its input to its output. So a blank line can serve to separate paragraphs. Given all of this, running ctext on the example shown might produce the following output:

It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.

However little known the feelings or views of such a man may be on his first entering a neighborhood, this truth is so well fixed in the minds of the surrounding families, that he is considered as the rightful property of some one or other of their daughters.

2.2. Using No-Fill Mode

Output generated in fill mode is called filled text; its analogue, as you might guess, is unfilled text, which is output in no-fill mode. In this mode, each output line is constructed from a single input line. Also, each output line is left-justified, that is, output starting at the left margin with exactly the spacing it had when input. No-fill mode, too, can be entered without any explicit commands. A line that starts with whitespace is automatically output unfilled. You can therefore prepare the line in your input file exactly as you want it to appear in your document. An input file using no-fill mode to set a quotation might look like:

```
In examining
the tormented life of this misguided genius,
we are led reluctantly to a discouraging conclusion:

    A brain weight of nine hundred grams is adequate
    as an optimum for human behavior.  Anything more
    is employed in the commission of misdeeds.
        -- Earnest Hooton

To assess the truth
of the statement,
we need but examine the aphorism.
```

which would produce this output:

> In examining the tormented life of this misguided genius, we are led reluctantly to a discouraging conclusion:
>
> A brain weight of nine hundred grams is adequate as an optimum for human behavior. Anything more is employed in the commission of misdeeds.
> -- Earnest Hooton
>
> To assess the truth of the statement, we need but examine the aphorism.

Note that every line started with whitespace is output in no-fill mode. Notice, too, that each of these lines turns on no-fill mode only for itself. If the next line does not start with whitespace, fill mode is reestablished.

2.3. Using Tab Characters

One final input file convention concerns tab characters. These characters are significant to ctext, and (unless you issue a command to the contrary), are used to output space up to the next tabstop. A tabstop is just a pre-defined point within the output line. By default, tabstops are set every four columns, up to column 80, so a tab character in your input will get translated into from one to four spaces -- however many are needed to get to the next tabstop.

This is useful in no-fill mode, since simple tables can be formatted without any commands being given. But it also means that tab characters should be avoided in filled text; here, their effects are unpredictable (because there is no simple correlation between input line position and output line position).

3. COMMAND CONVENTIONS

Though the conventions described up to now are useful, they won't get you far in creating documents of any complexity. They have a reasonable amount of influence on the resulting output, but very little information can be passed to ctext by them. To give ctext more information you must use explicit commands. These commands are indicated by special strings of characters, or markup, that you intersperse with the text to be formatted.

Markup does not appear directly in your output document. Instead, it causes ctext to take some action that will affect the document. This indirection is important. Instead of spelling out precisely what actions ctext is to perform, you can use markup to tell ctext what you want to create, leaving to the processor the decision of how to create it.

3.1. Command Formats

By default, ctext recognizes two kinds of markup.

1) any input line that begins with a period '.' is treated as a command, and not as ordinary text. The command is taken to be the entire line.

2) for more flexibility still, the characters "<:" are also taken to be the start of a command, which continues until a matching ":>" is found. Notice that the characters "<:" begin a command wherever they are found, even within an input line. And the command continues until a ":>" is found, even across more than one input line.

3.2. Command Names

A command has two primary parts. First is a command name, which is usually just a string of letters designed to bring to mind what the command does. (Digits and underscores can also be part of a command name, but are not used in any of the command names we will discuss.) Thus, either of these lines is a command:

```
.DE
<:LP:>
```

Notice that the second command is on a line by itself. If a command that opens with "<:" begins at the start of an input line, then it may be followed by a newline that will be included as part of the command, just as a "." command would be. On the other hand, the following line of input text contains two commands (named "b" and "r"):

```
This is <:b:>not<:r:> acceptable to us.
```

3.3. Command Arguments

The second component of a command is an optional argument list, which is a set of strings separated by whitespace. Normally, each string is a separate argument; if you want a string containing whitespace to be counted as a single argument, you enclose it in quote characters. (The whole string is then called a quoted argument).

The first form of command has only the double-quote (") as a quote character. The second form of command has five pairs of quote characters; any of the characters "'([{" or double-quote may be used to begin a quoted argument, which is ended by the corresponding character from "')]}" or double-quote. Thus both of these commands contain four arguments:

```
.SH AN INCORRECT SECTION HEADING
<:BH A Similar Block Heading:>
```

While these commands contain only one argument:

```
.SH "A CORRECT SECTION HEADING"
<:BH {An Analogous Block Heading}:>
```

3.4. Types of Arguments

There are two major types of arguments. One kind is a string argument, that specifies some text for use by a command (often as a heading). The other kind is a numeric argument.

Often, a numeric argument specifies a distance to be measured in your output document. Horizontal distances, like for setting line length or tab-stop locations, are frequently measured in "characters", though other units like "inches" or "picas" could also be used. Vertical distances, like for setting page length or the amount of space around headings, are usually measured in lines, though other units are sometimes more convenient.

A number that names a distance is called a measurement. You make a number into a measurement by following it with a unit specifier, that indicates in what units the distance is being measured. Some common unit specifiers are:

NAME	MEANING
ch	characters
ln	lines
in	inches
pc	picas
pt	points

For example, the measurement "5ch" names a distance of five characters, while "2ln" names a distance of two lines. Measurements may use fractional numbers, too: ".5in", for instance, names a distance of half an inch. Frequently, where a measurement is expected a default unit specifier will be used if you do not give one explicitly. This will always be indicated in the command description.

3.5. User-defined Commands

Most of the commands you're likely to use have names that consist of two upper-case letters, like **PP** or **LP**. A name of this kind implies that a command is user-defined -- that is, not built into ctext, but added to the basic system through a macro package.

Each of these commands is called a macro, and is basically a form of shorthand. A macro has two components. First, of course, is the name by which you call -- or invoke -- it. Second is its definition, which is a string of text that is substituted in place of each call. Usually, this definition is much longer and more complicated than the call was. More to the point, when you call the macro you do not know, nor need you care, what definition will eventually be used for it. All you need to consider is the operation you want to perform, and what macro will perform it. And to change how the operation is performed, only the macro definition is altered; any call to the macro can be unaffected. This is one key to the flexibility that the use of commands provides.

Other essays in this manual describe the standard macro packages more fully, and explain how to create new macros. For now, you don't need to worry about whether a given command is a macro, or is built into ctext. You use both kinds of commands in the same way. But we will use the term "macro" to name any command that is user-defined, so the difference will be explicit if you wish to note it.

4. MACROS FOR CREATING PARAGRAPHS

We've already seen that you can create paragraphs with ctext just by inserting blank lines in your input file. However, by using commands instead to indicate the start of each paragraph, you can very easily get prettier output (with the macros in the standard package), and can ease the process of changing your document's appearance after the input file is prepared (should you decide to change the macro definitions later on).

All three of the macros for starting paragraphs make sure that a half-line of space will precede the paragraph. (For output devices where a half-line of space is unobtainable, a full line of space results.) They also set up the default series of tabstops mentioned above: one every four columns through column 80.

4.1. Creating 'Plain' Paragraphs

The command **PP** is used to begin a 'plain' paragraph, in which the first line is indented by a fixed amount, and all later lines begin at the left margin. If we used this macro in our initial sample input file, the file would look like:

```
.PP
It is a truth universally acknowledged,
that a single man
in possession of a good fortune,
must be in want of a wife.
.PP
However little known
the feelings or views of such a man may be
on his first entering a neighborhood,
this truth is so well fixed
in the minds of the surrounding
families,
that he is considered
as the rightful property
of some one or other of their daughters.
```

And the new file would produce the following output:

> It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.
>
> However little known the feelings or views of such a man may be on his first entering a neighborhood, this truth is so well fixed in the minds of the surrounding families, that he is considered as the rightful property of some one or other of their daughters.

Notice that the two paragraphs now have a bit less space between them. This is a half-line instead of a full line of whitespace.

4.2. Creating Left-Justified Paragraphs

The command **LP** is used to begin a left-justified paragraph, in which every line (including the first) begins at the left-hand margin. We forego an example here: this essay was formatted using **LP**.

4.3. Creating Indented Paragraphs

The command **IP** is used to begin an indented paragraph, in which the first line may start at the left margin, and all later lines are indented by a fixed amount. One subtlety: though **IP** sets up the same default tabstops as **PP** and **LP**, the tabstops are measured from the indented margin, that is, from where the indented lines of the paragraph begin.

IP is used in two different ways. You can call it with no arguments, just like you would **PP** or **LP**, in which case the whole paragraph that follows

will be indented by the same amount. For instance, the quotation in our second example could be formatted with a call to **IP**:

```
.LP
In examining
the tormented life of this misguided genius,
we are led reluctantly to a discouraging conclusion:
.IP
A brain weight of
nine hundred grams is adequate as an
optimum for human behavior.
Anything more is employed
in the commission of misdeeds.
-- Earnest Hooton
.LP
To assess the truth
of the statement,
we need but examine the aphorism.
```

If **IP** were used, the following output would result:

> In examining the tormented life of this misguided genius, we are led reluctantly to a discouraging conclusion:
>
> > A brain weight of nine hundred grams is adequate as an optimum for human behavior. Anything more is employed in the commission of misdeeds. -- Earnest Hooton
>
> To assess the truth of the statement, we need but examine the aphorism.

4.4. Creating Itemized Paragraphs

IP can also be used to format an itemized paragraph, in which some short piece of text, or tag, is highlighted at the paragraph's start. The tag is given as the first argument in the call to **IP**, and is placed at the left margin of the first line. The rest of the first line is then indented like the later ones. If possible, the tag is also formatted in boldface (or at any rate is highlighted) so that it stands out from the rest of the paragraph. A simple set of itemized paragraphs might be:

```
.IP "an input file -"
is an ordinary textfile
(as the host operating system defines it),
containing ordinary text interspersed with markup.
.IP "markup -"
is an invocation of a command.
.IP "a command -"
is an instruction to ctext to do something,
and is not placed in the output file;
commands are either builtin or user-defined.
.IP "a user-defined command -"
is usually a macro,
consisting of a name and a definition,
which is inserted into the input file wherever the name
is used as a command.
```

Formatted, this would become:

an input file - is an ordinary textfile (as the host operating system defines it), containing ordinary text interspersed with markup.

markup - is an invocation of a command.

a **command** - is an instruction to ctext to do something, and is not placed in the output file; commands are either builtin or user-defined.

a **user-defined command** - is usually a macro, consisting of a name and a definition, which is inserted into the input file wherever the name is used as a command.

Notice that a tag does not have to fit in the space between the left margin and the indented margin. If a tag is longer, it is followed by a single space, then by the rest of the first line.

To make indented paragraphs easier to use, all three paragraphing macros remove any indentation left over from the last indented paragraph before they do anything else. Thus successive indented paragraphs all line up under one another, and a paragraph started with **PP** or **LP** moves back to the old left-hand margin.

4.5. Nesting Indented Paragraphs

Sometimes, however, it is desirable to nest indented paragraphs, that is, to have a series of paragraphs indented further than some enclosing paragraphs. This might occur in any hierarchically arranged document. With the standard macro package, this nesting is done through indentation levels.

The macro **IL** begins a new indentation level. What actually happens is that the column used to indent the last preceding indented paragraph is used as the new left margin, so that any future indented paragraphs will be indented from there, and **PP** and **LP** will return the left margin to there.

The macro **IX** ends the current indentation level. The indentation remaining from the last indented paragraph is removed, and the left margin moves back to where it was when that paragraph was started.

IL should be used before a set of nested paragraphs, and **IX** should be used afterwards. If you pair the two in this manner, ctext makes the left margin come out correctly. Suppose we wanted to insert the itemized paragraphs from the last example into a larger document. One way to do so would be:

```
.IP A.
Running ctext.
.IP B.
Input file conventions.
.IP C.
Command conventions.
To contain the following definitions:
.IL
.IP "an input file -"
is an ordinary textfile
(as the host operating system defines it),
containing ordinary text interspersed with markup.
.IP "markup -"
is an invocation of a command.
.IP "a command -"
is an instruction to ctext to do something,
and is not placed in the output file;
commands are either builtin or user-defined.
.IP "a user-defined command -"
is usually a macro,
consisting of a name and a definition,
which is inserted into the input file wherever the name
is used as a command.
.IX
.IP D.
Macros for creating paragraphs.
```

which would yield the following result:

A. Running ctext.

B. Input file conventions.

C. Command conventions. To contain the following definitions:

an input file - is an ordinary textfile (as the host operating system defines it), containing ordinary text interspersed with markup.

markup - is an invocation of a command.

a command - is an instruction to ctext to do something, and is not placed in the output file; commands are either builtin or user-defined.

a user-defined command - is usually a macro, consisting of a name and a definition, which is inserted into the input file wherever the name is used as a command.

D. Macros for creating paragraphs.

5. MACROS FOR CREATING HEADINGS

Another way to impose structure on a document is to use headings. Commands are provided for three different levels of headings, so that a hierarchy can be used in which a document component is given a heading that matches its importance. Where possible, all headings are output as highlighted. Headings are also preceded and followed by a fixed amount of space, so that they are always on a line set apart from any surrounding text.

The heading to be output is given as the first argument in the call to these macros. If the heading contains more than one word, it must be given as a quoted argument.

5.1. Creating a Section Heading

The macro **SH** creates a high-level heading called a section heading. **SH** makes sure that two lines of space will precede the heading, then outputs the heading in boldface (or in any case as highlighted). The heading is followed by a full line of space.

Thus the command:

```
.SH "FROM RAGS TO RICHES"
```

would produce the output:

FROM RAGS TO RICHES

The text of the heading output with **SH** may be anything you wish; however, a convention often followed is to put the heading all in capitals, so that it stands out as much as possible.

5.2. Creating a Block Heading

The macro **BH** creates a medium-level heading, called a block heading. **BH** makes sure that a line of space will precede the heading, then outputs the heading in boldface (or in any case as highlighted). The heading is followed by a half-line of space.

For example, the command:

```
.BH "Boyhood Beginnings"
```

would yield the output:

Boyhood Beginnings

A convention often followed with **BH** is to capitalize each important word in the heading output, in keeping with its intermediate importance.

5.3. Creating a Paragraph Heading

The macro **PH** creates a low-level heading, called a paragraph heading. **PH** makes sure that a half-line of space will precede the heading, then outputs the heading in italics (or in any case as highlighted). The heading is followed by a half-line of space.

For instance, the command:

```
.PH Greed
```

would produce the heading:

Greed

A convention often followed with **PH** is to capitalize only the first word in the heading output, leaving the rest all lower-case.

6. COMMANDS FOR OUTPUTTING SPECIAL TEXT

Frequent mention has been made of text output as "highlighted", as in a heading or in an indented paragraph tag. The mechanism ctext uses to do this is also directly available to you, so that text of your choice can be output as boldface, in italics, or both.

6.1. Using Stylecodes

All text processed by ctext is output using a stylecode. A stylecode is a command that designates one particular style of output for your eventual output device. Stylecodes permit you to name a style by the function it will serve, without worrying about how that style will be translated for a given device. The stylecodes normally defined are: **r**, for ordinary or roman output; **b**, for boldface output; **i**, for italic output; and **s**, for boldfaced italic (or "special") output.

By default, ctext uses the stylecode r, which, of course, creates text in a "normal" style for the output device. But you can alter the stylecode currently in effect just by using the command for the stylecode you want. Thus in a line of filled text, to put the word "not" into boldface, you could enter:

```
This is
.b
not
.r
acceptable to us.
```

or equivalently:

```
This is <:b:>not<:r:> acceptable to us.
```

either of which would give the output:

This is **not** acceptable to us.

If you give a new stylecode as a command, it remains in effect until you name another stylecode to replace it.

6.2. Using Style Control Macros

Three commands are provided to change stylecodes for a specified number of input lines, then revert to the stylecode that was in effect before the command was called. These are often more convenient to use than a stylecode alone, because you do not have to restore the previous stylecode explicitly after temporarily changing it.

These three macros are: **BD**, to make text boldface; **IT**, to make text italic; and **RM**, to make text roman. The first argument to these macros specifies the number of input lines they are to affect; if no argument is given, then the macros affect just one line. Our last example, using these macros, would become:

```
This is
.BD
not
acceptable to us.
```

Note, too, that these macros will read your input file until they have gotten the number of ordinary input lines you specified. This means that other commands may appear interspersed with the lines of text that the

command is reading. For instance, to make one word appear in roman, within a sentence set in boldface, the following would suffice:

```
.BD 3
This is
.RM
normally
extremely important.
```

Specifically, this asks **BD** to read three input lines and set them in boldface, except that the middle one (under the influence of a one-line **RM**), gets set in roman. Notice, though, that the middle line is still counted by the **BD** command as one of its input lines. This input would produce:

This is normally **extremely important.**

7. MACROS FOR LINE FORMATTING

The standard macro package provides a set of commands to change the characteristics of the lines output to your document. These commands are used to turn fill mode on or off, among many other things. They give you more explicit and tighter control over the appearance of your output than anything else we've so far discussed.

Many of these commands cause a line break when they are encountered. That is, any output line currently being constructed is output before the command is executed, and a new output line is started. Though we have not noted the fact, a line break is also caused whenever vertical whitespace is output (like between paragraphs or around a heading).

Line breaks happen in all of these cases because starting a new output line is usually what you want to do. Line breaks can be suppressed, however, with the command **NB**. Use **NB** just before any command that usually causes a line break, and the line break will not happen.

From here on, if a command causes a line break, its description will note the fact.

7.1. Changing Fill Mode

The first two of these commands provide an explicit way to enter or leave fill mode. The macro **FI** turns fill mode on, while the macro **NF** turns fill mode off (i.e., enters no-fill mode).

If you use **NF** to enter no-fill mode, your document is formatted in no-fill mode until fill mode is explicitly re-entered. However, even if you enter fill mode explicitly with **FI**, an input line starting with whitespace will still be output in no-fill mode.

Both **FI** and **NF** cause a line break.

7.2. Changing Text Justification

Two major settings determine how an output line appears in your document. One, as we have seen, is whether or not fill mode is on. The other is how the text in the line is justified, that is, how the text is positioned within the line.

By default, lines are body-justified: both right and left margins are kept even by expanding the text in the line until it exactly fits the current line length. Only lines output because of a line break (like the last line of a paragraph) are not treated this way. These lines are output left-justified: the text begins at the left margin, and is not expanded to the right-hand margin, no matter how short the resulting line.

One pair of macros alters this default justification by controlling whether the right-hand margin is even or ragged. The command **NA** (for "no adjustment") causes every line to be output left-justified. Note that whether or not fill mode is on is not affected by this. The only effect is to create a ragged right margin. The command **AD** (for "text adjustment") causes output lines to be body-justified once more. Again, the command has no effect on fill mode.

If, for instance, we wanted our first example to be output with a ragged right margin, we could add a call to **NA** at the start of the input file:

```
.NA
.PP
It is a truth universally acknowledged,
that a single man
in possession of a good fortune,
must be in want of a wife.
.PP
However little known
the feelings or views of such a man may be
on his first entering a neighborhood,
this truth is so well fixed
in the minds of the surrounding
families,
that he is considered
as the rightful property
of some one or other of their daughters.
```

to produce the following output:

> It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.
>
> However little known the feelings or views of such a man may be on his first entering a neighborhood, this truth is so well fixed in the minds of the surrounding families, that he is considered as the rightful property of some one or other of their daughters.

A third command will causes lines of input text to appear centered in your document. The macro **CE**, like the style-control macros, takes an argument that specifies the number of input lines it is to affect. If no argument is given, then **CE** centers the next input line. Once the lines of text specified have been read, output line justification reverts to whatever it was before **CE** was called.

In addition to centering each input line, **CE** turns off fill mode, so that each line appears as a separate output line in your document. Thus a two line heading could be created with the following input:

```
.CE 2
Notice of Tentative
Rent Increase
```

which would yield the output:

Notice of Tentative
Rent Increase

The **CE** command can also be used very effectively with the style-control commands to produce more striking headings. The following input asks for the next two lines to be centered and set in boldface:

```
.BD 2
.CE 2
NOTICE OF
CONDOMINIUM CONVERSION
```

for the output:

NOTICE OF
CONDOMINIUM CONVERSION

All three of the macros **NA**, **AD** and **CE** cause a line break before they are executed.

7.3. Changing Other Line Characteristics

The standard macro package also gives you control over several other aspects of how output lines look.

The macro **LL** sets the total line length of your output. The length is given as the first argument to the command, and should be a measurement. For instance,

```
.LL 6in
```

sets up a line length of six inches, while

```
.LL 72ch
```

sets up a line length of 72 characters. If the number you give contains no unit specifier, the number is taken to mean "characters". By default, a line length of 6.5in is used.

The macro **IN** sets the left-hand indentation for output lines, each of which will begin with the amount of space you specify. The indentation is given as the first argument to the command, and should be a measurement. If the number you give contains no unit specifier, the number is taken to mean "characters". By default, no indentation is used. The indentation you specify is counted as part of the total output line length. Thus the commands

```
.LL 6.5in
.IN .5in
```

set up line length of 6.5in with an indentation of half an inch, leaving six inches for the actual text in each line.

Both **LL** and **IN** can specify a new value relative to the current one. If you give a number preceded by "++", it is added to the current value to get the new one; a number preceded by "--" is subtracted in the same way. So to increase line length by five characters, you could use:

```
.LL ++5ch
```

while the following would decrease indentation by half an inch:

```
.IN --.5in
```

Indentation has two important side effects: both centered text and tabstops are output using the current line length minus any indentation specified. So centering occurs over the part of the output line actually available for text output, and tabstops are measured from where text starts on the line.

Once you have established the line length and indentation you want, another characteristic you may want to change is line spacing, that is, the amount of space between output lines. By default, lines are output with no space between them (in other words, single-spaced). However, the macro **LS** can be used to specify other amounts of space. The desired spacing is given as the first argument to **LS**, and is an ordinary number. The number is taken as a multiple of the height of each output line, and determines the amount of space left for each one. For instance, the command

```
.LS 2
```

would set up double spacing (because the space left for each output line would be twice its height). Fractional values may also be given, and will give results as accurate as permitted by the output device. The following requests a half-line of space between output lines:

```
.LS 1.5
```

All three of the commands **LL**, **IN** and **LS** cause a line break before they are executed.

One final characteristic you can control is how ctext will perform hyphenation. Whenever adding the current word to a line of filled text

would make the line exceed the current line length, ctext tries to hyphenate the word, to see if some part of it will fit at the end of the line.

The command **HY** sets the minimum number of characters that must remain in each part of a hyphenated word in order for ctext actually to hyphenate it. The number is given as the first argument to the command. Thus,

```
.HY 3
```

would force both halves of any hyphenated word to contain at least three characters. Giving a number of zero turns off hyphenation. By default, each part of a hyphenated word must contain two characters.

8. MACROS FOR GALLEY CONTROL

So far, we have discussed formatted output in terms of individual lines of text. We will now describe commands that do more than affect single output lines. Here, it is useful to look at formatted output in a different way.

The process of creating a document can be viewed as formatting an output line, then adding it to the end of a galley, an endless column of output. As we will see, ctext breaks this galley into fixed-length pages, and (by default) outputs each page with a user-specified heading and footing. But the notion of the galley itself is important: it is the formatting environment in which individual output lines appear. A number of commands add space to the end of the galley, or both add space and change some group of formatting characteristics to set up a specific kind of output.

8.1. Generating Whitespace

One series of commands is provided to insert vertical whitespace into your document, at the end of the current galley. All of these commands cause a line break before they are executed.

The first of these macros, **BR**, just causes a line break. No whitespace is output after the line break. Thus, a new line of text is started immediately below the previous one.

Two macros serve to output a specified amount of whitespace at the end of the galley. The command **HS** outputs a half-line of space, if that is possible for your eventual output device (otherwise, **HS** outputs a full line). The command **SP** outputs the amount of whitespace named by its first argument, which should be a measurement. Or, **SP** can be called with no argument, in which case it outputs one line of space. The input for a letter with two lines of space after the address and one line after the salutation might read, in part:

```
Ms. Emma Knightley
Hartfield Manor
Highbury, Hertfordshire
.SP 2ln
My dear Ms. Knightley:
.SP
.FI
While sympathizing with your distress at marrying
a man so much your senior, . . .
```

and would produce the output:

```
Ms. Emma Knightley
Hartfield Manor
Highbury, Hertfordshire


My dear Ms. Knightley:

While sympathizing with your distress at marrying a
man so much your senior, . . .
```

If a number given to **SP** does not contain a unit specifier, it is taken to mean "lines".

Both **HS** and **SP** always output exactly the amount of vertical whitespace requested of them. Often, however, it is more useful to output space conditionally, that is, to check the amount of whitespace already at the end of the galley, and add only the additional space needed to bring the total up to a certain amount. If, for example, you wanted to follow a section heading with two lines of space, you could try to use:

```
.SH "FIXED ASSETS"
.SP 2ln
```

only to discover that three lines of whitespace would follow the heading. Why? Because **SH** follows the heading with one line of space, and **SP** asks for two more.

You can simplify the process of requesting vertical space by using the macro **CS**. **CS** (for "conditional space") acts just like **SP**, except that it checks the whitespace already at the end of the galley, and adds only what is needed to bring the total amount up to what you request. Thus an easier way to follow a section heading (or anything else) with exactly two lines of space is to say:

```
.SH "FIXED ASSETS"
.CS 2ln
```

If a number given to **CS** does not contain a unit specifier, it is taken to mean "lines".

Another command allows you to check whether or not a certain amount of space still remains on the current output page. The macro **NE** asserts your need for the amount of vertical space named by its first argument. If at

least this much space remains on the current page, **NE** does nothing. If not enough space is left, **NE** ends the current page and starts a new one. **NE** is most often used to assure that a group of lines will appear on the same page of output. For instance, to keep together the lines of a quotation, you could use:

```
.NE 4
Up the airy mountain,
Down the rushy glen,
We daren't go a-hunting
For fear of little men . . .
```

which would assure that the four lines needed are available on the current page. If a number given to **NE** does not contain a unit specifier, it is taken to mean "lines".

8.2. Special Formatting Commands

Several commands are provided to set up the galley for a particular kind of output. All of these commands cause a line break, and create whitespace at the end of the galley, before they do anything else.

The first of these, the macro **LH**, simplifies creating a letter by preparing output for letterhead stock. Specifically, it advances to the top of a page, sets the page number to one, removes any footing that was to appear on page one, then leaves eight lines of whitespace (to skip past the presumed letterhead), and enters no-fill mode. After using **LH**, you are ready to enter the first text that will appear in the letter. So our last example could be fleshed out as:

```
.LH
10 February 18--
.SP 21n
Ms. Emma Knightley
Hartfield Manor
Highbury, Hertfordshire
.SP 21n
My dear Ms. Knightley:
.SP
.FI
While sympathizing with your distress at marrying
a man so much your senior,
we regret to say that we have examined
the documents pertaining to the marriage,
and can find nothing that would permit
their invalidation.
.LP
We wish your peace.
.SP 21n
Most humbly yours,
.SP 21n
.NF
Suckling, Suckling, and Bragge
Solicitors
```

And this input would produce the letter:

10 February 18--

Ms. Emma Knightley
Hartfield Manor
Highbury, Hertfordshire

My dear Ms. Knightley:

While sympathizing with your distress at marrying a man so much your senior, we regret to say that we have examined the documents pertaining to the marriage, and can find nothing that would permit their invalidation.

We wish your peace.

Most humbly yours,

Suckling, Suckling, and Bragge
Solicitors

(The amount of space left by **LH** was reduced to improve the presentation.)

Another common kind of output is a display, a body of unfilled text that appears in the output document broken into lines in the same way as it was in your input. A table is the most frequent kind of display, though any text whose appearance is to be preserved in the output can be treated as one. The macro **DS** is used to mark the start of a display; the macro **DE** is used to mark its end. **DS** makes sure that half a line of space exists at the end of the current galley, sets up a default set of tabstops, then enters no-fill mode. All that **DE** does is to re-enter fill mode; it adds no whitespace to the end of the galley, so that you can follow the display with anything you want, without any whitespace interfering with it. A simple table could be produced with the input:

```
The following commands set up special formatting
conventions:
.DS
.BD
    NAME        MEANING
     LH    start output for letterhead
     DS    start a display
     DE    end a display
     EX    start an extract
     XE    end an extract
.DE
.LP
These commands
may be freely intermixed with all the others,
in particular the heading and style control commands.
```

Notice that **DS** is not usually preceded by anything special; the half-line of space it outputs is generally sufficient to separate the display from what comes before it. **DE**, however, is often followed by whatever command is needed to generate the next item in the document. Here, a left-justified paragraph follows the display, so an **LP** command follows the **DE**. The input shown would generate the following display:

The following commands set up special formatting conventions:

NAME	MEANING
LH	start output for letterhead
DS	start a display
DE	end a display
EX	start an extract
XE	end an extract

These commands may be freely intermixed with all the others, in particular the heading and style control commands.

Another kind of output is an extract, a body of filled text set off from its surroundings by spacing and indentation. A common use of extracts is to format extended quotations, but any text that needs to be highlighted can be effectively presented in this way. The macro **EX** is used to start an extract; the macro **XE** is used to end one. **EX** makes sure that half a line of space exists at the end of the current galley, increases left-hand indentation by 5ch, and decreases line length by the same amount. **XE** restores the previous formatting characteristics, increasing line length by 5ch, and decreasing left-hand indentation by the same amount. **XE** then inserts a half-line of whitespace at the end of the galley. To return yet again to our second example, its quotation could be set very simply by using **EX**:

```
.LP
In examining
the tormented life of this misguided genius,
we are led reluctantly to a discouraging conclusion:
.EX
A brain weight of
nine hundred grams is adequate as an
optimum for human behavior.
Anything more is employed
in the commission of misdeeds.
-- Earnest Hooton
.XE
To assess the truth
of the statement,
we need but examine the aphorism.
```

This time, the output would be:

> In examining the tormented life of this misguided genius, we are led reluctantly to a discouraging conclusion:
>
> > A brain weight of nine hundred grams is adequate as an optimum for human behavior. Anything more is employed in the commission of misdeeds. -- Earnest Hooton
>
> To assess the truth of the statement, we need but examine the aphorism.

EX and **XE** are a convenient way to highlight whole paragraphs of filled text. Since they affect the left and right margin equally, using them leads to more pleasing results than using **IP** for the same purpose. Also, of course, using commands for only one purpose distinguishes what you meant by using one instead of another. If you use **EX** and **XE** to "flag" any extracts you make use of, you can change the way extracted text appears by giving either command a new definition, without affecting anything else in your document.

9. COMMANDS FOR FORMATTING TABLES

Several of the examples up to now have made use of tabstops; one has even contained a simple table. Inside paragraphs or displays, the standard macro package sets up a default set of tabstops, one every four columns, that make it very easy to produce simple tabular material. So far, we have used tabstops much as we would on a typewriter or a VDT: tabstops have been equally spaced a fixed distance apart, and we have used them to position the beginning of a piece of text. ctext, however, permits you to do much more with them. We will look at how tabstops can be established, and at the special commands that make use of them, once they're set.

9.1. Setting Tabstops

A tabstop names a specific point within an output line. The easiest way to set a tabstop is to use the command **TA**. With **TA**, you specify a tabstop as a measurement from the indented left margin of the output line. That is, you measure a tabstop starting at the point where left-justified text would begin. If, for instance, we wanted to use explicit tabstops in the table of commands we showed earlier, we might try:

```
The following commands set up special formatting
conventions:
.DS
.TA 12ch 20ch
.BD
\NAME\  MEANING
\ LH\start output for letterhead
\ DS\start a display
\ DE\end a display
\ EX\start an extract
\ XE\end an extract
.DE
.LP
These commands
may be freely intermixed with all the others,
in particular the heading and style control commands.
```

Here, and in the next several examples, the character '\' is used to indicate where a tab character appears in the input text. Instead of using default tabstops, we have set tabstops that will approximately center the table, to produce the following:

The following commands set up special formatting conventions:

NAME	MEANING
LH	start output for letterhead
DS	start a display
DE	end a display
EX	start an extract
XE	end an extract

These commands may be freely intermixed with all the others, in particular the heading and style control commands.

Notice that the **TA** command used to set the additional tabstops is given after the **DS** command that starts the display. This is because the default tabstops set by **DS** must be overridden by the new ones.

If this table appeared within indented text, the tabstops would be measured from wherever the indentation ended. If, for instance, we extended our example by placing the table within an indented paragraph, the tabstops would move to the right, because they would be measured from the indented left margin. Thus the input:

```
.IP "special formatting -"
the following commands set up special formatting
conventions:
.DS
.TA 12ch 20ch
.BD
\NAME\  MEANING
\ LH\start output for letterhead
\ DS\start a display
\ DE\end a display
\ EX\start an extract
\ XE\end an extract
.DE
.IP
These commands
may be freely intermixed with all the others,
in particular the heading and style control commands.
```

would result in the output:

special formatting - The following commands set up special formatting conventions:

NAME	MEANING
LH	start output for letterhead
DS	start a display
DE	end a display
EX	start an extract
XE	end an extract

These commands may be freely intermixed with all the others, in particular the heading and style control commands.

You can specify as many tabstops as you like, by giving each one as a separate argument to **TA**. Note, however, that **TA** clears any tabstops that were previously set before it establishes the new ones, so successive calls to **TA** will not keep adding tabstops to the ones already in effect. Giving a tabstop of zero clears all tabstops currently set.

You can specify tabstops in two other ways, as well. For tables with several columns, all fairly similar in width, the command **tabcols** can be used to break up the indented line length into equal sections, then set a tabstop at the start of each section but the first. You tell tabcols the number of columns you want; the first column begins at the indented left margin, and each later one at a tabstop. A table with four such columns could be nicely formatted with:

```
This history is summarized in the table following:
.DS
.tabcols 4
.BD
YEAR\REVENUE\GROSS PROFIT\NET PROFIT
.HS
1982\$10.8 million\$0.7 million\$0.2 million
1981\  5.2\ 0.9\ 0.6
1980\  2.3\ 1.2\ 0.9
.DE
```

for the output:

This history is summarized in the table following:

YEAR	**REVENUE**	**GROSS PROFIT**	**NET PROFIT**
1982	$10.8 million	$0.7 million	$0.2 million
1981	5.2	0.9	0.6
1980	2.3	1.2	0.9

Tables prepared using **tabcols** can be made snazzier by "indenting" the table, either by using **IN**, or by specifying more columns than you actually intend to use, then leaving an equal number of columns empty on each side of the table. The **tabcols** command always clears any tabstops already set.

The final way to specify a tabstop involves a special kind of command that we haven't seen before. This is an unnamed command. Like other commands, unnamed commands are used by typing the characters that always begin a command invocation: either a '.' at the start of a line, or the string "<:" anywhere at all. But instead of having a name like **LP** or **TA**, these commands are identified by a single special character immediately after the introductory string.

The unnamed command "<:^" ("set dynamic tabstop") sets a tabstop at the current position on the output line. This permits you to establish tabstops according to the text you happen to be entering, usually with respect to the longest entries in each column, then to let the other entries sort themselves out. This is especially convenient in tables where the headings (or other entries on the first line) are longer than the entries on the later lines. The table of commands presented earlier could easily be expanded by using these dynamic tabstops to indicate the start of each column:

```
The following commands set up special formatting
conventions:
.DS
.TA 0
    <:^NAME  <:^FILL MODE    <:^  MEANING
.HS
\ LH\   off\start output for letterhead
\ DS\   off\start a display
\ DE\   on\end a display
\ EX\   on\start an extract
\ XE\   on\end an extract
.DE
.LP
These commands
may be freely intermixed with all the others,
in particular the heading and style control commands.
```

In this form, the table would become:

The following commands set up special formatting conventions:

NAME	FILL MODE	MEANING
LH	off	start output for letterhead
DS	off	start a display
DE	on	end a display
EX	on	start an extract
XE	on	end an extract

These commands may be freely intermixed with all the others, in particular the heading and style control commands.

Note that the default tabs set up by **DS** had to be cleared with a ".TA 0" command in order for this to work; otherwise, the dynamic tabstops would just be added to the ones already set.

9.2. Using Tabstops

We have seen that tabstops may be set in a variety of ways. The ways they may be used are just as numerous. The traditional way of using a tabstop involves typing a "tab" character (ASCII HT) to advance to the next stop, then typing text that will be left-justified beginning at that stop. ctext does two things to improve on this approach.

First, ctext provides an alternate way of advancing to the next tabstop. The HT character, though handy to type, is hard to interpret once entered. With many -- if not most -- editors, an HT is often indistinguishable from a string of spaces. This makes tables using it hard to read, and harder still to modify. The fact that HT is a whitespace character can also make it difficult to get into your input file.

We have already used the character '\' to make tab characters "visible", and hence easier to read. ctext defines an unnamed command for the same

purpose. The command "<:\" ("advance to next tabstop") can be used instead of HT as a tab character. This command is (obviously) completely printable, and has no other meaning to anyone. The HT character and "<:\" are completely equivalent, but use of the "visible tab character" is encouraged.

The second improvement involves how a tabstop affects the text surrounding it. Since a tabstop is just a defined point on the output line, it can be used for things other than positioning the start of a string of text. A tabstop can also signal the right end of a right-justified string, or signal the endpoint of a space in which text is to be centered. Two more unnamed commands provide these capabilities. The command "<:+" ("begin right-justified text") is used to mark the start of text to be right-justified against the next tabstop. The command "<:/" ("begin centered text") is used to mark the start of text to be centered between the current output line position and the next tabstop. The text to be centered or right-justified is terminated by the tab command that advances to the tabstop.

Aside from letting you create fancy output, these commands also solve some basic problems in creating any table. For example, it is often desirable to center entries within a column. Usually, this is done by inserting spaces as needed in front of short entries. The command "<:/" provides a much easier way of getting a comely column. Here is what our growth history would look like if entered with the proper unnamed commands:

```
This history is summarized in the table following:
.DS
.tabcols 4
.BD
<:/YEAR<:/REVENUE<:/GROSS PROFIT<:/NET PROFIT
.HS
<:/1982<:/$10.8 million<:/$0.7 million<:/$0.2 million
<:/1981<:/5.2<:/0.9<:/0.6
<:/1980<:/2.3<:/1.2<:/0.9
.DE
```

Admittedly, the profusion of commands takes some getting used to. But with a bit of practice, this table is simpler to read and far easier to modify than the other ones presented. (Would you care to guess how many spaces preceded the the middle column in the last example?) This input also illustrates that both "<:+" and "<:/" will act as tab characters in certain contexts. If you use more than one of these commands on the same line, a later command will automatically terminate the text started by the last previous command, and advance to the next tabstop, before it marks the start of a new string of text. All of which is to say that, for example, instead of typing:

```
<:/1980<:\<:/2.3<:\<:/1.2<:\<:/0.9<:\
```

we were able to omit the tab character between each pair of "<:/", and get the same effect:

```
<:/1980<:/2.3<:/1.2<:/0.9
```

Here, at any rate, is what this input would produce:

This history is summarized in the table following:

YEAR	REVENUE	GROSS PROFIT	NET PROFIT
1982	$10.8 million	$0.7 million	$0.2 million
1981	5.2	0.9	0.6
1980	2.3	1.2	0.9

The obvious case in which "<:+" is useful is in right-justifying columns of figures. A simple price list, for instance, might be entered as:

```
.DS
.tabcols 3
.CE
.SH "FALL PRODUCE"
<:\Chestnuts<:+1.49<:\ pound
<:\Rutabaga<:+.59<:\ bunch
<:\Persimmons<:+.29<:\ each
<:\Squash<:+2.79<:\ dozen
<:\Warheads<:+57,000.00<:\ kiloton
<:\Silos<:+2,000.00<:\ cubic yd.
.DE
```

to yield the output:

FALL PRODUCE

Chestnuts	1.49	pound
Rutabaga	.59	bunch
Persimmons	.29	each
Squash	2.79	dozen
Warheads	57,000.00	kiloton
Silos	2,000.00	cubic yd.

Two final conventions serve to reduce the number of tab commands you need to enter. Any field of text (right or left-justified, or centered) terminates at the end of the output line, if it is not explicitly terminated before then. Also, the right margin is used as a final endpoint for right-justified or centered text, if no tabstop is left for that purpose. The following fairly simple input, for instance, would center two addresses in parallel, one in each half of the page:

```
.DS
.tabcols 2
.CE
.SH "WHERE TO FILE"
.BD
<:/If claiming a REFUND:<:/If making a PAYMENT:
.HS
<:/Mass. Income Tax<:/Mass. Income Tax
<:/P.O. Box 7000<:/P.O. Box 7003
<:/Boston, MA  02204<:/Boston, MA  02204
.DE
```

to produce the output:

WHERE TO FILE

If claiming a REFUND:	**If making a PAYMENT:**
Mass. Income Tax	Mass. Income Tax
P.O. Box 7000	P.O. Box 7003
Boston, MA 02204	Boston, MA 02204

9.3. Specifying Tab Fill Characters

We have covered how tabstops are specified, and how they are used to position text. There is, however, one more feature available in using them. Normally, the width that a tab character moves in advancing to a tabstop is filled with whitespace. Two last commands let you specify an arbitrary string that will be used instead to fill up the width of a tab. Both of these commands expect the fill string to be given as their first argument. If no further arguments appear, the string is used to fill the width of every tab. Or, you can give a series of tabstop positions, to specify that the string is to be used only when tabbing to the tabstops named.

The command **tabfill** causes the fill string to be inserted at the start of the width of the tab, and repeated until the width is filled. The command **tablead** does the same, except that the fill string is inserted at the first point on the output line that is a multiple of the length of the string. Thus fill strings line up on successive lines. To illustrate the effects of **tabfill**, consider the price list presented before:

```
.DS
.tabcols 3
.tabfill ". " 2
.CE
.SH "FALL PRODUCE"
<:\Chestnuts<:+1.49<:\ pound
<:\Rutabaga<:+.59<:\ bunch
<:\Persimmons<:+.29<:\ each
<:\Squash<:+2.79<:\ dozen
<:\Warheads<:+57,000.00<:\ kiloton
<:\Silos<:+2,000.00<:\ cubic yd.
.DE
```

Note that we asked for the fill string to be used only in tabbing to the second tabstop set, so that whitespace will still be used to fill up to the first tabstop. Notice, too, that in using **tabfill**, we changed nothing except to add the command. No additional information is needed on the line where the tabs occur. Here is the revised price list this input would produce:

FALL PRODUCE

```
Chestnuts. .  1.49 pound
Rutabaga. . .  .59 bunch
Persimmons. .  .29 each
Squash. . .   2.79 dozen
Warheads 57,000.00 kiloton
Silos. .  2,000.00 cubic yd.
```

The fill characters here do not line up vertically, which can be bothersome. Using **tablead** instead of **tabfill** will take care of that:

```
.DS
.tabcols 3
.tablead ". " 2
.CE
.SH "FALL PRODUCE"
<:\Chestnuts<:+1.49<:\ pound
<:\Rutabaga<:+.59<:\ bunch
<:\Persimmons<:+.29<:\ each
<:\Squash<:+2.79<:\ dozen
<:\Warheads<:+57,000.00<:\ kiloton
<:\Silos<:+2,000.00<:\ cubic yd.
.DE
```

and will produce this:

FALL PRODUCE

```
Chestnuts  .  1.49 pound
Rutabaga . . . .59 bunch
Persimmons . . .29 each
Squash . . .  2.79 dozen
Warheads 57,000.00 kiloton
Silos  .  2,000.00 cubic yd.
```

10. MACROS FOR PAGE CONTROL

We have noted already that ctext breaks the output document into fixed-length pages as it is created. Several commands permit you to control the appearance of the pages output. You also have the option of specifying headings, footings, or both, to appear on each page.

10.1. Changing Page Appearence

Each page output has an overall length. This length, in turn, is divided into three regions. Equal-sized areas are reserved at the top and bottom of the page for headings or footings, and the remaining area is used for the actual text of the document.

The macro **PL** is used to specify overall page length. The length is given as a vertical measurement. If the number given contains no unit indicator, it is taken to mean "lines". The default page length is 11in.

The macros **OS** and **IS** are used to specify the amount of space preceding or following a heading or footing. The command **OS** (for "outer space") gives the distance between a heading or footing and the extremity of a page; **IS** (for "inner space") gives the distance between a heading or footing and running text within the page. Both of these distances are given as vertical measurements; if a number given contains no unit indicator, it is taken to mean "lines". The default inner space is 2pc; the default outer space is 3pc.

The total page length and the space around headings are maintained independently of each other. Whenever the page length is changed, the space left for running text is altered to compensate, so that inner and outer space around headings stays constant. In the same way, if this inner or outer space is changed, the space left for running text is altered to compensate, so that the overall page length stays constant. As a special case, if both inner space and outer space are set to zero, all heading and footing output is suppressed, to produce a continuous column of running text.

ctext maintains a page number that it automatically increments as each page is output. By default, of course, this page number is initialized to one and counts up from there. The page number can be set explicitly, however, with the macro **PN**. Give the new number as the first argument to the macro. Note that the new number will take effect immediately, even if output is in the middle of a page. This can lead to surprising results.

Safer is to use the macro **BP**. **BP** checks to see whether output is currently at a page boundary. If not, **BP** causes a line break, then finishes up the current page with the old page number still in effect, advancing to a page boundary. Any argument given to **BP** is taken to be the page number of the new page. If no new page number is given, then one is added to the old number to get the new one.

The macro **FP** (for "first page") is provided as shorthand. This command is equivalent to ".BP 1".

10.2. Specifying Heading or Footing Contents

ctext maintains three pairs of headings and footings: one pair for the first page of a document, and separate pairs for even and odd-numbered pages other than the first page. Each of the three headings is set up independently, as is each footing.

By default, the standard macro package provides a chaste first-page footing consisting of the page number, centered, between hyphens. The same contents are provided for headings on even and odd-numbered pages.

First-page headings, and footings on later pages, are left blank.

A set of six macros allows you to specify fancier three-part headings or footings where appropriate. All of these macros operate the same way, and so will be discussed together. They are:

NAME	SPECIFIES
EF	even-numbered page footing
EH	even-numbered page heading
FF	first page footing
FH	first page heading
OF	odd-numbered page footing
OH	odd-numbered page heading

Each of these commands takes three arguments. The first argument specifies text that will appear at the left margin. The second argument names text to be centered, and the third names text that will appear right-justified at the right margin. An empty quoted argument may be used to leave one of these components blank while filling in later ones; if no arguments at all appear, the entire heading or footing will be blank.

For instance, these commands would set centered headings on all pages consisting of "Draft Reference":

```
.FH "" "Draft Reference"
.EH "" "Draft Reference"
.OH "" "Draft Reference"
```

And the following commands would set up footings consisting of a centered revision number on the first page, and a date at the outer margin on all pages:

```
.FF "" "Revision 4A-7" "19 Jan 83"
.EF "19 Jan 83"
.OF "" "" "19 Jan 83"
```

The text given for heading or footing contents may have markup embedded in it. In particular, you may use the command **pval** to get the current page number inserted in the heading or footing. There is one subtlety, however. If you just specify the markup as you normally would, it will be executed when the heading or footing command is first encountered. In order to delay the interpretation of markup until the heading or footing is output, you must use the "<:" ... ":>" notation, and must double the opening "<:". Thus the following commands would cause the current page number to be centered in even and odd-numbered page footings, in addition to the date we added earlier:

```
.EF "19 Jan 83" "<:<:pval:>"
.OF "" "<:<:pval:>" "19 Jan 83"
```

11. WHERE TO GO FROM HERE

The standard macro package provided with ctext contains a few additional macros that were not covered here. Once you have experience with the macros we have gone over, you may want to read the next essay (**Macros**), which is a detailed reference to the complete contents of the standard package. That essay also describes two additional macro packages that provide fancier headings than the ones we have discussed. One of the additional packages provides numbered headings, the other, headings for "Whitesmiths-style" manual pages.

The ctext processor itself also has capabilities that were beyond the scope of this introductory essay. We have mentioned in passing that **cth19** and **ctdbl** are really command files. They run the actual programs that process your input files. The first program they run is the formatter itself, which is a program named **ctext**. Then they run one of several device drivers that convert the formatter's intermediate file into an output file for a specific device. The script **ctdbl** runs the driver **dbl**; the script **cth19** runs the driver **vt52**. The formatter and drivers are all described in Section II of this manual, just as the command files are. The formatter ctext, in particular, has many command-line options that may be of interest. All are described there.

For more details on formatter features, as well as on topics like programming new macros or adding new device support, the reader is referred to the system reference (**Ctext**) later in this section.

NAME

Macros - the packaged ctext macro sets

FUNCTION

This essay summarizes the macros shipped with Edition 2.2 of ctext. These macros are grouped into three "packages": ctmac.s, the standard macro set used by the other packages; ctmac.h, a set of macros providing enhanced heading generation; and ctmac.w, a set of macros useful for producing "Whitesmiths-style" manual pages. The three macro packages are described in the order just given.

These descriptions necessarily presume some familiarity with a basic vocabulary of formatting terms. If you need to, read the introduction to ctext (**Intro**) before going further.

The description of each macro is in four parts. First, its name is given, followed on the same line by a summary of what it does. If the macro accepts arguments, a synopsis of them appears on the next line. Finally, an indented paragraph fully describes the macro's purpose, and to a lesser extent its operation.

Macro names (like **SP** or **LP**) always appear in boldface. Macro arguments (like <tag> or <indent>) are given in italics. An argument is always named by a "metanotion", that is, a mnemonic string enclosed in angle brackets, for which you substitute some actual value when giving the argument. If an argument is optional, it will appear inside square brackets on the synopsis line.

Many macro descriptions are followed by a paragraph marked "Internals". **Casual users should skip the "Internals" paragraphs; they are provided as an aid to experienced users who wish to customize the macro packages.** Here are documented many of the internal counters and strings used to set formatting parameters, so that they can be modified if desired. These also appear in boldface and are followed by their initial value (as **iIND** [initial value 5ch] or **iSC** [initial value <:b:>]).

A few conventions used throughout the packages are worth mentioning. First, any vertical whitespace not explicitly requested via **SP** is output using the <:cspace:> directive, meaning that if at least the whitespace desired already exists at the end of the current galley, no further space is output. Second, the descriptions here of whitespace generation are accurate only to the degree that the output device permits. For example, a macro described as outputting a half-line of space will output a full line to devices incapable of smaller vertical motions.

1. THE STANDARD MACRO PACKAGE

1.1. Paragraphing Macros

All of these macros assure that at least half a line of whitespace exists at the bottom of the current galley, clear any tabstops currently set, then assure that space remains on the current page for two lines of text.

1.1.1. LP - begin left-justified paragraph

Begin a paragraph in which the first and succeeding lines start at the left margin. This macro resets any indentation still in effect from the most recent **IP**.

1.1.2. PP - begin 'plain' paragraph

Begin a plain paragraph in which the first line alone is indented. This macro resets any indentation still in effect from the most recent **IP**.

Internals: The amount by which the first line is indented is given by the counter **iIND** [initial value 5ch].

1.1.3. IP - begin indented paragraph

[<tag>] [<indent>]

Begin a paragraph in which lines after the first are indented from the left margin. If <tag> is given, then it is placed at the start of the first line of the paragraph, beginning at the left margin. The rest of the first line of the paragraph is indented like the later lines. The amount of indentation is specified by <indent>, or, if <indent> is not given, defaults to 5ch.

Internals: <tag> is formatted in the stylecode named by string **iSC** [initial value <:b:>]. If <indent> is not given, the indentation for the paragraph is specified by the counter **iIN<n>** [initial value taken from **iIND**], where <n> represents the current indentation "level". The counter **iIN<n>** is updated from each <indent> argument specified.

1.1.4. IL - start new indentation level

[<indent>]

Specify the start of a nested set of indented paragraphs. By using **IL** and **IX** to start and end levels of nesting, you assure that the paragraphing macros will maintain a correct left margin throughout. If given, <indent> specifies the default indentation of future **IP** macros at the new indentation level.

If an **IP** macro is given with no explicit **IL** preceding it, an **IL** is performed automatically.

1.1.5. IX - exit indentation level

End the current level of indentation, by removing the indentation remaining from the latest **IP**.

1.2. Heading Macros

The standard package contains three macros that process headings. All of these assure that at least one line of whitespace exists at the bottom of the current galley, then output a heading followed by at least a half-line of whitespace, after assuring that two lines of text following the heading will fit on the current page. If the enhanced heading package is used, these macros will incorporate the additional features it provides (see

section 2). In any case, all of these macros remove any indentation remaining from the last previous **IP** before the heading is output.

1.2.1. SH - generate a 'section heading'

<heading> [<level>] [<before-mac>] [<after-mac>]

Create a highest-level heading. This macro assures that two lines of whitespace will precede the heading, and follows the heading with a full line of whitespace. The other arguments to **SH** are examined only if the enhanced heading package is used; they are then treated like the arguments given to **NH** (see section 2.1.1).

Internals: <heading> is output using the stylecode named by the string **hSC** [initial value <:b:>].

1.2.2. BH - generate a 'block heading'

<heading> [<level>] [<before-mac>] [<after-mac>]

Create a next-lower level heading. Assures that one line of whitespace will precede the heading, and follows the heading with a half-line of whitespace. The other arguments to **BH** are examined only if the enhanced heading package is used; they are then treated like the arguments given to **NH** (see section 2.1.1).

Internals: <heading> is output using the stylecode named by the string **hSC** [initial value <:b:>].

1.2.3. PH - generate a 'paragraph heading'

<heading> [<level>] [<before-mac>] [<after-mac>]

Create a lowest-level heading. Assures that a half-line of whitespace will precede the heading, and follows the heading with a half-line of whitespace. The other arguments to **PH** are examined only if the enhanced heading package is used; they are then treated like the arguments given to **NH** (see section 2.1.1).

Internals: <heading> is output using the stylecode named by the string **pSC** [initial value <:i:>].

Note: when the enhanced heading package is used, the macros **SH**, **BH** and **PH** use the counters **hBS** [initial value 1ln] and **hAS** [initial value .5ln] to determine how much whitespace comes before or after the heading, as follows. **SH:** **hBS** + 1ln (before), **hAS** + .5ln (after); **BH:** **hBS** (before), **hAS** (after); **PH:** **hBS** - .5ln (before), **hAS** (after).

1.3. Style Control Macros

Quite presentable documents can be created using no more than the macros already discussed. However, the standard package contains many macros designed to give you finer control over various aspects of your document's appearance. The first group of these selects stylecodes in a way sometimes more convenient than the usual procedure of naming stylecodes as commands in your input text. These macros have the advantage that they change the stylecode in effect for some number of input lines, then restore it to whatever it was previously.

1.3.1. BD - embolden lines of input text

[<line-count>]

Read a number of input lines, formatting them using stylecode <:b:>. If given, <line-count> specifies the number of input lines to be emboldened; otherwise, one line is read and emboldened. After the input lines are read, the current stylecode reverts to whatever it was beforehand.

1.3.2. IT - italicize lines of input text

[<line-count>]

Read a number of input lines, formatting them using stylecode <:i:>. If given, <line-count> specifies the number of input lines to be italicized; otherwise, one line is read and italicized. After the input lines are read, the current stylecode reverts to whatever it was beforehand.

1.3.3. RM - make lines of input text roman

[<line-count>]

Read a number of input lines, formatting them using stylecode <:r:>. If given, <line-count> specifies the number of input lines to be made roman; otherwise, one line is read and made roman. After the input lines are read, the current stylecode reverts to whatever it was beforehand.

Note that stylecode <:r:> is the one normally in effect; the macro **RM** is used to obtain roman text inside a larger body of text set in some other stylecode.

1.4. Line Control Macros

The next group of macros control the appearance of individual output lines in your document, by altering various galley characteristics.

1.4.1. FI - enter fill mode

Turn on horizontal filling, set up compaction of whitespace embedded in input lines, and set horizontal justification to "body" (i.e., block-justified except for the last lines of paragraphs, which are left-justified). This mode is in effect at the start of a run, and is the norm for formatting running text.

1.4.2. NF - enter no-fill mode

Turn off horizontal filling, preserve whitespace embedded in input lines, and set horizontal justification to "left". This mode is most often used for formatting tabular material.

1.4.3. AD - turn on line 'adjustment'

Set horizontal justification to "body", without affecting the state of filling.

1.4.4. NA - turn off line 'adjustment'

Set horizontal justification to "left", without affecting the state of filling.

1.4.5. CE - center lines of input text

[<line-count>]

Read a number of input lines, formatting them with horizontal filling turned off and horizontal justification set to "center". If given, <line-count> specifies the number of input lines to be centered; otherwise, one line is read and centered. After the input lines have been read, filling and justification revert to whatever they were beforehand.

1.4.6. LL - specify line length

<length>

Change the total length of output lines (including left-hand indentation) to <length>. If <length> starts with "++" it will be added to the existing line length to derive the new one, and a <length> starting with "--" will be similarly subtracted to obtain the new value. If <length> does not end with a units indicator, it is taken to be in characters ("ch").

1.4.7. IN - specify left-hand indentation

<indent>

Set indentation from the left margin. If <indent> starts with "++" it will be added to the existing indentation to derive the new one, and an <indent> starting with "--" will be similarly subtracted to obtain the new value. If <indent> does not end with a units indicator, it is taken to be in characters ("ch").

1.4.8. TA - set tabstops

[<tabstop-list>]

Clear any existing tabstops and set new ones. If given, <tabstop-list> names a series of points where tabstops are to be set. Each point is given as a measurement from the current start of the output line, that is, a measurement taken from the current left-hand indentation. If no tabstops are given, the old ones are simply cleared.

1.4.9. TD - set default tabstops

Clear existing tabstops and set a default series of stops, in every fourth column through column 36; columns are counted from the current left-hand indentation.

1.4.10. LS - specify line-spacing ratio
<ls-ratio>

Specify the spacing, baseline to baseline, of adjacent lines of output text. The value <ls-ratio> is used as a proportion of the current text size, so no spacing between lines is specified with a value of one (and an <ls-ratio> of two specifies double-spacing, and so on).

1.4.11. SZ - specify text size
<size>

Specify the size of output text; this is meaningful only for devices that can scale a devicestyle to more than one size. The value <size> is given as a vertical measurement; if it does not end with a units indicator, it is taken to be in points ("pt").

1.4.12. HY - set hyphenation control
<hy-length>

Set the minimum number of characters in a hyphenated portion of a word, not including any hyphen added during hyphenation. If <hy-length> is zero, hyphenation is disabled.

Note: All of the line control macros except **HY** and **SZ** cause a "line break" when used; that is, any pending output line will be output, and a new line started, before the macro is executed. A line break may be prevented by using the macro **NB** (see section 1.9.3) in front of the macro that would cause the break.

1.5. Galley Control Macros

These macros cause some special action to occur in the current galley, usually involving the output of vertical whitespace.

1.5.1. NE - check space remaining on current page
[<distance>]

Assure that vertical space of at least <distance> exists on the current page (i.e., assert your "need" for that much space); if not enough space remains, start a new page. This macro does a line break before checking the remaining space. If <distance> is not given, space one line in length is checked for. If <distance> does not end with a units specifier, it is taken to be in lines ("ln").

1.5.2. BR - cause a line break

Break the current output line; that is, if a partial line is pending, output it.

1.5.3. HS - output a 'half-line' of whitespace unconditionally

Output a half-line of whitespace. When the output device does not permit this, a full line of whitespace results.

1.5.4. SP - output whitespace unconditionally

[<distance>]

Output vertical whitespace of length <distance>. If <distance> does not end with a unit indicator, it is taken to be in lines ("ln"). If <distance> is not given, one line of whitespace is output.

1.5.5. CS - output whitespace conditionally

[<distance>]

Check that whitespace of at least length <distance> exists at the end of the current galley. If not enough does, output as much as is needed. If <distance> does not end with a unit indicator, it is taken to be in lines ("ln"). If <distance> is not given, one line of whitespace is checked for.

1.6. Special Formatting Macros

Other macros are provided that use the galley control macros, among others, to set up special formatting parameters. These are a convenient shorthand for performing frequently-needed operations.

1.6.1. DS - start a 'display'

Prepare to output lines of unfilled text, exactly as they appear in the input. This macro assures that a half-line of whitespace exists at the end of the current galley, assures that two lines of text will fit on the current page (by calling **NE**), sets up default tabstops (by calling **TD**), then enters no-fill mode (by calling **NF**).

1.6.2. DE - end a 'display'

End a set of lines output literally from the input. This macro simply re-enters fill mode (by calling **FI**).

1.6.3. EX - start an 'extract'

Start a body of filled text set off from the surrounding text (an extended quotation, for instance). This macro assures that a half-line of whitespace exists at the end of the current galley, assures that two lines of text will fit on the current page (by calling **NE**), increases left-hand indentation (by calling **IN**), then decreases line length by the same amount (by calling **LL**). By default, this amount is 5ch.

Internals: the amount by which the line is shortened on each side is given by the counter **iIND** [initial value 5ch].

1.6.4. XE - end an 'extract'

End a body of filled text set off from the surrounding text. This macro increases line length, decreases left-hand indentation by the same amount, then assures that a half-line of whitespace exists at the end of the current galley. By default, the amount of change is 5ch.

Internals: the amount by which the line is lengthened on each side is given by the counter **iINO** [initial value 5ch].

1.6.5. LH - start output for 'letterhead'

Set up for output on letterhead stock. This macro establishes default parameters by calling **RS** (see section 1.9.1), turns off any first-page footing by calling **FF** (see section 1.8.3), enters no-fill mode (by calling **NF**), begins a new page numbered one by calling **BP** (see section 1.7.5), then outputs eight lines of whitespace.

1.7. Page Control Macros

Several macros are provided to control the appearance of pages, and to advance conditionally to a page boundary.

1.7.1. PL - specify page length
<length>

Specify the total length of output pages, including any headings or footings. The value <length> is a vertical measurement; if <length> does not end with a unit specifier, it is taken to be in lines ("ln").

1.7.2. IS - specify 'inner space' for heading or footing
<distance>

Specify the amount of whitespace between a heading or footing and the text of an output page. The value <distance> is a vertical measurement; if <distance> does not end with a unit specifier, it is taken to be in lines ("ln"). If both the "inner space" and the "outer space" around headings and footings are set to zero, heading/footing output is suppressed.

1.7.3. OS - specify 'outer space' for heading or footing
<distance>

Specify the amount of whitespace between a heading or footing and the extremity of an output page. The value <distance> is a vertical measurement; if <distance> does not end with a unit specifier, it is taken to be in lines ("ln"). If both the "inner space" and the "outer space" around headings and footings are set to zero, heading/footing output is suppressed.

1.7.4. PN - specify page number
<number>

Use <number> as the number of the current page.

1.7.5. BP - begin new page
[<number>]

Assure that output is at a page boundary, then use <number> as the number of the new (empty) page. If output was not initially at a page boundary, the old page is finished with the previous page number still in effect.

1.7.6. FP - begin first page

Assure that output is at a page boundary, then begin a page numbered one. This macro is equivalent to a **BP** with a <number> of 1.

1.8. Heading/Footing Macros

This group of macros provides three-part headings and footings. Three different sets of headings and footings are maintained: for a "first page" whose number is specifiable at runtime, for even-numbered pages other than the "first page", and for odd-numbered pages other than the "first page". The text given to all of these macros may contain embedded markup. Note, however, that markup given directly will be interpreted when the heading or footing macro is first encountered; to delay the interpretation of markup until the heading or footing is used, double its command open sequence.

1.8.1. EF - specify even-numbered page footing

[<left>] [ ] [<right>]

Use the three arguments given as components of footings on even-numbered pages other than the "first page". The string <left> will be output starting at the left margin, while will be centered and <right> will be right-justified against the right margin. Any of the three may be omitted, or (if a later argument needs to be given) may be specified as a null string; the corresponding part of the heading will then be blank.

1.8.2. EH - specify even-numbered page heading

[<left>] [ ] [<right>]

Use the three arguments given as components of headings on even-numbered pages other than the "first page", as described under **EF**.

1.8.3. FF - specify 'first page' footing

[<left>] [ ] [<right>]

Use the three arguments given as components of footings on the "first page" given at runtime, as described under **EF**.

1.8.4. FH - specify 'first page' heading

[<left>] [ ] [<right>]

Use the three arguments given as components of headings on the "first page" given at runtime, as described under **EF**.

1.8.5. OF - specify odd-numbered page footing

[<left>] [ ] [<right>]

Use the three arguments given as components of footings on odd-numbered pages other than the "first page", as described under **EF**.

1.8.6. OH - specify odd-numbered page heading

[<left>] [ ] [<right>]

Use the three arguments given as components of headings on odd-numbered pages other than the "first page", as described under **EF**.

1.9. Miscellaneous Macros

1.9.1. RS - reset formatting parameters to defaults

Re-establish the parameters in effect at the start of the run (this macro is called automatically whenever the standard package is read). These parameters are: first-page footing, and odd and even-page headings, consisting of the page number centered between hyphens; no first-page heading or odd or even page footings; an indentation of 0; default tabstops; a line length of 6.5in; a page length of 11in, with inner space of 2pc and outer space of 3pc; fill mode turned on; end-of-sentence processing turned on for ".?;:!"; and leading whitespace and blank lines both set to "break".

1.9.2. SO - use new file as input source

<filename>

Begin reading input from <filename>. The old input file remains stacked beneath the new one; when EOF is reached on <filename>, input is again read from the old file. Input files may be stacked to arbitrary depth.

1.9.3. NB - suppress next line break

Suppress the next line break that would normally be generated by the execution of a command.

1.10. Internal Counters and Strings

The following counters and strings are used to parameterize various values needed by the standard macros. Any of these may be altered as needed.

1.10.1. iIND - default indentation value

Specify the default indentation used by the **IP**, **PP**, **EX** and **XE** macros. This counter is initialized to 5ch.

1.10.2. iIN<n> - default indentation for current indent level

Specify the default indentation for indent level <n>, where <n> counts up from one. This counter is initialized from **iIND** whenever a new indent level is entered, and is updated from each explicit <indent> given to the macro **IP**.

1.10.3. iSC - default stylecode for IP tags

Specify the stylecode used to output each <tag> given to the macro **IP**. This string contains an invocation of the stylecode, and is initialized to "<:b:>".

1.10.4. hSC - default stylecode for headings

Specify the stylecode used to output headings with the macros **SH** and **BH**. (This string is also used by the enhanced heading macros.) Contains an invocation of the stylecode, and is initialized to "<:b:>".

1.10.5. pSC - default stylecode for PH

Specify the stylecode used to output paragraph headings with the macro **PH**. This string contains an invocation of the stylecode, and is initialized to "<:i:>".

2. THE ENHANCED HEADING MACRO PACKAGE

2.1. User Definitions

All of the macros in this package are oriented toward numbered headings, in which a counter is maintained for headings at each of an arbitrary number of levels, and is incremented and output with each successive heading. However, if the enhanced heading package is used, the three heading macros from the standard package (that is, **SH**, **BH** and **PH**), are redefined to take advantage of its level administration features.

The highest level is numbered one, and "lower" levels count up from there. Counters for each level are output from level one to the "current" level. Changing to a higher-numbered level resets the highest-numbered counter to zero. The counter at the current level is always incremented before the heading is output.

2.1.1. NH - generate a numbered heading
<heading> [<level>] [<before-mac>] [after-mac]

Create a numbered heading. If given, <level> specifies the level of the heading; otherwise, the level of the latest heading output is maintained. The heading counter at the current level is then incremented. If given, <before-mac> names a macro to be executed before any processing occurs for the heading (except that the level will have been changed and the correct counter incremented beforehand). If given, <after-mac> names a macro to executed after all processing for the heading is complete. **NH** assures that at least two lines of text after the heading will fit on the current page, then, by default, assures that one line of whitespace will precede the heading, and follows the heading with a half-line of whitespace.

Any indentation remaining from a previous **IP** macro is removed before the heading is output. Also, if **NH** is used with no explicit **NL** preceding it, an **NL** is executed automatically.

Internals: the whitespace preceding the heading is indicated by the counter **hBS** [initial value 1ln]. The stylecode used to output <heading> is named by the string hSC. The whitespace indicated by the counter **hAS** [initial value .5ln] will follow the heading.

The string **hHD** is set to the contents of the heading output (including the heading number). The string **hBM<n>**, where <n> is the current heading level, is set to contain an invocation of <before-mac>, while **hAM<n>** is analogously set to contain an invocation of <after-mac>. This means that calls to **NH** at a given <level> automatically use the <before-mac> and <after-mac> most recently set up by a previous call at that level.

The string output for the heading number may be modified. Text to be output immediately before each number is contained in the string **hBT** [initial value undefined], while text to be output immediately after each number is contained in the string **hAT** [initial value ". "].

2.1.2. NP - begin a numbered paragraph

[<tag>] [<indent>] [<level>] [<before-mac>] [<after-mac>]

Begin a paragraph in which lines after the first are indented from the left margin, and which starts with a "paragraph number" derived from the same counters as the heading numbers used by **NH**. The current number, followed by any <tag> given, is placed at the start of the first line of the paragraph, beginning at the left margin. The rest of the paragraph is formatted as for the macro **IP** (see section 1.1.3). The last three arguments to **NP** are used like the corresponding arguments to **NH**; <tag> is processed in the same way as the **NH** <heading>, except obviously no whitespace is left after it.

Note, too, that **NH** and **NP** share the same counters, so that calls to the two may be intermixed (though for stylistic reasons this is not usually done on the same level).

2.1.3. NS - set heading counters explicitly

[<ctr-list>]

Assign explicit values to the counters named in <ctr-list>; the next heading that uses the changed counters will use the values given here. The first counter named corresponds to level one, the second to level two, and so on. Counters omitted from the list are not changed; a counter may be skipped by putting a hyphen "-" in the list position corresponding to it.

2.1.4. NL - start new heading level

Specify the start of a set of headings at a lower level. This macro and **NX** provide an alternative to giving explicit level numbers in each call to **NH** or **NP**. Using **NL** has the same effect as giving a <level> to **NH** or **NP** of one more than the current level.

2.1.5. NX - exit heading level

Specify the end of a set of headings at a lower level. This macro and **NL** provide an alternative to giving explicit level numbers in each call to **NH** or **NP**. Using **NX** has the same effect as giving a <level> to **NH** or **NP** of one less than the current level.

2.2. Internal Counters and Strings

2.2.1. hAM<n> - specify macro for use after heading at level <n>

Give an invocation of the macro that is to be run after each heading is processed at level <n>. This string is initially undefined.

2.2.2. hAS - specify spacing after heading

Give the spacing to be performed after a heading output by **NH**; this also affects the macros **SH**, **BH** and **PH** from the standard package (see section 1.2.3). This counter has an initial value of .5ln.

2.2.3. hAT - specify text suffix to heading numbers

Give the text that is immediately to follow each number string output by **NH** or **NP**. This string is initialized to ". ".

2.2.4. hBM<n> - specify macro for use before heading at level <n>

Give an invocation of the macro that is to be run before each heading is processed at level <n>. This string is initially undefined.

2.2.5. hBS - specify spacing before heading

Give the spacing to be performed before a heading output by **NH**; this also affects the macros **SH**, **BH** and **PH** from the standard package (see section 1.2.3). This counter has an initial value of 1ln.

2.2.6. hBT - specify text prefix to heading numbers

Give the text that is immediately to precede each number string output by **NH** or **NP**. This string is initially undefined.

2.2.7. hHD - record the last heading output

Record the most recent heading output with any of the heading macros (that is, **NH**, **SH**, **BH** or **PH**), or with **NP**. This string will contain the complete text of the heading, including any heading number. It is initially undefined.

2.2.8. hSC - default stylecode for headings

Specify the stylecode used to output headings with the macros **NH** and **NP**. (This string is also used by the standard heading macros.) Contains an invocation of the stylecode, and is initialized to "<:b:>".

3. THE WHITESMITHS HEADING MACRO PACKAGE

3.1. String Definitions

The first component of this package is a set of strings that may be used as shorthand for commonly occurring words or phrases. The descriptions following simply show the name of each string and the word or words it expands to; all of the words are preceded by soft hyphens, so will never be broken across two output lines.

3.1.1. A - Agreement

3.1.2. C - Customer

3.1.3. F - User Manual

This string is used by the macro **NM** as part of a first page heading (see section 3.2.2).

3.1.4. S - Supplier

3.1.5. W - Whitesmiths, Ltd.

3.2. Heading Definitions

The other component of this package is a set of macros that generate the standard headings used in Whitesmiths manual pages, together with the basic macro that does the formatting for all the particular headings. After this basic macro, the remaining macros are presented in the same order as the corresponding manual page sections would appear.

3.2.1. HD - generate a manual page heading
<heading>

Create a heading as for a Whitesmiths manual page. This macro assures that one line of whitespace exists at the end of the current galley, takes indentation 4ch to the right, then outputs <heading> in stylecode <:b:>. Indentation is then returned to its previous position.

3.2.2. NM - specify name for and initialize manual page
<name> <description>

Begin processing for a manual page. This macro resets formatting characteristics to their initial defaults (by calling **RS**), then clears all footings and sets up headings as follows: first page heading becomes <name> at each margin, and the contents of string **F**, centered. Even and odd page headings consist of <name> at each margin, and the current page number, between hyphens, centered. The macro then advances to a page boundary and sets the page number to one. Indentation is set to 4ch, then **HD** is called to output "NAME". Finally, a summary line is output consisting of <name>, the string " - ", and <description>, which consists of the remaining arguments to the macro. Fill mode is then turned on.

This macro should always be used before any of the other heading macros in this package, since it alone performs the initialization they all require.

3.2.3. SY - output a synopsis heading

Call HD to output "SYNOPSIS", and enter no-fill mode.

3.2.4. FU - output a function heading

Call HD to output "FUNCTION", and enter fill mode.

3.2.5. RT - output a returns heading

Call HD to output "RETURNS", and enter fill mode.

3.2.6. EG - output an example heading

Call HD to output "EXAMPLE", and enter no-fill mode.

3.2.7. FL - output a files heading

Call HD to output "FILES", and enter fill mode.

3.2.8. SA - output a see also heading

Call HD to output "SEE ALSO", and enter fill mode.

3.2.9. BU - output a bugs heading

Call HD to output "BUGS", and enter fill mode.

3.2.10. CO - output a course outline heading

Call HD to output "COURSE OUTLINE", and enter fill mode.

3.2.11. PR - output a prerequisites heading

Call HD to output "PREREQUISITES", and enter fill mode.

3.2.12. SS - output a price heading

Call HD to output "PRICE", and enter fill mode.

NAME

Ctext - a processor for text

FUNCTION

ctext is a user-programmable system for creating formatted text suitable for reproduction by a variety of output devices. The system consists of two parts. At its base is a formatter that reads input text interspersed with formatting commands, or markup, which is read according to user-settable parameters. This formatter produces a device independent stream of graphics, cursor motions, and output control sequences, which is interpreted by one of a set of drivers for specific output devices, such as displays, line printers, incremental printers, or phototypesetters.

1. OVERVIEW

ctext produces output documents consisting of a series of pages, each containing a single column, or galley, of running text, optionally bordered by a heading, a footing, or both. The length and width of pages, and the contents and appearance of headings and footings, are user-specifiable. Markup contained in the input text falls into two classes. One class consists of predefined directives that cause some action internal to ctext, like changing a galley parameter, or changing some more general formatting characteristic. The other kind of command names a user-defined object. The most common of these are macros, that when named cause the insertion of text (perhaps containing further, "embedded" markup) into the input stream of the galley. User-defined commands are named exactly as predefined directives are, so that the user need not concern himself with the origin of the markup he happens to be using.

All markup has a tightly-defined format. By default, markup begins with the two-character sequence "<:", and ends with the sequence ":>". Everything in between is interpreted by ctext as a command. The user can specify his own command formats, either in addition to or instead of this default format. Markup may but need not be constrained to begin in the first column of an input line (like the traditional '.' commands of earlier processors). Markup may also be terminated by the end of an input line, instead of by a second special sequence of characters. Markup commands may take arguments, with the argument list parsed according to further user-specified parameters.

After all command processing has occurred, the resulting input text (perhaps now containing special characters denoting local formatting operations) is further modified to produce a "token stream", in which potentially ambiguous text strings (blank lines, leading whitespace) are deleted, passed through, or translated into explicit command sequences, as requested by parameters set in the galley.

This stream is then fed into the linotype associated with the galley. The linotype sets a single line of text, perhaps with hyphenation and justification, into a two-dimensional region called a box, and passes it back to the galleymaker, which currently just outputs it to STDOUT, creating a new page if the current one was already full.

2. MARKUP

The format of the markup used in manuscript files is highly user-configurable; nearly all of its essential characteristics can be defined by means of the <:init:> command, explained below. Each markup format given can be described by the following productions:

```
<command>      ::= <copen-seq> <name> [ <arglist> ] <cclose-seq>
<arglist>      ::= <aopen-seq> <aterm-string>
                       [ <arg> <aterm-string> ]* <aclose-seq>
<arg>          ::= <quoted-arg> | <simple-arg>
<quoted-arg>   ::= <qopen-seq> <char-sequence> <qclose-seq>
<simple-arg>   ::= <char-seq-no-aterm>
<aterm-string> ::= <aterm-seq> [ <aterm-seq> ]*
```

A <name> is always a string of letters, digits, and underscores, starting with a letter or underscore. A given markup format must define exactly one <copen-seq>, and must define a matching number of aopen/aclose sequences and qopen/qclose sequences. A format must define one or both of an <aclose-seq> and a <cclose-seq>. (The interrelation of the two is explained under the <:init:> command, below.) And naturally, a <simple-arg> is considered to end at the first <aterm-string> encountered, or at the end of the argument list, whichever is reached first. Any of the delimiting sequences indicated can be an arbitrary string of characters. In addition, more than one instance of a given sequence can be part of the same markup format. (Thus, the default format defines any one of the characters "([{" as <qopen-seq> sequences, and any one of ")]}" as <qclose-seq> sequences.) Multiple calls to <:init:> may be used to specify alternate command formats; any of the formats will then be recognized.

Note that ctext will always try to complete a low-level construct before trying to complete those above it. Thus a quoted argument, for instance, may contain instances of <aclose-seq> and <cclose-seq>; until a <qclose-seq> is encountered, they will not be recognized. Also, all delimiting sequences except copen/cclose nest to arbitrary depth.

However, despite the considerable variety of markup this scheme can handle, it does have its limitations. For example, it does not allow the specification of troff escape sequences, which ctext will not directly support. (Instead, a filter is used to convert them into a more regular format that ctext can interpret.)

3. EXPRESSIONS

3.1. Numeric Constants

An expression may be used in any context where a numeric value is expected. Ultimately, an expression may involve only constant operands; however, as input it may contain references to numeric variables called counters, or other commands that eventually evaluate to strings legal in the context of the expression.

Numeric constants can be divided into "measurements" and ordinary numbers. A measurement is given by following a constant with a units indicator, in-

dicating in what linear units the value of the constant is expressed. Measurements are used to specify distances, and are immediately converted into device-dependent horizontal or vertical raster units, depending on what direction the distance extends in. (A raster unit is the smallest motion of which an output device is capable, either horizontally or vertically.) Constants with no units indicator suffer no conversion. Thus measurements generally should not be combined with other constants, and should be used only in expressions measuring distances. Distances measured downward or rightward are expressed as positive, while those measured upward or leftward are negative.

The allowable unit indicators follow:

CODE	UNIT EXPRESSED
cm	centimeters
em,qd	em spaces (or quads)
en,ch	en spaces (or "characters")
gw	galley width (see below)
hr	device horizontal rasters
in	inches
ln,sr	"lines" (as affected by current <:spread:> value)
pc	picas
pt	points
sp	current vertical spacing (set by <:spacing:> alone)
vr	device vertical rasters

The indicator "gw" specifies a distance relative to the width of the current galley (e.g., .1gw names a distance one-tenth of the galley width).

3.2. Grammar

The grammar used for expressions is largely a subset of that used by the language C:

```
<arith-expr> ::= [ { -- | ++ } ] <log-expr>
<log-expr>   ::= <log-term> [ '||' <log-term> ]*
<log-term>   ::= <log-factor> [ && <log-factor> ]*
<log-factor> ::= <bit-term> [ '|' <bit-term> ]*
<bit-term>   ::= <bit-factor> [ & <bit-factor> ]*
<bit-factor> ::= <rel-term> [ { == | != } <rel-term> ]*
<rel-term>   ::=
                 <rel-factor> [ { '<' | <= | >= | '>' } <rel-factor> ]*
<rel-factor> ::= <term> [ { + | - } <term> ]*
<term>       ::= <sfactor> [ { * | / | % } <sfactor> ]*
<sfactor>    ::= [ { ! | ~ | + | - } ] <factor> | ( <log-expr> )
<factor>     ::= <constant>
```

The one deviation from the standard C grammar is that an expression beginning with a "++" or a "--" will be used to increment or decrement a counter, where appropriate, instead of replacing the counter's previous value. These operators, if present, do not affect expression evaluation.

Though the expression parser will handle arbitrary whitespace between tokens, an expression generally must appear as a single argument in a command invocation, and so must be quoted. The effect of expression evaluation is to replace the text of the expression with the numeric string representing its value.

3.3. String Constants

Given rules for command invocation as flexible as those of ctext, the notion of a "quoted string" loses meaning. ctext does, nonetheless, permit special string constants in known contexts. The only significant difference between such constants and any other text is that string constants may contain escape sequences to represent byte-sized values not representable as printable ASCII characters.

These sequences are a superset of those defined in the language C, namely a backslash followed by one of: a standard upper-case mnemonic for an ASCII "control character", or one of the letters "btnvfr" (to represent ASCII BS, HT, LF, VT, FF, CR), or one to three digits representing the value of the byte. A digit string is interpreted in octal if preceded by a '0', as decimal otherwise. The recognized mnemonics for control characters are described in Appendix D. Thus within a string constant "\n", "\LF", "\012", or "\10" all represent a single newline character. A '\' preceding any string that cannot be taken as one of the above causes the character after the '\' to be taken literally, so that "\\" may be (and should be) used to represent a literal backslash safely in any context.

String constants may appear in expressions only as arguments to the builtin directive <:ordstr:>, which is explained below.

4. FONTS

A font, as ctext uses the term, is a series of n-to-1 mappings between ASCII characters in the user's input file, and specific graphics that an output device is capable of generating. This mapping proceeds in three distinct stages. First, the user specifies a devicestyle, a set of symbols all available on the same physical medium on the output device. Next, he specifies a charset, a mapping from ASCII input characters to symbols defined in one or more devicestyles. Then, he specifies (what ctext calls) a font, which names a set of correspondences between charsets and stylecodes. A stylecode, in turn, is usually a standard single-letter code used to access analogous charsets in different fonts simply by changing the current font. Finally, a series of fonts may be grouped together in a fontset, so that analogous fonts in different sets may be accessed simply by changing the current fontset. In more detail:

4.1. Specification

Most of the information described above is provided to ctext by table files, files of a fixed format that follow consistent naming conventions. (The conventions are described in Appendix A.) A table file always begins with a <:mark:> directive, followed by lines of file-specific information. More than one <:mark:> directive may (and sometimes must) appear in a single file; each one must be given on a line by itself.

4.2. Devices

In order to format text for a device, ctext must be informed of a small set of device specifications. These are given by means of a ".ds" file, whose name corresponds to the device name given by the user in a

<:device:> command. A specification file consists of a <:mark:> directive of the form:

<:mark devicespec <name>
<hraster> <vraster> <size-expr> <char-list>:>

where <name> is the name of the device, <hraster> and <vraster> specify (in some known units) the size of horizontal and vertical "rasters" (smallest specifiable motions) for the device, <size-expr> is an arithmetic expression relating "relative character widths" to horizontal motion output, and <char-list> is zero or more of the following sequences of arguments:

fixed - specifies that this device cannot scale a single devicestyle to more than one size (and hence that the <:size:> directive explained below should be ignored). This argument should always appear in the devicespec for any device to which it applies, since size changes can otherwise lead to undesirable results.

letter <code> - specifies the letter identifying devicestyle, charset, and fontset files associated with this device. If not given, the letter is taken to be the first character of the device name.

pcwid - specifies that the actual width of each output symbol is to be sent to the output device, in addition to the symbol itself. The width is sent as a two-byte integer (counted in device horizontal rasters), less significant byte first. The width precedes the output for the associated symbol. If pcwid does not appear, no per-character width information is output.

<size-expr> is an expression involving the relative character widths defined in devicestyle files for the device, and is used to determine the actual width (in device horizontal rasters) of each output symbol. In this expression only, the directive "<:sz:>" may be used to refer to the point size current at the time the output is generated, and "<:wd:>" to refer to the relative width of the current output symbol. Currently, only two cases are recognized: <:wd:> (to make actual width equal relative width), and <:wd:>*<:sz:> (to multiply relative width by point size in deriving actual width).

4.3. DeviceStyles

A devicestyle corresponds to the traditional notion of a type style; that is, a single set of symbols present on the same physical medium on an output device, like a Diablo typewheel, or one of the styles available on a photocomposer type disk. A devicestyle is specified by means of a ".xd" file, where x is a device-specific letter, and 'd' signifies "devicestyle". A devicestyle file starts with a <:mark:> directive of the form:

<:mark devicestyle <df-name> <char-list>:>

where argument 1 is the keyword "devicestyle", argument 2 is its full name, and succeeding arguments specify zero or more of the following:

baseline <offset> - specifies the distance from the top of a character cell to the baseline of the style; <offset> is a proportion of the size of the style (e.g., .75 would specify a baseline three-fourths of the way "down" each character cell). The default is 1.

bold - identifies this style as bold.

final <string> - specifies a string of characters to be placed in the output whenever this style is left.

init <string> - specifies a string of characters to be placed in the output whenever this style is entered.

italic - identifies this style as italic.

like <df-name> - defines this devicestyle as containing the same symbol definitions as devicestyle <df-name>, which must already have been encountered. The lines describing each symbol in this devicestyle can then be omitted.

size <value> - specifies the height of a character cell in the current style, overriding the <:size:> characteristic of the galley in which the style is used; <value> should be given as a measurement. If omitted here, then the style size is determined by the galley. In general, size should be given in the ".xd" file if and only if a device is incapable of scaling a devicestyle to more than one size.

spacing <offset> - specifies the distance (baseline to baseline) between successive rows of text set in this style, as a proportion of the current style size, overriding the <:spacing:> characteristic of the galley in which it is used (e.g., 1.25 would add space between lines of one-fourth the size of the font). If omitted here, then the spacing is determined by the galley.

Following the <:mark:> directive must be lines describing each symbol in the devicestyle. Each line has the format:

<index> <width> [<output> [<repeat-count>]]

where <index> is the ASCII input character corresponding to a given devicestyle symbol, <width> is the relative width of the devicestyle symbol, and <output> specifies a series of bytes to be sent to the device in order to generate the symbol. If <output> is omitted, it is presumed to equal <index>.

Since devicestyles sometimes specify long linear correspondences between devicestyle indices and output sequences, any line may contain an optional fourth field, to specify that <repeat-count> indices are to map onto <output>, with its last byte incremented by one for each successive index. In this case, <width> is presumed to be identical for each output string. <index> and <output> are treated as string constants, while <width> and <repeat-count> are scanned as integers.

To avoid the repetition of identical sets of devicestyle indices for different devicestyles, the keyword "like" can be used in later devicestyles to refer to the first one defined.

A single set of devicestyle indices may also be "shared" by placing all the devicestyle <:mark:> commands that refer to the set immediately in front of it. So long as all of the <:mark:> commands are on successive

lines, any of them will be taken to refer to the single set of indices. (This may only be done if all the devicestyle names map to the same table file -- see Appendix A.)

4.4. CharSets

The charset mechanism is meant to remedy the fact that devicestyles, as defined by a hardware manufacturer, may not themselves contain sets of characters optimal for a given application. A charset is defined by a ".xc" file, where x is a device-dependent letter and 'c' signifies "charset". A charset file begins with a <:mark:> directive of the form:

<:mark charset <cs-name>:>

where argument 1 is the keyword "charset", and argument 2 is the name of the set.

Following the <:mark:> directive must be lines describing each character in the charset, of the form:

<char> <index> <df-name> [<repeat-count>]

where <char> is the ASCII input character to be mapped, <index> is the ASCII index in a devicestyle to which <char> is to be mapped, and <df-name> is the name of the target devicestyle, as given in a devicestyle file. Since charsets often specify long linear correspondences between input characters and devicestyle characters, any line may contain an optional fourth field, to specify that <repeat-count> charset members are to be mapped onto successive elements of the devicestyle.

4.5. Fonts

ctext fonts, alone among the components of the font mechanism, are not specified by table files, but are set up by means of the ordinary directive <:deffont:>. A font establishes a correspondence between a series of charsets and a series of stylecodes, standard single-letter directives that may be used in input text to request generic typeface changes, regardless of what font is currently in use.

A <:deffont:> directive has the format:

<:deffont <font-name> <code-pair> [<code-pair>]*:>

where argument 1 is the name of the font being defined, and each succeeding pair of arguments names a single stylecode/charset correspondence, in the form:

<stylecode> <cs-name>

where <stylecode> may be one of the standard codes given below, and <cs-name> names a charset defined in a ".cs" file. Note, however, that any identifier may be declared to be a <stylecode>. The standard ones follow:

b - requests boldface printing.

i - requests italic printing.

r - requests ordinary (roman) printing.

s - requests bold italic printing.

4.6. FontSets

A fontset is used to group a set of fonts together under a common name, so that identically-named fonts from different sets can be used simply by changing the fontset selected. Fontsets are designed to promote the consistent use of whole series of fonts, by enabling a font name to be assigned according to the role the font plays in the document (e.g., TextFont, HeadingFont), then associated with a specific devicestyle family (e.g., Times Roman, Helvetica) only by the <:fontset:> directive, so that nothing else need be altered to change the family selected.

A fontset is defined by a ".xf" file, where x is a device-dependent letter, and 'f' signifies "fontset". A fontset file begins with a <:mark:> directive of the form:

<:mark fontset <name>:>

where argument 1 is the name of the fontset being defined. Following the <:mark:> directive is a series of <:deffont:> directives, as described above, one for each font in the set.

A fontset is named for a given run by giving a <:fontset:> directive, of the form:

<:fontset <name>:>

Also, <:deffont:> directives may appear directly in the input, without being associated with a fontset.

5. GALLEYS

A galley can be viewed as an environment defining characteristics that determine the appearance of formatted output. Currently, ctext defines two galleys -- one for the running text in a given document, and one for headings and footings. Many ctext predefined directives affect a single galley characteristic. These are summarized in the first two sections below. These general and tab-related characteristics are what <:push:> and <:pop:> save and restore.

5.1. General

Lines of text are output to a galley in two ways: "normally", or because of a "line break". When output line filling is not in effect (i.e., hfill is off), a line is output normally at the end of each input line. When filling is in effect, a line is output normally when it is "full", that is, when appending the next available word to it would cause it to exceed the current galley width. A line break causes any text accumulated for the current line to be output immediately, regardless of its length. (Successive line breaks after the first have no effect.) Line breaks can be caused by blank lines in the input text; they are also caused when certain galley characteristics change, when vertical whitespace is output to the galley, and at end of input.

<:blines <state>:> - specify the effect of blank lines encountered in the input text. If lhwhite is set to kept or break, a blank line is one containing no characters (i.e., two consecutive newlines). Otherwise, a blank line is one containing no characters other than space or tab (ASCII HT or ' '). Values: ignored, to suppress blank lines; kept, to retain them in the input stream. If kept, each blank line causes a line break. Default: kept. This directive causes a line break.

<:ehwhite <state>:> - controls processing of spaces embedded in an input line. Values: ignored, to suppress them (seldom advisable); kept, to pass them through to the galley; compact, to reduce each string of spaces to a single space, then expand it as required by hjust and eosproc. Default: compact in the text galley, kept in the heading galley.

<:eosproc [<chars>]:> - controls whether extra space is inserted into the galley after a sentence break; a sentence break is deemed to occur after any of the characters in <chars> (this character may be followed by an arbitrary number of quotes, apostrophes, or right parentheses; the break comes after them). Extra space will be inserted only if the break occurs at the end of an input line, or if it is followed by two or more spaces in the input text. Values: the set of characters that can cause a sentence break; or no value at all, to disable sentence break processing. Default: processing is disabled.

<:filter <state>:> - controls whether the input stream is filtered to remove characters that are not printable ASCII (i.e., ones with octal values below 040 [space] or above 176 [~] except for tabs and newlines). The filtering is applied to input from user files only. Values: on; off. Default: on.

<:font <name>:> - names the font ctext switches to when setting text in this galley. Values: any currently defined font. Default: the first font named in the current fontset.

<:hfill <state>:> - controls whether each line output to the galley will be filled from one or more input lines, or simply consist of a single input line. Values: on; off. Default: on in the text galley, off in the heading galley. This directive causes a line break.

<:hjust <state>:> - controls the justification of each line output to the galley. Values: block, to block-justify each line; left, to justify each flush-left; right, to justify each flush-right; center, to center each within the galley; body, same as block, except that lines output because of a line break are left-justified. Also, the additional space needed for body justification is inserted at opposite ends of successive output lines; for block justification, it is always inserted on the left. Finally, the space for body justification is always inserted to the right of the rightmost tab in the output line. Default: body in the text galley, left in the heading galley. This directive causes a line break.

<:hyph <count>:> - controls whether hyphenation will be performed on input words, and gives the minimum number of characters permissible in the hyphenated portion of a word. Values: zero, to disable hyphenation,

or the minimum length of a hyphenated string (excluding any trailing hyphen). Default: two in text galley, zero in heading galley. (See Appendix E for a discussion of how hyphenation is performed.)

<:indent <value>:> - sets the indentation (i.e., leading horizontal whitespace) to be added to each line output to the galley. This indentation is subtracted from the remaining line width, and is not seen by lhwhite. Values: a horizontal measurement. If the measurement is preceded by "++", it is added to the old indentation to obtain the new one; if the measurement is preceded by "--", it is subtracted from the old indentation to obtain the new one. Default: 0. This directive causes a line break.

<:lhwhite <state>:> - controls processing of leading whitespace on an input line. Values: ignored, to suppress it; kept, to pass it through to the galley; break, to pass it through to the galley and process the rest of the line with hfill off, hjust set to left, ehwhite set to kept, and eosproc disabled. (Setting lhwhite to break emulates the handling given by earlier processors to input lines with leading whitespace; all characteristics are reset to their previous values after the line is processed.) Default: break in text galley, ignored in heading galley. This directive causes a line break.

<:lvwhite <state>:> - controls processing of vertical whitespace occurring at the top of a page. Values: ignored, to suppress it; kept, to retain it in the galley. Default: ignored.

<:size <value>:> - specifies the point size to be used to set text in the galley; has no effect on current font. Values: a vertical measurement (generally in points). If the measurement is preceded by "++", it will be added to the old size to obtain the new one; if the measurement is preceded by "--", it will be subtracted from the old size to obtain the new one. Default: size is initialized to 1pc. For devices that can scale a devicestyle to different sizes, an explicit <:size:> directive should precede any input text. (Any device whose definition names it as "fixed" will ignore this command, wherever it is given.)

<:skew <value>:> - specifies the distance from "normal" position of the baseline of text set into the current galley. Normally, text is set at the bottom of the region reserved by <:spacing:>; if a font characteristic changes in mid-line, the new text is set so that its baseline aligns with that of the previous text. Values: a vertical measurement, giving the offset from normal of the baseline of new text. As always, a negative value indicates an upward motion, a positive value, a downward motion. If the measurement is preceded by "++", it will be added to the old skew to obtain the new one; if the measurement is preceded by "--", it will be subtracted from the old skew the obtain the new one. Default: zero.

<:spacing <offset>:> - sets the normal spacing between baselines of successive lines of text output to the galley. Values: a number used as a multiple of the current <:size:> value. Default: 1. This directive causes a line break.

<:spread <offset>:> - specifies additional spacing (beyond that given by <:spacing:>) between output lines of text. Values: a number used as a multiple of the current <:spacing:> value. Default: zero.

<:stylecode <code>:> - names the stylecode ctext switches to when setting text in this galley. Values: any stylecode defined in the current font. Default: the first stylecode named in the current font.

<:width <value>:> - sets the prevailing width of the galley. This characteristic alone is "shared" between the text and heading galleys; changing it in either galley will change it in both. Values: a horizontal measurement. If the measurement is preceded by "++", it will be added to the old width to obtain the new one; if the measurement is preceded by "--", it will be subtracted from the old width to obtain the new one. Default: 6in. This directive causes a line break.

5.2. Tabs

ctext also provides several directives that deal with tabstops. A tabstop is a defined point within an unjustified output line, which is specified as a horizontal measurement from the current left margin (as affected by <:indent:>). All tab-related formatting is performed on an output line before justification (i.e., the line is treated as left-justified). Text may be left-justified beginning at a tabstop; text may also be right-justified ending at a tabstop, or centered between a tabstop and some previous point on the output line. The width of the associated tab (i.e., the space it adds to the output line) can be filled with unpaddable space (the default), or with any other string. These capabilities are provided through the directives described below, and through several of the unnamed directives described in the next section.

<:tabcols <count>:> - sets tabstops as required to divide the current galley width (minus <:indent:>) into <count> columns of equal width. Once set, these stops are identical to those set by <:tabset:>. If <count> does not evenly divide the galley width, the leftover space is placed in the rightmost column. Any tabstops previously set are cleared. This directive causes a line break.

<:tabfill <string> <tab-list>:> - causes <string> to be used in filling the width of a tab; <string> is replicated as necessary. If the tab width is not an exact multiple of the length of <string> unpaddable space is used to fill the remainder. <tab-list> is an ordered list of positions in the current list of tabstops; <string> will be used to fill up to the tabstops in the list only, or, if the list is null, up to every stop set. For instance, <:tabfill . 2 3:> would cause fill composed of '.' to be used in tabbing to the second and third stops currently set. Default: unpaddable space is used to fill up to each tabstop.

<:tablead <string> <tab-list>:> - like <:tabfill:>, except that <string> is inserted on an output line only at a position equal to some integral multiple of its length; thus fill characters are made to line up on successive output lines. Only one of filling and leading can be in effect for a given tabstop. Default: no leading is done.

<:tabset <tabstop-list>:> - sets tabstops in the current galley. A tabstop of zero clears all pre-existent stops; each <:tabset:> otherwise adds tabstops to those already in effect. New stops are given as horizontal measurements, measured from the current left margin (as affected by <:indent:>). Default: no stops are set. This directive causes a line break.

5.3. Unnamed Directives

The following directives are specified, not by name, but by a command open sequence immediately followed by the special character shown.

<:<copen-seq> - two successive command open sequences cause the insertion of a single open sequence into the input stream.

<:<space> - causes the insertion of an unpaddable space. Output lines will never be broken at such a space, nor will it be expanded with additional whitespace.

<:<newline> - evaluates to a null command, thus causing <newline> to be ignored. Used to enter a newline for cosmetic purposes (as in a macro definition) that is not be passed to the text formatter.

<:- - indicates a "soft" (or "discretionary") hyphen. If the surrounding "word" (i.e., an otherwise unbreakable string) is to be hyphenated, hyphenation will occur only at the points indicated.

<:& - null command. Does nothing but occupy space in the input stream.

<:| - forces an output line break.

<:* - discards input characters until a character other than whitespace is encountered. Markup that ultimately adds no non-white characters to the input stream, and that causes no line breaks, will also be processed. "Whitespace" here indicates spaces, tabs, and newlines.

<:^ - sets a "dynamic" tabstop, at the current horizontal position on the unjustified output line. Once set, such a tabstop is identical to one set by <:tabset:>.

<: - visible tab character, used to invoke the next tabstop available. The tab fills the output line as needed up to the position of the tabstop. If no tabstops remain unused, the tab fills the width of a single space. If the accumulated width of the output line is already at or beyond the next tabstop available, the tabstop following is tried, until none remain or one is found beyond the end of the line. The conventional tab character (ASCII HT) can also be used to invoke a tabstop, though visible tabs are much easier to interpret.

<:/ - marks the start of text to be centered in the space between the current horizontal position and the next tabstop; the end of the text is signalled by the next tab-related command read. If no tabstops remain, the text is centered between the current position and the right margin; if no tab-related command is encountered before the end of the output line, the centered text terminates there.

<:% - marks the start of text to be centered in the space between the current horizontal position and the next tabstop; the end of the text is signalled by the next tab-related command read. Unlike "<:/",

"<:%" inserts the proper fill to the left of the centered text, but inserts nothing to its right, leaving the end of the output line at the end of the centered text. If no tabstops remain, the text is centered between the current position and the right margin; if no tab-related command is encountered before the end of the output line, the centered text terminates there.

<:+ - marks the start of text to be set flush-right against the next tabstop; the end of the text is signalled by the next tab-related command read. If no tabstops remain, the text is set flush-right against the right margin; if no tab-related command is encountered before the end of the output line, the justified text terminates there.

<:! - memorizes the current unjustified output line position, for possible later recall with "<:$". Like tabstops, this "cursor" is defined on a per galley basis; a separate instance is defined for each pushed copy.

**<:$** - return to the output line position memorized by "<:!". The motion necessary may be either rightward or leftward. If no position has been marked, "<:$" returns to the start of the line.

A "<:" preceding any other character that cannot begin an identifier and that is not a digit is deleted during command processing, and otherwise has no effect. Naturally, any character string defined as a command open sequence will be treated the same way.

6. PAGES

The page mechanism in ctext allows for single-column pages, the length of which is user-specifiable. Pages may have headings, footings, or both, the length and contents of which are also programmable. Text inserted into headings or footings may contain markup, which will be interpreted (when the heading or footing is output) exactly as if it appeared in running text. A separate galley is devoted to heading/footing production, so that galley characteristics may be set therein without affecting the galley used for running text. The directives that control page production are the following:

6.1. Page Parameters

Several directives together determine the length of a page and of its major components. The user can specify overall page length, and the length of headings and footings. The difference between the two, of course, is the space available for running text on each page. Whenever the page length or heading length is changed by the user, this space is altered to compensate, so that the other length can stay constant.

<:ispace <value>:> - controls the distance between a heading or footing and the running text of a page (the "inner spacing" of the heading). If both <:ispace:> and <:ospace:> are set to zero, pages are output contiguously with no vertical whitespace whatever between them. Value: a vertical measurement. Default: 2pc.

<:newpage [<number>]:> - advances to a page boundary, then sets the page number of the new page to <number>. If ctext is already at a page boundary, this command just sets the new number. Otherwise, a line break is generated, and the current page terminated (with the old page number still in effect). Value: the desired number of the new page; if <number> is omitted, the old page number is incremented to derive the new number.

<:ospace <value>:> - controls the distance between a heading or footing and the extremity of the page (the "outer spacing" of the heading). See <:ispace:>. Value: a vertical measurement. Default: 3pc.

<:plen <value>:> - specifies the overall length of page, counting headings and footings (if any), and the space reserved for running text. Value: a vertical measurement. Default: 11in.

<:pnum <number>:> - specifies the page number of the next page started. Value: the desired number. Default: 1.

6.2. Heading/Footing Parameters

The directives following are used to insert text into headings or footings. Three different heading/footing pairs are provided: for "first pages", and for other even or odd-numbered pages. The page number that is to activate the "first page" heading/footing is specifiable at runtime. Text given to these directives is inserted into the input stream when appropriate, then evaluated like any other user input. This means that it may contain formatting commands. One subtlety: simply naming commands in the heading/footing text will cause them to be executed when the heading/footing directive is first encountered. To delay their interpretation until the heading or footing is output, each command open sequence must be doubled.

<:ef <text>:> - causes <text> to be used as a footing when needed on even-numbered pages other than the "first page".

<:eh <text>:> - causes <text> to be used as a heading when needed on even-numbered pages other than the "first page".

<:ff <text>:> - causes <text> to be used as a footing when needed on pages with the "first page" number specified.

<:fh <text>:> - causes <text> to be used as a heading when needed on pages with the "first page" number specified.

<:of <text>:> - causes <text> to be used as a footing when needed on odd-numbered pages other than the "first page".

<:oh <text>:> - causes <text> to be used as a heading when needed on odd-numbered pages other than the "first page".

7. OBJECT DEFINITION

ctext defines only one namespace for all named objects, both pre-defined and user-defined. Any name may be redefined; the most recent definition hides all pre-existent ones until removed (with <:undef:>).

<:defctr <name> <arith-expr>:> - define an integer counter named by argument 1, initialized to the value of argument 2, which must be an arithmetic expression. <:defctr:> is also used to alter the value of a pre-existent counter. If not explicitly initialized, a counter is set to zero. A direct reference to a counter is replaced by the numeric string representing its value. Note that, to increase their ease of use, counter names do not "stack"; that is, a <:defctr:> will define a new counter only if one does not already exist with the name given.

<:defdir <name> <value>:> - define a directive (named by argument 1) as the contents of argument 2. ctext treats <value> as "quoted"; that is, in scanning it ctext will not interpret any embedded commands, which will be left untouched inside. This implies that <value> must be surrounded by a qopen/qclose sequence, since otherwise the closing sequence of the first embedded command would be interpreted as the end of the defdir command itself. When the directive being defined is invoked, each argument in the invocation is obtainable by following a command open sequence with the position of the argument in the argument list (e.g., the value of argument 1 would be obtained with <:01). Argument numbers count up from 1, and must be given as two-digit integers.

<:defenv <name> <value>:> - define an environment (named by argument 1) as the contents of argument 2. ctext treats <value> as "quoted"; that is, in scanning it ctext will not interpret any embedded commands, which will be left untouched inside. (See <:defdir:>.) When invoked, the environment cannot directly access the arguments in the invocation; instead, the environment definition is inserted into the input stream, then the text of the invocation is inserted following. In addition, the current galley characteristics are memorized when the environment is entered, and restored when it is left (i.e., when the invocation arguments are exhausted).

<:deffont <name> <code-pair> [<code-pair>]*:> - define a font named by argument 1, containing the stylecode/charset associations named by succeeding pairs of arguments. See above.

<:defqdir <name> <value>:> - define a directive (named by argument 1) as the contents of argument 2. This directive differs from <:defdir:> in that any invocation of the directive being defined will be treated as "quoted", in the same way that <:defdir:> and <:defqdir:> themselves are, except that references in the definition to arguments will be expanded from the invocation.

<:defstr <name> <value>:> - define a string (named by argument 1) as the contents of argument 2. <value> is not treated as "quoted"; in addition, it is scanned as a string constant (i.e., any of the escape sequences defined above will be interpreted within it). Arguments are available at invocation as described under <:defdir:>.

<:undef <name>:> - removes the most recent user-supplied definition of <name>; no builtin name definition may be removed. "Undefining" an already-undefined name is not considered an error.

8. GALLEY MANIPULATION

The following directives supplement the galley characteristics described earlier, by causing the action noted:

<:cspace <distance>:> - conditionally inserts space into the galley, by ensuring that vertical whitespace of at least height <distance> exists at the end of the galley. If not enough does, the necessary whitespace is inserted. Value: a vertical measurement. Default: 1sp.

<:need <distance>:> - requires that vertical space of at least <distance> remain on the current page. If not enough does, a new page is started. Value: a vertical measurement. Default: 1sp. This directive causes a line break.

<:pop:> - restores the set of characteristics for the current galley that were most recently saved with <:push:>. <:pop:> may be used with either <:push:> or <:pushf:>, and will restore tabstops as appropriate; since <:pop:> goes to a great deal of trouble to restore the previous font and stylecode, however, <:popf:> should be used where possible. If no characteristics are stacked, <:pop:> does nothing.

<:popf:> - like <:pop:>, except that the current font and stylecode are left in effect. <:popf:> should be used only if these have not changed since the corresponding <:push:> or <:pushf:>.

<:push:> - saves the characteristics of the current galley, for later retrieval with <:pop:> or <:popf:>. <:push:> saves the current tabstop settings in addition to other galley characteristics; as this is a non-trivial operation, <:pushf:> should be used instead whenever the previous tabstops will not be needed after the push.

<:pushf:> - like <:push:>, except that tabstop settings are not saved. All tabstops previously set are cleared in the new galley; any declared in the new galley are still local to it, and do not affect the previous galley.

<:space <distance>:> - inserts vertical whitespace of length <distance> into the current galley. Value: a vertical measurement. Default: 1sp.

9. STATE MANIPULATION

<:device <name>:> - format text for output device <name>. This directive must precede any other input, and causes ctext to look for a ".ds" file with a matching name.

<:fontset <name>:> - establishes <name> as the fontset to be used for this run. Must precede any user text. Must follow a <:device:> directive (since the latter establishes parameters the font mechanism needs to know about).

<:if <expr> <text> <else-chain>:> - evaluates argument 1 (a boolean expression), then, if it is true, inserts the contents of argument 2 into the input stream. <else-chain> is zero or more "else" clauses, each of which is a series of arguments having one of the forms:

```
elseif <expr> <text>
else <text>
```

where <expr> and <text> are ordinary arguments. The <text> of an elseif clause will be inserted if the corresponding <expr> is true, and if no earlier <expr> in the <:if:> command was true. An else clause must be the last of the <else-chain>; its <text> is inserted only if no preceding <expr> was true.

<:include <filename>:> - inserts the contents of <filename> into the input stream.

<:init [clear] <attrib-pair> [<attrib-pair>]* :> - initializes the format of command markup. If the keyword "clear" is specified, all previously-defined formats are discarded before the new one is added. In the absence of this keyword, multiple calls to init will register alternate formats, any of which will then be recognized. Each <attrib-pair> consists of one of the following keywords, followed by an argument giving the character sequence to be used in the specified context. Keywords:

copen - names the command sequence signalling the start of a command. Default: "<:".

cclose - names the sequence signalling the end of a command. Default: null.

aopen - names the sequence signalling the start of an argument list. Default: none.

aclose - names the sequence signalling the end of an argument list. Default: none.

aterm - names the sequence one or more of which can terminate arguments within the argument list. Default: ' ', '\t', '\n'.

qopen - names the sequence signalling the start of a quoted argument. Default: '"', '(', '[', '{', apostrophe.

qclose - names the sequence signalling the end of a quoted argument. Default: '"', ')', ']', '}', apostrophe.

Each sequence may be an arbitrary string constant; all of the normal escape sequences are therefore recognized. In addition, if the first character of a sequence is a '^', the sequence will match input only at the start of an input line, and the sequence itself is deemed to start with the next character. A sequence may begin with a literal character '^' by writing it as "^^". A given keyword/sequence pair may appear more than once in a given call; any of the sequences given will then be recognized in that context. A sequence appearing in a call will be associated only with the other sequences in that call; and each open or close sequence is associated only with the complementary sequence having the same position in the set of sequences the call mentions (e.g., in the default format, a qopen

character of '[' will be matched only with a qlose character of ']').

Either aopen/aclose or cclose (but not both) may be omitted. If cclose is omitted, then an invocation is considered to end at the close of the argument list, if any, else at the end of the command name. If aopen/aclose are omitted, then the argument list is terminated by the end of the invocation.

To ease the entry of directive definitions containing calls to other commands, the parsing implied by a command format is augmented as follows: If the "last" sequence in an invocation (cclose, or aclose if cclose is undefined) is '\n', nothing more happens. Otherwise, if an invocation began at the start of an input line or at the start of a directive definition, the character following the end of the invocation is examined; if the character is a newline, it is read and discarded. To prevent nested invocations from scanning successive newlines after the top-level invocation, newlines are not scanned past the end of a directive definition.

<:line <count>:> - forces <count> lines to be read from the input stream. The lines are fully interpreted as they are read. To maximize its usefulness inside macro definitions, <:line:> pops the current (i.e., topmost) definition, if any, off the input stream before reading, then replaces it afterwards.

<:mark <arg-list>:> - signals the start of a device specification, or of a charset, devicestyle, or fontset definition. May appear only in table files, as described above.

<:nobrk:> - prevents the output of a line break by the next command that would normally cause one.

10. MISCELLANY

The following commands provide conversion facilities and various environmental information.

10.1. Conversion

<:comment <args>:> - <args> are simply ignored by the processor, after being parsed in the usual way.

<:isdef <name>:> - inserts into the input stream a "1" if <name> is currently a user-defined symbol, or a "0" otherwise.

<:ordstr <s1> <s2>:> - treats its first two arguments as string constants, comparing them character-by-character for their order in the collating sequence. Inserts into the input stream: -1, if <s1> is less; 0, if <s1> and <s2> are equal; or 1, if <s2> is less. An empty argument (or a missing <s2>) compares less than any non-null string.

<:remark <args>:> - each argument in <args> is output to STDERR, with adjacent arguments separated by a space. No further formatting is performed on the output.

10.2. Predefined Counters and Strings

<:argc:> - the number of arguments given (hence, "argument count") to the most recent user-defined directive whose contents are still in the input stream.

<:argv [<first>]:> - the list of arguments (hence, "argument vector") to the most recent user-defined directive whose contents are still in the input stream. This is returned as a string in which the contents of adjacent arguments are separated by spaces. If given, <first> is the number of the first argument to be included in the string; by default, all are included.

<:devname:> - the name of the output device specified for this run.

<:fontname:> - full name of font currently in use.

<:indentval:> - the indentation currently in effect from the left margin of the current galley. Recorded in device horizontal rasters.

<:pcnt:> - count of the number of pages so far output, including the current page.

<:pval:> - current page number, as set by <:pnum:> or <:newpage:>.

<:stylename:> - full name of the current stylecode.

<:widthval:> - the width of the current galley. Recorded in device horizontal rasters.

II. Ctext Component Programs

NAME

Programs - components of the ctext system

FUNCTION

This section describes the individual programs that make up the ctext system. The descriptions are in the form of "manual pages": summaries of how the programs work, and how they are run. In particular, the flags accepted by each program are documented here. These programs fall into four groups:

1) the command scripts that run ctext for a specific output device. These are ctdbl, to run ctext for plain or Diablo-compatible printers; cth19, to run ctext for VT-52 compatible VDT's; and ctvt, to run ctext for Varityper Comp/Edit devices.

2) the ctext formatter itself. Like the other programs, the formatter can be run separately; its intermediate output can then be saved in a permanent file for later submission to one of the device drivers.

3) device drivers. These programs translate the intermediate output from the formatter into the command stream expected by a particular class of output device. They are: vt52, for VT52-compatible VDT's; dbl, for plain or Diablo-compatible printers; and vtyper, for Varityper Comp/Edit devices driven through the STS interface.

4) utility programs. Currently, this is just the program toctext, which converts troff-style escape sequences into ctext commands.

Normally, ctext is run using the command scripts. If the scripts are used, then only the conversion program toctext, if wanted, needs to be run separately. The other programs are available for special-purpose use, however, and this section tells you everything you need to know in order to run them.

NAME

alarm - send alarm signal periodically

SYNOPSIS

alarm -[p# s#]

FUNCTION

alarm is used by vtyper, the device driver for Varityper Comp/Edit photocomposers, to provide timeout capability on reads by vtyper from the typesetter.

alarm simply sends a signal to the process whose id is given on its command line, after pausing for the number of seconds also given there. It is started by vtyper before each read from the typesetter. When the read is complete, the alarm process is killed. If, however, the read is not complete after the delay specified, alarm interrupts vtyper, indicating a timeout has occurred.

The flags are:

-p# send alarm signal to process #.

-s# sleep for # seconds between sending signals.

If the signal cannot be sent, alarm exits.

RETURNS

alarm loops until unable to send the signal, or until killed. In either case, it then returns failure.

NAME

ctdbl - drive ctext for Diablo or plain printers

SYNOPSIS

ctdbl -[c d* fs* f# h hi i* ms# o* p# t* u vi w +# #] <files>

FUNCTION

ctdbl is a shell script that runs first ctext, then the device driver dbl, to produce formatted text for Diablo-compatible printers, or for "vanilla" printers with no special capabilities. The entire command line given to ctdbl is passed directly to ctext, except for any occurrence of the four flags explained below.

Four flags are interpreted by ctdbl instead of being passed to ctext. These are:

-h include the enhanced heading macro package, ctmac.h, on the command line to ctext.

-hi enable the use of incremental horizontal spacing in the Diablo driver dbl.

-vi enable the use of incremental vertical spacing in the Diablo driver dbl.

-w include the Whitesmiths-style macro package, ctmac.w, on the command line to ctext.

By default, only the standard macro package, ctmac.s, is passed to ctext.

If either "-hi" or "-vi" is given, ctext will format with a Diablo in mind, using device "dbl". These flags are then passed through to the device driver dbl. As shipped, the devicespec for device "dbl" specifies both horizontal and vertical incremental spacing, so if one of these flags is given, both should be.

If neither "-hi" nor "-vi" is given, ctext will format for a "vanilla" printer, using device "tty", and the output of ctdbl is not Diablo-specific.

To lessen the demand ctdbl places on Idris, ctext and dbl are run sequentially, communicating via a temporary file. On systems where this is not an issue, the two programs may be run in a pipeline, permitting dbl to output text to the printer while ctext is still formatting.

RETURNS

ctdbl returns success if both ctext and dbl ran successfully.

BUGS

It would be nice if ctdbl recognized and passed to dbl the other flags that the driver accepts.

NAME

ctext - a processor for text

SYNOPSIS

ctext -[c d* fs* f# i* ms# o* p# t* u +# #] <files>

FUNCTION

ctext is a user-programmable utility for creating formatted text suitable for reproduction by a variety of output devices. This first pass of the ctext system is a formatter that outputs a device-independent stream of graphics and control information, suitable for interpretation by a driver program for a specific output device.

ctext takes input from each file named in <files>; if any file named does not exist, ctext does not complain but goes on to the next file in sequence. If no <files> are given, STDIN is read. A filename of "-" also causes STDIN to be read, at that point in the list of files.

The formatter accepts these flags:

-c continue formatting (without output) after page given with "-#".

-d* format text for device *. The string given is used as the argument to a <:device:> command. A device given with "-d" will override any specified elsewhere; <:device:> commands after the first are ignored.

-fs* use fontset *. The string given is used as the argument to a <:fontset:> command. A fontset given with "-fs" will override any specified elsewhere; <:fontset:> commands after the first are ignored.

-f# make # the page number that will activate the "first page" heading and footing commands. The default, of course, is 1.

-i* make * the name of the initialization file ctext will read at the start of the run. To the name given is appended the string ".ci"; the resulting filename is then searched for as a table file would be. The default name is "ctext", meaning that the file "ctext.ci" is looked for.

-ms# specify the maximum size of a legitimate macro invocation; # is given in bytes, and may be no larger than 8160. ctext will complain and abort if it encounters a longer definition than specified. By default, a maximum size of 1024 bytes is imposed. A value of 0 disables macro size checking.

-o* write intermediate output to file *, and error output to STDOUT. By default, intermediate output goes to STDOUT, and error messages to STDERR. This flag is mandatory on non-Idris/UNIX systems that distinguish between text and binary files, since STDOUT is treated there as a textfile.

-p# make # the number of the first page formatted.

-t* use * as a series of prefixes (presumably directory names) that ctext will prepend to table filenames in the order given. Prefixes are separated by a vertical bar '|'; the current directory is indicated by a null prefix. The default string is the Idris/UNIX "|/usr/ctext/" (i.e., current directory, then directory /usr/ctext).

-u warn of user-defined names that are undefined when used. By default, commands with names having no current definition simply disappear.

+# start output at each page numbered #. Earlier pages will be formatted, but will generate no output. By default, all pages are output.

-# stop output after each page numbered #. If "-c" is not given, ctext exits after the first such page is completed.

RETURNS

ctext returns success if all file and macro processing was performed with no errors, that is, with no diagnostic messages produced. If ctext finds an error, it will issue an appropriate message, and will return failure.

FILES

ctext searches the directories specified by "-t*" for the ".ci" initialization file it needs, and for the table files implied by the first <:device:> and <:fontset:> commands it encounters.

NAME

cth19 - drive ctext for H19 (VT52) terminals

SYNOPSIS

cth19 -[c d* fs* f# h i* ms# o* p# t* u w +# #] <files>

FUNCTION

cth19 is a shell script that runs first ctext, then the device driver vt52, to produce formatted text for terminals compatible with the DEC VT52. The entire command line given to cth19 is passed directly to ctext, except for any occurrence of the two flags explained below.

Two flags are interpreted by cth19 instead of being passed to ctext. These are:

-h include the enhanced heading macro package, ctmac.h, on the command line to ctext.

-w include the Whitesmiths-style macro package, ctmac.w, on the command line to ctext.

By default, only the standard macro package, ctmac.s, is passed to ctext.

To lessen the demand cth19 places on Idris, ctext and vt52 are run sequentially, communicating via a temporary file. On systems where this is not an issue, the two programs may be run in a pipeline, permitting vt52 to output text to the screen while ctext is still formatting.

RETURNS

cth19 returns success if both ctext and the device driver ran successfully.

NAME

ctvt - drive ctext for Varityper Comp/Edit devices

SYNOPSIS

ctvt -[c d* fs* f# h i* l* ms# o* p# t* u w +# #] <files>

FUNCTION

ctvt is a shell script that runs first ctext, then the device driver vtyper, to produce formatted text for Varityper Comp/Edit typesetters, using the slave typesetter (STS) protocol. The entire command line given to ctvt is passed directly to ctext, except for any occurrence of the three flags explained below.

Three flags are interpreted by ctvt instead of being passed to ctext. These are:

-h include the enhanced heading macro package, ctmac.h, on the command line to ctext.

-l* use * as the name of the communications link to the typesetter. By default, the name built into ctvt is used.

-w include the Whitesmiths-style macro package, ctmac.w, on the command line to ctext.

By default, only the standard macro package, ctmac.s, is passed to ctext.

To lessen the demand ctvt places on Idris, ctext and vtyper are run sequentially, communicating via a temporary file. On systems where this is not an issue, the two programs may be run in a pipeline, permitting vtyper to output text to the printer while ctext is still formatting.

RETURNS

ctvt returns success if both ctext and the device driver ran successfully.

BUGS

It would be nice if ctvt recognized and passed to vtyper the other flags that the driver accepts.

NAME

dbl - drive plain or Diablo-compatible printers for ctext

SYNOPSIS

dbl -[bi hi h# l# p vi v# +# #] <files>

FUNCTION

dbl converts the ctext intermediate language into a command stream suitable for "vanilla" printers or CRT's, or printers compatible with the Xerox Diablo. A "vanilla" output device is presumed to be capable of generating overstruck lines (by accepting successive text lines terminated only by a carriage return), but to have no other special capabilities. Specifically, dbl reads an intermediate command stream from each file in <files>, and a writes a converted command stream to STDOUT. If no <files> appear, STDIN is read; a filename of "-" also causes STDIN to be read, at that point in the list of files.

dbl accepts these flags:

-bi use incremental spacing only in producing boldfaced text, in the manner described below.

-hi use incremental spacing horizontally. Implies a horizontal raster of 1/120 of an inch.

-h# output # characters per inch horizontally. The "hmi" is initialized to correspond to this value. The normal value used is 12.

-l# make "pages" # lines in length; used in conjunction with the "+#" and "-#" flags described below. Default is 66 lines.

-p pause after each page of output. dbl will pause until a newline is read from STDOUT (presumably the terminal).

-vi use incremental spacing vertically. Implies a vertical raster of 1/48 of an inch.

-v# output # lines per inch vertically. The "vmi" is initialized to this value, and reset to it after each incremental spacing (if any). The normal value used is 6.

+# start output at page #, where pages are measured by "-l#" and numbered from one. Default is first page.

-# stop output after page #, where pages are measured by "-l#" and numbered from one. By default, all pages are output.

Note that the "page numbering" here is just a count of the pages processed, where page length is given by "-l". No numbering information is passed from ctext to the driver.

dbl will output text as overstruck whenever the current font has an internal index with the "bold" bit (1-weighted bit) set. Overstriking is accomplished by outputting the text three times in place. dbl will output text as underlined whenever the current font has an internal index with the "italic" bit (2-weighted bit) set. Currently, only alphanumeric characters are underlined.

If none of -[bi hi h# vi v#] is specified, dbl as far as possible avoids outputting Diablo-specific escape sequences, and is thus usable as a

general-purpose driver. In this case, dbl outputs spaces (ASCII ' ') and backspaces (ASCII BS) for right or left cursor motion, line feeds (ASCII LF) and -- for want of anything better -- "reverse line feeds" ("\ESC\LF") for down or up cursor motion.

If -hi is given, dbl will use incremental spacing in outputting horizontal whitespace, and will offset the first overstrike by one increment in overstriking text (to enhance emboldening). If -bi is given, only this offseting will occur. If -vi is given, dbl will similarly use incremental spacing in outputting vertical whitespace. Naturally, "-hi" or "-vi" may be used only if the front end was run with a devicespec that allowed for them (by specifying the appropriate horizontal and vertical raster units).

If "-h#" or "-v#" is given, dbl will set the default horizontal or vertical spacing as specified; they are reset at the end of the run to the "normal" values given above.

No other escape sequences are ever output; further, if none of -[hi h# vi v#] is given, dbl is guaranteed never to output any Diablo-specific commands.

dbl expects to be installed with a filename that contains the name of the device used by ctext in preparing the input to the driver. In the standard package, this driver is installed under the names "dbl" and "tty".

RETURNS

dbl returns success if able to read all of the named input <files>. It will return failure (but output no disagnostic) if unable to do so.

NAME

toctext - convert nroff/troff escape sequences to ctext markup

SYNOPSIS

toctext -[+a* a* c2* cc* +c* c* ec* eo* hc* ho* +q* q* +t +u] <files>

FUNCTION

toctext is intended to ease the conversion of documents originally meant for formatting with nroff/troff into a format suitable for use with ctext. For the most part, this amounts to converting in-line escape sequences to a more regular format that ctext can interpret. toctext also recognizes the troff requests meant to cause some change in the format of troff markup, and provides other conversion facilities as options.

Note that, aside from a few rudimentary heuristics, toctext is purely a translator from one markup syntax to another. It does nothing to provide ctext-compatible definitions for any troff escape sequences.

Most of the flags accepted by toctext serve to change the components of the markup syntax it outputs. By default, toctext converts escape sequences to the default markup syntax described in the System Reference Manual, and passes through line-oriented requests unchanged. (The initialization file "ctext.ci" shipped with the package defines a markup format that allows ctext to recognize line requests.)

Flags are:

+a* make * the output argument open sequence.

-a* make * the output argument close sequence.

-c2* make the first character of * the initial troff secondary command character.

-cc* make the first character of * the initial troff command character.

+c* make * the output command open sequence.

-c* make * the output command close sequence.

-ec* make the first character of * the initial troff escape character.

-eo turn off troff escape character initially.

-hc* make the first character of * the initial troff hyphenation character.

-ho turn off troff hyphenation character initially.

+q* make each character of * a potential open quote character in the output markup.

-q* make each character of * a potential close quote character in the output markup.

+t convert .ta or .TA requests by prepending a "0" to the list of tabstops given, and appending the unit specifier "ch" to each tabstop value.

+u change lowercase alphabetic request names to uppercase.

toctext prepends two characters to the names of most of the escape sequences it encounters. This is done both to bring names into at least

closer conformance with the rules for ctext identifiers, and to differentiate objects that in troff are in separate namespaces. Two escape sequences, "\|" and "\^", are transformed outright into "trs" and "ths" (for "narrow space" and "half-width narrow space"). The other troff escape sequences named by a single special character ("\&", "\ ", and so on) all have direct equivalents in ctext to which they are translated. The prefixes used for the remaining sequences follow:

PREFIX	USED FOR
tc	character names "\(xx"
ts	strings
tn	number registers
tf	font names or numbers
te	all other escape sequences

Thus the sequence "\n(AC" would become the command "<:tnAC:>", "\f1" would become "<:tf1:>", and so on.

toctext also translates line requests introduced by the secondary command character into ordinary requests preceded by the command ".nobrk". Finally, toctext does two kinds of argument processing. In line requests or escape sequences, numeric arguments of the form "+n" or "-n" are rewritten as the ctext-style "++n" or "--n". And in line requests, a closing quote is added to quoted arguments that lack one.

toctext will output warning messages, each identifying the current input file and line number, if confronted with a request or escape sequence it is unable to convert. Nominally, however, anything legal in troff will be converted to a corresponding ctext command.

RETURNS

toctext returns success if able to read its input <files>, and if able to convert every escape sequence it encounters.

NAME

vt52 - drive VT52-compatible terminals for ctext

SYNOPSIS

vt52 -[l# +# #] <files>

FUNCTION

vt52 converts the ctext intermediate language into a command stream suitable for any terminal compatible with the DEC VT52. Specifically, it reads an intermediate command stream from the files in <files>, and a writes a converted command stream to STDOUT. If no <files> appear, STDIN in read; a filename of "-" also causes STDIN to be read, at that point in the list of files.

vt52 accepts these flags:

-l# make "pages" # lines in length; used in conjunction with the "+#" and "-#" flags described below. Default is 66 lines.

+# start output at page #, where pages are measured by "-l#" and numbered from one. Default is first page.

-# stop output after page #, where pages are measured by "-l#" and numbered from one. By default, all pages are output.

Note that the "page numbering" here is just a count of the pages processed, where page length is given by "-l". No numbering information is passed from ctext to the driver.

vt52 will turn on reverse video when it encounters a font change command to any font with a non-zero internal index; vt52 turns reverse video off when the current font is changed back to the one with font index zero. Whitespace is also output with reverse video off; this is done purely for aesthetic reasons.

vt52 outputs spaces (ASCII ' ') and backspaces (ASCII BS) for right or left cursor motion, line feeds (ASCII LF) and "cursor up one line" commands ("\ESC[1A") for down or up cursor motion.

No other escape sequences are ever output.

vt52 expects to be installed with a filename that contains the name of the device used by ctext to prepare the input to the driver. In the standard package, this driver is installed under the name "vt52".

RETURNS

vt52 returns success if able to read all of its input <files>. It returns failure (but outputs no diagnostic) if unable to do so.

NAME

vtyper - drive Varityper Comp/Edit photocomposer

SYNOPSIS

vtyper <dev> -[cl## f* i* pl## t* +# #] <files>

FUNCTION

vtyper converts the ctext intermediate language into a command stream suitable for a Varityper Comp/Edit phototypesetter, driven through the STS (Slave Typesetter) interface. vtyper reads an intermediate command stream from each file in <files>, writing a converted command stream to <dev>, which vtyper treats as a filename that names the communications link to the typesetter (presumably a serial line). If no <files> appear, STDIN is read; a filename of "-" also causes STDIN to be read, at that point in the list of files.

vtyper reads a private table file to learn what typestyles are mounted at what turret positions in the typesetter. (ctext itself has no access to this information.) This table file consists of one or more lines, each containing two fields: first, the full name of the typestyle (as it appears in a ctext devicestyle table file); and second, the "style number" corresponding to the turret and row position where the typestyle appears. Style numbers count up from one (turret A, row 1), with adjacent rows on a type disc having consecutive numbers and adjacent type discs having consecutive ranges of numbers (e.g., style number 4 is turret A, row 4; style number 5 -- on a model 5410 -- is turret B, row 1). The typestyle name is scanned as a single string with no embedded whitespace; the style number is scanned as a decimal string, and is separated from the name by whitespace.

Flags are:

-cl## specify a "cassette length" of ## inches. When a vertical motion command is encountered that would cause vertical leading to exceed this amount, vtyper presumes the paper takeup cassette on the typesetter is full. vtyper leads up to the full cassette length, then displays a "cassette full" message to the typesetter console, issues a "suspend processing" command, and waits for a "resume processing" command to be typed at the console keyboard. Default is 110 inches (ten 11-inch pages).

-f* use * as the name of the typestyle location table file. By default, the name "vtyper.in" is used.

-i* write the string *, followed by a carriage return (ASCII CR), to <dev>, before entering the slave typesetter protocol.

-pl## specify a "page length" of ## inches; this length is used only in conjunction with the "+#" and "-#" flags described below. Default is 11 inches.

-t* treat * as a series of directory prefixes, with adjacent prefixes separated by a vertical bar '|'. Each prefix in turn will be prepended to the typestyle location table filename in searching for the file. By default, the single Idris/UNIX directory /usr/ctext is used.

+# start output at page #, where pages are measured by "-l#" and numbered from one. By default, output begins with the first page.

-# stop output after page #, where pages are measured by "-l#" and numbered from one. By default, all pages are output.

vtyper uses a relatively small set of the commands defined by the slave typesetter interface. Specifically: text, of course, is output with "escape then expose" triplets, a one-byte character position followed by a two-byte horizontal escapement, with the escapement output as a two's-complement integer, less-significant byte first. Changes in point size are output as "select point size" (PS, or 0x93) commands; changes in devicestyle are output as "select style" (FT, or 0x92) commands. Vertical motion is output as one or more "feed exposure media" (LD, or 0x94) commands; horizontal motion is output as one or more "escape then expose" triplets, with a (non-existent) character index of 0x72 given -- this causes an escapement only.

Several other commands are used to administer the line protocol to the typesetter. The "suspend processing" command (SP, or 0x98) is used to halt the typesetter so that the paper cassette can be changed, or when an error condition reported from the typesetter (like being out of paper) requires it. Any "suspend processing" command is always preceded by a "display message" command (DM, or 0x9b), to notify the operator of why the suspension is occurring. When the typesetter notifies vtyper of an error condition, other commands may be used in remedying the condition. These are: "request status" (RS, or 0x9c), "resume processing" (RP, or 0x99), and "initialize" (GO, or 0xa2). Finally, vtyper always terminates a run by sending an "end of job" command (ET, or 0x97).

vtyper expects to be installed with a filename that contains the name of the device used by ctext in preparing input to the driver. In the standard package, this driver is installed under the name "vtyper".

RETURNS

vtyper returns success if able to read all of its input <files>, and able to transmit them successfully to the typesetter. No diagnostic is issued if an input file does not exist; any problem in talking to the typesetter, however, will be indicated with an appropriate message.

BUGS

vtyper has been tested with versions of the slave typesetter interface up to revision seven (released in March, 1983). As of that revision, the DM command did not work properly. The use of that command by vtyper is believed correct; however, this remains unproven. vtyper was developed using a Model 5410 Comp/Edit system; hence, there was no opportunity to use the new commands for the Model 6400 digital typesetter.

III. Ctext Device Interface Functions

NAME

Functions - ctext device interface functions

FUNCTION

Almost all of the ctext system is portable, not only across different implementations of Idris, but across the full range of operating systems where Whitesmiths C is supported. The only dependencies on an underlying operating system are in the device driver programs, and these dependencies, in turn, have been isolated into two areas.

One system dependency involves how a terminal line can be made seven or eight-bit transparent. The other dependency is how (and if) a mechanism can be constructed to time I/O, providing a timeout capability if an operation is not complete within a specified time limit.

The descriptions in this section are portable specifications of the functions used by the device drivers to provide these capabilities. Any implementation of ctext should provide functions compatible with these descriptions, and (hopefully) providing the functionality described.

NAME

mktran - make or unmake transparent data transfer connection

SYNOPSIS

```
COUNT mktran(fd, mode)
    FILE fd;
    COUNT mode;
```

FUNCTION

mktran modifies the file associated with fd, presumably designating a terminal, to enable or disable the transparent transfer of data through the file. If (mode == 1), mktran sets up 7-bit transparent transfer of data; if (2 <= mode), mktran sets up 8-bit transparent transfer of data. If a preceding call to mktran was made with a mode of 1 or 2, then a call with (mode == 0) will restore the terminal characteristics in effect before the first such call; otherwise, a call with (mode == 0) will set up a "normal" (i.e., interpreted) terminal state, which, depending on the underlying operating system, may not be 7-bit transparent.

If (mode < 0), mktran does not change the terminal state; such a call simply reports the current state.

RETURNS

mktran normally returns the mode value (0, 1, or 2) best corresponding to the state of fd before the current call. mktran returns -1 if unable to read or (when required) write the state of fd.

BUGS

There is necessarily some variation in what mktran does under different operating systems; in general, it changes as little as it can to assure the selected transparent connection.

NAME

tmcall - name function to be called on timer interrupt

SYNOPSIS

```
BOOL tmcall(pfn)
    VOID (*pfn)();
```

FUNCTION

tmcall tries to assure that the function whose address is passed in pfn will be called whenever a timer interrupt occurs. (The function tminit initiates the timer; unless tmclr is called before the period specified has elapsed, a timer interrupt will be generated.)

If (pfn == NULL), tmcall will cause timer interrupts to be ignored. If (pfn == 1), tmcall will cause the handling of timer interrupts to revert to the system default. If a timer interrupt occurs before tmcall is called, this default handling will also be in effect.

RETURNS

tmcall returns YES if the handling implied by pfn was successfully associated with the timer interrupt, or NO if any errors occurred.

SEE ALSO

tmclr, tminit

BUGS

On some systems, this routine (and its kin) may be dummies.

NAME
tmclr - clear timing

SYNOPSIS
BOOL tmclr()

FUNCTION
tmclr clears any pending timer interrupts, by causing the timer to be halted. If no timing is being performed, tmclr does nothing.

RETURNS
tmclr returns YES if timing has been halted, NO if any errors occured in trying to do so.

BUGS
On some systems, tmclr (and its kin) may be dummies.

NAME

tminit - initiate timing

SYNOPSIS

```
BOOL tminit(delay)
    COUNT delay;
```

FUNCTION

tminit attempts to cause a timer interrupt to be generated after the number of seconds specified by delay. Presumably, this interrupt will cause control to be transferred to the routine already named to tmcall.

Only one timer may be in use at any given time; that is, either a timer interrupt or a call to tmclr must occur between each call to tminit.

RETURNS

tminit returns YES if timing was successfully initiated, or NO if any errors prevented it.

SEE ALSO

tmcall, tmclr

BUGS

On some systems, tminit (and its kin) may be dummies.

NAME

tmwait - wait for period given

SYNOPSIS

```
BOOL tmwait(delay)
    COUNT delay;
```

FUNCTION

tmwait attempts to wait for the number of seconds specified by delay. No interrupt of any kind occurs at the end of this period; tmwait simply returns to its caller.

RETURNS

tmwait returns YES if the wait occurred, NO otherwise.

BUGS

On some systems, tmwait may be a dummy.

Appendix A

CTEXT AND THE FILESYSTEM

Much of the information ctext needs about individual output devices and the fonts associated with them resides in table files in the host filesystem. A table file is an ordinary ASCII textfile, whose first line is a special ctext command called <:mark:>, which identifies what the file defines. Following the command are lines of information, generally of fixed format, which may be interspersed with further <:mark:> commands, due to the file naming restrictions explained below. In any case, the contents of a table file are strongly line-oriented, and each new information unit or command must begin on a new line.

All ctext table files live in one of the directories on a user-specified "search path". By default, the Idris/UNIX path "|/usr/ctext" is used, which looks for files first in the current directory, then in the directory /usr/ctext. For maximum portability, this directory contains no further directory structure, but only ordinary files, with names of the format "xxxxxx.xx" (six-character name, two-character suffix). The first part of each filename is simply the first six characters of the full name of whatever the file defines. The first letter of the suffix identifies the class of device to which the definition applies, while the second letter gives the type of definition. An exception is a devicespec file, the extension of which is always ".ds". Device class letters are user-selectable (because defined in <:devicespec:> commands); default device letters are:

LETTER	DEVICE CLASS
d	Diablo-compatible printer
t	"tty" -- either plain printer or VDT
v	Varityper Comp/Edit photocomposer (5810/5900/6400)

File types are:

LETTER	FILE TYPE
c	charset definition
d	devicefont definition
f	fontset definition
ds	device specification file

Thus the file db640.ds might define the specifications of a Diablo model 640 printer, while goudyo.vd might define the devicefont Goudy Oldstyle for the Varityper photocomposer.

Because of the severe restraint on name length, a given table file may have to define more than one charset, devicefont, or fontset (if you want access to two or more whose names share the same first letters). So long as each <:mark:> command in the file begins on a new line, the corresponding definition will be correctly processed.

Appendix B

CTEXT INTERMEDIATE OUTPUT

The output from the formatter itself is not suitable for direct reproduction on any device. Instead, it is intended for submission to a driver for the output device desired. This intermediate output contains formatted text, interspersed with command bytes. Because the formatted text is not necessarily ASCII characters, the interpretation of command bytes is context-dependent.

Specifically, the components of an intermediate output file will always be the following:

1) a "device name follows" command, containing the name from the <:device:> command read at the start of the run.

2) a series of "font initialization follows" commands, one for each devicestyle eventually used by the <:fontset:> command read at the start of the run.

3) the formatted text, consisting of horizontal and vertical motion commands interspersed with escaped buffers of characters. That is, bytes to be sent to the output device will always be included as part of the buffer named by an "0240" command, so that they cannot be misinterpreted as command information.

Further notes about intermediate output:

1) the strings specified as "init" or "final" parameters for a devicestyle will be sent as buffers separate from any other output.

2) font and size change commands will appear between buffers of formatted text.

3) a "box level 1 ends" command follows the intermediate output for each line; a "box level 2 ends" command follows the intermediate output for each heading/footing, and the running text for each page; a "box level 3 ends" command follows the output for the entirety of each page.

Broadly speaking, command information serves three functions: it specifies horizontal or vertical motion, alters some driver state, or passes user information through to the driver for its own consumption. These three classes of command bytes are detailed below.

B.1. Cursor Motion

030x specify horizontal motion. The low five bits of the command byte are added to the left of the next byte in the output stream, to produce a two's-complement integer specifying a distance in device horizontal rasters.

034x specify vertical motion. The low five bits of the command byte are added to the left of the next byte in the output stream, to produce a two's-complement integer specifying a distance in device vertical rasters.

B.2. State Manipulation

0200 change font. The next byte in the output stream specifies the internal index of the new font (as established by font initialization).

0201 change size. The next two bytes in the output stream specify the new size in which the current font is to be set, in device vertical rasters. The size is written as unsigned, less significant byte first.

B.3. Information Pass-Through

0240 escape a buffer of characters. Use next byte in the output stream as a count of the number of characters immediately following that are to be passed to the output device with no interpretation.

0241 escape a string of characters. Use next byte in the output stream as a delimiter, then pass bytes to the output device, with no further interpretation, until the delimiter is seen. The delimiter itself is discarded. Two consecutive delimiters cause one delimiter to be passed to the device, and escaping to continue.

0244 device name follows. The next bytes in the output stream, up to a terminating ASCII NUL, contain the name of the output device this stream is intended for, as given on the command line to ctext.

0245 font initialization follows. The next byte in the output stream is an internal font index; following it is the full name of the "font" (i.e., devicestyle) to be associated with the index, which is taken to continue up to a terminating ASCII NUL. The two low-order bits of the index indicate the bold/italic characteristics of the font: if the 1-weighted bit is on, the font is bold; if the 2-weighted bit is on, the font is italic.

0251 box level 1 ends. Currently, this byte signals the end of the intermediate output for each output line.

0252 box level 2 ends. Currently, this byte signals the end of the intermediate output for the interior ("running text") of each page, and for each heading and footing. If heading and footing output is disabled (by setting <:ispace:> and <:ospace:> to zero), the corresponding control bytes are not sent.

0253 box level 3 ends. Currently, this byte signals the end of the intermediate output for the entirety of each page. It is always sent at that point.

Appendix C

ASCII CHARACTER MNEMONICS

Within string constants, ctext accepts special escape sequences to provide symbolic equivalents for arbitrary eight-bit bytes that are not printable ASCII characters. An escape sequence consists of a backslash '\' followed by one of: 1) one of the letters "btnvfr" (in lower case), to represent ASCII BS, HT, LF, VT, FF, CR; 2) one to three digits representing the value of the byte (these are taken as octal if preceded by a '0', and as decimal otherwise); or 3) one of the ASCII mnemonics listed below for a "control character" (these must be given in upper case).

MNEMONIC	VALUE	MNEMONIC	VALUE
NUL	000	DLE	020
SOH	001	DC1	021
STX	002	DC2	022
ETX	003	DC3	023
EOT	004	DC4	024
ENQ	005	NAK	025
ACK	006	SYN	026
BEL	007	ETB	027
BS	010	CAN	030
HT	011	EM	031
LF	012	SUB	032
VT	013	ESC	033
FF	014	FS	034
CR	015	GS	035
SO	016	RS	036
SI	017	US	037

Appendix D

INTERNAL LIMITS AND STANDARDS

The representation of data within ctext has two significant areas of impact on users: first, the processing of user-defined names in the symbol table and elsewhere, and second, the handling of numeric values in expressions and elsewhere.

D.1. User-defined Names

Names connected with fonts (i.e., fontset, font, stylecode, charset, or devicestyle names), are significant to at least 32 characters. All other names are significant to 14 characters. Longer names may be given, in either context, but characters beyond these maxima will be ignored.

D.2. Numbers

Numeric input is initially treated as floating-point (i.e., as a C-language "double", roughly, a double-precision number). If a number is immediately followed by one of the recognized "unit specifiers", it is taken to mean a distance, and is converted to device rasters with the appropriate multiplier. The use of either horizontal or vertical rasters is unambiguously implied by all commands that expect a measurement as an argument. Within a <:defctr:>, or anywhere else where a measurement cannot be anticipated, a measurement is taken to mean a vertical distance.

Expressions are permitted wherever a numeric constant may occur; numbers are retained as floating-point until any surrounding expression has been evaluated. Then, they are converted to signed short integers by rounding away from zero. If the integer part of the result is not representable in 15 bits plus sign, a warning message is output, but processing continues. (To be precise, results in the range [-32767, 32767] are permitted, because the value -32768 is pre-empted by the processor for its own use.)

Four values associated with output line control are maintained differently. The baseline and spacing characteristics of a devicestyle (given in the <:mark:> command for the style), and the <:spacing:> and <:spread:> characteristics of a galley, are maintained as signed short integers with an implied fractional part of three decimal digits. Thus numbers given for these values are significant to only three fractional digits, and numbers outside the range [-32, 32] may not be stored correctly.

Appendix E

HYPHENATION

ctext attempts to hyphenate words when, in creating a line of filled text, the next word to be output will not fit entirely within the current galley width. ctext performs hyphenation as two separate operations. First, potential hyphenation points in the word are marked; then, the latest possible point is selected for actual hyphenation.

Hyphenation points come from three sources. First, of course, are the "soft" hyphens indicated by the user in his input text. If a word contains soft hyphens, then no further hyphenation points are sought. Otherwise, any pre-existent hyphen in the word is considered a hyphenation point, so long as at least one vowel appears both before and after the hyphen. Thus, "pre-existent" contains such a hyphenation point; "1-weighted" or "<cs-name>" does not. Finally, if the word contained no soft hyphens, ctext tries to generate more potential hyphenation points from each non-hyphenated portion of the input word. It does so by conditional suffix stripping, followed by the marking of likely hyphenation points between adjacent pairs of letters in the resulting word prefix. The suffix stripping is often conditioned on the letter immediately preceding the suffix, to try to reduce invalid suffix removal.

This algorithm is applied to any non-white string appearing at the end of a line of filled text; the only special treatment added is that any of a set of "punctuation" characters appearing at the end of a word are removed before suffix stripping. The punctuation characters are ".,;':)!?]" and double-quote.

Appendix F

ERROR MESSAGES

Three kinds of diagnostic messages may be output during a ctext run. First, ctext outputs a few messages as warnings; these indicate that an unexpected but not fatal condition has arisen. After such a message, processing continues, though the resulting output may no longer be correct.

A larger class of error messages are output only when an error has occurred that makes further formatting impossible. Most of these concern the font mechanism; driven as it is from many different table files, the possibilities for erroneous setup are numerous.

Finally, aside from the messages documented here, ctext may (but should not) output abort messages indicating that some inconsistency in its internal state has been caused by nominally correct input. These messages will always contain an '!', and should be immediately reported to the maintainers.

F.1. Warning Messages

can't include <fname> - the file <fname>, given as the first argument to an <:include:> command, cannot be read. Note that ctext tries to open a file with exactly the name given; the table file prefixes (specified with "-t*" on the command line) are not used by the <:include:> command.

integer conversion overflow - the result of an arithmetic expression cannot be represented as a signed short integer. It has been stored in truncated form, but obviously not in its full magnitude, and may cause future arithmetic to go awry.

undefined name: <user-name> - the command name <user-name> was invoked, but had no definition at the time of invocation. This message can be output only if "-u" is given on the command line to ctext.

F.2. Fatal Error Messages

<:deffont:> must precede user text - no <:deffont:> command was encountered before the first input that creates formatted text. Generally, <:deffont:> commands are included in the ".xf" files named by a <:fontset:> command.

<:device:> must precede <:deffont:> - no <:device:> command was encountered before the first <:deffont:> command read. The <:device:> command provides the name of a ".ds" file containing a devicespec <:mark:> command. Since this <:mark:> command gives the "device letter" that helps to determine the names of all other table files referring to that device, no font processing can be done until the ".ds" file has been read.

can't invoke font <ft-name> - the name <ft-name>, which should have occurred as the first argument to a <:deffont:>, is not currently defined as a font name. Hence the <:font:> command that tried to invoke <ft-name> can't be executed.

can't open font table file: <fname> - the ".xf", ".xc" or ".xd" file <fname>, one of the table files used by the font currently being defined, could

not be found in any of the directories given by the command line flag "-t*".

can't open initialization file: <fname> - the ".ci" file <fname>, used to specify an initial set of commands that ctext will automatically read at startup, could not be found in any of the directories given by the command line flag "-t*".

can't read charset: <cs-name> - the ".xc" file for the charset <cs-name> exists, but the <:mark:> command for <cs-name> was not found inside.

can't read devicefont: <df-name> - the ".xd" file for the devicefont <df-name> exists, but the <:mark:> command for <df-name> was not found inside.

can't read devicespec: <ds-name> - the ".ds" file for the devicespec <ds-name> exists, but the <:mark:> command for <ds-name> was not found inside.

can't read fontset: <fs-name> - the ".xf" file for the fontset <fs-name> exists, but the <:mark:> command for <fs-name> was not found inside.

ef contents not complete - the processor state at the conclusion of an ef (even-page footing) command did not match the processor state at the start of the command. Typically, this indicates that a macro invocation was left incomplete inside the footing. This message is output when the footing is actually used.

eh contents not complete - the processor state at the conclusion of an eh (even-page heading) command did not match the processor state at the start of the command. See "ef contents not complete".

empty font invoked: <ft-name> - the name <ft-name>, used in a <:font:> command, is currently defined as a font; however, no stylecodes are defined in the font, and ctext is unable to establish a new stylecode for further processing.

ff contents not complete - the processor state at the conclusion of an ff (first-page footing) command did not match the processor state at the start of the command. See "ef contents not complete".

fh contents not complete - the processor state at the conclusion of an fh (first-page heading) command did not match the processor state at the start of the command. See "ef contents not complete".

macro invocation too long: <macro-name> - an invocation of <macro-name> has been read that is longer than the maximum specified by "-ms#" on the command line. This message is output as a debugging aid; the length counting mechanism can be disabled by specifying "-ms0".

no devicestyle precedes user text - font initialization did not succeed in setting up even one devicestyle. Since none are defined, no output can occur. This message should be output only if neither a <:device:> command nor a <:fontset:> command precedes the first input that creates formatted text. (If any error occurs once ctext has started to define devicestyles, one of the preceding messages will result.)

of contents not complete - the processor state at the conclusion of an of (odd-page footing) command did not match the processor state at the start of the command. See "ef contents not complete".

oh contents not complete - the processor state at the conclusion of an oh (odd-page heading) command did not match the processor state at the start

of the command. See "ef contents not complete".

unrecognized <size-expr> in devicespec - a <size-expr> was given that ctext cannot deal with. Though arbitrary expressions may be specified in devicespec <:mark:> commands to relate relative character width to output character width, only the simple special cases <:sz:> and <:wd:>*<:sz:> can currently be interpreted.

Whitesmiths, Ltd.
97 Lowell Road, Concord, Massachusetts 01742