1618 Clawthorpe Avenue,
Victoria, B.C., Canada
V8T 2R8
May 3, 1987

Mrs. Susan DeAngelis,
Department of Computer Science,
University of Waterloo,
Waterloo, Ontario
N2L 3G1

Dear Mrs. DeAngelis:

This letter is my order for one copy of Research Report #CS-81-12. I am enclosing a cheque for $13.00 in payment for this report.

Thank you.

sent report
MAY 13 1987
S.D.

Yours truly,

[illegible]

Alan Haines

University of Waterloo

Waterloo, Ontario, Canada
N2L 3G1

Faculty of Mathematics
Department of Computer Science
519/885-1211

Telex Number
069-55259

April 21, 1987.

Mr. Alan Haines,
1618 Clawthorpe Avenue,
Victoria, B.C.
V8T 2R8

Dear Mr. Haines:

Thank you for your letter of March 30th, 1987.

I am out of stock of the report CS-81-12 but I could have it reprinted for you at a cost of 6¢ per page. The total cost would be $13.00. If interested would you please make your cheque or money order payable to the Computer Science Department, University of Waterloo and forward to my attention.

I have enclosed our 1986 research report listings and hope they are of interest to you.

Yours truly,

Susan De Angelis

Susan DeAngelis (Mrs.),
Research Report Distribution,
Computer Science Dept.

/sd

Encl.

1618 Clawthorpe Avenue,
Victoria, B.C., Canada
V8T 2R8
March 30, 1987

Department of Computer Science,
University of Waterloo,
Waterloo, Ontario
N2L 3G1

Gentlemen:

I recently came across a reference to a University of Waterloo publication, AN INCREMENTAL TEXT FORMATTER, #CS-81-12, by Mark Stuart Brader.

189 pages = 11.34 + 2.00 postage $13.34

Is this publication still available? Is it available to the public? Is there a charge? I would appreciate anything you can tell me.

On a related topic, is there any information available describing the internal methods and algorithms used in Waterloo Script?

?

Thank you.

Yours truly,

A.J. Haines

Alan Haines

BROWN UNIVERSITY *Providence, Rhode Island • 02912*

Department of Computer Science
Box 1910
(401) 863-3300

July 30, 1986

Mrs. Susan DeAngelis
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1

Dear Mrs. DeAngelis:

Thank you for your letter of 24 July. I have enclosed a check for $11.63 US ($15.50 CDN at .75 exhcange rate) which I hope covers reproduction of technical report CS-81-12 ("An Interactive Reformatter").

Sincerely,

Matthew Kaplan

Matthew Kaplan

Sent out
Aug. 11/86
S.D.

University of Waterloo

Waterloo, Ontario, Canada
N2L 3G1

Faculty of Mathematics
Department of Computer Science
519/885-1211

Telex Number
069-55259

INVOICE

July 24, 1986.

Mr. Matthew Kaplan,
Department of Computer Science,
Box 1910, Brown University,
Providence, RI 02912

Dear Mr. Kaplan:

I am in receipt of your request for our technical report CS-81-12.

I regret that this report is currently out of stock. The cost to have it reprinted would be $13.34 CDN (6¢ per page plus $2.00 postage and handling). If you are interested, would you please make your cheque payable to the University of Waterloo, Computer Science Department.

Thank you for your interest in our department.

Yours truly,

Susan DeAngelis

Susan DeAngelis (Mrs.),
Technical Report Secretary.

/sd

PROVIDENCE, RI 029
PM
10 JUL
1986

Dept. of Computer Science
University of Waterloo
Waterloo, Ontario

Matthew Kaplan
Department of Computer Science
Box 1910, Brown University
Providence, RI 02912

Dear

I would greatly appreciate ______ reprint(s) of your paper(s) Mark Stuart Brader, "An Incremental Text Formatter," CSTR-81-12

which appeared in ______

Thank you for this courtesy. Please bill if necessary.

Sincerely yours,

An Incremental Text Formatter

by

Mark Stuart Brader
Department of Computer Science
University of Waterloo
Waterloo, Ontario

Research Report CS-81-12
April, 1981

ABSTRACT

Conventional text formatting programs require an entire document to be reprocessed when it is amended in any way. EZ27 was an attempt to avoid this problem by producing an intermediate result corresponding to a galley proof, which could be edited before the final output was produced from it. Though it worked after a fashion, it lacked the flexibility achieved by existing formatters, and was abandoned.

ACKNOWLEDGEMENTS

To my supervisor, Prof. W. Morven Gentleman, for his advice and encouragement; to Profs. Kellogg S. Booth and Thomas A. Cargill, for their useful suggestions;

And to my wife, Catherine, for her patience.

CONTENTS

FIGURE

1. INTRODUCTION

"A formatter is an important tool for anyone who writes (including programmers describing their programs), because, once correct, material is never re-typed. This has some obvious cost benefits, and helps ensure that the number of errors decreases with time. Machine formatting eases the typing job, since margin alignment, centering, underlining, and similar tedious operations are handled by the computer, not by the typist. It also permits drastic format changes in a document without altering any text. But perhaps most important, it seems to encourage writers to improve their product, since the overhead of making an improvement is small and there is an esthetic satisfaction in always having clean copy available." [Kernighan 1976-S, page 219]

From the user's point of view, making a local change to a computer-formatted document is indeed a low-overhead procedure: read file into editor, make change, write to file, invoke formatter. But the conventional type of formatter then has to reprocess the entire document, though most of it will not change at all (except in rare "pathological" cases).

The amendment of a few words may affect only one paragraph: every other character in the document will be unmoved. Usually, even if the number of lines in the amended paragraph does change, elsewhere intact lines will merely be moved between adjacent pages, and nothing more.

An analogous situation exists with most programming language implementations. A small change forces the recompilation of at least an entire subprogram, and often the entire program. This is sometimes avoided by the use of an incremental compiler, which produces in addition to object code some type of intermediate result, by reference to which local changes can be compiled rapidly.

When a book is published, it is first typeset into galley proofs. These contain the full text of the book, but not in pages; they are long sheets of continuous text, divided into paragraphs, with all horizontal formatting complete. The galleys are proofread and the text edited at this point. Pages are then made up and vertically formatted, page number references are filled in, and the text can be edited again.

This thesis describes an attempt to make an incremental text formatter. It consists of Controller and Editor components and two formatting passes. Pass 1 takes input text with formatting commands, and produces a sequence of paragraphs, each properly formatted, along with commands controlling their vertical placement; the Editor may interactively revise the text; and Pass 2 then produces complete pages ready for the output device. The Controller interacts with the user, invoking the other components as desired.

Since the output from Pass 1 is more or less equivalent to a galley proof, the file on which it is written is named the _galley file_; the formatter's name EZ27 was chosen in honor of the proofreading robot in the story "Galley Slave" [Asimov 1957].

2. TEXT AND COMMANDS

When text is on a computer for formatting, its parts must be distinguished and the formatter instructed how they differ: where a paragraph starts, what is a centered line, what is underlined, what is a footnote. The commands supplying this information can take several forms.

2(a) No Explicit Commands (Automatic Recognition)

Suppose the text is entered into the computer exactly as it would be typed on a typewriter. Its layout could simply imply various commands to the formatter [Kimura 1978; Vergès 1972]. For instance, one line indented at the left would begin a new paragraph; a blank input line would give a blank output line; a line indented on both margins could be centered exactly. Tabular matter could also be recognized and treated specially.

If a formatter could identify and ignore on input whatever hyphens, formfeeds, and such things that it might insert in the text, then formatting a formatted file could reproduce the identical file as output: the formatter would be idempotent. In that case, only the most recent, formatted version of a document would need to be stored. Editing could be done directly on it, and the file simply passed through the formatter again: only what had to be adjusted due to the editing would change.

While automatic recognition is theoretically interesting, its practicality is doubtful. Infallibly distinguishing centered headings, for instance, from short paragraphs seems difficult at least, and if there exist cases where the recognition fails then the user must learn them, which negates the principal advantage of the method. Idempotency would also cause difficulties with footnotes, say, or automatic hyphenation: what is to show whether a word hyphenated at the end of a line was divided at an existing hyphen? Macros could be implemented only by a separate preprocessor (e.g. Trac [Cole 1976; Mooers 1965] or Max [Nudds 1977]). Finally, certain formatting functions would be most awkward to implement: changes of page dimension, or of font or size in typeset text. (Anyway, typeset output would probably be incompatible in coding with the input.)

Where this method can be useful, though, is in combination with any of the others described below: even if there is another way to produce a blank output line, for instance, a blank input line can conveniently have the same effect.

2(b) Commands From Special Keys

Word processors, self-contained devices for text editing and formatting, are now being produced in many varieties [Right 1978; Wohl 1977]. They are cheap enough for businesses to use them for their correspondence, and have thus had considerable impact on offices [Berenyi 1977; MacDonald 1978; Mitchell 1977].

The text is entered on a keyboard, and where some command is desired, generally a special key for that purpose is used (naturally, the available functions vary widely); other special keys edit the text. As befits a single-purpose device, this is a highly specialized method: the functions are actually designed into the keyboard. Accordingly, it cannot be easily adapted if some new function is wanted; nor is the method suitable for a general-purpose computer system unless special terminals are constructed. (Formatting can be done anew whenever the text is being printed, since the processor is otherwise idle then; the formatted text need never be stored, so the question of idempotency does not arise.)

2(c) Commands as Invisible Text

With the preceding method, the text is stored unformatted; therefore, when a formatting key is used, the fact must be retained by inserting a corresponding code in the text. These codes are the commands actually used by the formatter. They do not appear when text is printed, and are ignored as far as possible when it is edited, though they may be inserted or deleted along with adjacent text.

Such invisible commands can be used on a general-purpose computer; by retaining them in its output (and introducing new ones indicating, for instance, where a hyphenation was performed), a formatter can be truly idempotent. However, editing and printing software would have to be written to ignore commands in text, yet provide a way to enter them in the first place. (Nevertheless, this is the method adopted by the team now working under the International Standards Organization towards a family of languages for all levels of text processing [Card 1979].)

A variant of this method [W. M. Gentleman, private communication] would use short invisible commands as markers of significant places in the document. An auxiliary file of formatter instructions, read in parallel with the text file, would detail what to do at each place. This file could be edited in the ordinary way, making practical more complex, thus flexible, commands than in the methods described above.

2(d) Commands as Normal Text (Conventional Method)

The commands can also be character sequences that are special to the formatter but not to the editor used to enter and amend the text. Thus, again, the commands can be edited in the ordinary way, so they can have considerable complexity and flexibility. The formatter and editor can then be entirely separate, written, maintained, and even used individually. (In particular, such a formatter can easily be added to a computer system that has none, but does have an editor. Using the same editor for text and programs also saves learning time [Mashey 1976].)

Because of these virtues, this has become the conventional organization for text processing on general-purpose computers. The fullest advantage has been taken on the UNIX system, where, in addition to a formatter (Nroff, described below) and editor, a host of auxiliary programs [Kernighan 1978-D] can be used on the same files, some as preprocessors or postprocessors for Nroff. It may be noted here that this preprocessing does conflict somewhat with the "file insertion" (discussed below) that is widely used with Nroff and other formatters.

(On the other hand, there are also formatters combined with an editor that acts only as an input preprocessor, designed for non-interactive systems; since the formatter could stand alone, these too are regarded in this thesis as being of the conventional type.)

Since this kind of formatter cannot be idempotent, the unformatted input text must be kept on file for editing; some users consider this defect serious.

2(e) The Specialized Editor-Formatter

The conventional method has another disadvantage: a separate, general-purpose editor may be less useful than one especially for editing (natural language) text. This presupposes a specialized data structure for text, which may as well serve as the input to the formatter; thus the editor and formatter best form a single package. A necessary adjunct to this is a facility to take input text and transform it into the specialized structure. The structure could be designed for formatted as well as (or instead of) unformatted text; then, transformation out of the structure, for output, would be required.

Since this method virtually requires that a complete text handling system be written, the designer has more flexibility than with a conventional formatter. The commands can be made invisible where this is desired, and can be entered by any of the methods described above, or even interactively while text is not being entered. The formatter can be idempotent as long as the text files considered are in the special structure. QUIDS [Coulouris 1976], discussed below, is an example of such a system.

3. AN ORGANIZATION FOR THE NEW FORMATTER

The distinctive property of EZ27 was to be the ability to edit formatted text with minimal reprocessing. A program that is only a formatter obviously cannot do this; clearly a combined editor-formatter approach is necessary, and EZ27 was designed that way.

EZ27 was implemented on the University of Waterloo Mathematics Faculty Computing Facility Time-Sharing System (TSS); this system already supports an excellent general-purpose text editor, QED [QED 1980] (now being replaced by a variant FRED [Gardner 1981]), which is used with conventional formatters, chiefly Roff (described below). Designing an editor for a new data structure that would compete satisfactorily with QED would be a major task. Instead, EZ27 was designed to accept as initial input text and commands organized as for a conventional formatter, so that QED could be used to prepare input. Of course EZ27 also includes its own Editor component, but this is a relatively simple affair intended for local corrections rather than major changes.

3(a) The Galley File

EZ27 was designed to operate in two passes, with editing permitted between them. The output produced by Pass 1 is written on the galley file, which therefore must contain all the information needed for Pass 2 to produce the final output. It also must be directly editable, with local changes in the text producing only local changes in the file. For practicality, Pass 1 should do most of the work.

In apportioning the processing between Passes 1 and 2, one should consider just what is likely to be affected by a change, say, of one word in a document. Within the affected paragraph, every word following the change will likely be moved to a new location. If this does not change the number of output lines in the paragraph, there will be no other effects anywhere. If it does, the output location of every subsequent paragraph may change, with some lines being moved between pages; this may change the total number of pages in the document; but that will almost certainly be all. (Far-reaching effects are possible, if unlikely. For instance, a piece of text may be associated with a footnote, or its page number may be mentioned elsewhere in the document. If that text happens to be moved to another page, the associated changes could cascade indefinitely.)

This suggests that the natural dissection of the formatting task is between paragraph formatting as Pass 1, and page formatting as Pass 2. The galley file should contain a list of paragraphs, with the complete text and formatting information for each. (As mentioned above, this is the reason for the name "galley file", and thus for "EZ27".) In the case of simple running text, Pass 2 then has merely to place the paragraphs to fill up pages, and handle the dividing of paragraphs between pages.

3(b) Paragraph Numbers

Since it is essential that the galley file be editable, each paragraph must be addressable; one fairly natural (and easy to implement) way is by <u>paragraph numbers</u>, much like the line numbers in many text editors.

In QED, a line's number is merely the number of lines before it plus 1, so it can vary with changes elsewhere.

QED works well only because most of the time the user does not specify an explicit line number: the operand line is addressed by default, or in terms of the last one, or is located by a context search for some pattern. It does not seem a good idea to allow context searches of the entire galley file, for it will not be in internal memory; the user should locate the desired paragraph by other means (from a printer proof, or by examining nearby paragraphs). Therefore, numbering with consecutive integers, as in QED, would not be best for EZ27.

Most other editors regard a line's number as its invariant label. Multiples of, say, 100 may be used initially, so that insertions can use integer numbers in the appropriate range. Alternatively, inserted lines may have numbers with decimal places, and consecutive integers be used initially; the number of places may be limited, so the numbers are really scaled integers, and then there is no functional difference between the two variants. The APL function editor's approach was chosen on esthetic grounds for EZ27's paragraph numbering. This uses the second alternative, but whenever the editor terminates it first reassigns the line numbers to consecutive integers to facilitate future insertions (among other reasons). Actually, this last effect should really be optional, in case the user is working from a printer proof.

In EZ27, then, paragraphs are assigned consecutive integers as they are put on the galley file, and decimals are used to specify intermediate numbers for insertions; but when one has finished dealing with a particular galley file, its paragraphs are renumbered with consecutive integers.

3(c) Practical Considerations

Though the galley file consists in theory of a simple sequence of numbered paragraphs, usage considerations must affect its implementation. The following operations are frequent, and each should be reasonably efficient: sequential reading of paragraphs; random reading of one paragraph; replacement of one paragraph by a new version; random deletion of one paragraph or a group of consecutive ones; insertion of one or many paragraphs at a random point; locating the final paragraph (for insertions after it); moving a paragraph.

Most processing of the text and commands should be done in Pass 1, but there are some formatting operations that inherently must be deferred to Pass 2: starting a new page, for instance, or handling a footnote. Thus, commands for these must be put in the galley file along with the paragraphs. The approach adopted, in order to keep Pass 2 simple, was to parse and validate the commands in Pass 1, and write on the file a coded form, called an <u>opcode</u>. Most opcodes are logically located between particular paragraphs; therefore, a structure should be chosen where they can resemble paragraphs, and opcodes and paragraphs can be numbered and otherwise handled in the same fashion.

However, some Pass 2 commands (such as footnotes) must be associated with particular positions in the text, even in mid-paragraph, and may require considerable amounts of text as operands. The latter property can be provided by a form of opcode that resembles a paragraph in that it has text, but is still marked as an opcode in some way; for the former, an invisible command must be inserted in the text indicating the place associated with the opcode.

4. A HYPOTHETICAL FORMATTER

The terminology of text formatting is used quite divergently by different writers. This thesis attempts to be consistent, rather than following its sources, and introduces some new terms. Any underlined term anywhere in this thesis is taken to be defined by the context in which it then occurs, and will be used consistently with that meaning throughout. (By contrast, a few terms in quotation marks are local to particular sections.)

Likewise, many writers have described the features they feel are desirable in text processing systems [Landau 1971, especially pages 135-156 and [Schatzkin 1971; Seybold 1971; Tunnicliffe 1971]; Muir 1972]. Here, the author lists what in his opinion are the essential functions of a good conventional formatter, in terms of a hypothetical one called Hypo. Some additional features that are useful but not essential are described as belonging to Hyper, considered as an enlarged variant (a superset) of Hypo.

The subheadings in this and the following section are included for ease of cross-reference, as all topics are covered in the same order throughout. Their exact meanings are not to be taken too strictly.

Commands and Names

The input consists of _general text_, which is _plain text_ interspersed with commands, distinguished syntactically. When several commands occur with no plain text intervening, a slightly clearer and more compact syntax called a _command list_ can be used. The input is in the character set of the implementation computer; the output, _formatted text_, in the appropriate code to drive a (certain) typesetting machine.

For every Hypo and Hyper command that affects the processing of subsequent text, there is another command, or another option of the same one, that has the opposite effect; this will generally not be mentioned below. The _control characters_, which distinguish commands and such things (not to be confused with, say, ASCII control characters), can be reassigned by the Control command.

Most commands (wherever it makes sense) that take a numeric argument redefining a formatting parameter (margin position, type size, etc.) or a variable (see below) allow it to be specified as a _relative value_, by implication added to, or subtracted from, the previous value. In Hyper, expressions can be used in place of numbers, and a _scale designator_ can be appended to numeric arguments so that physical dimensions can be expressed in inches, centimeters, points, picas, etc., and in _relative units_ proportional to the type size.

Symbolic names, _macros_, can be assigned to general text by the Define command. The syntax denoting an _insertion_, or _invocation_, of a macro allows arguments; the macro text supplied to the Define command may include markers indicating where the text of each argument is to be inserted. Symbolic names can also be assigned to integer values; these are _variables_. In Hyper, symbolic names can also be assigned to formatted text; since these cannot be invoked with arguments, they do not qualify as macros, so they are called _strings_ (in this case, _formatted strings_; they are not to be confused with _character strings_, which are simply sequences of characters). All these symbolic names may be reassigned many times. Macros and variables may be inserted within commands as well as in plain text. In Hyper, there are some predefined functions that are invoked like macros but produce a numeric result like variables; they compute such things as the output size of some text.

The variable "pnum" always contains the current page number, and may be reassigned.

Ordinary Horizontal Effects

Any type size and font available on the output device can be obtained by using the Size and Font commands respectively; the latter's possible arguments are predefined symbolic names. Hyper also has an Under command, which causes characters to be underlined on output.

By default, each output line is filled with as many words as fit, and then pad justified, meaning that the left and right margins are aligned by expanding blanks in the line.

The Break command interrupts filling, forcibly terminating the output line currently being built. Several other commands (those for which it makes sense) cause such a break as a side effect. Filling can be turned off altogether by the Nofill command: in nofill mode, each newline in the input implies a break.

Other justification modes, selected by the Just command, are left, right, and center, in which each output line is not padded but may be shifted to be aligned by one margin or the center. (In pad mode, any line ended by a break is left justified.) Hyper also has in and out modes, which are respectively equivalent to right and left on odd-numbered pages, but vice versa on even. The Title command produces a three-part title line, of which the first part is left justified, the second centered, and the third right justified: this is handy for titles, and simple table entries.

When filling lines, words may be automatically hyphenated to improve the fit. The Hyph command activates this, and can also specify <u>discretionary</u> hyphenation points different from those chosen automatically.

The left and right output margins can be reset by the Margin command. The Indent command gives <u>temporary</u> margin changes, for just one output line, as at the beginning of a paragraph. A default <u>indentation</u> (amount) can be specified, so that it need not be given anew at each <u>indent</u> (instance). In Hyper, temporary indents may apply for any (specific) number of output lines, and successive ones to the same margin may be nested.

<u>Special Horizontal Effects</u>

Hypo's Tab command sets a <u>tab character</u> and <u>tab stops</u> for formatting tabular data. This works best in nofill mode. At each occurrence of the tab character, the current output position is calculated, then advanced to the next tab stop. Associated with each tab stop is a justification mode, which is applied to the line fragment that begins there, using the tab stops as alignment margins.

Hyper's Table command is quite different and rather more general. It is given one or more <u>table formats</u>, each being a list of the justification modes, and optional width information, for each column in the table; then this is followed by a list of table entries, which can be general text, not just simple line fragments. After the whole table has been assembled, the appropriate column widths analogous to tab stops are computed, and then the table is produced.

Hyper's Column command gives <u>multiple-column output</u>: the output page is partitioned into two or more more columns of equal width, the output simply being formatted as though the page width were rather smaller, and the small pages then set in parallel. Hyper's Newcol command skips output to a new column.

Hypo's Merge command defines a merge pattern. This is a line of characters that is compared with each output line; wherever the regular output line has blank space, the character in corresponding position in the merge pattern is printed. Vertical rules can be obtained this way, as well as leadering, but the latter is somewhat awkward since extraordinary blanks (see below) may have to be used in the text to prevent leadering being inserted where it does not belong. Hyper has a Leader command that specifies the exact position to insert leadering, and a string to repeat as the leadering. In Hyper, another way to get vertical rules is with the Table command.

Vertical Effects

The Space command leaves a vertical space of specified size in the output, or, in Hyper, it may advance to a certain position on the page. The body or body size of the output type, which is the usual line spacing as measured from baseline to baseline, can be reset by the Body command.

As implied above, the output is finally divided into pages, which are numbered, normally, consecutively from 1. The Page command forces output to skip to the next page, or the next odd or even page. The Height command resets the size of a page.

By default, each page has a certain amount of top margin and bottom margin which is left blank; the sizes may be reset by the Marg command. The Head or Foot command, respectively, sets a heading or footing to be printed in those margins of every page; Hypo allows multiple headings and footings, and they can be three-part titles. The current page number may be inserted into the heading or footing when it is printed. In Hyper, headings and footings are considered simple _reusable traps_. Traps may be set at any page location, and may include general text (any commands in the trap are executed when it is _tripped_), and thus also, in effect, formatted text.

Footnotes also can contain general text, and are set by the Footnote command. They are placed at the bottom of the first page where there is room, after the command is encountered. In Hyper, footnotes are generalized to _one-time traps_, which are like reusable traps, except that they disappear after being tripped. Both kinds are set by the At command.

Text may be considered a sequence of _paragraphs_ delimited by breaks. Hypo will not allow a paragraph to be divided between pages if one portion would be very short. Instead, a blank line is placed at the bottom of the page, and the line that would have been printed there starts the next page. This _widow elimination_ feature may be disabled, or the critical "very short" length increased above one line, by the Widow command. All this also applies in Hyper to a paragraph divided arcund a trap or between columns. In Hyper, _glued text_, a section of general text that must not be divided, may be specified explicitly by the Glue command; Hyper's Float command gives _floating glued text_, which must not be divided but need not print immediately (other plain text could fill the page).

Input/Output Effects

General text may be inserted not only from a macro but from a file, by the Read command. File insertion enables a large document to be edited in parts of handy size (this thesis comes from over 30 files); also, it allows a set of common macros or initializing commands to be used with several different documents.

The Index command appends formatted text to an index file, as for a table of contents (with page numbers obtained from "pnum"). In Hyper, there is also a Write command, which appends text still in the input form to a work file, which could later be used in a Read command. Any variables and macros invoked in the text are inserted, but commands are copied and not executed. Hyper also has a System command, which is like Write except that the text is passed as a command to the system instead of written to a file, and (where the installation allows) a Pipe command, in which the text is sent to another program.

The If command gives conditional processing of some general text, and thus conditional execution of commands; the Note command is for unconditional omission, giving comments. Hyper also has a While command that causes some general text to be inserted in the input repeatedly until some condition is false. Conditions are specified by relations among numerical expressions, or by string equality or inequality.

Since some input devices may have a limited character set, the Case command is available to provide automatic translation of case escape sequences. That is, a case escape character can be chosen, and each letter in the input translated to upper case, or lower case, or simply the opposite case, except when preceded by that character, which reverses the case of the letter and itself vanishes.

Since the typesetter's character set may be larger than the computer's, the Char command is available to obtain the additional characters. It can change the normal output association of some input character otherwise unused (such as an ASCII control character, if ASCII is the character set), to be some output character otherwise unavailable; another option simply makes the command behave like a string whose value is the desired character. Also obtainable with Char is the _extraordinary blank_, which prints as a blank but is otherwise treated as an ordinary graphic character.

The Literal command takes the following (specified amount of) input quite literally, ignoring any characters with special syntactic meaning. (If this would be needed frequently, the Control command should probably be used.) In Hyper, the Trans command controls automatic _transliteration_ of output, which provides an alternative to the Char command, and another way to output control characters: any output character can simply be mapped into any other.

5. A PARTIAL SURVEY OF TEXT FORMATTERS

Before choosing the commands and syntax of EZ27, various conventional formatters were studied. Those, and some others, are compared in this rather long (but perhaps interesting in its own right) section. First, CypherText, DIP, FORMAT, NROFF (with -ms, TROFF, EQN, and TBL), PAGE-1, PROFF, QUIDS, Roff (and Vroff), RUNOFF, SCRIPT, TEX (with Basic format), and TYPE are described in detail; then, some distinctive features of various other formatters are described.

For uniformity's sake, the various names will be written with only the first letter capitalized (TEX will be called Tex, though properly it would have to use Greek script letters in lower case). Likewise, the present tense is used throughout, though, as the reference dates indicate, some descriptions are surely out of date, albeit the most recent found. (The Honeywell Computer Journal is defunct, for instance.) Some of these names have been reused for several variant or even entirely different formatters, for instance "Runoff" [Roistacher 1974], but each description makes clear which formatter is being described.

Each description (including Hypo, as mentioned above) lists topics in the same order. Several features that are more or less identical among most or all of the formatters are described below; each detailed description assumes these, unless otherwise stated.

The formatters are of the conventional type. Input is in the form of a file of ordinary characters, consisting of general text, made up of lines separated by newline characters (sometimes represented as "<nl>"; thus "<nl>:" means a newline followed by a colon). The output from the formatters falls into two types: monospaced (having all characters the same width, like these) and typeset.

Filling and pad justification are default, but nofill and left and center modes are available. There is a break command, and some other commands also cause breaks. Automatic hyphenation is optional, and discretionary hyphenation points are supported.

The formatters of monospaced text support underlining; those with typeset output allow selection from a range of type sizes and, independently, fonts. Body size can be selected, and vertical space (blank lines) left where desired.

The left and right margins can be reset; the left margin can be changed temporarily for the first line of a paragraph.

Output is automatically divided into pages. The page height and top and bottom margin sizes can be reset. There are commands to start a new page, and to reset the page number.

File insertion is supported.

5(a) Cyphertext

Cyphertext, designed at the Cyphernetics Corporation (Ann Arbor, Michigan), is a general-purpose formatter for typeset text. It can drive various typesetters and also produce monospaced proofs. The implementation language is PDP-10 assembler because "FORTRAN was felt to be too awkward and inefficient" for the purpose and nothing else was available.

The source [Moore 1970] for this section is not a complete manual; certain points are not explained.

Commands and Names

A "/" begins or ends a command list; commands within a list are separated by ";"'s. A command may be followed by a space and one or more arguments delimited from each other by ","'s. Extra blanks are allowed at least after a "," or ";", and newlines appear to be equivalent to blanks everywhere. The "/" at least can be set to another character, but apparently not within a document.

Strings and macros are both supported, but macros may only be used as commands, whereas strings can be inserted only as command arguments or in plain text. (And there is the "map" command, which assigns a string to a 1-character name; every time that character occurs in the input, the string is inserted at once, before any syntax scanning for commands.) To insert a string (of the ordinary kind) into text, one uses an "@" followed immediately by the string name (presumably delimited by a following blank), or alternatively the string name is mentioned in an "include" command: "@string" or "/include string/". Macros, or strings used as arguments, are simply referred to by name in the command list: "/macro argument, argument/" or "/command string, otherargument/".

The text assigned to a macro or string is given within the command list as an argument, delimited itself by either "″" or "'" characters (the reference is actually typeset with "“" and "”" characters, but these are very rare on computer systems). A macro's parameters are referred to by local names listed in parentheses after its name. Thus "/define gap (amount), 'space amount'/" defines a synonym for the "space" command. (The macro text has no "/"'s since it can only be used as a command anyway.)

Strings in Cyphertext serve also as variables. The values assigned to them, and other numeric arguments of commands, may be given as expressions.

A predefined string gives the page number, but it is read-only: it cannot be reassigned. Other predefined names give the date, time, and output position.

Ordinary Horizontal Effects

There is no way to specify discretionary hyphenation points.

Special Horizontal Effects

Tabular output appears to be supported. Assigning (by "set") a value to the reserved name "field" creates a list of output columns, each with associated margin positions and justification mode (left, center, right, or pad). Partitioning of output between fields is by the "nextfield" command. It is not clear whether this mechanism can be applied to give right justification, and permanent changes to the left margin, in ordinary running text; no other way to get these effects is described. Useful in tables is the "leader" command, which repeats its single-character argument as a leader to fill the output line.

Vertical Effects

Cyphertext can automatically leave vertical space between paragraphs, as well as indenting the first line of each.

Reusable traps are supported at the top and bottom of the page.

Input/Output Effects

Conditional input can be obtained by the "skipif" command, which takes as arguments a relational operator and two numeric expressions (in that order). If the relation is true, commands and text are ignored (though parsed) until an "endif" command is encountered.

5(b) Dip

Dip was written at the Tata Institute for Fundamental Research (Bombay, India). It contains a device-independent portion that can handle monospaced or typeset text, and postprocessors for various typesetters; there is even a monospacing postprocessor for typeset text, as for proofing. Dip's intended user community was compositors rather than computer people: it "implements a minimal but complete set of composition primitives." The implementation language is FORTRAN IV, for portability.

The source [Mudur 1979] for this section is not a complete manual; certain points are not explained.

Commands and Names

A command occupies one line starting with "*", except that a "<" on that line effectively starts a new line; in other words, "<nl>*", and sometimes "<*", starts a command and "<" or a newline ends it. (This is not quite accurate since a "*" at the very beginning of the input would also indicate a command. For purposes of discussion it is being assumed, in fact, that a virtual newline always precedes the input, so that a newline does precede each line.)

The command name follows the "*" and may be given either in a long form or as an acronym: "alter left margin" or "alm". In the long form, 5 characters of each word are significant; the blanks between words are therefore mandatory. It is also possible to define a synonym for a command, or rename it altogether. Scale designators are allowed for physical dimensions.

Commands may always be specified with associated "scopes". The characters "<" and ">" form pairs, with nesting allowed; when "<" terminates a command, its scope is the general text up to the matching ">". This is recommended for relatively short sections of text. Also available are "blocks", apparently defined by paired commands, which render any commands within them as local to them. The character "*" at least can be reset.

Two blank lines in succession indicate a new paragraph, but "of course" this feature is optional; Dip also supports some special handling of punctuation on input, but just what is not explained, except that it can be changed by command. Since India is multilingual, Dip was designed not to be restricted to English text.

Macros are available, but no details of syntax are explained; the only example given is "in pseudo Dip (to avoid further explanation of Dip commands)".

Ordinary Horizontal Effects

Subscripting and superscripting, regarded as additional fonts, are supported.

Right justification is available.

The automatic hyphenation facility does not support discretionary hyphenation points, but does have a built-in hyphenation exception dictionary, whose use is optional. There is a command to update the exception dictionary.

Vertical Effects

There are commands for headings, footings, footnotes, and "blank space for figures". Vertical justification is available. More general "trap" concepts were specifically excluded as inappropriate to a compositor-oriented system.

Input/Output Effects

There are commands for conditional input, which are not discussed in detail. The current output position can be tested, as can whether the page is a right- or left-hand one. The conditional text is apparently specified like other commands' scopes.

There is a command to translate output into block capitals; no other case translation commands are mentioned.

Ordinary file insertion is not supported, but macro libraries (on random-access files) can be accessed dynamically.

5(c) Format

Format [Berns 1969] was written at IBM and "is a single, production, batch processing program ... best suited for documents which do not require the highest quality printing and which are characterized by high revision rates." Its output is monospaced. The implementation language is IBM FORTRAN IV.

Commands and Names

Input is initially in "control card mode", wherein each line is taken as containing one command. (The first three letters are significant; the rest of the line is only examined for arguments.) The "GO" command switches to "text mode", where input is taken as general text, and newlines are totally ignored (except in nofill mode). In text mode, the sequence ")" initiates a command list, and another blank terminates it; each command consists of one or two characters only. The "V" command switches back to control card mode.

On input, multiple blanks are interpreted the same as single ones; the "0-8-2 multipunch" character is an extra-ordinary blank.

Ordinary Horizontal Effects

There is no automatic hyphenation, though lines may be broken at actual input hyphens.

Functions affecting the document layout are generally separated by Format into two kinds of commands: "layout control" commands (of the control card type) specifying what is to happen, and "text control" commands (of the single-character type) indicating where. Margin changes of all kinds, and tabbing, work that way: one control card has a list of pairs of left and right indentations, another has a list of tab stops, and a third gives the indentation for the beginning of each paragraph. The corresponding text control commands then select a pair of indentations from the list (either permanently or as a hanging indent), advance the output to the next tab stop (filling with either blanks, or ".''s as leaders), or start a new paragraph.

Special Horizontal Effects

Multiple-column output is available.

Vertical Effects

A heading may be defined, though perhaps only one line is allowed; either footings (on a similar basis) or footnotes are also supported, but not both. (These points are not explained clearly, nor is the syntax of the relevant commands.) Page numbers may be printed in either one of the top corners, or both alternately.

A pair of commands can delimit a section of glued text; the alternative need makes the following (specified number of) output lines into glued text.

Input/Output Effects

Format does not support file insertion.

The "DIC" command generates an alphabetical list of all words of 4 or more letters in the input.

The main output can be duplicated (including side-by-side printer copies), sent to tape, or punched.

There is no conditional input, nor any way to enter an ordinary blank followed by a ")" literally.

Input can be from tape, and can be specified as being in upper and lower case, or IBM Model 026 or 029 keypunch characters. When it is in upper case, it is mapped to lower case, except as indicated by the case escape, which is the "¢"; if this is used as a command it starts or ends block upper case output. There is also a mode where the first letter of each word is capitalized on output, as for a name, and a mode where Format attempts to recognize new sentences and capitalize the first letter of each one automatically. Special characters can be entered as two hexadecimal digits preceded by "!": e.g., "!B2" gives an EBCDIC "2".

This batch-oriented formatter has its own editor, of the preprocessing type. Invoked by the "EDI" control card, it takes commands such as "$INSERT", with an input card number and word number to indicate a position. The editor command "$LOCATE" lists where a given word is used; applied to words selected from the "DIC"'s output, this could be helpful in (manually) preparing an index.

5(d) Nroff (and Troff)

The names Nroff and Troff both refer to a general purpose formatter [Ossanna 1977; Kernighan 1978-A] written at Bell Telephone Laboratories (Murray Hill, New Jersey) and modeled somewhat after Roff but with an eye to greater flexibility. Troff's output is typeset (on a Graphic Systems phototypesetter), but monospaced proof output is available; Nroff's is monospaced (but variable-width blanks are supported when the output device permits). For input compatibility, Nroff generally ignores functions relating to typeset text only. Both versions are being discussed below, except where Troff is mentioned by name: "Nroff" is used for both. The implementation language is C.

<u>Commands</u> <u>and</u> <u>Names</u>

In Nroff, either "<nl>." or "<nl>'" starts a command (when "'" is used, the command never causes a break), and a newline ends it. After the control character, the next non-blank character and the character after that are the command name; the rest of the input line may contain arguments, generally delimited by blanks.

Some other commands, called "escape sequences", are initiated by the single character "\" so that they can occur in the middle of an input line; the following character names the escape sequence. Any arguments follow immediately. (They are mostly short and never need a terminating newline. For instance, "\s10" gives type size 10 points in Troff.) Some escape sequences produce single characters: "\ " gives an extraordinary blank and "\\" a literal "\". There are commands to reset each of the control characters ".", "'", and "\".

There are also certain implicit effects needing no command: a wholly blank (or null) input line causes a break and a blank output line, while a line of text with some leading blanks causes a break and a corresponding-sized temporary indent. Also, the punctuation marks ".", "?", and "!" are special in that if they occur at the end of an input line but are placed in the middle of an output line by filling, they are taken as end-of-sentence indicators and followed by extra space.

Relative values are often allowed for numeric arguments, and if these arguments are omitted the action is to restore the previous value of the parameter, where this makes sense. Numeric expressions may be used for arguments as well as constants, and relational operators such as ">" are permitted in expressions (as in APL and B, they give an integer truth value of 0 or 1). Arguments that refer to physical distances may (and, in some cases a naive user might not expect, practically must) include a scale designator. Some scale units have somewhat different meanings in Troff, including some relative units. Also, where it makes sense, arguments can be specified with a preceding "|" to indicate absolute physical position: "<nl>'sp |6i" causes the next output line to be spaced down to 6 inches below the top of the page.

Some commands operating on considerable amounts of text use a specified number of following input lines as their operands: "<nl>.ce 3" centers the next 3 lines, breaking on each. Other syntactic forms are described below.

Nroff has several commands to define macros and strings. The basic "de" (define macro) takes for the macro text each input line up to (and excluding) one starting with ".."; another command (or macro) name can be specified as a delimiter on the "de" command, and then the text taken is everything up to the first occurrence of that command, which is then executed. Also, the "di" command diverts output into a formatted string (with minor restrictions in Troff); the diversion runs until another "di" command occurs, and a "di" with no argument restores normality. The "ds" (define string) command takes for the contents of a string the rest of the input line following the string name and delimiting blanks (an initial """ will be dropped, so that the string may begin with blanks, but it can never contain a newline). For each of "de", "di", and "ds", there is a corresponding command that works the same way except that the text is appended to the macro or string.

Nroff also supports variables. Each may be assigned an individual output format (such as Roman numerals) and "auto-increment size" (see below; default is 1).

Macro, string, and variable names are restricted to a 2-character maximum. The invocation of a macro or formatted string is syntactically the same as a command: "<nl>." (or "<nl>'") followed by the name and perhaps arguments. The places to insert the arguments on invocation are indicated in the macro text by "\$" (which must be entered as "\\$" on definition) followed by a digit denoting the argument number: "<nl>.de pw <nl>\\$2th power of \\$1 <nl>.. <nl>.pw two 5" would result in "5th power of two".

A "ds"-defined string is invoked by "*", followed by the string name if it is 1 character, or by "(" and the string name (no ")") if it is 2 characters. A variable is invoked similarly, except that "\n", "\n+", or "\n-" is used instead of "*"; the two latter forms first add to, or subtract from, the variable its auto-increment size.

One use of macros is the provision of predefined packages which may simplify preparation of a particular kind (or several kinds) of document by providing relatively specific functions which the formatter does not support directly, though the user could produce them with appropriate macros or combinations of formatter commands. Such a package, of fairly general scope, is available on UNIX for use with Nroff, under the name -ms [Lesk 1978]. Some of its features will be described below.

Although an input newline is normally taken as a word space, and a break if in nofill mode, preceding it with "\c" cancels these effects, so that commands may be placed within a word of text. This is called *interrupted text processing*. On the other hand, the escape sequence "\<nl>" (a *concealed newline*) is totally ignored, thus permitting arbitrarily long logical input lines.

Nroff has many predefined variables, including the page number, date (by components, including day of the week), distance to next trap or end of page, and number of arguments with which the macro currently being inserted was invoked. Some are read-only. A predefined function "\wƒstringƒ" ("ƒ" can be almost any character) computes the width of a character string (and sets variables to indicate the presence of ascenders or descenders) and can be used in place of a number in expressions.

Ordinary Horizontal Effects

The Graphic Systems phototypesetter supports various special characters, each available in Troff by "\(" followed by the character's 2-character name: for instance, "\(dd" gives "‡". These special characters include ligatures for "ff", "ffi", "ffl", "fi", and "fl"; besides their escape sequences, the ligatures may optionally be generated automatically by the corresponding letter sequences in plain text. When not in Troff, the special characters are represented with combinations of normal characters as far as possible. With -ms, strings are available for various accents to be properly placed: for instance, the string "`" places that accent, so "\*`a" gives "à".

Continuous underlining is available as well as the usual underlining of alphanumeric characters only. In Troff, though, either kind is interpreted as a font change. (Conversely, the corresponding font's escape sequence in Troff gives underlining when not in Troff.) Also, in Troff, a font can be assigned a single type size, or can be emboldened by overstriking each character with itself slightly offset.

There is an escape sequence that causes a break with pad justification on the partial line it terminates (unlike a normal break).

It is possible to request that hyphenation never separate 2 letters from the rest of the word, and that words never be hyphenated onto the next page (or around a trap).

Special Horizontal Effects

With -ms, the user can automatically produce section headings with structured numbering (e.g., 3.1.4 for section 3, subsection 1, item 4).

Nroff allows three-part title lines anywhere. The command requesting them is followed on the same line by a sequence like "ʃxʃyʃzʃ", where "x", "y", and "z" are the three parts, and "ʃ" is any character not found in any of the parts. The user may also specify a short hyphenation exception list.

With -ms, a paragraph may be specified with a hanging indent of the left margin, or with indents of both margins. There is also a -ms macro which will take a group of lines, center the longest one, and align all the others with it by the left margin.

In addition to the tab character, the ASCII TAB, Nroff has a "leader character", the ASCII SOH. Either one is interpreted on input, before formatting, and causes the following text to be advanced to the next tab stop (tab stops are set by command); the intervening space is filled with repetitions of the _tab replacement character_ (normally the extraordinary blank), or the "leader replacement character" (normally "."), respectively. The two replacement characters are resettable, independently, but the tab and leader characters are not. (There are escape sequences giving literal tab and leader characters.)

A flexible justification mechanism usable with tables is controlled by resettable "field delimiter" and "padding indicator" characters. Occurrences of the former are used to partition an input line into fields corresponding to tab stops, much as tab characters do; but instead of left justifying the line fragment within each field, the places marked by padding indicator characters are blank-padded equally to fill the available space in the field.

As mentioned above, a number of other programs are used in conjunction with Nroff on UNIX [Kernighan 1978-D]. One is Tbl [Lesk 1979], a general program for table formatting. It is a preprocessor: its output is normally passed (via a UNIX "pipe") to Nroff. The pseudo-command "<nl>.TS" indicates a table; "<nl>.TE" ends it. Everything not in a table is passed though unaltered.

Since Tbl is not part of Nroff proper, it will not be described fully here, but a simple example will be given. The "TS" line might be followed by a table format such as "c s c, l n n."; this means that the first line of the table contains a field centered over two columns ("s", for "span", means this column is combined with the previous one, for this line), then another centered field, while each subsequent line contains a left justified field and two numeric fields to be aligned by their units digit. Following the format line would be the data, with a tab character delimiting each entry and a newline each new row.

In Tbl, table entries can be general text, not just line fragments; column widths and spacings can be either specified or deduced; various boxes and lines can be drawn; the completed table can be centered as a whole.

With -ms, multiple-column output is available.

For each nonblank output line, Nroff can print in the right margin a "margin character", and in the left margin a line number (which may be blanked out when not a multiple of some specified value).

Nroff supports _local motions_, i.e., explicit changes in the positioning of text (in any direction); abbreviated escape sequences provide common motions such as forward and reverse half linefeeds for superscripting. The user can also space up the page to a position explicitly marked by command. A character can be repeated to fill a specified dimension vertically or horizontally, or to match the size of some text (and -ms can use this to draw a box around it). "Tall characters" requiring extra blank space above or below them are supported, as well as various variations of over-struck and vertically piled characters.

Another preprocessor is called Eqn or Neqn (for Troff or Nroff respectively) [Kernighan 1976-A; Kernighan 1978-T]. It is intended for mathematical "formulae", but is sufficiently general to also be used for chemical equations [Edelson 1977]. The formulae can be delimited by pseudo-commands like Tbl's, or the user can define left and right math delimiter characters (perhaps the same character) and put the formula between them. A formula does not cause a break, so it can be run in line.

One example will give the flavor of Eqn: "x = {-b +- sqrt{b sup 2 -4ac}} over 2a" is the formula for the roots of "ax sup 2 +bx+c". Here "sqrt" gives a square root sign (with vinculum), "over" a displayed fraction, and "sup" a superscript. (If the letters "sup" were actually wanted, as in a supremum, enclosing quotes would be necessary: ""sup"".) The blanks in the example are necessary delimiters: as in programming languages, there are binding rules to resolve the operands of "over", "sup", and so on. The "{" and "}" characters produce no output but override the binding rules; characters such as parentheses are not special to Eqn and have no effect on binding (after all, "(0,x]" is a legitimate formula). Anything interpreted as an ordinary math variable is set in italics unless otherwise requested. "+-" becomes "±", and other special characters are similarly available by reserved character sequences. Accents and multi-line arrangements are also supported.

In Nroff there are several formatting _environments_ identified by numbers. Each has its own values for many formatting parameters such as margin settings, justification mode, fill mode, tab stops, hyphenation, body size, tab replacement character, and control characters. All commands affecting such parameters do so only within the current environment. The user can switch to another environment with the "ev" command, which pushes and pops a (limited) stack of current environment numbers (not of environments, so the same one can be on the stack more than once).

Vertical Effects

Any macro can be associated, as a trap, with the end of the input, or else with a position in either the output page, the current output diversion, or the input; when that position is reached, the trap is tripped and the macro invoked. Traps associated with the output page are reusable; the other kinds are inherently one-time. A trap containing plain text ought to switch environments to make the trap text independent of external matters. In -ms, traps are used to implement headings, footings, footnotes and multiple-column output; none of these facilities is provided by direct Nroff commands.

Nroff has a need command. There is also a command "sv" ("reserve space") that gives vertical space if there is room before the end of the page, or the next trap; but if there is not, the command is simply set aside. An "os" command ("output saved space") retries the last "sv" if it could not be executed before: "os" is intended to be used in traps, giving "sv" practically the effect of floating glued space.

<u>Input/Output Effects</u>

In addition to the usual file insertion, there is a command to skip the rest of the current input file and insert another one instead.

There is a command to accept general text from the standard input (which may be the terminal) until an empty line is read. Also, messages can be sent to the terminal. Another command (not in Troff) will pipe the formatter output to another program for postprocessing. There is also a command to terminate execution.

The "if" command takes a condition argument, then the rest of the input line is taken as general text (i.e., one command or plain text) to be processed or skipped; several lines can be attached to one "if" command by enclosing them in "\{" and "\}"). The condition may be an arithmetic expression (true if positive), a string equality comparison (syntax similar to three-part titles), or a predefined condition (the parity of the current page number, or whether the formatter is Troff). Any condition may be preceded by "!" to reverse the test. The command "ie" (if-else) is just the same as "if" except that the comparison result is also saved on a stack; later, an "el" command pops that stack and acts like an "if" with the opposite test to the corresponding "ie"'s. There is also a command (syntax like "de") to ignore input (except that "\n+" and "\n-" sequences still alter variables) up to a specified delimiter, and there is an escape sequence for in-line comments.

Nroff supports automatic character transliteration on output; if the formatted text is stored (with "di"), translation takes place then.

On input, macro definitions and any line beginning with "\!" are processed in "copy mode", where most commands (and tab and leader characters) are taken literally but string and variable invocations are executed.

5(e) Page-1

Page-1 [Pierson 1971; Pierson 1972] was developed at RCA, with book publishing in mind. It produces output for a Videocomp typesetter.

Commands and Names

Page-1 considers its input as a continuous stream, not broken into lines. A command list is enclosed between "[" and "]"; commands in the list are separated by ";"'s. Each command name is 2 characters; it may be followed by arguments, delimited by ","'s.

Certain commands that require a quantity of text, such as those defining strings and traps, have the text enclosed in an inner set of "[" and "]" characters. That text is general text: it may include commands, nesting another level of "[" and "]".

String names must also be 2 characters, specifically a letter followed by a digit; furthermore, the initial letters "a" through "s" are reserved for predefined ones, only "t" through "z" being free. A string is invoked by its name enclosed in "[" and "]", just like a command. There is also a mode wherein all text is both processed normally and saved in a special string (one of two, identified by numbers and invoked by a command).

Variables are numbered, not named. In addition to "general" integer variables, there are "indirect" variables containing integers that are the numbers of general variables. The "gv" and "iv" commands define the two types.

There are a few special variables that do have names, which also serve as the commands to assign them (if not read-only): "pn", for instance, is intended for the page number, but need not actually be used for that purpose.

The "dc" command inserts in the text the value of its argument, which is an expression: a variable, an integer, or one of a group of arithmetic commands, such as "ad" (add), whose arguments are in turn expressions: "[dc,[ad,1,iv95]]" would insert 1 plus the value of the general variable whose number is the value of indirect variable 95.

Ordinary Horizontal Effects

The "hs" command affects the output by dividing all requested type sizes by 2.

Since input has no lines, there can be no nofill mode.

Right justification is available.

Page-1 has an option where every output line can be padded, even if terminated by a break. There is also a command which causes a break (and pads the partial line thus formed) only if the current horizontal output position is close (the threshold specified in relative units) to the right margin; this horizontal analog of the need command is used before constructs which Page-1 might treat as two words but which must actually stay on the same line, such as words with superscript numerals (footnote references) attached.

The amount by which blanks can be altered in width for justification is limited and may be set by the user.

Page-1 has a hyphenation exception list built in; its use is optional.

Special Horizontal Effects

For tables, three commands, "xc", "xr", and "xt" (tab center, tab right, tab) cause the text following them to be respectively centered or right or left justified between two (possibly specified) tab stops as margins. There are also commands for horizontal local motions, the distances being given in relative units. The command "fl" is replaced by as many repetitions of its 1 or 2 argument characters as fit on the line, for leadering.

Vertical Effects

In Page-1, not only is output divided into pages, but pages are divided into "text blocks". Page-1 supports traps at the end of text blocks: the "ab" and "wb" (after block, when block) commands respectively set a one-time trap tripped when the current block ends, and a reusable trap for the end of every block. These can be used to implement headings, footings, and (with "ib", described below) footnotes; none of these facilities is provided directly, nor is automatic page numbering. Another form of trap, tripped simply by the completion of a filled output line, is set by "ar" and "wr" (after return, when return), again respectively one-time and reusable.

When there is nothing more to go in a text block, vertical justification can be applied in several ways, analogous to the horizontal justification modes. The text can be vertically centered, or justified against the top or bottom margins of the text block; or it can be padded to be justified against both margins, either by expanding the body size of each line or by padding at specified places. (There are two commands to leave vertical space: "dj" does, and "dn" does not, also specify a vertical padding place.)

Output need not go continuously to one text block until it is filled. Many text blocks, identified by numbers, can be active at a time; however, one is designated the "primary" and a second, perhaps, the "interrupting" text block. The "cb" (continue block) command directs output to the primary text block. The "ib" directs it to the interrupting block; if there is none, an "sb" (switch block) must follow immediately to create one. Finally, "sb" directs output to a specified text block, which becomes either the primary or the interrupting text block, according as "cb" or "ib" was executed last.

Input/Output Effects

Page-1 does not support file insertion.

The command "su" begins a "trial set". That is, the text read is processed, but the formatted output is merely saved; the trial set is ended by a "us" command, according to whose expression argument the saved output is either used at once or totally ignored.

Conditional processing of general text is available based on comparisons between two numeric expressions.

Page-1 has a series of ccmmands which serve as a preprocessing editor.

5(f) Proff

Proff [Beach 1976] was designed at the University of Waterloo with the aim of easily converting existing Roff-oriented input to be run on its Phcton 737 Econosetter. It is actually a preprocessor for the primitive formatting program contained on the small computer in that typesetter [Buccino 1980]. The implementation language is B.

Commands and Names

In Proff, "<nl>." initiates a command, and another newline terminates it. The command name is the 2 characters following the ".", and may be followed by arguments delimited by blanks. Arguments may be enclosed in the _quote character_ """, so that they may contain blanks themselves. The "." and """ can be reset.

Expressions and relative values are supported for numeric arguments; when representing physical distances, they may also have scale designators.

Macros in Proff can be invoked in two ways: by the same syntax as commands (except that macro names need not be exactly two characters long), and by the use of a resettable _insertion character_ followed immediately by a 1-character macro name or by "(", the macro name and arguments, and ")". The insertion character allows a macro to be inserted in the middle of a word or unfilled line. The places to insert the arguments are marked when the macro is defined by a resettable _parameter character_, a single character used exactly like Nroff's "\$". (By default there is no insertion or parameter character.)

Variables may be assigned individual output formats; they also are inserted using the insertion character syntax.

It is possible to undefine a macro or variable.

The "at" (assign text, meaning define macro) command takes as the macro text everything up to an "en" with the same macro name as argument; this syntax facilitates macros containing "at" commands. Other commands that operate on a considerable quantity of text take it from (a specified number of) following input lines: "<nl>.ce 3" means the same as in Nroff.

Ordinary Horizontal Effects

The Photon Econosetter has "ff", "fi", and "fl" ligature characters; Proff, optionally, interprets these sequences in plain text input as requesting the ligatures.

Special Horizontal Effects

Proff has a tab facility like Hypo's: each tab stop has an associated justification mode, left, center, or right. In fill mode, any input line containing a tab character causes a break, while lines not containing one are interpreted as continuations of the last field, which can be typeset as several lines of filled text bounded by the tab stop and right margin positions. The tab character is initially the ASCII TAB but can be reset.

Proff supports multiple-column output.

Proff's "program mode" simultaneously sets nofill mode and left justification, and turns off hyphenation and ligature recognition, as for computer program listings.

Input/Output Effects

The "ig" command causes all input to be ignored until an "en" whose argument matches the "ig"'s argument occurs; the "if" acts the same as "ig" if its second argument, an expression, evaluates to zero, but has no effect otherwise.

Proff has commands to pass characters through to the typesetter's formatter unchanged except for character set translation (transliteration). This is the only way to obtain some of the special characters on the device, as well as certain other effects.

Proff does not support page numbering, nor any effect that would require the formatter to know the position of output on the page.

5(g) Quids

Quids was developed at Queen Mary College of the University of London (London, England). It operates on monospaced text, not as a conventional formatter but as a combined editor-formatter. It is mostly written in "a high-level system implementation language".

The source [Coulouris 1976] for this section is not a complete manual; certain points are not explained. That paper describes itself as having been produced "with the assistance of Quids", but it is typeset.

Commands and Names

There is no general text with Quids; (almost) all commands are entered interactively, and some cause plain text to be read from the terminal. That text is, as in Page-1, taken as a continuous stream of input, not broken into lines; from time to time a typed blank produces a displayed newline, to prevent the terminal's lines from overflowing, but internally it is still a blank. An ASCII ESC terminates the input, to allow more commands, as explained below; the text is then stored in a galley-file-type structure in memory. (When it is written to a file, apparently a sequential-access form of the structure is used.)

Commands are named by single letters, each the initial of a mnemonic word or phrase; when the letter is typed the full mnemonic is displayed on the terminal, along with, in some cases, a generated serial number (as for a new paragraph), or an interactive request for arguments or a confirmation. Some commands may also be preceded by arguments as mentioned below.

Two special sequences are recognized in plain text, thus being in effect commands: "[ref." and "[fig.", followed by a number and "]". Also taken as special in plain text is the character "_"; text bounded by two of them is underlined on output.

Quids supports strings identified by numbers, but only for bibliographic references (like "[Coulouris 1976]"). Apparently the string contents must actually begin and end with "[" and "]". The string is inserted by "[ref.".

Ordinary Horizontal Effects

No automatic hyphenation is mentioned.

There is a command to specify in advance, along with an identifying number, a temporary indent of either margin for several lines; it takes effect following a "[fig." with that number.

Special Horizontal Effects

Tab stops may be set for the nofill, left justified mode; a tab character (apparently the ASCII TAB) causes the text to be advanced to the next tab stop.

Quids allows text to be inserted as a marginal note to the left of the output.

Vertical Effects

Headings are supported, at least in the form of one centered line, and page numbers may be printed automatically. Page numbers may not be reset, nor may the top and bottom margin sizes.

The user may divide the document, by commands, into sections, and those into subsections. Each has a title, which will (or may) be printed centered, optionally preceded by the section or subsection number.

There is a command to produce a table of the section and subsection headings, or of the bibliographic references, or of the "index entries". The format of the table is not explained, nor are index entries.

Input/Output Effects

There are several "modes" of interaction, each with its own set of commands (whose names, but not mnemonics, may conflict with other modes). The initial "context edit" mode is for operations on paragraphs: they can be inserted, deleted, edited, displayed on the terminal, written to a file. Its commands are modeled somewhat after the text editor QED: optional preceding arguments may specify a range of operand paragraphs, the default generally being the one last referenced. There are also commands to search for a paragraph containing a particular character string and to pass to the next or previous paragraph. Paragraphs are addressed by three numbers, section, subsection, and paragraph, separated by "."'s. Truncating this, when it makes sense, denotes an entire section or subsection. As in QED, any insertion or deletion immediately changes all following addresses; the numbers are always consecutive. The first paragraph is number 0.0.1 if no section or subsection heading precedes it.

As well as paragraphs, headings, etc., the text on the galley file may include certain opcodes; all types of item are numbered in the same address space, so that "paragraph" number 1.0.1 would actually be the first section heading. There are commands to edit opcodes, but no details are explained.

"Context edit" mode has a command to replace all occurrences of a particular character string, in a range of paragraphs, by another. All other editing is done by switching to "local edit" mode for a particular paragraph. The commands for this mode are named by various ASCII control characters, and have no mnemonics; as soon as a command is given, the displayed text changes to show the result. The operations are character- and word-oriented; there is no way in any mode to make one paragraph into several, or several into one, and there are no operations to relocate text.

The "context edit" mode commands that cause insertions of paragraphs switch the interaction to "input text" mode. In this mode, the commands most resemble in function those of a conventional formatter; some initiate the entry of plain text, which is actually done in "local edit" mode so that errors may be corrected at once. An ASCII ESC terminates the paragraph and switches back to "input text" mode for more commands.

There is a command in "context edit" mode to insert text from a file, but what is allowed in that input is not explained. (Some changes of mode, or equivalent, must be permitted, if more than one paragraph can be read at a time.)

5(h) Roff (and Vroff)

Roff, a derivative of Runoff intended for general publication formatting using monospaced text, was written at Bell Telephone Laboratories (Murray Hill, New Jersey), where it has been replaced by Nroff. The University of Waterloo version [Roff 1978], described here, has several enhancements; its implementation language is B. It was used (with very minor alterations) for this thesis.

Vroff is a variant of Roff written at the University of Waterloo and designed to be portable between various PDP-11 operating systems; its implementation language is assembler. The version [Vroff 1976] described here runs on the University of Waterloo Mathematics Faculty UNIX system. Except where Vroff is explicitly mentioned below, the discussion applies to both Roff and Vroff.

Commands and Names

The command syntax is the same as Proff's (but there is no default quote character); the forms of commands operating on considerable quantities of text are also the same. Interrupted text processing can be obtained by preceding a newline with the insertion character.

Numeric arguments may be specified as relative values, with multiplication and division of the old value by the argument allowed as well as addition and subtraction. Arguments may be specified as expressions, which may (not in Vroff) include comparison (giving a truth value, 0 or 1), maximum, and minumum operators; also (not in Vroff), a character string bounded by the quote character may stand in place of a number, and its length in characters is used.

Various implicit effects are supported exactly as in Nroff: input lines beginning with blanks (or null lines) cause a break and vertical space or a temporary indent, while certain punctuation marks ending an input line can get extra space on output. This effect in Roff applies to the ":" as well as ".", "?", and "!".

Roff supports variables and macros. The definition and invocation syntaxes are identical with Proff's, including the use of insertion, parameter, and quote characters, except that a macro must have a name exactly 2 characters long to be invoked by the same syntax as a command (names may have up to 4 significant characters, 10 in Vroff), and in Vroff macros invoked by the insertion character may not take arguments. Variables may never have negative values. Each variable may be associated with an output format.

Predefined variables, besides the page number, include (not in Vroff) the various components of the time and date (and a macro is available to calculate the month name).

Ordinary Horizontal Effects

Roff supports a simulated boldface, obtained by printing each character 3 times overstruck.

Roff's automatic hyphenation may be turned partially on, detecting only discretionary hyphenation points and existing hyphens, or these things and certain suffixes, as well as off or fully on. (Automatic hyphenation was not implemented in Vroff.)

Special Horizontal Effects

Roff's tab facility is syntactically the same as Proff's, but takes effect on input, before formatting, as in Nroff. Left, center, and right justified fields are available, but there is no special treatment of the last field as in Proff. Vroff supports a resettable tab replacement character.

Roff (not Vroff) allows text to be inserted as a marginal note to the right of the output.

Roff supports merge patterns.

Output lines can be automatically numbered in the left margin, either continuously or starting anew with each page.

Vertical Effects

In Roff the top and bottom page margins are each divided into two parts, each independently resettable; one is always clear space and the other may contain headings or footings. Headings and footings are three-part title lines; several such lines, identified by digits, are permitted in either margin (if it has been set large enough), each line being separately resettable, independently on even and odd pages. The character "%" in a heading or footing is replaced by the page number at output time; "%" is the name of the page number variable, but in this case it is used without an insertion character preceding, and a literal "%" cannot be obtained there. (Despite all this flexibility, Roff cannot automatically provide the page numbering style required in a University of Waterloo thesis where a major section begins; for this one, it was fudged, but Nroff or Tex could do it with general text in a trap.)

Roff supports footnotes, and will automatically divide them between pages if necessary; also, a three-part title line can be specified to be the _footnote separator_, which is printed just above the first footnote on each page having any.

Roff supports a need command, and also (not in Vroff) floating glued text; the latter carries its own formatting environment, as do footnotes in Roff, since it is not expected to be part of the main text of the document.

Input/Output Effects

Roff (not Vroff) has a command to delay processing while one line (which is ignored) is typed on the terminal. In Vroff messages can be sent to the terminal, while in Roff (not Vroff) commands can be sent to the system. General text (with macro invocations first performed) can be written to one or another of a set (in Vroff, one) of work files.

For conditional input or comments, the "if" and "ig" commands are like Proff's, except that "if" is only in Vroff, and allows a "!" before the expression argument, to reverse the test.

Roff can reverse the case of input (Vroff can map it into all upper or all lower case instead), and supports a case escape character. Output transliteration is supported and is the only way to get an extraordinary blank. There is a command to take the following input lines as plain text, even if they begin with "."; Roff actually does so anyway, unless the characters following the "." form an actual macro or command name.

Roff has a command to suppress output for a specified number of pages.

5(i) Runoff

Runoff [Saltzer 1965] was written at the Massachusetts Institute of Technology (Cambridge, Massachusetts). A simple formatter producing monospaced output directly on the terminal, it has given rise to numerous descendants and imitators, including Nroff, Proff, Roff, and Script; being a rather simple formatter, it is included here for this historical reason only.

Commands and Names

In Runoff as in Proff, "<nl>." initiates a command; the command name may be given as a long or short form, as in Dip. Some commands take an argument, separated by a blank; the rest of the line is ignored. The "center" command takes the following input line as its operand, while the "header" command irregularly uses the rest of its own input line.

Ordinary Horizontal Effects

Runoff has no automatic hyphenation.

Only negative (relative) temporary indents are supported.

Special Horizontal Effects

There are no commands for underlining or tables, but Runoff passes input tab and backspace characters through untouched; the terminal may handle them.

Vertical Effects

Only single and double output line spacing are available.

The top and bottom margin sizes cannot be reset. Page numbers can optionally be printed on each page, but there is no choice of where. The "header" command prints a heading line on each page.

Input/Output Effects

The manual is ambiguous as to whether file insertion is supported or there is a command to skip the rest of the current input file and insert another one.

5(j) Script

Script, a derivative of Runoff intended for general document formatting, was developed at IBM. The version [Waterloo 1978] described here runs at the University of Waterloo Department of Computing Services. Its monospaced output can be adapted for terminals with special facilities such as printing backwards.

Commands and Names

The simplest form of a Script command or command list has "<nl>." followed immediately by the 2-character command name; the rest of the line is taken up by the command's arguments, usually delimited by blanks. However, whenever a line begins with ".", any occurrence of ";" in it is taken as a logical end-of-line, so that what follows can be another command if the next character is ".", or plain text otherwise. If a line begins with ". ", everything up to the next nonblank is ignored, but ";" is still a logical end-of-line. The "." and ";" characters may be reset by command, even within an input line; another character (there is none by default) may be defined to have the same function as Nroff's "'".

For numeric arguments, expressions can be used as well as constants. Relative values are supported.

An input line starting with a blank or EBCDIC TAB causes a break.

Script has the philosophy that one command should be used for various related functions, distinguished by keyword arguments (which often can be abbreviated). Thus, while "<nl>.ce 3" means the same as in Nroff, alternatively "<nl>.ce on" could precede the text to be centered and "<nl>.ce off" follow it, while ".ce" at the beginning of each line of text would be another possibility. ("Begin", "End", "Nosave", "QUit", "SAve", "STop", and "Yes" are each synonymous with either "ON" or "OFf", and for each of these keywords the characters here shown in lower case are optional.)

Likewise, the "dm" (define macro) command may be followed on the same line by a character not used in the macro, then by the complete text of the macro using repetitions of that character to stand for newlines, or else the macro text may be placed after the command line, terminated by another "dm" (which is supposed to have arguments of the macro name and "end", analogous to "en" in Proff, but the verification is not implemented). A macro having somewhat different properties (see also below under environments and traps) may be defined with the "rm" (remote) command; the text is again placed after the command line, and terminated by "<nl>.rm". Script macros may have names of up to 8 characters.

A macro may be invoked explicitly by the "si" (signal) command, or by using the macro name itself as a command; the latter syntax may be switched off, for macros defined by "dm" with the "ms" (macro substitution) command, and for those defined by "rm" by using ".." instead of the "." before the command. With either invocation syntax, arguments are delimited by blanks and referred to in the macro text as strings named "1", "2", etc.; the number of arguments is string "0" and all the arguments as one string is "*".

Strings in Script serve also as variables, with the natural representation for integer values. The "sr" (set reference) command defines them. Following this on the same line are the string name (up to 10 characters plus an optional subscript in parentheses), an optional "=", and either a numeric expression (whose constants may be in binary, decimal, or hexadecimal, or as EBCDIC character equivalents) or a character string (enclosed in """, "'", or any of 5 other characters, optionally if no ambiguity would result; the delimiting character may even occur in the string unless a blank follows it.) The name's subscript, if any, may range from -32767 to 32767; if none, 0 is assumed. A null subscript ("( )"), however, means that the same name with a subscript of 0 contains a number and is to be used as the subscript after 1 is added to it. (Thus if string "latin(0)" contains "5", "<nl>.sr latin( )=sex" sets "latin(0)" to "6" and "latin(6)" to "sex".)

Strings cannot be referenced except by command. The "ur" (use reference) command affects the rest of its input line, which is then reprocessed as if a complete input line. A string is invoked in that line by an "&" followed by its name and (optionally if no ambiguity would result) a ".". (So, after the previous example, "<nl>.ur .ce vi=&latin(6)" finally gives "vi=sex" centered.) If "L'" or "T'" precedes the "&", instead of the string its length (in characters) or type ("C" for character, "N" for numeric) is inserted. "&&" gives a literal "&". There are special subscripts to list all the elements defined under one name with negative and/or with positive subscripts, all separated by ", ".

The "su" (substitute) command works like "ur" except that instead of only invoking the strings directly mentioned in the line or lines, the process is iterated until no string invocations remain. Thus if string "indir" had value "&&list(&sub)", "sub" had value "10", and "list(10)" had value "macname", then "<nl>.su .si &indir" would finally be interpreted as "<nl>.si macname", with ".su" acting the same here as ".ur .ur .ur". There is also a command "se" (also for set reference), which essentially means "su .sr".

The user can change the names of commands.

Ordinary Horizontal Effects

There is a command that alters the list of characters affected by underlining, and can also specify a character that switches the underlining off and on when placed within a line to be underlined. Overstriking of each character with a character other than the "_" is also supported. Simulated boldface by overstriking of each character with itself is available, both for small portions of text as with underlining and for the whole output. Also, the usual vertical spacing commands, given with a 0 argument, will cause two successive complete output lines to be overstruck, if possible without horizontal backspacing on the output device.

In addition to left, center, and pad justification modes, there are in, out, right, and "half". The last leaves the left margin straight but pads spaces half as much as pad mode would, so that the right margin is not as ragged as in left mode.

Script supports a hyphenation exception list, and a large predefined one is available by file insertion. The user can specify the maximum number of consecutive output lines to hyphenate, the minimum number of letters to split off either end of a word being hyphenated, and how much an output line can be harmlessly padded before hyphenation need even be attempted. Hyphenation can be turned off temporarily, to be reactivated automatically at the next break.

In addition to the usual margin changes, Script supports an automatic hanging indent, whereby every following output line until otherwise specified is indented by the specified amount except the line after each break.

Special Horizontal Effects

Script's tab facility is similar to Roff's, but with two additional features. A tab replacement character or character string can be associated with each tab stop, for applications such as leadering; and not only can left, center, or right justification be associated with each tab stop, but there is also "character alignment", where the entries in the column are aligned by occurrences of a specified character, such as a decimal point.

Script has a command for drawing boxes around text, using either the "corner" characters "┌", "┐", "└", "┘", "┼", and or the more widely available "+", together with "-" and "|". The user indicates what positions to draw the vertical sides at, and each "bx" command causes a break and draws a horizontal side.

Script supports multiple-column output. The optional column balancing facility causes the columns of a page not full of text to be made as equal in height as possible.

Script supports merge patterns operating on input as well as on output text. Either kind can be set to automatically cancel itself after a specified number of lines, and further patterns can be queued to replace it. Also, any part of a document may be labelled by command with a numeric "revision code", and each revision code can be associated with a string (up to 8 characters) to be printed in the left margin of each corresponding output line. A section with one revision code may be nested within one with a different code, as for successive versions of a document.

Output lines can be numbered automatically, in any column position, but only for the entire document with numbering starting anew on each page. Blank lines may be counted optionally.

Script allows formatting environments to be saved, but only on a stack. The environment may optionally be stacked automatically while a macro defined by "rm" is being inserted.

Vertical Effects

There is a command to space vertically downward to a particular line number, on the same page if possible, or else on the next one.

Page numbers with two parts, separated by a "." on output, are supported. (Where the parity of the page number matters, the parts are summed.) This mode is entered by explicitly specifying such a number, or by a command option that automatically causes this effect beginning with the next odd-numbered page.

In Script the top and bottom page margins are each divided into three parts, two of which, and the total of all three, are independently resettable; the middle part may contain headings or footings and the others are always clear space. Headings and footings work the same way as in Roff, except that there is a default right-justified heading of "PAGE" and the page number on each page after the first, and the insertion of page numbers in headings is slightly different: the character "%" indicating a page number is resettable in Script, and strings of up to 8 characters, to be inserted before and after the page number when it replaces that character, can be assigned. (In "sr" commands, the current page number can be obtained as either just "&", just "%", or just the character replacing "%" in headings if any.) Script's footnotes and footnote separators work just the same as Roff's, except that Script allows multiple footnote separator lines and by default has three, the second line with several "-"'s centered and the others blank.

Script also supports a "headnote". This is a block of formatted text placed at the top of each output page until cancelled by command, and anywhere else explicitly requested by command. Odd and even pages may have separate headnotes. Headnotes, like footnotes, go in the text area of the page, not the margins; the environment is stacked while they are formatted.

Macros defined by the "rm" command may be identified with numbers instead of names. In that case, they may be used as traps. The number indicates the output page position to trip the trap. These macros may still be invoked directly with "si"; if the number exceeds the page height, this is the only way. Multiple traps set at the same location are queued. Traps may be one-time or reusable, and an intermediate type (that works for a specified number of times, then vanishes) is also available.

Script supports indices, as in books. The user may specify entries at up to 3 levels ("Newton, Sir Isaac, calculus") in each of up to 9 separate indices. The page number is filled in automatically unless another string is specified to replace it. The indices are automatically kept alphabetized. Any index can be printed or cleared by a single command, but "su" or "ur" must be in effect when it is printed, because the system inserts "&SYSIXREF." just before the page number or replacement string (the string "sysixref" is initialized to ", "). Similarly, the user is expected to have defined certain macros (with names of like flavor), which the system invokes before each entry in the index and when the first letter changes.

There is, likewise, a table-of-contents facility.

Script supports 10 different "head levels", each independently having various formatting and other options (the defaults varying from one level to another). Entries under each head level may be directed to the main output and/or to any one of up to 10 different tables of contents. Other lines, including commands, may also be inserted in a table of contents. The page numbers are filled in automatically and any table of contents can be printed or cleared by command. Indices, head levels, and tables of contents are all identified by digits.

The user can divert sections of formatted output, as for bibliographic references, to a queue for later printing, as after a chapter. Each such item is bounded by "fb" (floating block) commands with "begin" and "end" arguments; the same command with a "dump" argument prints the contents of part or all of the queue.

Script has two need commands, one which may force a new page and the other only a new column (if in multiple-column mode). However, column balancing takes priority over needs. Script also has glued text and optional automatic widow elimination.

Input/Output Effects

In addition to the usual file insertion, there is a command to skip the rest of the current input file and insert another one instead. Another command simulates the end of the current input file.

Script has a command to delay processing while a specified number of lines (which are ignored) are typed on the terminal, as well as commands to read data from the terminal, either a specified number of lines to be taken as normal general text input or one line to be assigned to a specified string. A line of text can also be sent to the terminal, or to the system as a command.

Script supports work files.

The "if" command takes as arguments numeric expressions or (recognized the same way as with "sr") two character strings, separated by a relational operator. The rest of the input line (or if none, the next line) is taken as general text (i.e., one command or plain text; several such lines can be attached to one command by surrounding them with "do" commands with "begin" and "end" arguments). The text is processed if the relation is true, skipped if false. The following input line may be an "el" command, which acts like an "if" with the opposite test. (It works properly even if a file or macro is inserted because of the "if".)

Portions of a document may be designated as "conditional sections", each of which may bear a number, much as they may be assigned revision codes (but without nesting). The user indicates independently which numbers indicate that the section is to processed, and which that it be skipped.

Unnumbered conditional sections are taken as comments, as is any text following a "cm" command on the same logical input line, or following a "*" command on the same physical line.

Iterated input is supported: the "pe" (perform) command is followed by the number of times the rest of its physical input line is to be repeatedly processed. A "pe" with an argument of "delete" aborts the iteration. The "pe" and "if" family of commands support some nested command structures.

Output lines may be translated to upper case, as for headings; one command gives upper case and underlining.

The "li" command can cause the following (specified number of) input lines to be taken literally as plain text. It can also cause all input to be taken as plain text until the sequence "<nl>.li off" occurs. (It is another option of the same command that changes the "." character.)

Special characters can be entered as two hexadecimal digits separated by a user-specified "hex join character": thus if it was "!", "B!2" would give an EBCDIC "²". If either character is not a valid hexadecimal digit, the hex join character is interpreted as a backspace.

Script supports output transliteration, with the characters optionally being specified in hexadecimal; input translation is also available, but the user must specify an escape character, which is deleted whenever it occurs, only the character following it then being transformed.

A line of input may be labeled with the "lb" or ".." command; the "go" command acts like a transfer of control causing the input to be advanced or backed up by a specified number of lines, or to a specified label or absolute line number. If the "lb" command is given with a number instead of a label, it causes an error if it does not occur on the line with that absolute number.

Script can automatically process the entire input two or more times in succession, producing output only on the last pass, so that forward references can be automated: for example, tables of contents can be produced at the beginning.

There are two commands that cause Script to terminate processing (one without flushing footnotes and so forth).

5(k) Tau Epsilon Chi

Tex [Knuth 1979] was written at Stanford University (Palo Alto, California) with the aim of making mathematical publications look more beautiful, as they did before photocomposition. Its output is therefore typeset.

In fact, Tex is specifically intended for use for the modern typesetting equipment that produces each character with a fine raster scan [Barry 1977; Walter 1969]. An associated program, Metafont, is used to design new fonts for such a device; each character is described in terms of a program of actions to be followed by pens (and erasers) of specified shapes, travelling in straight lines and cubic splines through specified points. "R. W. Gosper has observed that this is the opposite of Sesame Street: instead of 'This program was brought to you by the letter S,' we have 'This letter S was brought to you by a program.'" [Knuth 1978]

Commands and Names

A command is indicated by "\" and named by the following characters, any number of letters (delimited by the next nonletter) or any one nonletter. The syntax for arguments or operands varies somewhat from command to command.

Tex requires scale designators for physical dimensions. The user may specify the size of one new unit.

The "group" delimiter characters "{" and "}" are used in the input somewhat like mathematical parentheses: a group can act as an operand instead of a character, and some commands require groups. Groups also, simultaneously, define environments: the font, justification parameters, control characters, and even some macros, specified within a group have no effect outside it. Groups are thus similar to the "scopes" in Dip, but more flexible as they can occur anywhere.

Tex supports macros, variables, and formatted strings. Macro names, and macro invocation, have the same syntax as commands; variables and formatted strings, however, are identified only by digits, and there are explicit commands to insert them. Formatted strings are boxes (see below). Arithmetic on variables is limited to addition or subtraction of a constant or another variable, and addition of 1 to the magnitude. The last is provided because, while a positive variable is inserted in the usual Arabic format, a negative one has its magnitude inserted as lower case Roman numerals. (No other formats are available.)

The syntax for each macro's arguments must be specified with its definition. For instance, "\def \msb ( )#1#2[ ]#3:{Most significant #3 of the #2 bits of #1}" would define a macro "msb" that would have to be invoked followed by "( )" (or it would be a syntax error); the next argument would be one character or group, the next would be everything up to (and excluding) the sequence "[ ]", and the third would be everything up to a ":". The arguments must be labeled by digits in sequence, as in the example. "##" gives a literal "#".

With the "def" command, the macro is local to any group within which it occurs, and the macro text is taken literally when it is defined. The similar commands "gdef" and "xdef" give global definitions; with "xdef", macro references in the macro text are expanded at once, so that a macro can be redefined in terms of itself.

Besides "\", "{", "}", and "#", other control characters, as used in this description, are "$", "ˇ", "^",

and "⊗" (circled "x"), respectively for math, subscripts, superscripts, and tabbing. Tex has no default control characters, but the very first character of input is taken as "\"; the user is expected to begin by inserting a file defining the other control characters to be used, as well as the fonts and perhaps common macros. One predefined file for this purpose is called the "Basic format"; it is written using "\", and sets the other control characters as indicated above, except for the last two which are respectively defined as the up and down arrows (these and "⊗", it seems, are commonly available on terminals at Stanford).

Actually, any character can be defined to have any (one) type. Therefore there can be synonymous control characters and additional logical end-of-line or space characters; even the contents of the alphabet can be revised (so that a macro could be named, say, "p.d.q."). The "chcode" command, which does all this, requires the character and its new type to be specified numerically.

Ordinary Horizontal Effects

In some ways Tex tries to mimic hot-metal typesetting. Where other formatters generally expect the typesetter to produce various type sizes by geometric expansion or reduction, Tex assumes each font has only one size (except for some special math characters), which encourages the design of fonts especially for large or small type sizes. With the same philosophy, Tex supports automatic kerning: combinations such as "AV" get closed up to reduce the space between the diagonal strokes. Likewise, the sequences "ff", "ffi", "ffl", "fi", and "fl" become ligatures. Such specially treated sequences are specified in each font's description passed to Tex; the ligatures listed above are those applicable in the fonts selected by the Basic format. The sequences "--" and "---" are handled like ligatures in these fonts, respectively giving the en and em dash characters. (Ligature recognition can be suppressed only by grouping, e.g. "{f}f" or "f{}f".)

Tex supports underlining, but only in math mode. However, ordinary text can be inserted into math (see below).

Tex does recognize input lines, but end-of-line is treated almost the same as a blank. There are only two exceptions: any end-of-line character except a newline (set by "chcode") causes the rest of the line to be taken as a comment, and a line containing no text causes a break. (Double-spacing the input, therefore, produces an effect similar to nofill mode, which Tex does not support directly.)

Tex considers that formatted text is made up of two types of entity: "boxes" of fixed size possibly containing text, and "glue" of adjustable size. A box can contain just one character, or black ("rule") or white space; or it can be assembled from a list, running horizontally or vertically, of boxes and globs of glue. The exact physical length (in the direction of assembly) that a box being constructed should occupy is usually specified or known in advance, and this requirement is satisfied by adjusting the glue sizes; the glue is then said to have been "set", and the constructed box is treated as a unit thereafter. Its other dimension (vertical if from a horizontal list, or vice versa) is usually fixed by the largest dimension of a component box.

Boxes actually have four dimensions: width, height above baseline, depth below baseline, and "italic correction" which is the amount by which the contents protrude beyond the box at the right. Glue has three dimensions, all applying in the direction of the list (being assembled into a box) containing it: normal size, shrinkability, and stretchability. (For instance, the glue after the end of a sentence is somewhat more stretchable and less shrinkable than between most words.) Occasionally a box may happen, or be commanded, to be set at its "natural size" (sum of box sizes and normal glue sizes), but usually the glue must be adjusted, and then each glob is altered in proportion to its own shrinkability or stretchability, as appropriate. Box and glue dimensions can always be specified explicitly, and there is considerable scope for variation in the defaults governing the glue inserted (between words, lines, and paragraphs) in ordinary text.

In particular, glue cf very large stretchability can be used for a broad range of effects such as right justification and three-part titles; multiple globs of glue of different, but all very large, stretchability can give still more general effects. Also, the "ragged" command resets a parameter, r, where the amount by which glue in paragraphs is stretched or shrunk gets multiplied by 100/(100+r): the default r=0 gives pad justification, very large r left justification, intermediate values intermediate results (r=100 gives Script's "half" mode). For special effects (like local motions), boxes can be offset from their usual alignment, and can have dimensions not matching their contents, even negative dimensions.

Paragraph formatting is another field where Tex emulates less automatic methods: it waits for a complete paragraph before considering where to start each output line. (Meanwhile, the text has the form of a long horizontal list of boxes, usually single-character boxes, and glue, usually between words. Each output line finally becomes a box.) A nontrivial algorithm is used to minimize the "badness" of the paragraph, meaning hyphenations, glue greatly altered from its natural size, line breaks at points the user suggested (by command) would be poor, and other such things. This approach reduces the number of hyphenations considerably, which in turn enables Tex to use a hyphenation algorithm that only breaks words at places where it is almost surely correct, rather than trying to find almost every syllable. Therefore Tex's hyphenation errors are mostly of omission; if one is bothersome, the user is expected to insert a discretionary hyphen. (If the badness is too large, Tex halts with an error message; the user then inserts discretionary hyphens or raises the badness threshold.)

Tex indents the first line of each paragraph by default; unless the indentation is set to zero, this effect must be explicitly cancelled for each paragraph where it is not wanted. Tex can also produce a temporary indent of the beginning (specified number of output lines), or all but the beginning, of a paragraph. These temporary effects apply only to the left margin. On the other hand (literally!), permanent indents apply only to the right margin; on the left, boxes of white space would be used.

Special Horizontal Effects

The box concept provides a flexible table facility. The "halign" command constructs a list of boxes each containing one row of the table (afterwards handled like lines of a paragraph). The table is actually made from a matrix of constructed boxes; the glue within each row is set so as to align each column properly, producing an effect much like Tbl with Nroff. (A very similar command, "valign", exchanges the horizontal and vertical directions.)

Syntactically, the main argument of the "halign" command is a group, containing one or more portions each terminated by a "cr" command (and each balanced with respect to "{" and "}"). The first portion is the table format, and each subsequent portion produces one row of the table. Within each portion, "@"'s separate elements corresponding to the table columns. Each element of the format must contain exactly one "#"; the boxes which go into the table are produced by inserting each data element in place of the format element's "#" and formatting the result.

Tex treats mathematical "formulae" quite specially. There are 4 "styles": "display", "text", "script", and "scriptscript". Display style is used for a math formula that goes on a line by itself, text style for one run in with the rest of the paragraph. The other two styles are for subscripts, and their subscripts, and other such small things. Display and text styles use the same size for simple expressions, but text style requests smaller styles in a greater number of constructions (such as fractions and summations with limits). A formula to be run in with the text is indicated by "$" before and after, while a displayed formula uses "$$" instead. The style automatically chosen for each component of a formula can be overridden by command. The user must specify 10 fonts to use for math mode: roman, italic, and symbol fonts for text (used also for display) and for each of the other styles, plus one "ex font" containing oversized or variable-sized characters. Not all need be different, though in Basic format they are. As in Eqn with Troff, ordinary variables automatically go in italics, subscripts are reduced in size, etc. Tex's usual fonts have many math characters, each available by its own command such as "Rscr" (script R; case is significant in the first letter of a command name), "zeta", "union", and so on.

A simple math formula that goes on one line is parsed into 7 types of element, each a box, and Tex inserts glue between them according to their types (but never because of input blanks, though; in math mode they are totally ignored, except as delimiters). However, the type of a box can always be explicitly specified: "\mathrel{+}" gives a "+" parsed like an "=". The Basic format defines a number of macros such as "sin" and "log" that get parsed as single elements of the same type as, say, the "∫". Any constructed box (thus, absolutely any formatted text) can be inserted into a formula, and parsed as a single element of any type.

Several commands put one part of a formula over another. Their syntax is unusual: only one of them may be used at a particular level of grouping, and its two operands are everything before and after it in the group, mimicking the way one thinks of the mathematical expression. For instance, x+1 over yz would be "{x + 1 \over y z}", and n choose k is "{n \comb( ) k}". The two characters after a "comb" command surround the pair of lines. (They must be chosen from a limited set such as parentheses, brackets, the vertical line, and "." which gives a blank here, but as with Eqn they need not be conventionally paired.)

Superscripts and subscripts are obtained in Tex by control characters, not commands. They operate on the single character or group following: "a ˇ b c" (or "aˇbc") is a-sub-b, times c, while "a ˇ {bc}" is a-sub-bc. Similarly "\int ˇ0 ^n" gives a definite integral from 0 to n. This use of these operations for limits suggests that "\sum ˇ

{k=0} ^n" similarly be used as the sum from k=0 to n; but in display math at least, normal practice is to typeset the limits above and below the summation symbol, not in the sub-script/superscript positions. Tex evades this problem by defining an alternative position for subscripts and super-scripts, namely above and below what they modify. The alternative position is used specifically in display math style, when the thing modified is either a single ex font character (see above) or a constructed box, and either is followed by a "limitswitch" command or has a zero italic correction but not both. (Thus "sum" and "int" are treated differently by default; the "log" macro in Basic format is a constructed box with zero italic correction but includes "\limitswitch".)

Other math commands include a series to put various ac-cents, underlines, and so on, on characters (or on groups, or thus on boxes), and a pair "left" and "right" that produce large delimiters around what they enclose, much like "comb". For instance, the absolute value of x-hat over x-bar, cubed, would be "{\left| {\A x \over \overline x} \right|} ^ 3". The "left" and "right" commands must be paired, but the actual characters used have only the same restrictions as with "comb". In fact, since "atop" is like "over" but produces no fraction bar, "p \comb\{. q" could be written "\left\{ {p \atop q} \right.": either gives p above q bracketed by a literal left brace ("\}").

Tex has commands for both horizontal and vertical leadering, which each produce a glob of glue that does not print as white space but as a rule or as repetitions of a specified character string (repeated uses of the same one are automatically aligned).

Vertical Effects

Tex supports, in effect, a trap tripped each time a page is completed. The argument of the "output" command is a group formatted at that time, becoming the box that is actually typeset. The actual contents of the page must be inserted within that box using the "page" command, or they will not be printed. Headings and footings may be obtained only with "output". It can also be used with formatted strings to produce multiple-column output: one saves "\page"'s on consecutive executions of the trap, then finally retrieves and prints them side by side.

The "botinsert" command creates a box that is placed as a footnote; there is also an analogous "topinsert".

Tex handles completed lines vertically much as it handles characters horizontally. Each line is taken as a box, and glue is inserted automatically between paragraphs (also around displayed equations, above footnotes, etc.); then, after there is too much text to fit on the page, Tex chooses where to actually end it, much the way it chooses where to start new lines; this allows some flexibility in avoiding widows (as in avoiding hyphenations). Since the user can specify glue that stretches vertically (though never within filled text), vertical justification has much the same flexibility as horizontal.

When one wants to have a heading that changes (as at chapter boundaries, or for dictionary guide words), simply modifying "output" or having it reference a macro will not work, for the trap invocation is not synchronized with the input. Tex therefore provides a "mark" command, which associates a formatted string with a position in the text. The most recent "mark" strings before the page's top and bottom can be retrieved for "output" by commands.

Input/Output Effects

There is no command requesting terminal i/o when a file is being processed, but the normal input stream itself comes from the terminal; all error situations are handled interactively, with a message sent to the terminal and on-the-spot corrections accepted (provided the error was detected soon enough that no backtracking is necessary). The end of all input is indicated by command.

Tex supports conditional input based on the equality of two characters (possibly obtained as macro references) and the parity or sign of a variable (identified by digit). Conditions always take the form of a command followed by two groups with an "else" command between them. Comments are also supported, as mentioned above.

Case translation on input is supported, but with no provision for case escape characters, and with command names taken unchanged. There is no literal plain text input facility, but of course the control characters can be turned off individually with "chcode".

The Tex program can interface with a user-supplied subroutine, which is called by the "x" command.

5(l) Type

Type [Type 1979] was written in B at the University of Waterloo (Waterloo, Ontario) and can produce output either monospaced or typeset on either the Photon 737 Econosetter or the APS-V.

Commands and Names

A command in Type is indicated by the character "{". Then come the command name and arguments, delimited by blanks, and a "}" ends the command. Commands can occur anywhere, even within another command: the "{" and "}" characters are nested. This is useful because some commands "return a value" like macros or variables: that value is actually inserted into the input (plain text or command, wherever the command occurred).

Some other commands, like Nroff's "escape sequences", are initiated by "\" and named by the following character. For instance, "\<nl>" is a concealed newline and "\ " an extraordinary blank, just as in Nroff.

Expressions can be used as well as constants for numeric arguments; as well as the usual and the relational operators, logical ones are also supported (returning a truth value 0 or 1, like the relational operators). Scale designators are supported, and sometimes practically required, for physical distances.

As well as a simple sequence of nonblanks, three other forms are supported for a non-numeric argument. It may be enclosed in """ characters, enabling it to include blanks. If "'" characters are used instead, the string is taken literally (not scanned for commands and other special sequences). Finally, it may be enclosed in "[" and "]", whereupon it is also taken literally except that newlines (and ASCII TABs) are deleted; this construct also allows nesting, so the "[" and "]" must be properly paired. Thus the syntax enables commands not only to be inserted anywhere but also to take arbitrarily long arguments.

Macros in Type are defined by the "ds" (define string, meaning macro) command, and invoked by the same syntax as commands. There are actual commands returning the arguments, and the number of arguments, with which the macro currently being inserted was invoked; these are, of course, used in the macro definition where the arguments are to be placed. The "["-"]" argument syntax allows commands and other macro definitions to be tidily given within the macro text.

Type also supports formatted strings: its "di" command works much like Nroff's, except that "di" diversions can be nested in Type. Formatted strings are inserted, or concatenated, by the "pr" (print) command.

Another kind of string is defined by the "ac" (associate character) command. This works just like Cyphertext's "map", except that in Type the insertion takes place after command scanning and macro insertion rather than before.

Type also supports variables. As in Cyphertext, some formatting functions are controlled by assigning values to reserved names (of variables, in Type), rather than by commands.

There are commands to link (supply a second name for the same referent) and to undefine names, applicable to both macros and commands, and there are commands to find out whether a particular name is in use (for anything), which return a truth value of 0 or 1.

Ordinary Horizontal Effects

There is no underlining, even when text is monospaced.

Right justification is available.

The amount by which blanks may be padded for justification is limited and may be set by the user.

Special Horizontal Effects

Type supports local motions, which provide the only way to get a temporary indent.

Explicit environment switching is supplied, much as in Nroff (but environments are named, not numbered).

Vertical Effects

Vertical spacing of output can optionally be altered automatically, much as in Tex, so that characters never overlap even though the requested spacing of baselines would cause them to.

Reusable traps are supported. A trap may be associated with the end of each output line, and another with a vertical position in the output or the current diversion; but no more may be in effect at a time. As in Nroff, headings, footings, and footnotes are not supported directly, nor are adjustable top and bottom margin sizes.

Input/Output Effects

There is a command to read one line of plain text from the terminal.

Both index and work files are supported. Files must be opened by command, and are thereafter referred to by local names.

Commands can be passed to the system.

The "if" command gives conditional input. It takes a numeric argument and one or two string arguments, and processes the first string if the number is nonzero, otherwise the second (if any).

There is a command which, if it is executed while a macro is being inserted, causes the rest of the macro text to be skipped, and can optionally have the same effect on any macros from which the present one was invoked.

Case translation is supported, but not case escape characters.

The "ch" command returns a substring of its character-string argument, as specified by its numeric argument(s).

The sequence "\c" followed by a number is used to obtain special characters.

There are alternate commands for inserting a file, inserting an argument in a macro, and invoking a macro, which, used instead of the usual ones (or the macro name), cause the text to be taken literally as plain text, not scanned for commands. Any single character preceded by "\l" is also taken literally.

5(m) Points of Interest in Other Formatters

Mathematics and Graphics

The established conventions for printing mathematical formulae are demanding and somewhat complicated [Chaundy 1957]. Eqn (see under Nroff above) and Tex probably have the two most general, and mnemonic, facilities for formatting math by computer. However, several typesetting systems have simple math facilities. The system at the Jet Propulsion Laboratory (Pasadena, California) [Korbuly 1975] permits quite general formulae to be typeset, but while the user need not account for the actual widths of the characters, they do have to be counted manually to obtain the proper alignment. The system at Mack Printing (Easton, Pennsylvania) [Varley 1977] has a macro facility and one can enter the sum from k=1 to n as "SIGM n,k=1". Proff supports a mathematics character set, and [Beach 1977] indicates how to use macros for similar formulae.

The American Chemical Society (Washington, DC) has a more general system [Kuney 1966; Kuney 1969], with a complicated but quite versatile coding for formulae. It can also handle (2-dimensional) chemical structure diagrams. These are typed on the Army Chemical Typewriter, a device with special characters including extensive support for subscript and superscript characters; the paper tape it punches is scanned by the formatter to deduce the coordinate positions of the atomic symbols.

The formatter RED, at the Lawrence Livermore Labs (Lawrence, CA) [Beatty 1979] was written in TRIX, an interactive language based on the macro processor TRAC [Cole 1974, Mooers 1965] and on SNOBOL. This formatter handles formulae with some generality, but the relevant commands are syntactically awkward in that their operands must be simple subformulae: complicated expressions are built up by passing macro references as arguments. A postprocessor, REDPP, enables the output from RED to be handily combined with that from the graphics language PICTURE, so that computer-generated diagrams of all sorts may be included in a document and placed automatically.

The formatter at Computype (New York City) [Boehm 1976] has points in common with Tex. It considers that material is either text, tabular, or math, but that these can also be nested: math in text or vice versa, text or math in tables. The math facility has considerable generality, automatically handling such things as enlarged parentheses; its "nested math frames" may be delimited either by special symbols (as "$1" below, used like "{" and "}" in Eqn and Tex), or by mathematical constructs. In math, variables go in italic, and font requests apply to single characters only; a "special math hyphenation subroutine" prevents "sin x", say, from being divided between two lines of output. All commands in this formatter are unmnemonic: "*" or "$" followed by a short sequence of alphanumerics. This formatter may be compared with Tex and Eqn for the formula "theta-sub-12, squared, equals e to the power (a-sub-1 times x, plus b-sub-1 times y, plus c-sub-1 times z), divided by (a-sub-2 times x, plus b-sub-2 times y, plus c-sub-2 times z), all squared"; only mandatory blanks are shown in each case.

<u>Computype</u>:

```
 $(*gq'1'2"2=($fe"$1a'1x+b'1y+c'1z$1$sa'2x+b'2y+c'2z$t)"2$)
```

Here "*g" gives Greek, and "$f", "$s", and "$t" a fraction.

<u>Eqn</u>:

```
$theta sub 12 sup 2 = left ( e sup {a sub 1 x + b sub 1 y
+ c sub 1 z} over {a sub 2 x + b sub 2 y + c sub 2 z} right
) sup 2$
```

The begin-math and end-math characters are set to "$".

<u>Tex</u>:

```
    $\theta˘{12}^2={\left(e^{a˘1x+b˘1y+c˘1z}\over
                a˘2x+b˘2y+c˘2z\right)}^2$
```

Control characters are set as in the Tex description above.

Hyphenation

Hyphenation of words divided between lines arose hundreds of years ago; it is still used today because pad justification can otherwise cause unpleasantly wide word spaces when a formatted line has few words [Justus 1972]. Algorithms to select places to hyphenate a word have been developed [Knuth 1979, appendix H; Moitra 1979; Ocker 1971; Rich 1965], but since they look only at single words they share the defect that the "correct" hyphenation points, between syllables, cannot always be predicted: "I will re/cord his results and pre/sent him the rec/ord as a pres/ent." Even for words with only one syllabification, English is so complicated and irregular that any algorithm is likely to have exceptions, to say nothing of text containing different languages. This kind of problem is why formatters supporting automatic hyphenation invariably have some sort of facility to override it, often including hyphenation exception lists.

Autoscript, at Western Electric (Winston-Salem, North Carolina) [Stuckey 1969; Stuckey 1973], and the system at the Central Intelligence Agency (Washington, DC) [Kunzel 1966] need no automatic hyphenation. If justification would pad blanks beyond a certain threshold, they justify lines by altering the set width as well; thus each letter in the line may be followed by additional space proportionate to the width of the letter itself. This produces a reasonably pleasant appearance, distinctly better than simply inserting equal space between letters; whether it is better than hyphenation (or, indeed, whether that is better than greatly padded word spaces) is a matter of taste. (And would it be better, in set width expansion, to insert the proportional space before as well as after the letter?) In Autoscript, there is a limit on set width expansion also; exceeding it is treated as an error, forcing manual hyphenation (much as with Tex).

On the other hand, the Linotron Page Formatting Computer [Makris 1966], from Mergenthal Linotype (New York City), actually has a hyphenation algorithm built into its central processing unit.

Preprocessing, and Human Intervention

There are systems where a formatter is routinely used with a preprocessor to alter the form of its input. At the American Institute of Physics (New York City) [Alt 1973], Page-1 is the formatter; since its rules for strings are restrictive, a preprocessor is used to allow alternate forms. Particularly useful are mnemonics for "compound characters": "=(i-breve)=" gives an "ĭ". The preprocessor is designed to accept input in a format that facilitates indexing, which Page-1 does not support. (More recently, however, some AIP publications have used Troff extensively [APS 1977].)

In the system used at the Honeywell Computer Journal [Bemer 1973], text is entered using the Honeywell 6000 Text Editor, which contains its own formatter [Honeywell 1972, pages 31-34] (something between Roff and Runoff); the commands are given in the format of that formatter, then passed through a preprocessor to Page-2 (a later version of Page-1). As well as translating the commands to Page-2's less mnemonic forms, the preprocessor handles several things specially: "^" becomes a fixed en-space; """ becomes a "“" or "”", whichever is correct, or "¨" if it is overstruck (by backspacing) with a letter; "o" becomes "•" if it is the first word in a paragraph beginning with a certain temporary indentation.

That journal's editors have found that "reader attraction and satisfaction is increased significantly by tight control of page layout," so they lay out each article in rough, decide where they want each column of each page to end, and edit the text to fit. They "have to get into the guts of the author's meaning and say it shorter and clearer, without altering the flavor or meaning in any way! Being forced to do this ... yields a big dividend in increased readability." Thus the computer has no say in where to end pages. (Such attention is reminiscent of the best newspaper practice [Hutt 1967], or the weekly version of Life magazine [Hamblin 1977, chapter 7].)

Setlst [Messina 1970] is an automatic-recognition program to translate tabular computer output into the appropriate codes to drive a formatter, complete with case shifts and font changes. Setlst was developed by the U. S. government so that the books of tabular data it has to print might be typeset (thus be smaller and more legible) directly from the computer (thus have fewer errors).

Two-Pass Formatting

Autoscript (see above) formats text in two phases. After the first pass, a galley proof can be produced, with paragraphs and lines numbered; corrections can then be keyed in, addressed with these numbers, and the affected paragraphs are reformatted. The second pass divides output into pages, inserts headings and footings, and so on.

The CIA system (see above) provides a similar galley proof, with each word numbered, likewise allowing corrections at that level.

IBM's Text/360 [Ziegler 1969], a system designed for large documents, also allows some corrections after formatting. An unusual feature is "logical-sheet mode, [which] serves to freeze a change to certain pages so that the change is not allowed to propagate through following pages." Thus an insertion in page 8 may cause it to grow onto page 8.1, with page 9 being unchanged until otherwise requested.

In none of these systems are the operations upon the galleys interactive as in EZ27.

<u>Specialized Editor Only</u>

The system at Bell-Northern Research (Ottawa, Ontario) [Reed 1977] is unusual not for its formatter (a modification of Script) but for its editor Ted. Text is considered as a sequence of lines ("there has to be some organization imposed on the file"), but Ted does not see them, treating a newline as a blank: "/The Lodger ... Family Plot/" would match the text beginning with the next occurrence of "The Lodger" up to the occurrence after that of "Family Plot", even though that text, or even either delimiting phrase, is divided between lines.

<u>Special Uses of Optical Character Recognition</u>

This thesis generally does not mention the medium of text entry, for the formatter usually need not know it. One medium often used (so that text can be typed cheaply offline [Mack 1975; Schneider 1974]) is OCR. The Computype system (see above) initially has its input typed double-spaced; the OCR can then read corrections typed between the lines! Another special use is on the system at Perry Publications (West Palm Beach, Florida) [Perry 1966], which supports not only input but also page make-up by OCR; for the latter, a form with a grid of "□"'s is pencil-marked to indicate where on the page to begin each item. (The demanding requirements for page make-ups of newspapers and such things have also led to systems using special terminals for this purpose [Boissavy 1973; Newspapers 1977].)

Portability

Scribe, designed at Carnegie-Mellon University (Pittsburgh, Pennsylvania) [Reid 1980], achieves high portability by carefully excluding device- or installation-dependent constructs: for instance, actual filenames may not be referenced in commands, indirect names being used instead. Scribe attempts to relieve the user of specifying typographic information by providing a wide range of predefined formats; the user selects from these by name, as in "@Style {References CACM, Footnotes "*", Doublesided}".

Author's Experience

The author has designed one earlier formatter, Set [Brader 1979]. This was designed to closely resemble Proff and Roff, producing output on the University of Waterloo's Photon 532 Fontmaster typesetters. Its only innovations are that temporary indents can be nested, and that files of formatted text, already coded for the typesetter, can be inserted by command. Still, the experience with Set may have influenced EZ27 through the "second-system effect":

"An architect's first work is apt to be spare and clean. He knows he doesn't know what he's doing, so he does it carefully and with great restraint. ... Sooner or later the first system is finished, and the architect, with firm confidence and a demonstrated mastery of that class of systems, is ready to build a second system ... the most dangerous system a man ever designs ... using all the ideas and frills that were cautiously sidetracked on the first one. The result, as Ovid says, is a 'big pile.'" [Brooks 1975, page 55]

6. THE SYNTAX OF INPUT TEXT

EZ27's input, as mentioned above, is general text, i.e., commands and plain text intermixed, like the input to a conventional formatter. Its syntax was designed to avoid certain problems perceived in existing formatters.

6(a) Plain Text

Plain text can be considered either as a simple sequence of characters, or as a sequence of *words* delimited by *space characters* such as blanks and newlines (others might include tabs and formfeeds). With the first interpretation, " " and " " are different: each blank produces a certain amount of space on output. Similarly, while a single newline has to be more or less equivalent to a blank, multiple ones may have additional effects. This is the natural method for a formatter incorporating some approach to idempotency or automatic recognition: "Ideally a document containing *no* formatting commands should be printed sensibly. ... Blank lines cause fresh paragraphs. ... If a line begins with *n* blanks followed by text, it causes a break and a temporary indent of +*n*." [Kernighan 1976-S, pages 220 and 223]

The second interpretation, text as a sequence of words, treats any non-null sequence of space characters as equivalent. This has the advantage of simplicity, especially since the atom of natural language is in fact the word; it also follows modern programming language practice (as in C and PASCAL).

Most formatters actually take some intermediate position. Roff and Nroff, for instance, generally follow the first interpretation, but ignore blanks that precede a newline; Tex generally follows the second, but "<nl><nl>" and such sequences cause breaks. For EZ27, it was felt that one interpretation or the other should be used strictly, and the second one was chosen (with the ASCII TAB as a third space character) for the reasons mentioned above and because of some practical considerations:

Interpreting " " on input as a double-width output blank encourages users to think in terms of fixed-width characters, to think that text aligned on input must be aligned on output; but with typeset output, that is false.

A formatter of monospaced text, such as the one in the paper quoted above, may occasionally be called upon to process text which was originally typed with no intention of being machine-formatted and which therefore contains no commands. In the case of a formatter for typeset output, this seems rather less likely, simply because the user community will be more sophisticated. In particular, users wanting wide blanks will probably use the orthodox formatter commands; extra blanks on input will mostly be typographical errors (such is the author's experience, anyway), in which case the formatter should absorb them.

Finally, allowing meaningless extra spaces following newlines is convenient with the particular syntax of EZ27 for reasons that will be described shortly.

6(b) Command Initiation

The obvious, albeit not the only, way to distinguish a command from the plain text surrounding it is to precede the command by a specific short character string, a <u>command indicator</u>. 1-character command indicators used by existing formatters include "[" (Page-1), "{" (Type), "/" (Cyphertext), and "\" (Nroff, Tex, and Type); 2-character ones used or proposed include Format's ")", and newline followed by "*" (Dip), "'" (Nroff), "?" [Wetherell 1978, chapter 4], and of course "." (Nroff, Proff, Roff, Runoff, Script, and [Kernighan 1976-S, chapter 7]). Many of these formatters allow the control characters that are not space characters to be reset. Several support additional control characters (tab, insertion, parameter, discretionary hyphenation, etc.) that are not command indicators since they basically act alone; these also are often resettable. This seems unduly complicated.

Input might come from a device with a limited character set, so the characters of the command indicator are best chosen from among blank, newline, and the more widely available graphic characters. However, the command indicator should not be a character string likely to occur often in plain text. (If it can ever occur, there must be a way to change it, and/or an escape allowing it to be entered; this will be inconvenient if it has to be used often.) By these considerations, a 1-character command indicator is clearly impractical: any particular character from the limited available set is simply too likely to occur too frequently (especially in technical text, which tends to draw on a wide range of symbols).

There are, however, 2-character strings that are most uncommon in plain text, such as those cited above. Their mandatory use of a space character is awkward, though, for changes of <u>typestyle</u> (font, type size, and body size) and the like may be wanted in mid-word; interrupted text processing is rather inelegant, as is Nroff's use of more than one command indicator. Also, when a newline is required, commands occurring close together become obtrusive (though a command list facility can alleviate this).

EZ27's command indicator was chosen to be 2 characters, each resettable but graphic by default. (Users could reset the first character to a newline if they preferred that, thus reintroducing the problem of whether a command list ends a word.) As the default command indicator, "/*" was tried. This has high visibility in ordinary text, and has the connotation of an interruption since it designates a comment in, for instance, PL/I and B. (The null command was then defined as a comment, so that "/* " actually does introduce one; the next "/" ends it, so "/* comment */" is perfectly correct.)

In what follows, the characters "/" and "*" will be used for the sake of readability, rather than "the first and second command indicator characters".

6(c) Command Lists

Command lists seem useful, though Script, for instance, has an unnecessarily complicated syntax for them. One character should suffice to introduce a second command in a list. EZ27 uses "*" also for this purpose, and "/" also to close the list: thus if "a", "b", and "c" were commands, "/*a*b*c/" would be a valid command list. This is straightforward and requires no additional control characters; also, if "/" is set to newline, a command list occupies exactly one input line.

Formatters such as Runoff that initiate a command with "<nl>." and terminate it with another newline allow the latter to be part of the "<nl>." that begins the next command. Otherwise, an unnatural blank line would be necessary where no plain text came between commands; this strange requirement would lead to mistakes. Likewise, in EZ27 the "/" closing a command list can be part of the "/*" initiating another one, so "/*a/*b/" is the same as "/*a*b/".

6(d) Command Arguments

Most other formatters delimit short command arguments with blanks, if at all. EZ27 extends this naturally to allow any sequence of space characters, in keeping with the word delimiters of plain text. These short arguments are of two kinds: numbers, optionally preceded by a "+" or "-" specifying relative values (relative values by multiplication and division were intended originally, too, but "*" and "/" got pre-empted as control characters, so the other use was dropped, in intent temporarily); and keywords specifying options particular to the command. Keywords can be abbreviated, normally to the first letter. Space characters are also allowed, and recommended for clarity, between commands (and before the quoted arguments described below): "/*a *b/" and "/*a<nl>*b/" are the same as "/*a*b/".

Some commands, such as those for footnotes and macros, operate on considerable quantities of text. There are also the _transformational_ commands, which change some mode or parameter of formatting, or of the interpretation of input; these can affect all the text following them, so it really is their operand. In EZ27, it can be given as a _text argument_ to the command, and such arguments can be general text, allowing further commands nested within. (Where a transformational command is given with no text argument, its effect is _permanent_: it acts on all text until countermanded.) This notion was inspired by a similar but more restricted construct in Page-1; it was developed independently from Dip's scopes, which work almost the same way, and Tex's groups, which can produce a similar effect although they differ syntactically. (The nested structures of Type resemble scopes and text arguments only superficially, since their semantics are quite different.)

At first, any character not in a text argument was allowed to delimit it, as with substitutions in QED: "/*a #text#/" and "/*a $text$/" would mean the same thing. This avoided creating any more control characters. Naturally, a nested command taking an argument would require a different delimiter:

```
/*a #a-text /*b !a-b-text!/ more-a-text# *c #c-text#/
```

In typing a complicated group of nested commands, the fact that blanks after a newline are meaningless may be useful: commands at different levels of nesting may be typed on different lines indented by proportionally varied amounts, as in programming language practice.

One often wants to have several formatting commands operate on the same argument text: an introductory quotation, say, could be indented left and right, right justified, and in a different font, type size, and body size from the main text. This might well lead to something like (not actual EZ27 commands):

```
/*in 40 $/*ir 5 %/*rj '/*ft i @/*sz 8 #/*bd 9
                 =Et tu, Brute=/#/@/'/%/$/
```

This is not only grotesque, but difficult to use, for the choice of the various delimiters is restricted to characters not in the text of the real argument, and they must be paired correctly besides.

Because of this problem, the syntax was revised so that if several transformational commands occur consecutively in the same command list, with no other commands and no text argument intervening, then a following text argument applies to all the commands in the group. That grotesque example then simplifies to:

```
/*in 40 *ir 5 *rj *ft i *sz 8 *bd 9 'Et tu, Brute'/
```

(Under the original syntax, this would have meant that the first five commands applied permanently, and the text argument related to the last command only; now, to get that effect, a "/" would have to be inserted before "*bd", to start a new command list.)

However, attempts to actually use the syntax showed that the delimiter problem still occurred when trying to format text of some complexity. The syntax was therefore revised again to delimit text arguments by a particular character, defaulting to """. Then the first nested example becomes:

```
/*a "a-text /*b "a-b-text"/ more-a-text" *c "c-text"/
```

However, the advantage gained from the 2-character command indicator is now lost! While initiating a command list may require 2 characters, resuming it after a general text argument requires only one: the """. In the last example, the third, fourth, and sixth """ characters resume command lists, and two are directly followed by a closing "/". Now, a """ could be taken literally unless followed by a command list's "/" or "*" (and intervening space?), just as a "/" is taken literally unless a "*" follows it; or a 2-character string could also be used to delimit text arguments; both seem rather unappealing, and no change was actually made.

Some commands require one other form of argument, an arbitrary character string not interpreted as text to be formatted: it may be a filename, a macro name, even the contents of a macro. These are called string arguments, and are delimited by "″" just like text arguments, but they are taken quite literally, not examined for command lists within. Whether a particular quoted argument is interpreted as string or text depends on the individual command.

6(e) Macro and Variable Invocation

In Proff and Roff, a macro can be invoked by a syntax similar to that for executing an ordinary command, but this forces it to start on a new input line, so the alternative insertion character syntax is available for inserting a macro anywhere in the input. In EZ27, on the other hand, a command need not start on a new line; therefore the macro invocation syntax, as initially defined, was exactly that which executes a command.

However, the insertion character can be used to insert a macro in mid-command as well as in plain text. For instance, Proff identifies type sizes by number only, but if, say, macro "brevier" was defined as "8", and the insertion character set to "#", one could say "<nl>.pt #(brevier)". This is impossible in EZ27 as defined so far. An actual command could be created for this effect, something like "/*pt *insert "brevier"/", but this is verbose and inelegant, as well as inconsistent with the simple "/*macro/" to insert a macro in a text.

To avoid creating more control characters (such as an insertion character), a new combination "**" of existing ones is used to denote macro insertion within a command list (the macro must not contain a closing "/" and plain text). The example becomes "/*pt **brevier/". Or macro "brevier" could be defined as "*pt 8", then invoked just by "/**brevier/": in Proff, "<nl>.#(brevier)". (Of course, if the macro was defined as "/*pt 8/", it could be invoked simply by "/*brevier/", but then it could not take a text argument.) "/** " and "/***" were defined to give comments, the same as "/* ".

So far this is reasonably elegant and terse, but there is a problem: "**" cannot be used on a macro with arguments, for "**macro arg" in a command list must be interpreted as the contents of the macro followed by "arg". However, one does not often want such an insertion; macros inserted in commands tend to be simple, as in the previous paragraph. Therefore a somewhat awkward syntax would not hurt here, and, rather than alter the existing syntax or define any additional control character, "**"" was defined to take everything up to a closing """ as the name of a macro to insert in the command list followed by the arguments to invoke it with: "/*command **"macro m-arg" c-arg/".

Variables are inserted the same way as macros.

6(f) Literal Text

Before the necessity of distinguishing text and string arguments had become clear, "literal text arguments", opened by "*"" and closed by another """ (originally with any delimiting character for """), were defined; these would be similar to text arguments, but occurrences of "/*" would not indicate commands. Now, string arguments are already taken literally; "*"" simply gives a string argument. However, if

"*"" does not follow any command (as in "/*""), the text up to the following """ is taken as plain text, again, literally with "/*" not being recognized.

These literal modes (though they survive as relics) were not really satisfactory once """ was introduced, since it could not be entered literally. A FORTRAN-like """"" was considered, but it would be quite inconsistent with the way a literal "/*" was obtained (and relatively awkward to implement). Instead, a fourth control character, "\", was defined. It removes special meaning from the control character or space character following it (so "\\" and "\ " mean the same as in Nroff), and is simply ignored preceding anything else. The "\" was also intended to be resettable.

6(g) Remarks About the Syntax

At this point further study of the command syntax was laid aside until a good working formatter existed, which is to say, permanently. Still, an innovative syntax had been developed, and some experience was gained with it in the course of program development. Some choices were less than ideal: probably """ to open <u>and</u> close text arguments could be improved, for instance. Still, there may be room for a really good syntax to be developed from that of EZ27.

On the other hand, Tex (which was being developed at the same time) may be a much better basis to work from.

7. ORIGINAL SPECIFICATION

This section details the set of commands originally specified for EZ27. Some were extended shortly after the specification was first written, and these extensions are included here. The descriptions have been shortened from their original forms since they would repeat material elsewhere in this thesis.

Where commands take multiple arguments, they generally can come in any order: "IN AT 10 BY .1" could be given as "IN BY .1 AT 10". In displaying the syntax of various commands, the notation is consistent throughout this section. "#" represents a number, which, where it makes sense, may be preceded by "+" or "-" indicating a relative value, and may have decimal places. Anything bracketed by "[" and "]" is optional. Things separated by "|" are alternatives, one of which is required unless the whole is bracketed. (If "|" begins a line, the whole line is an alternative.) "..." means that additional things of the same type are permitted. Other nonalphabetic characters are meant as written; so are words in upper case. Words in upper and lower case may be abbreviated by omitting any or all of the lower-case letters. Words in lower case ("filename") indicate syntactic categories.

7(a) Controller Commands

The control component of EZ27 operates on paragraphs on a galley file. It is driven by interactive commands from the terminal. The USE command must be issued first, to specify the galley file, before anything else can be done.

Each command may be entered on a separate line, or several may be entered on one line delimited by ";"'s.

The commands are:

<u>DEL</u> # [<u>TO</u> #]

Delete the specified paragraph, or range of paragraphs. If more than 7 paragraphs are included in the range, the user is asked to verify the command.

<u>DONE</u>|<u>QUIT</u>

Leave EZ27, renumbering the paragraphs in the galley file.

<u>EDIT</u> #

Edit the specified paragraph at the word level.

<u>IN</u> [[<]<u>filename</u>] [<u>AT</u> #] [<u>BY</u> #]

Input is taken from the named file, if any, or else from the terminal (terminated by the Break key). If "AT" is used, the input is placed beginning with the paragraph whose number is given; otherwise, it goes at the end of the file. If "BY" is used, the paragraphs entered are numbered with the specified interval; if not, the interval is 1 if at the end of the galley file, and .01 otherwise.

<u>REP</u> # [<u>TO</u> #] [[<]<u>filename</u>] [<u>BY</u> #]

Exactly equivalent to "DEL # [TO #]" followed by "IN AT # [[<]filename] [BY #]".

<u>SET</u> <u>macroname</u> <u>paramname</u>...

Define a macro. The macro text is read from the terminal (terminated by the Break key). The macro becomes available for use in future insertions anywhere in the file. It must not already be defined.

USE filename

Select the galley file to use. If the file already exists, EZ27 assumes that it is a galley file which is going to be edited (perhaps added to); if not, it is created and initialized. Only one galley file can be in use at a time, and another USE command after a file is selected closes the first file, renumbering its paragraphs exactly as with DONE.

VIEW [#|FROM #] [>filename]

The entire galley file is displayed (in readable format), or just one paragraph, or all the paragraphs beginning with a certain one. The output may be directed to a file, and the Break key will cut it off.

!systemcommand

The command line (all text up to the next newline, regardless of ";"'s) is passed to the system.

7(b) Edit Subcommands

The EDIT command mentioned above invokes an interactive editor that operates on the words (as originally delimited by space characters) of a paragraph. Space characters between edit subcommands are ignored. The subcommands are:

D|Q

Leave the editor: the paragraph is fixed.

P

Print the paragraph, with repeated occurrences of a word numbered, and operations like typestyle changes in a readable form.

#P

Print the last part of the paragraph, # telling where to start printing (not decided: whether in words, lines, or percentage of paragraph length). The Break key will cut off the printout.

S*∫word∫text∫

Substitute the text for the word throughout the paragraph. With the S subcommands, "text" is a sequence of zero or more words delimited by blanks, and almost any character not found in the word or the text can be used as the delimiter "∫".

S#∫word∫text∫

Substitute the text for the "#"'th occurrence of the word in the paragraph.

S∫∫text∫

Substitute the text for the next occurrence of the same word in the paragraph.

S*∫∫text∫
S#∫∫text∫

These forms have the natural analogous meanings.

!systemcommand

The command line is passed to the system.

7(c) Formatting Commands

The input read by the IN and REP commands consists of general text, with commands whose syntax is described above. In the descriptions below, ["text"] with no explanation refers to the optional text argument of a transformational command.

Command names have been chosen to be 2 to 5 characters; there is no such restriction on macro and variable names. The commands are:

/*<u>macroname</u> "<u>argument</u>".../

Invoke macro, with the argument strings specified.

/*<u>varname</u> ["<u>format</u>"]/

Insert variable. The format string indicates how to print the number 1, as "i", "1", "00001", "   1", etc.; the leading blanks in the last form are an estimation of how many digits the number will use, for justification occurs before the variable is evaluated.

/*<u>AT</u> [<u>Near</u>] [<u>All</u>|<u>Singleparator</u>] [<u>Even</u>|<u>Odd</u>]
<u>Top</u>[+#]|<u>Head</u>[+#]|<u>Foot</u>[-#]|<u>Base</u>[-#]|#|-#
"<u>traptext</u>" ["<u>text</u>"]/

The trap text will be inserted at the given position on a page. If "A" is used, the trap is reusable, otherwise one-time. If "S" is used, at most one copy of the text will print on the same page "N"ear the same place, if identical *at commands occur. (This would be useful for footnote separators.) If "E" or "O" is used, the text will be printed only if the page number is even or odd, respectively. "T", "H", "F", and "B" refer to the margins defined under *marg.

EZ27 "Traps" contain formatted text, not general text.

When another trap is set for the same place as an existing one, the latter is dropped, unless "N" is used on one of them, in which case it will take a new position adjacent to the other. (If both have "N", the earlier one has priority.)

/*BAL [#] "text"/

Formats the text into lines of length as nearly equal as possible before justification. If a number is included, at least that many lines will be produced. (A break is implied before and after the text.)

/*BCOL [#]/

Jump to top of next column, or specified column.

/*BREAK/

Break.

/*BSP [#]/

Backspace horizontally by specified (or default) amount.

/*CASE [Lower|Upper|Reverse|Normal] [Escape "char"] ["text"]/

Controls case translation on input.

/*CENT ["text"]/

Center justify each output line. (A break is implied, before and after the text if any.)

/*CHAR #/

Expands to the non-ASCII printing character represented by the number, in a device-dependent code.

/*CLOSE "filename"/

Close the named work file.

/*COLS # [Width #] [At #...]/

Output in (specified number of) multiple columns. "W", if used, specifies the width of each; otherwise, the largest width that will fit in the *width setting, with minimal gutters, is chosen. "A", if used, is followed by a sequence of numbers giving the left-hand margin position of each column.

/*DROP [Near] [All|Singleparator] [Even|Odd] Top[+#]|Head[+#]|Foot[-#]|Base[-#]|#|-# ["traptext"]/

Remove the trap set by the *at command with matching arguments. The actual trap text need be included only if there is ambiguity! If "N" was not used, a fresh *at with null ("""") or omitted text will also drop the trap.

/*END/

Ignore the rest of this file.

/*FILL ["text"]/

Cancels *nfill. (A break is implied, before and after the text if any.)

/*FLOAT [#] ["text"] [More|Out]/

This command diverts output to a formatted string. Several can exist simultaneously, distinguished by numbers (if omitted, -1 is assumed). When a *float command is executed, the text argument is formatted and appended to the specified string (the command does not cause a break). If "M" was not used, the string is then placed in the output, immediately if "O" was used, or as floating glued text if not. (If two such come up for output simultaneously, low numbers take priority.)

/*FONT # ["text"]/

Specify typeface.

/*<u>GLUE</u> ["<u>text</u>"]/

If there is an argument, it becomes glued text; if not, automatic widow elimination is switched on.

/*<u>HEIGH</u> #/

Set overall page height. (This takes effect as soon as possible after the next break.)

/*<u>HSP</u> [#]/

Space forward horizontally by specified (or default) amount.

/*<u>HYPH</u> [<u>A</u>t "<u>char</u>"] ["<u>text</u>"]/

If "A" is used, the character argument following it becomes a control character, the discretionary hyphenation character: words containing it will be hyphenated only where it occurs (even if they contain hyphens already). Otherwise, the command cancels *nhyph.

/*<u>IN</u> "<u>filename</u>"/

Insert the contents of the named file into input as general text.

/*<u>IND</u> [<u>R</u>ight|<u>B</u>oth] # [<u>F</u>or #|"<u>text</u>"]/

Indent (by the specified amount). If "R" is used, the right margin is indented; if "B", left and right (equally); otherwise, the left only. If "F" is not used, this takes effect at the beginning of the next output line; if it is, a break is implied, and the indentation expires after the specified number of output lines, or at the next break, or at a *ind without "F" affecting the same margin.

/*INDEX ["text"]/

If there is an argument, the specified text is diverted into a work file. Macro and variable references are expanded, so that page numbers and the like are filled in, but other commands are not executed, so that the result is still general text. When there is no argument, that work file is closed, and inserted as if with a *in command.

/*JOIN/

Even though another command would imply a break at this point in the text, this command prevents it.

/*LEFT ["text"]/

Text is to be left justified. (A break is implied, before and after the text if any.)

/*MARG [Top #] [Head #] [Foot #] [Base #]/

Reset top or bottom page margin sizes. "H" and "F" refer to the total margins outside running text on a trapless page, while "T" and "B" refer to outer zones left blank beyond usual headings and footings. "H" refers to total margin above running text on a normal page, including "T" amount, and "F" to same at bottom. (The command takes effect as soon as possible after the next break.)

/*NFILL ["text"]/

Nofill mode is used: each input newline gives a break. (So does the command, before and after the text if any.)

/*NGLUE/

Turns off widow elimination, which is off by default.

/*<u>NHYPH</u>/

Turns off automatic hyphenation.

/*<u>NLINE</u>/

Like *break, but in pad justification the output line it terminates is padded.

/*<u>PAD</u> [<u>At</u> "<u>char</u>" [<u>With</u> "<u>string</u>"]] ... ["<u>text</u>"]/

If "A" is not used, this turns on pad justification, the default. (A break is implied, before and after the text if any.)

If "A" is used, any output lines containing (any of) the specified character(s) that need to be padded for justification will be padded only at each such occurrence; if "W" is used, there will be leadering with repetitions of the specified character string, instead of blank space.

/*<u>PAGE</u> [<u>Even</u>|<u>Odd</u>|#]/

Begin new page. New page number may be specified or not, or if "O" or "E" is used it will advance 1 extra number if necessary to make it odd or even respectively.

/*<u>RIGHT</u> ["<u>text</u>"]/

Right justify each output line. (A break is implied, before and after the text if any.)

/*<u>SET</u> "<u>macroname</u> <u>paramname</u>..." "<u>macrotext</u>"/
| "<u>varname</u>" # [<u>Format</u> "<u>format</u>"]/

The first form shown defines the macro named, assigning it the "macrotext". If it is to have parameters, they are given local names, separated by blanks from the macro name as shown, mimicking the invocation. They are referred to in the macro text as if they were (argumentless) macros themselves.

The second form shown defines the variable named, assigning it the value given and, optionally, a default output format (as described above). It should be noted that macros are expanded as early in processing as possible, are treated basically as source diversions in fact, while variables are left unevaluated as late as possible.

Each *set command is associated with the position where it occurs with respect to the text. The macro (or variable) named can be used with the value given for future insertions between that position and the next *set command affecting the same macro, and for insertions to earlier points if it is the first command for the macro.

/*<u>SIZE</u> # [<u>On</u>] # ["<u>text</u>"]/

Specify output type size and effective body size. If the "O" is omitted, the arguments are assumed to be in the order shown, but it is recommended to be used for clarity.

/*<u>SPACE</u> [#|<u>To</u> #]/

Space downwards by specified amount, or to specified position on (the same or next) page. "/*SPACE To 0/" is equivalent to "/*PAGE/".

/*<u>SUB</u> [<u>By</u> #] "<u>text</u>"/

Set the text argument as a subscript, displacing its baseline down by the given amount, or a default of 1/4 the type size (subject to later tuning). No change of type size is implied.

/*<u>SUPER</u> [<u>By</u> #] "<u>text</u>"/

Analogous to *sub, but giving a superscript.

/*SYST "systemcommand"/

The argument is directed to the system (during the final pass).

/*TAB [Tab "char"] [Newline "char"] [Format "format"]
["text"...] [Format "format" "text"...] .../

General tabular output command, adapted from Tbl (see under Nroff above). A table format generally consists of a sequence of items separated by blanks and ";"'s, each item corresponding to an element of one of the text arguments. The elements are delimited in the text by ASCII TAB characters (or the substitute defined by "T"), and a new row of table entries is started by a newline (or the substitute defined by "N").

When an additional text argument occurs with no new format preceding, the same format is understood to repeat from its beginning. When a new format occurs after one or more text arguments, it is understood to describe additional text which is still part of the same table. If there is no format before the first text argument, the command is understood to use the last format previously defined in a *tab command, whether there was any text in that command or not. When a command with no text includes "T" or "N", the values so set are permanent.

The basic formats are (upper or lower case): C, L, N, P, R, and S. These describe a table entry that is to be, respectively, centered, left justified, numeric-aligned (units' column matching), pad justified, right justified, and continued into an additional column.

In the first line of the first format of the table, qualifiers may be appended to each format: <#, >#, =#, and @#, indicating respectively (as the number) the maximum column width, minimum column width, fixed column width (short for <#>#), and fixed starting position; and a simple # between two format items indicates the amount of space to leave between the columns if the default is not desired.

Any single character placed between two formats (or between a format and a #) represents a vertical rule between two columns; it may be preceded by V to prevent ambiguity. If the contents of one or more columns are to be replaced by a horizontal rule, H followed by a character may be used, with S used to extend it across the necessary columns; the other columns will then usually be blank, represented by B. A rule need not be a simple line (as "|"), but may be any printing character, including one produced by the expansion of a macro or *char command, repeated as necessary.

As the blank corresponds (partially) in the format to the tab in the text, so the ";" corresponds to the newline. A single character between two ";"'s is understood to represent a horizontal rule extending across the entire table. Where there are more lines in the text (delimited by newlines) than in the format (delimited by ";"'s), the last format line (excepting any consisting solely of horizontal rules and the like) is repeated as necessary, unless any portion of the format is enclosed in "(" and ")", in which case that portion is repeated, with an implied ";" preceding it if necessary.

/*<u>TITLE</u> "<u>ltext</u>" ["<u>ctext</u>"] "<u>rtext</u>"/

Produces a three-part title. The middle part is optional. Equal-width columns are used; each part may be set as multiple lines if sufficiently long, in which case the parts will be top-aligned with each other. (A break is implied before and after the title.)

/*<u>TRANS</u> "<u>char</u>" "<u>string</u>" ["<u>text</u>"]/
| "<u>chars</u>" "<u>chars</u>" ["<u>text</u>"]/

Translate characters on input. The first form allows any single character (including an otherwise unused ASCII control character) to be replaced by an arbitrary string. The second form allows single characters to be transliterated respectively to other characters. The first argument must consist of a single character or exactly as many characters as are in the second argument.

/*<u>ULINE</u> [<u>eXcept</u> "<u>chars</u>"] ["<u>text</u>"]/

If there is a text argument, that text will be underlined on output. If "X" is used, the characters in the list given with it will not be underlined; if there is no text argument, the "X" list applies until a different one is given, but if there is a text argument, the list applies only to the underlining of that text. It is not permissible to omit both arguments.

/*<u>WIDTH</u> # ["<u>text</u>"]/

Set the output page width. (A break is implied, before and after the text if any.)

/*<u>WRITE</u> "<u>filename</u>" "<u>string</u>"/

Append the character string to the work file. An AFT-name or complete pathname may be used for the file. A *close command for the same file will enable it to be included in input by *in, and will cause the next *write to overwrite it rather than appending.

8. IMPLEMENTATION OF THE GALLEY FILE

Since the size of the galley file is important, as well as the ease of its use, this section will describe its format in detail.

EZ27 was implemented in the language B, for TSS on the Honeywell 66/60. The words of this machine are 36 bits long and when used for ASCII characters may contain up to 4 each. (BCD characters are also supported at 6 per word, but with no lower case they are of little use for textual applications.) The basic unit of disk i/o is the <u>sector</u> of 64 words, and B supports random-access reading and writing of sectors (addressed within a file by consecutive integers from 0).

8(a) The B-Tree

Since randomly located changes to the galley file are important, no sequential or indexed-sequential storage organization would do; since sequential access is also necessary, hash or trie methods would be awkward. Some form of tree seems to be best. One having good properties for external files is the B-tree [Knuth 1973-S, section 6.2.4]; each node of a B-tree of order n has from n/2 to n entries (the root may have fewer), and at the next lower level (if any) one more successor node than it has entries, all ordered by values of a key. New entries are inserted at the leaf (lowest) level, and whenever a node overflows it is split in two, one entry being transferred to the next higher level (perhaps a new one) and inserted there. If the branching order is high, not many nodes need be examined for a particular entry, so each node can simply occupy 1 sector without excessive i/o; since all leaves stay at the same level, there are no degenerate forms (as of, say, a binary tree whose elements were inserted in order); and since nodes can vary from full to half-full, there is a good compromise between use of filespace and frequency of node additions (or deletions).

The EZ27 galley file was therefore implemented as a random-access file organized as a B-tree with each tree node occupying one sector. The B-tree keys are, of course, the paragraph numbers. The 36-bit word allows a maximum integer value of 34,359,738,367; the paragraph numbers, to fit in 1 word, were implemented as integers scaled by 100,000, ranging from 0.00001 to (for simplicity) 99999.99999.

To keep the B-tree's order high, its entries must be small. A <u>paragraph descriptor</u> includes a pointer to the paragraph's contents, and certain information about its format (for Pass 2 and the Editor, stored as several small integers packed together). An opcode fits in the same space, being distinguished by a negative value where the text-pointer would be. A trap acts like an opcode, but is stored like an ordinary paragraph (a few bits in the descriptor distinguish it).

Originally the paragraph descriptor was planned to be 3 words, 1 for the text-pointer and 2 for the format information; the latter proved inadequate and 1 more word was added. Since each B-tree node must also include the key of each paragraph and pointers to successor nodes, this change reduced the B-tree order from 12 to 10: in a 64-word sector, 11 words for successor-pointers, 10 for keys, and 40 for 10 paragraph descriptors. (B-tree entries need not actually be of fixed size [McCreight 1977], but here it seems best.)

Several independent improvements [Knuth 1973-S, section 6.2.4] were held in reserve in case filespace or i/o time proved to be limiting. The B-tree could be replaced by a B*-tree, wherein a node can overflow into the adjacent one at the same level, rather than being split (some insertions are slower, but filespace is conserved). Buffering in the program, to make much of the i/o virtual, would substantially improve performance, and the program was designed to facilitate its introduction. Also, there is no strong reason (though it does simplify the program) that each level of the tree have nodes of the same format: most of the nodes are leaves, and it would cause no great disturbance to place all the actual entries at that level, whereas the leaves use no successor-pointers. A related matter is that those pointers need not be full words, as sector numbers are consecutively assigned and will be small. If 3 pointers were packed per word, pointer fields omitted from leaves, and all opcodes and paragraph descriptors stored in leaves, the order could be 12 for the leaves and 47 for the other nodes. [Black 1980] details a scheme for adding redundancy to B-trees, for error detection and correction.

The paragraph descriptor was evolved in detail as the program was developed. It finally came to contain: the text-pointer (two half-word integers, the sector number and the word within the sector, but both could fit in one half-word if necessary); the paragraph's usual typestyle; the body of the first and last lines, and a bit indicating whether the body ever changes during the paragraph; whether the paragraph is a trap and, if so, of what kind; and whether it is part of a <u>block</u> of paragraphs (such as a multi-paragraph trap). Additional formatting information (see below) is stored elsewhere because its size varies.

8(b) The Text

For most operations on the text of a paragraph to have reasonable speed, it must be in internal memory in some kind of linked structure. However, copying a linked structure to an external file, and later recopying it to memory, will cause the address spaces to conflict, unless the addresses are given in some relative form. But this would force memory for the paragraph to be allocated in largish pieces, rather than sparingly, for relative addresses must operate within a known range. Rather than accept this nuisance (additionally inconvenient in B because the B library's dynamic memory allocator is very well suited to taking only what is needed), the structure was transformed during the transfer to and from the galley file.

Anyway, unless editing is allowed directly on the galley file, text on the file needs no linked structure, and a sequential one will use less storage. The most compact form retaining ASCII characters would use 4 characters per machine word, separating words of text with some special character. This is not convenient using B because the paragraph must be divided back into words of text when it is read, and B handles character strings best when they begin on machine word boundaries. Also, certain formatting information is associated with each word of text: its output width (WWIDTH), the width of blank space to leave before it (WSPACE), and, if it starts a new line, the body size to use (WDEPTH). As discussed below, these items are closely associated with the word when the paragraph is in memory; it is both simple and convenient to place them, as integers, immediately before it, but only if the data for each word of text start on a machine word boundary.

The implementation therefore represents the paragraph text as a sequentially stored list. First, normally, come two half-word integers, WWIDTH and WSPACE; if the word of text begins a new output line after the first one in the paragraph, those integers are preceded by WDEPTH as a full-word integer. Next comes the word of text as a normal character string, terminated (as always in B) by an ASCII NUL (all 0-bits), then garbage characters if necessary to fill out the machine word (they might be NULs, so a 0 word could occur here). The data for the next word of text follow immediately after; a 0 machine word there indicates the end of the paragraph. (The normal case, a new line, and end-of-paragraph can be discriminated by the number of 0 half-words from the word boundary following the previous word of text, for WWIDTH and WDEPTH will always be positive and the latter will never be large enough to use its upper half-word.) This scheme is quite economical of storage, and can be transformed into the internal-memory structure quite easily in B.

Some commands must be retained for Pass 2 or for editing, but cannot be opcodes because they can occur in mid-paragraph: typestyle changes and indents, for instance. These are coded as <u>markers</u>, which are one or two characters, the first chosen from the range octal 740 to 777, so as to be readily distinguished from real characters. Where there is a second character it is taken as an integer argument. Any word of text containing a marker also begins with one, for ease of identification. It was intended to use markers pointing to opcodes when such things as footnotes were implemented.

Besides the format information detailed above, a paragraph in EZ27 has stacks of left and right temporary indentations; these need variable amounts of storage, which forces the information out of the paragraph descriptor where it logically belongs. It is in fact stored with the paragraph's text. Each stack is of paired integers, the amount of each indentation (SDATA) and when to end it (SCTRL, which is never 0); the bottom entry has SDATA the permanent margin position and SCTRL "infinity" (34,359,738,367). On the galley file, each stack becomes a sequential list of half-word integers; the last SCTRL is changed to 0, indicating the end of the list. These two lists and the paragraph's text are simply concatenated to form one sequential structure.

8(c) Filespace Allocation

Transforming a paragraph into a sequential structure still does not allow it to be written directly to the galley file, for space must first be allocated on that random-access file. The data are simply split off into 63-word portions, each written to one sector of the file. The remaining word of each sector, in the text-pointer format, links to the next portion. It is both simpler and more efficient to do portioning without regard to words of text; when read into memory, the portions are logically concatenated. (There is a minor problem on deallocating storage using this layout: the only way to tell the length of the last portion is to read the whole paragraph and search for the ends of the two margin stacks, then for the end of the text, which may not be the first 0 machine word. Replacing the 0 link of the last entry by the negative of its length would avoid this, but of course would complicate the structure.)

It is clear that the file storage must be allocated dynamically. Paragraphs in general will not require an exact multiple of 63 words of storage; dividing them into such portions therefore creates for almost every paragraph a shorter "remainder" portion. Naturally, several such remainders are best stored on a file sector. Thus the sectors of the galley file fall into three varieties: containing one 63-word portion, or several remainder portions, or one B-tree node. Now, since two of the three are allocated as entire sectors, there is reason to treat this case specially; as implemented, there is one free-list for sectors and another for partial sectors. Each list is maintained in the free space itself.

The list of free sectors is worked as a simple stack of sector numbers. The free-list of partial sectors, however, is kept sorted, and its entries consist of 2 words (allocation is by double words, chiefly so that this simple format is possible): a pointer (in the text-pointer format) to the next free area, and the length in words of the current one. Since the free areas can vary in length, serious fragmentation could occur when the text is repeatedly edited. This problem is alleviated [Knuth 1973-F, section 2.5] by using first-fit allocation rather than best-fit (under which very small leftover areas proliferate), and by coalescing adjacent free areas (the list is kept sorted to facilitate this). When they coalesce to an entire sector, it is transferred to the other list; conversely, a sector is transferred from the other list when an allocation cannot be satisfied from this one.

Storing free-lists in the free areas is natural, but when the storage is on disk, each entry examined in the list necessitates a disk read. For the list of partial sectors, there is really no other logical place to store it, and first-fit allocation does relieve the situation somewhat. (If it proved serious in practice, areas could be allocated in multiples of, say, 8 words to further reduce fragmentation at the expense of some filespace.)

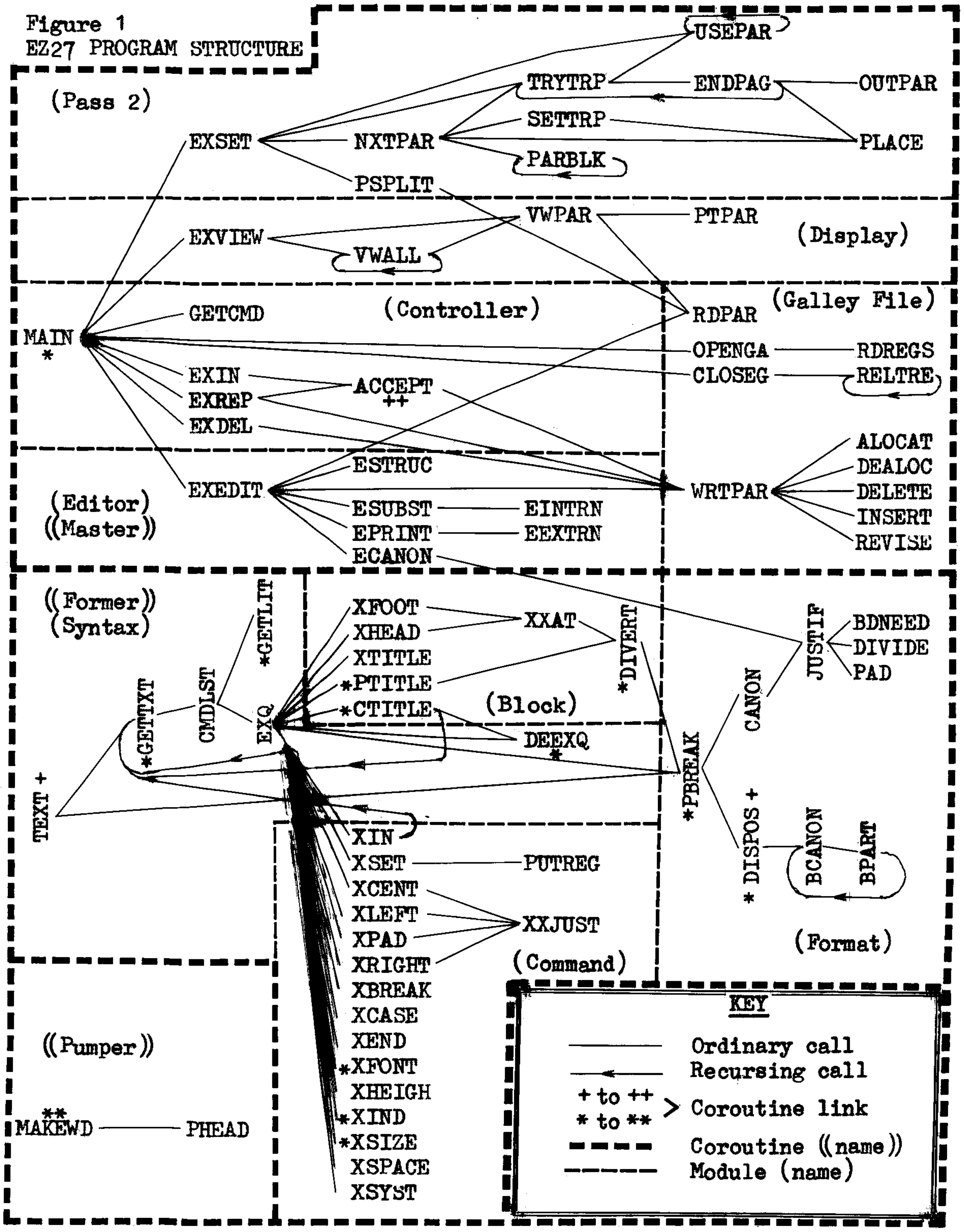
For the free-sector list, operating it as a stack minimizes the disk reads, but increases the likelihood that a file will come to contain a number of free sectors in the middle. Ultimately some kind of file compaction would probably be implemented, and this would be easier with a free-sector list in a different form: one or more machine words in a fixed location (or sequence of locations, if the file grew large) could be used as a bit vector, each bit corresponding to a particular sector of the file and being 1, say, if it was available.

8(d) Macros

As discussed below, the interpretation of macros in EZ27 requires that their values be saved on the galley file. Now, in EZ27 macros taking arguments are defined with named parameters, as in "/*set pw base exp "/*exp/th power of /*base/"/"; while the parameter references "/*exp/" and "/*base/" could be replaced in storage by markers, this is not actually done, and so the local names must be stored along with the macro name and contents. Each of these character strings is stored in the same fashion as the words of a paragraph, beginning on a machine word boundary and ended by a NUL.

Galley file storage for macros is allocated by entire sectors. As with portions of a paragraph, 63 words are used for data and the other one for a link; the logical concatenation of all the 63-word portions and the final shorter one gives the logical concatenation of all the macros stored. Each macro is stored simply as its name, followed by each parameter name, followed by a null string, followed by the macro contents. This scheme is quite workable and efficient because macros can only be assigned once, permanently: in effect the galley file acts as a log of *set commands.

Figure 1
EZ27 PROGRAM STRUCTURE



9. PROGRAM STRUCTURE

The EZ27 program, as finally implemented, is here described in terms of 10 "modules", the term being used loosely to refer to any set of related functions (B jargon for subprograms). This section should be read with reference to Figure 1, on the preceding page.

Only the more important functions are included here. Several utility functions are called by many others for simple tasks; some functions are trivial. The complete, commented source has been archived on TSS (archive tape 55, file number 0364); it uses 167 llinks (blocks) of filespace, contains 4526 lines, and is 41% comments.

As mentioned above, EZ27 includes Pass 1, which produces formatted paragraphs; Pass 2, which produces pages; a paragraph Editor; and an overall Controller. These form a simple hierarchy; other modules, performing utility functions, may contain components called from several places.

9(a) Controller Module

This module and the next four described comprise Coroutine Master.

The mainline (unique entry point) of the Control Module, of Coroutine Master, and of EZ27 itself, is MAIN. First it initializes Coroutine Pumper and some other things. Afterwards, it governs the opening and closing of galley files on command and, when a galley file is open, the execution of other commands read from the terminal by GETCMD. Functions EXDEL, EXEDIT, EXIN, EXREP, EXSET, and EXVIEW respectively execute the corresponding commands; some are mainlines of their own modules.

Pass 1 can be invoked from EXIN or EXREP by way of ACCEPT, which interfaces with Coroutine Former. The latter returns (pointers to) formatted paragraphs, and ACCEPT assigns them numbers and writes them on the galley file. The coroutines must pass status data cooperatively to prevent numbering collisions, since Former may generate more than one paragraph at a time.

9(b) Editor Module

EXEDIT is the Editor's mainline. After the selected paragraph is read, ESTRUC converts its format in memory to the special one (described below) intended to facilitate editing. Editing subcommands are then read from the terminal and executed. Finally, ECANON converts the paragraph back to the normal memory format and calls JUSTIF (of the Format Module) on it before it is rewritten.

ESUBST parses and executes the S subcommand; the actual substitution is done by one of a series of functions according to the number of words involved. EPRINT executes the P subcommand. Since markers are editable the same as other characters, EEXTRN translates them into visible form for EPRINT, while EINTRN makes the inverse translation for ESUBST.

9(c) Pass 2 Module

This module assembles pages by scanning the galley file sequentially; it handles traps and blocks of paragraphs, and splits paragraphs between pages. For appearance, when the body size decreases at the end of a paragraph (including after a trap), the larger size is maintained for one extra line; that is handled by EXSET.

EXSET is the mainline of the module. After some initializations, it gets a paragraph from NXTPAR, places it for output with USEPAR, and repeats indefinitely. When a paragraph has to be split (around a trap or between pages), PSPLIT does this, and USEPAR is called for each part.

What NXTPAR actually does is to read entries in sequence from the galley file B-tree until it finds an ordinary paragraph descriptor, and then to return that. (A block opcode is treated as an ordinary paragraph, after PARBLK is called to read the paragraphs it contains and attach them as a list. All the functions in this module can handle such a block, some by recursion.) Everything else NXTPAR may read, i.e., a trap or any other opcode, it acts upon; in particular, when it encounters a trap, it immediately calls SETTRP, which puts it in the appropriate list.

TRYTRP is called whenever the position on the page is advanced; it tests whether any traps should be tripped, and if so, does everything required. At the end of the page, it calls ENDPAG, which passes each paragraph (or part-paragraph) of the completed page to OUTPAR (a stub, intended to interface with the output device), then throws away any used one-time traps. PLACE maintains all the variables and lists associated with traps.

9(d) Display Module

EXVIEW is the mainline of the module. VWALL traverses the galley file B-tree, using recursion, for each paragraph calling VWPAR, which calls PTPAR, which displays the text in a form resembling the final output.

9(e) Galley File Module

OPENGA opens a galley file, calling RDREGS to bring into memory any macros stored on it, or initializing it if necessary; CLOSEG closes the file, calling RELTRE (which recurses for tree traversal) to forget the macros. SEARCH (which is omitted from Figure 1 because it is called from many places) reads a paragraph descriptor given the paragraph number; RDPAR, given the descriptor, reads the text of the paragraph, putting it in memory in the proper structure.

WRTPAR is given a paragraph number and perhaps the text of a paragraph; it deletes from the galley file the existing paragraph with that number, if any, then writes out the new one, if any. The two portions respectively use DEALOC and ALOCAT for the dynamically allocated storage of the text; one of DELETE, REVISE, and INSERT then updates the B-tree.

9(f) Syntax Module

This module and the next three described comprise Coroutine Former, which together with Coroutine Pumper forms Pass 1.

The mainline of Pass 1 and of Coroutine Former is function TEXT, which serves mainly to isolate Former's i/o from Master's.

The Syntax Module's work is done through GETTXT, which is called by TEXT and also, in indirect recursion, by several other Pass 1 functions. Each call of GETTXT corresponds to a level of input, either as a file or macro insertion, or as a text argument tc some command. GETTXT reads general text, passing each ordinary character on to

Coroutine Pumper (replacing space characters with ASCII NULs), and calling CMDLST when it encounters "/*". GETLIT is an otherwise similar function that does not recognize "/*".

CMDLST parses a formatting command, or command list, and governs its execution. Each command (or group of transformational commands) and its arguments is translated into an internal structure and passed to EXQ. (For text arguments, which may of course be indefinitely long, a flag is passed indicating that one occurred.)

EXQ looks up each command name. If a macro, it is opened and GETTXT is called; if a command, the corresponding function from the Command or Block Module is called, then, if there was a text argument, GETTXT reads the argument, and DEEXQ cancels the command's effects (as explained below).

The division of EZ27 into three coroutines allows this indirect, data-driven recursion of GETTXT (greatly facilitating the control of the program) in Coroutine Former to be quite uncoupled both from the linear text structure that is produced after the commands are executed (as in Pumper), and from the linear list of paragraphs finally written to the galley file (in Master).

9(g) Command Module

These functions mostly correspond to the various formatting commands. Some (such as XIND) produce markers, passed to Coroutine Pumper; some (such as XSPACE) produce opcodes; some (such as XCASE) only set external variables. XIN opens a file and calls GETTXT; XEND causes a B _reset_ (i.e., it truncates the calling stack, as though an arbitrary number of functions suddenly returned, until the desired, previously marked, function becomes the current one) to the most recent active call of XIN, or TEXT if none, thus simulating end-of-file. XSET assigns a value to a macro and also calls PUTREG to write it to the galley file. XXJUST handles four commands with similar effects.

If a command causes a break, this is indicated in an external variable, so the Format Module can be called later. When a general text argument occurred, the Command Module function returns the data necessary for DEEXQ to later undo the command.

9(h) Block Module

The *foot and *head commands can create blocks of paragraphs, and *title always does. They all also create diversions, where the formatting environment and the paragraph currently being formed in Coroutine Pumper have to be stacked until the active command's text argument(s) is/are read.

XFOOT and XHEAD have such similar effects that they both merely call XXAT. After calling DIVERT, which stacks the environment and flags that a block is being formed, XXAT starts the block off with an opcode. The text argument is then processed, each paragraph being appended to the block. Afterwards, DEEXQ restores the environment and PBREAK eventually handles the block.

The *title command is more complicated since, uniquely, it can take up to three text arguments. XTITLE only verifies the command and sets external variables for later use. EXQ sees these after the entire command group has been read (*title might occur in the middle of a transformational group), and calls PTITLE, which is somewhat like XXAT.

After the first text argument, EXQ calls CTITLE, which restores the environment as it was on entry to PTITLE before reading each of the other text arguments, and finally updates the title block opcode with cumulative data.

9(i) Format Module

When a command has caused a break, or when input is finished, or when a diversion is started, PBREAK is called. This snips off the list of words that Coroutine Pumper has been making and, unless a block is being built (as for a diversion), passes the paragraph (and any others accumulated, as from a block just ended, or opcodes) to CANON, then on to DISPOS.

CANON adjusts the structure of each paragraph in a block from the way MAKEWD creates it to the normal format, then calls JUSTIF on the paragraph. JUSTIF, which the Editor Module also uses, calls DIVIDE to partition the paragraph into output lines, BDNEED to position them vertically without overlapping characters, and PAD for pad justification if appropriate (handling other modes itself).

DISPOS interfaces with function ACCEPT of the Controller Module of Coroutine Master, as described above. It also calls BCANON to fill in cumulative information in block opcodes (blocks within blocks being handled by indirect recursion), in a format like paragraph descriptors, enabling Pass 2 to handle them easily. If Coroutine Master informs DISPOS that a collision is inevitable, so the paragraph cannot be written, DISPOS recovers gracefully and does a reset to TEXT.

9(j) Coroutine Pumper

The mainline of this coroutine is MAKEWD, which is passed all the plain text characters found by GETTXT and GETLIT, with NULs for space characters. MAKEWD simply appends continuously to a linked list of words of text, whose ends are stored in external variables so Coroutine Former can snip it off when it wants to. MAKEWD calls PHEAD to make a paragraph header when Former signals it to (by another external).

Markers are also passed to MAKEWD, and handled almost as ordinary characters, though another external warns MAKEWD of their status. Many of the links to MAKEWD diagrammed in Figure 1 are actually indirect through trivial functions, for purposes such as forcing a word to end before starting a diversion.

10. DATA STRUCTURES IN EZ27

The EZ27 program uses linked data structures, dynamically allocated in internal memory, with abandon, basically wherever the amount of data changes dynamically. Most of them, as is common with B, are made up of "vector" entries: sets of (machine) words occupying consecutive memory locations. Each word, usually, forms a "field" of the vector and is referenced by its offset from the start of the vector; a symbolic name (in block capitals) is generally assigned to the offset, and used in this description as the field's name.

The lists may be operated as queues or stacks or by whatever other discipline is appropriate in each case [Knuth 1973-F, chapter 2], but almost always the "0th word" (offset 0) is reserved for the link field, the pointer to the next item of the same kind; binary-tree or doubly-linked structures have their second link field in the following word. Both in memory and on the galley file, the value 0 is always used for a null link (no next item).

10(a) Paragraphs

A paragraph is stored as a "header" and a list of words of text. The header contains: a pointer HPARA to the list of words; part of the formatting environment, namely typestyle HFONT, HSIZE, and HBODY, fill and justification modes HFILL and HJUST, margin information HLMAR and HRMAR, and trap and block control information HTRAP, HWHERE, and HBLOCK; other information for Pass 2, namely the first and last body used in the paragraph HFBODY and HLBODY, whether the body ever changes HMBODY, and the vertical size HVSPAN;

and fields used as work areas by the Editor, HEBASE, and by Pass 2, HNUM, HPLACE, and HBACKUP (which actually share locations with fields not used by Pass 2). Most of these contain integers, but each bit of HTRAP has its meaning, while HLMAR and HRMAR are pointers to the stacks of temporary margin settings mentioned above. The headers correspond more or less to paragraph descriptors on the galley file.

In the normal format the list entry for a word of text contains: WWORD, containing the word itself (in B jargon, that means a pointer to the beginning of the character string); WWIDTH, its output width; WSEP, the width of space to leave before it; and WDEPTH, the body to use if this word starts a new output line, otherwise 0. (Originally the structure was more awkward: there was no WDEPTH, and WSEP and WWIDTH were packed in one word, which was made negative at the start of a new line, and the WWCRD field was of variable length and contained the actual characters of the word of text.)

Thus a paragraph is represented as:

Header: (HFONT, HSIZE, HBODY, HFILL, HJUST, HTRAP, HWHERE, HFBODY, HLBODY, HMBODY, HVSPAN, HEBASE, HNUM, HPLACE, HBACKUP; HLMAR: Stack of: (SDATA, SCTRL); HRMAR: Stack of: (SDATA, SCTRL); HPARA: Linear List of: (WWORD, WWIDTH, WSEP, WDEPTH)).

□ □ □

For reasons which no longer seem valid, there are two other variants of this structure. When the linear list is formed in Coroutine Pumper, there are four more fields: WLENGW, the amount of memory allocated for the string (MAKEWD allocates by chunks, not being clairvoyant); and

WFONT, WSIZE, and WBODY, the typestyle of the first character in the word (CANON uses these in converting to the normal format, to compute the paragraph's "usual" typestyle for its header; the meaning of typestyle change markers is also changed in CANON, so that they expire at the end of each word, which avoids pitfalls in editing.)

The other variant is used by the Editor. To reduce the number of (slow) string comparisons when doing a substitution, a special structure was conceived with a binary tree having one entry for each different word in the paragraph, and occurrences of the same word in the main list of words being linked together directly; the main list would also be doubly-linked to simplify deletions.

This violated the maxim "Keep it simple to make it faster" [Kernighan 1974, chapter 6]. The structure, once built, would indeed speed certain substitutions, but more perverse cases, like "S*/the/the best of the rest of the/", would involve complex updating: the difficulty of programming for these cases would offset any saving.

Instead, the double linking and the binary tree were retained, and the same-word-occurrence pointers dropped. In the editing for of the structure, the list of words lacks fields WWORD and WWIDTH, having instead a back pointer (WWORD=1, so it is in the 1st word) and WENCDE, which points to the corresponding entry in the binary tree. String comparisons are avoided by comparing WENCDE fields instead. The binary tree nodes have fields WWORD and WWIDTH, and EUSES, the number of occurrences of the word; their links fields are ELNEXT and ERNEXT (not the 0th and 1st words because WWORD=1). HEBASE in the header points to the tree root. The tree is initialized with the 6 commonest words so that it will be fairly well balanced.

Thus this variant is:
Header: (same as above, except ...; HEBASE: Binary Tree of: (WWORD, WWIDTH, EUSES); HPARA: Doubly-Linked Linear List of: (WSEP, WDEPTH; WENODE: entry in HEBASE's Binary Tree)).

□ □ □

It now appears that three forms of the same data structure are two too many. If the development of EZ27 was continuing, this diversity would be eliminated and only the normal form kept; it can be made to do what the others do, with no more fuss than the conversions now involve. (For instance, if string comparisons really cause a significant delay in editing, a check of the first character or two would rapidly eliminate most non-matches.) The differing structure of the text on the galley file, on the other hand, appears legitimately justified by the reasons given above.

10(b) Opcodes and Blocks

On the galley file, an opcode fits in the place of a paragraph descriptor, being 4 words long. The first word, always negative, indicates an opcode, and which opcode it is, and thus the format of the other words (which varies from opcode to opcode). All blocks of paragraphs, such as three-part titles, are preceded by opcodes containing cumulative data, resembling an ordinary paragraph descriptor in format; in Pass 2 these opcodes are in fact represented in memory like ordinary paragraph headers.

In Pass 1, all opcodes are represented in memory as on the file, but with one word inserted at the beginning of the vector, for linking the opcodes and paragraphs.

10(c) Galley File Buffers

Galley file i/o is by 64-word sectors. However, 65-word buffers are used, the extra word BWHENCE being set on reading to the sector address read from. This would facilitate the introduction of virtual i/o.

This is simply:
Buffer: (data: 64 words; BWHENCE).

□ □ □

For applications involving the B-tree, a doubly-linked list of buffers is maintained (linked through 2 extra words BNEXT and BPREV), their contents usually corresponding to a search path from root to leaf.

The fields of a B-tree node are called LINK x, KEY x, and DESCR x, where x is a number from 1 (0 for LINK) to the B-tree order. The DESCR fields are 4 words long.

This is:
Doubly-Linked Linear List of: Buffer: (LINK 0, LINK 1, ..., LINK ORDER; KEY 1, KEY 2, ..., KEY ORDER; DESCR 1, DESCR 2, ..., DESCR ORDER: 4 words each; BWHENCE).

10(d) Input Stacking and Macros

The B library provides a convenient implementation for source stacking, but the program has to keep track of the stack's state, for error messages, so it operates another stack in parallel. Its fields are: INAME, the name of the file or macro in use; ILINE, the line currently being read; ITYPE, a code indicating whether the source is file (or terminal), macro, or macro argument; and IARGCNT, which indicates how many arguments it takes if it is a macro.

This is:
Stack of: (INAME, ILINE, ITYPE, IARGCNT).

□ □ □

All macros defined in the currently open galley file are retained in a binary tree in memory. Its fields are: RNAME, the macro name; RCONTENT, the macro text, with parameters indicated as (argumentless) macros; and RARGLIST, a pointer to a list of parameter names.

That is:
Binary Tree of: (RNAME, RCONTENT, RARGLIST: Linear List of: (name)).

10(e) Command Lists

The list of commands that CMDLST passes to EXQ consists of entries with fields COMMAND, the command name, and CARGLIST; the latter points to a list with fields CARG, the argument itself, and CATYPE, a code filled in when the argument is classified.

This is:
Linear List of: Command: (COMMAND; CARGLIST: Linear List of: (CARG, CATYPE)).

10(f) Environment Stacking

When a transformational command is executed with a text argument, it pushes data onto a stack. The format is the same as the temporary indent stacks; here, SCTRL contains the actual address of the (external) variable affected, and SDATA its old value to save. The functions doing this, such as XCASE and XFONT, are passed as an argument the old stack top, and return the new one, so the stack can be local to EXQ. (Finally it is given to DEEXQ, which then undoes the commands' effects.)

At a diversion, as with XFOOT or PTITLE, the whole environment is saved. A stack frame is pushed with SCTRL 0 and SDATA pointing to a vector of values in a specific order. A constant vector Elist contains the respective actual addresses of the variables saved.

This is:

Stack of: (SCTRL; SDATA: value or vector of values).

10(g) Pass 2 Structures

As the output is assembled into pages, it takes the form of a list of paragraphs, or of part-paragraphs when PSPLIT was used. The list entries contain: QNUM, the paragraph's number (scaled as on the galley file); QPOS, the vertical position on the page of its beginning; QBACKUP, which, if nonzero, means that this paragraph goes on the same page beside, or higher than, the previous one, as for a three-part title; and QAFTER and QUNTIL, which, if nonzero, tell where to start and stop printing, as measured from the start of the paragraph (i.e. either one produces a part-paragraph, and QAFTER is the previous part's QUNTIL).

This is:

Linear List of: Part-Paragraph: (QNUM, QPOS, QBACKUP, QAFTER, QUNTIL).

□ □ □

Pass 2 also maintains two doubly-linked circular lists (with heads), containing the traps currently in effect. The two lists correspond to head and foot traps. The entries in each list are basically in the format of paragraph headers, whether the trap is a single paragraph or a block. Besides the back link, they use the field HPLACE, which PLACE assigns to the position on the current page where the trap will go. A number of external variables are maintained in conjunction with the trap lists.

11. HISTORY OF IMPLEMENTATION

11(a) Work Completed

Written and debugged first of all were the functions to manipulate the galley file B-tree. A test mainline called various functions individually, to insert, delete, renumber, and read back in sequence (dummy) paragraph descriptors on (interactive) request. The Controller was then partially implemented, so that the proper interactive commands could be used to drive the various functions of the program as they were written.

This work was then set aside, and the first version of Pass 1 was written. This accepted a paragraph from the terminal, translating it into the proper internal form, then formatted it with pad justification and printed the result on the terminal. The command indicator "/*" was recognized and the function to parse commands was called; general text arguments were recognized syntactically, but not acted upon. The first commands implemented were *size (type size only) and *font; a restricted form of *ind (permanent indents only) followed. Arguments were not verified for legitimacy.

The *cent, *left, *pad, and *right commands were then implemented; since these cause breaks, the program was altered to accept any amount of text from the terminal and then print the entire sequence of formatted paragraphs.

The implementation was revised at this point so that each command was executed from its own corresponding function, which first verified arguments for validity. The *break, *in, *end, and *case commands were added, then *syst, but acting in Pass 1. General text arguments, with their implied stacking of part of the environment, were then implemented.

Coroutines Former and Pumper were then created, so that the Former could follow the data naturally with recursion; this enabled important control information, which formerly had been in external variables or faked into the input, to be properly stored on the local-variable stack. The revisions to the data structure for a paragraph described above were also made at this time.

Next came the functions needed to write a paragraph to the galley file (with dynamic storage allocation) and read it back. This enabled the Controller to be linked (as Coroutine Master) to the formatting routines proper (Pass 1), invoking them on command. Afterwards, the functions to delete a paragraph from the file (freeing storage) were added.

To the existing interactive commands DEL, IN, REP, and VIEW, EDIT was added next, together with its complete set of subcommands. This entailed the variant data structure described above.

Vertical positioning of text was implemented next. This required the implementation of the body argument of *size, and the specification of the proper handling of body changes. At the same time, the B library's automatic release of dynamically allocated memory was adopted wherever possible.

The program was altered to allow blocks of paragraphs to be handled internally. Functions to save and restore the entire environment were then added. With these changes, traps (of formatted text) became possible, and the commands *foot and *head were implemented (intended as temporary substitutes for *at). The *heigh and *space commands, which produce opcodes, were implemented.

Next came the page make-up routines: Pass 2. The interactive SET command had been provided for when the Controller was written, but as a dummy (macros not being implemented yet); this name was pre-empted instead for the command to invoke Pass 2 (which had been conceived as a separate program). The necessary tables and output functions to use the Photon 532 Fontmaster typesetter were then added, and typeset output was actually produced for the only time (debugging being easier with monospacing).

The "To" argument of the *space command was implemented as another opcode. The Pass 2 functions were then rewritten in better style, and the *foot and *head commands extended to allow traps at any position on the page. The *title command was implemented, giving 3-part titles in their full generality, as blocks of paragraphs preceded by a new opcode. The opcodes were then revised so that one precedes every block, enabling blocks to be nested.

The last feature implemented was macros. First the *set command, to define a macro in memory, was installed; then the functions to write the macro text to the galley file (so it could be reused later) were added. However, no provision was made for reassigning a macro a new value (which is nontrivial as explained below).

11(b) Lesser Works Not Completed

These deficiencies of the last version of EZ27 from (or because of) the specification are listed roughly in order of decreasing importance.

The formatting environment is initialized when EZ27 is entered, and thereafter changed only through commands in input text. When a paragraph is inserted or replaced in an existing galley file, the environment should be set, when input begins, to that of, say, the preceding, or the deleted, paragraph: this might be quite different from the last text entered. For versatility, options for IN and REP, or new commands, should be added giving more control of the environment. Each galley file should perhaps carry a default environment.

EZ27 does not handle the Break key; using it during galley file writing can leave an invalid file. Such critical procedures should be protected against interruptions. During terminal input, the Break key should simply terminate that input, as provided in the specification. (Input can be terminated only by a *end or the B end-of-file signal "<nl><fs><nl>", "<fs>" representing an ASCII FS.)

Similarly, there is no recovery from overflow of the galley file; invalidity results. A module to reduce the file size by garbage collection, and perhaps to claim storage from the B-tree by increasing the average fullness of nodes, seems called for, but (as discussed above) the sector free-list arrangement would slow it.

No conditional input command was provided; this was an oversight, and would be reasonably simple in Pass 1.

Another oversight was the omission of any way to insert a macro inside a string argument; this makes it impossible to define a macro in terms of the present value of another one. The syntax specification would have to be revised to solve this problem.

No hyphenation facility was provided. This would be made relatively difficult by the organization of text as a sequence of words: the data structure as it stands makes no provision for words to be divided between lines. Also, the *hyph command as specified can define an additional control character, which is contrary to the spirit of EZ27's syntax; it would probably be better to support a hyphenation exception list rather than a discretionary hyphenation character.

Instead of the *at and *marg commands as specified, the implementation has the less convenient (intended as temporary) *foot and *head. Several keyword options of *at were never implemented. The option of *at taking a second text argument, so that the trap would be automatically dropped, was not implemented, nor was the *drop command. These improvements could be straightforward, done mainly with new opcodes, except for *drop and the "S" option of *at, which would require actual comparisons of the text of different traps. (Probably it would be better to permit a *at command, hence a trap, to carry a symbolic label, which reflects the way a *drop opcode would be implemented anyway.) Also, the *at options based on margin positions were not properly specified.

The Controller command EDIT, as implemented, may not be applied to any paragraph that is in a block; this would require reading the entire block's paragraph descriptors into memory, so the block opcode's cumulative information would be corrected. Another defect in the Editor is its restriction to replacing single, complete words only, and including adjacent non-space characters such as punctuation marks with each word, so that a simple substitution might have to be entered as "S/unconstitutionality!/unconstitutionality./"; this again is related to the organization of text as made up of words. Also, the Editor cannot combine paragraphs or make one into several. Still, since the Editor is a distinct module, it could be improved (or replaced) independently.

Likewise, no facilities were provided for moving or copying complete paragraphs, which would not be unduly difficult.

Multiple-column output would mainly be a matter of new opcodes.

Also, the originally specified Controller command SET was not implemented (though it is practically redundant with *set).

The *bal command would simply require extensions to the Format Module.

The trivial *fill and *nfill commands were not implemented.

The *float command, intended for conveniently building lists, as of references, as well as for printing figures, was not implemented. It would chiefly involve a new kind of environment change.

*Pad with the "A" option was not implemented, nor was *nline. These would cause no great difficulty, needing only additions to the Format Module and the definition of new markers. The *bsp, *hsp, *char, and *trans commands for special character effects were not implemented, for output was not (except in a test) interfaced to any typesetter; the first two would become markers and the others probably opcodes. Typesetting output is a minor change since the program was designed for it.

The *write and *close commands for work files were not implemented, nor ever properly specified for the TSS environment. (Does a written file remain in the AFT?) Also, the writing is constrained to occur in Pass 1, if the file is to be rereadable by *in, while it could be more useful in Pass 2 (*float gives a similar effect in Pass 1 anyway). But that would require opcodes to take arbitrarily long (string) arguments; so would *syst, if implemented as specified. A command *echo, analogous to *syst and *write but directing its output to the terminal, would be a handy addition to either pass.

The *join command would probably be implemented as a kind of block, since it effectively couples two (or more) paragraphs.

Superscripts, subscripts, and underlining would be practically trivial; the "X" option of *uline would become an opcode with a special format.

The *width command is redundant if (as implemented) the default width is the maximum, for *ind with the "R" option controls the same margin.

Allowing the control characters "/", "*", "#", and perhaps "\" to be reset is merely a matter of defining command syntax and semantics. (Do these effects disappear when the source stack is popped, as at the end of an inserted file?)

12. MAJOR PROBLEMS

Work on EZ27 was abandoned because its aims were seen to be impracticable, at least by the approach used.

12(a) Which Pass For Variables?

Autoscript and the CIA system successfully separated paragraph formatting and page formatting, producing a galley intermediately. Nroff, Script, Tex, and many other formatters successfully used reassignable macros and variables to produce a wide variety of automatic effects, such as the page numbering, section numbering, and table of contents of this thesis. EZ27 was to combine both effects, but they are really incompatible.

If variables are invoked in Pass 1, there is no way to handle page number references, for pages are not assigned until Pass 2. (As always in this section, automatic handling is tacitly assumed; naturally a user can always make up pages and fill in numbers manually, but this is usually not what one wants.) However, if macros are invoked in Pass 2, then large sections of general text may have to be parsed, as by Pass 1, while Pass 2 is executing, which ruins the separation of function.

The EZ27 specification attempted to avoid these problems by providing that macros be invoked in Pass 1 and variables in Pass 2. Since a variable must have a numeric value, the number of characters it will need on insertion cannot vary much, so justification in Pass 1 can proceed on a guess (deduced from the given format) and be adjusted as necessary in Pass 2. But a guess can fail badly. In a situation such as in index, there may be dozens of variable insertions close together; if the fit is poor, the number of lines in the paragraph may have to be changed, forcing extra work in Pass 2 and possibly even leading to a cascade of further changes elsewhere. A program to handle this properly would surely be most complex and accordingly large, even though most cases would cause no difficulty.

Furthermore, with variables in Pass 2, constructs such as "/*weekday **daynum/", using a macro that takes a variable as an argument, are rendered totally impossible, for the value of the variable must be available to evaluate the macro for insertion; the same applies to conditional text (of less than a paragraph), if the condition must depend on a variable.

In short, handling variables in Pass 1 prevents their use in applications dependent on vertical positioning or pagination, while handling them in Pass 2 can cause severe worst-case problems and complicate the program accordingly, and will lead to inflexibility since variable references may also be wanted in Pass 1. (The last problem could be alleviated by having two classes of variable, one for each pass, but this seems unpleasant, and anyway it does not affect the justification problem.)

12(b) Indexing

For similar reasons, a work file, as used for the table of contents in this thesis, cannot be produced in Pass 2, which is where it is needed: it would contain general text, like a macro, so possible inserted only by reinvoking Pass 1. The *index command could still be implemented, but it would have to generate an index file, whose entries would be formatted paragraphs, thus reducing flexibility.

12(c) Reassigned Macros

Even if macros are implemented in Pass 1, there is a problem: when one is reassigned, what happens to its old value? The specification requires that every value ever assigned to a macro be retained, associated with the paragraph number of the text where the *set command occurred. In this way, when text is later inserted by IN, the macro can be reused with the value that it would have had if the same text had been in the corresponding place in the initial input: this is analogous to the handling of environments discussed above. But this could guzzle storage (if a macro is reassigned several times), and might seem somewhat artificial; there are also questions as to the proper handling when the paragraph associated with a *set is deleted; yet any other method seems to obliterate data that may be wanted.

12(d) Generality

No method was implemented to copy part of a galley file, or partition it into several smaller ones; likewise, there is no way to insert text in galley file format, or text already in typesetter code (produced by another program) into an existing galley file. Without these facilities, the manipulation of large documents is unnecessarily difficult; but they could be taken care of with a little trouble.

(An interesting possibility would be to produce an output structure of a set of files that reflects the structure of insertions by *in in the input; thus a document of 3 chapters in different original input files, processed by

"/*in chap1 *in chap2 *in chap3/", might produce galley files "chap1g", "chap2g", and "chap3g", along with a main galley file containing 3 opcodes corresponding to the *in's but inserting galley files.)

More serious is the associated defect that the EZ27 program, in using its special galley file, must basically stand alone, and does not associate well with preprocessors and postprocessors in the manner of Nroff. This inhibits the addition of new features after the fashion of Eqn and Tbl, whose programming would be relatively complicated if done within EZ27. It also forces users to learn an editor which is not useful for anything but EZ27-formatted text.

12(e) Efficiency

"Size has no meaning," Yoda insisted.
"It matters not. ... Judge me by my _size_, do you?"
[Glut 1980, page 123]

EZ27 did achieve its objective of making Pass 2 run much faster than Pass 1, so that amending a document became relatively cheap. However, it did so at a large cost in internal memory, disk space, and disk i/o.

A test file was created by taking some 33,000 words of straight text (with a little use of macros) from this thesis. EZ27 was run on it and the processor time, file i/o count, and maximum program size measured. The same text was processed by Roff, which is the principal formatter on TSS, and on the TSS implementation of Troff [Troff 1980], which might provide a better comparison since Roff uses monospaced text. The processor time required respectively by EZ27, Troff, and Roff was in the ratio 2.7:4.1:1; the file i/o, 49.5:4.2:1; and the maximum program size, 1.75:1.75:1. The real time required was in the ratio 6.7:4.2:1, but this would vary more with the system load.

It seems clear that EZ27 was competitive in processor time, especially as other measurements indicate that over 70% of that time was spent on Pass 1, so Pass 2 alone is about as fast as Roff (the numbers make it a little faster, but the typesetter interface of Pass 2 was not completed, so the times were obtained by running a VIEW command and a SET command, which does more or less the same work).

But EZ27 had to do much more file i/o than Roff or Troff, which consumes real time. Pass 2 did about 1/3 of this i/o, but that alone is more than Troff or Roff. Still, this was expected, and the amount could be reduced by at least 40% by the better buffering mentioned earlier.

Where Roff and Troff took an input file of 30 llinks (blocks of 5 sectors, i.e., 1.25K characters each) and produced formatted output of 33 llinks, EZ27 produced a galley file of 72 llinks, or 2.2 times larger. And this, in a real application, would be kept on-line for the life of the document, supplemented when output was desired by a transfer file (produced by Pass 2), which would presumably be about 33 llinks. Thus even though EZ27 has a reasonably space-efficient layout for its galley file, sufficiently heavy use of it could significantly burden the file system.

The EZ27 program grew, on the test data, to the same size as Troff, which is 28K words, or quite a large program for TSS. (It started at 23K, but dynamic storage allocation caused the growth. Of the original 23K, 28% contains B library functions, and 14% is the Galley File Module. No other module is as large as 9% of the total.) In fact, 28K is large enough to cause a noticeable loss of real time while swapped out. But Troff is a complete formatter, and a highly sophisticated one at that, while EZ27 is not even finished. It has all the things listed above as "work not completed" to be added. Considering the difficulty of handling some of these items, it seems quite clear that the complete program would be too large to run, except by overlays; so it would, in Pass 1 at least, be very slow indeed.

13. CONCLUDING REMARKS

A two-pass text formatter using a galley file is a feasible approach (it is particularly natural, perhaps, with interactive page make-up, where the user must intervene at about the galley level anyway), but only if the level of automation is kept low, as in systems like Autoscript, avoiding features like variables and interactive editing. Therefore the galley file is not the right approach to a sophisticated incremental text formatter.

A more promising possibility may be the invisible command method. If the commands are embedded in the formatted document, it might be possible to design a formatter that could cheerfully handle formatted text as input, thus being idempotent, and yet would recognize what parts of the text would not change, and would proceed accordingly. This might not really gain any efficiency because there might be worst-case problems, similar to those with two passes and variables; and as with EZ27 there would be problems with retaining macros for later use. The method briefly suggested above, of producing a set of output files rather than a single one, might simplify editing of formatted text.

But processing is getting cheaper, and conventional formatters such as the Roff family are fast enough, even when documents must be reprocessed entire, that these programs are widely used. They gain much flexibility by running in one pass, and still more by readily meshing with preprocessors and with existing editors. Perhaps the conventional way is in fact the best way.

APPENDIX I: FEATURES OF FORMATTERS

This appendix contains, in the table on the following page, a direct comparison of the facilities provided by the 12 formatters covered in the survey section.

An entry of "0" indicates that the facility is not provided at all; "-" means that several commands may have to be used to produce the effect where another formatter uses just one. Positive numbers indicate that the facility is available directly; higher numbers indicate either greater versatility or greater convenience of use.

In this and the following appendix, topics are listed in the same order as in the survey section.

□ □ □

Each column in the table is aligned with the last letter of its heading.

FEATURE	Cyph't	Dip	Format	Nroff	Page-1	Proff	Quids	Roff	Runoff	Script	Tex	Type
Typeset (2=monosp. also)	2	2	0	2	1	1	0	0	0	0	1	2
Strings and macros . .	1	2	0	2	1	1	0	1	0	1	1	2
Variables	1	0	0	2	1	2	0	2	0	1	0	2
Formatted strings . .	0	0	0	2	0	0	0	0	0	0	1	2
Automatic hyphenation .	1	2	0	2	1	1	0	1	0	2	1	2
Auto. indent para. start	1	0	1	0	0	0	1	0	0	1	1	0
Three-part title lines .	0	0	0	2	0	0	0	1	0	1	2	0
Tables	1	0	1	1	1	1	1	1	0	2	2	0
Leadering	1	0	0	1	1	0	0	0	0	0	2	0
Multiple-column output .	0	0	1	–	0	1	0	0	0	2	–	0
Marginal notes . . .	0	0	0	0	0	0	2	2	0	1	–	0
Merge patterns . . .	0	0	0	0	0	0	0	1	0	2	0	0
Local motions	0	0	0	1	0	0	0	0	0	1	1	1
Mathematical formulae .	0	0	0	–	0	0	0	0	0	0	1	0
Explicit envir. switching	0	0	0	2	0	0	0	0	0	1	1	2
Vertical justification .	0	1	0	0	1	0	1	0	0	0	1	0
Headings and footings .	3	2	2	–	–	0	1	3	1	3	–	–
Footnotes	0	1	0	–	–	0	0	2	0	2	2	0
General traps	0	0	0	2	1	0	0	0	0	2	1	2
Automatic page numbers .	2	0	2	2	0	0	1	2	1	3	0	0
Automatic indexing . .	0	0	0	0	0	0	1	0	0	2	0	0
Need, widow elim., etc.	0	0	1	–	0	0	0	1	0	3	2	0
File insertion . . .	0	1	0	2	0	2	2	2	2	2	2	2
Terminal interaction .	0	0	0	2	0	0	3	1	0	2	3	2
Work and index files .	0	0	0	0	0	0	0	1	0	1	0	2
Commands to system . .	0	0	0	0	0	0	0	1	0	0	0	1
"Dictionary" production	0	0	1	0	0	0	0	0	0	0	0	0
Conditional processing .	2	2	0	2	2	2	0	1	0	3	2	2
Iterated processing . .	0	0	0	0	0	0	0	0	0	1	0	0
Output transliteration .	0	0	0	1	0	0	0	1	0	1	0	0

APPENDIX II: COMPENDIUM OF COMMANDS

This is a list of (almost) all the different command functions found in the 12 text formatters covered in the survey section. For brevity, they are given as imperatives, and inverses are not mentioned, though they almost always exist.

(i) Commands and Names

Command Control

Set control character(s) that indicate commands. Rename command; undefine command. Recognize commands in upper case also. Interrupt text processing.

Symbolic Names

Define macro; define macro locally; define macro taking text literally; terminate macro definition. Define string; insert referenced strings and define string. Define variable; insert and redefine variable by auto-incrementing. Undefine name; change name; test whether name is defined. Append to macro, or string. Set parameter character.

Insert macro, or string, or variable. Test number of arguments to macro; insert argument. Activate macro insertion; set insertion character. Insert strings in following text; insert 1 level of string references in following text. Set format for variable insertion, or for page numbers.

Specify character to be mapped into string.

(ii) Ordinary Horizontal Effects

Typestyle and Related Special Effects

Set font, or set font temporarily. Set size. Use ligatures. Produce subscript, or superscript. Artifically embolden by overstriking, or by overstriking with offset. Overstrike specified characters (producing a compound character).

Underline; underline continuously; specify what characters are affected by underlining. Underline and capitalize. Use another character instead of "_". Set vertical separation between text and line of underlines.

Filling, Justification, and Hyphenation

Fill. Break. Try not to break here. Don't break despite nofill mode. Break, and pad justify partial line. Set justification mode. Set limits on padding. When padding, don't expand leading blanks.

Hyphenate automatically, or with restrictions. Specify, or alter, hyphenation exception list. Set discretionary hyphenation character; treat "-" as that. Specify threshold of padding at which hyphenation should be considered. Don't hyphenate near end, or start, of a word.

Line Margins

Set line length. Shift all output to right, or on odd or even pages only. Indent left margin, or right, or both equally. Specify a list of pairs of margin settings; select one of them. Set length of three-part title line.

Indent left margin of next line; indent left margin of each line after next line ("hanging" indent) until a break.

Negatively indent left margin of next line. Select pair of margin settings (from list) to use as hanging indent, or to apply to next line. Indent first line of each paragraph; don't indent first line of this paragraph; apply hanging indent to each paragraph.

Terminate paragraph.

(iii) Special Horizontal Effects

Tables, and Other Columned Output

Set tab stops; set tab stops and associated justification modes or character strings. Set tab character; set field delimiter character; set padding indicator character. Set tab replacement character; set leader replacement character. Tab to next stop, or to specified stop. Advance to next tab stop, or to specified stop, filling with leadering. Tab to next stop, or to specified stop, with specified justification mode.

Produce three-part title line.

Activate multiple-column output. Set number of columns; set gutter size; set number of columns and their left margin positions; set width of each column.

Set gutter between text and marginal notes; set overall page width including marginal notes. Marginal note.

Merge Patterns, etc.

Activate merge patterns. Specify a merge pattern to apply on input, or output; specify a merge pattern and number of lines before it expires. Set character to print in right margin of each line. Set string to print in left margin of specified sections.

Repeat character, or box, to fill space as leadering.

Number output lines on each page, or continuously; number output lines including blank lines.

Miscellaneous

Local motion; alter size or alignment of box. Specify glob of glue. Reverse linefeed, or forward or reverse half linefeed.

Leave space for tall character.

Typeset math formula.

Don't fill and don't use ligatures and don't hyphenate and do left justify.

Change environment; push or pop environment stack.

(iv) Vertical Effects

Vertical Positioning

Set page height. Set body size; single space; double space. Specify where on the page a formfeed leaves the output device. Set vertical justification mode. Set conditions to omit blank lines, or pages.

Skip to new page, or odd page, or even page, or page of particular number; specify number of next page; skip to new column. Leave vertical space; leave vertical space unless at top of page; leave vertical space and indicate place for padding for vertical justification. Specify glob of (vertical) glue. Space downward to specified position. Try not to start new page here.

Page Margins, Traps, and Page Numbering

Set margin above headings; set margin including and below headings but above text; set total margin above text; set marginal region for headings; set marginal region below headings and above text. Likewise for footings.

Specify a line of a heading, or for odd or even pages only; specify it as a three-part title. Likewise for footing.

Specify a footnote (goes in text area). Specify a line of a footnote separator; specify it as a three-part title. Specify a headnote, or for odd or even pages only; print the headnote now.

Set a macro as a trap at specified location; trip trap indicated for another location. Set trap to be tripped at end of each page.

Set format of page numbers. Reset character indicating insertion of page number in headings and footings. Set strings to insert along with page number. Specify that page number have two parts, or that it begin to with the next odd page.

Miscellanous

Specify table-of-contents format. Make an entry in main text and/or table of contents. Print table of contents. Make entry in index. Print index.

Need. Automatically eliminate widows, skipping to new page, or only to next column, or around next trap. Specify glued text. Specify glued, or floating glued, text consisting of vertical space. Leave a blank line and need and temporarily indent left margin.

Append to block of floating text; terminate and print that block; print it immediately and not floating. Specify whether more than one floating block may exist at once.

Retain position on output page.

(v) Input/Output Effects

I/O and Insertion

Insert file. End current input file. End current input file and insert another.

Write "dictionary" of all words encountered. Suppress output for specified number of pages. Send message to terminal. Accept input from terminal; accept input from terminal into a string. Pause while text typed on terminal, but ignore it.

Pass command to system. Direct output to another program. Send output, or copy from input, to a work file.

Control Flow in Main Input

Define label; go to label. Repeat a line a number of times, or while a condition is true; abort repetition. Process or omit a line, or group of lines, according to condition. Use or omit text just processed, according to condition. Process or omit sections of input according to switch set by user; set the switch.

Terminate; terminate without flushing buffers.

Debugging

Abort if input line number not equal to specified number. Print in margin the number of each input line; print each command in margin. Underline on output first character of each input line. Trace macro expansions and command execution.

Transliteration and Literal Text

Specify general transliteration on output, or on input using escape character. Translate to upper, or lower, or opposite case; set case escape character. Produce extraordinary blank. Accept character in hexadecimal.

Take following text as literal plain text; pass text through formatter untouched.

APPENDIX III: SPECIMENS OF SYNTAX

This appendix simply presents the following passage as it might be entered for several formatters. (Some do not have boldface, and the commands may not be exactly correct in every case, but the flavor will be there. In the examples, commands have been started on new lines only when required, but of course a user may, and should, use newlines more freely.)

ACT 1, Scene 1

Elsinore. _A platform before the castle_.
Francisco on his post. _Enter to him Bernardo_.

Bernardo: Who's there?
Francisco: Nay, answer me: stand, and unfold yourself.
Bernardo: Long live the king!
Francisco: Bernardo?
Bernardo: He.

[Shakespeare 1604]

□ □ □

Cyphertext Style

```
/center; newtype bold/ ACT 1, Scene 1 /space 12; newtype
italic/ Elsinore: A platform before the castle.
/nextparagraph/ Francisco on his post. Enter to him
Bernardo. /space 12; set xber, "Bernardo:"; set xfra,
"Francisco:"; define xrol, "nextparagraph; newtype italic";
define xlin, "newtype roman"; xrol/ @xber /xlin/ Who's
there? /xrol/ @xfra /xlin/ Nay, answer me: stand, and
unfold yourself. /xrol/ @xber /xlin/ Long live the king!
/xrol/ @xfra /xlin/ Bernardo? /xrol/ @xber /xlin/ He.
```

<u>Dip Style</u>

```
* quad centre <* fount bold <ACT 1, Scene 1>>
* sink 12 points
* fount italic <Elsinore.
A platform before the castle.
* paragraph
Francisco on his post.
Enter to him Bernardo.>
* sink 12 points
* define xber <<* paragraph
* fount italic <Bernardo:>>>
* define xfra <<* paragraph
* fount italic <Francisco:>>>
* xber
Who's there?
* xfra
Nay, answer me:
stand, and unfold yourself.
* xber
Long live the king!
* xfra
Bernardo?
* xber
He.
```

<u>Format Style</u>

```
SPE
GO
)M ACT 1, Scene 1 )MPPU Elsinore.  A platform before the
castle.  )P Francisco on his post.  Enter to him Bernardo.
)PP Bernardo:  )U Who's there?  )PU Francisco:  )U Nay,
answer me:  stand, and unfold yourself.  )PU Bernardo:  )U
```

Long live the king!)PU Francisco:)U Bernardo?)PU Bernardo:)U He.

<u>Page-1 Style</u>

[ce;tf,3] ACT 1, Scene 1 [nl;ju;tf,2;dn,12] Elsinore. A platform before the castle. [nl] Francisco on his post. Enter to him Bernardo. [nl;dn,12;sy,x1,[[nl,tf,2] Bernardo: [tf,1]];sy,x2,[[nl,tf,2] Francisco: [tf,1]];x1] Who's there? [x2] Nay, answer me: stand, and unfold yourself. [x1] Long live the king! [x2] Bernardo? [x1] He.

<u>Tex Style</u>

{\:b \hfill ACT 1, Scene 1 \hfill \par} \vskip 12pt {\:i Elsinore. A platform before the castle. \par Francisco on his post. Enter to him Bernardo. \par} \vskip 12pt \def \xber {\par \:i Bernardo:} \def \xfra {\par \:i Francisco:} \xber Who's there? \xfra Nay, answer me: stand, and unfold yourself. \xber Long live the king! \xfra Bernardo? \xber He.

<u>Type Style</u>

{ev} {dn _ts 3} {dn _ju 1} ACT 1, Scene 1 {re}
{ev} {br} {sp} {dn _ts 2} Elsinore. A platform before the castle. {br} Francisco on his post. Enter to him Bernardo. {br} {sp} {re} {ds xber '{ev} {dn _ts 2} {br} Bernardo: {re}'} {ds xfra '{ev} {dn _ts 2} {br} Francisco: {re}'} {xber} Who's there? {xfra} Nay, answer me: stand, and unfold yourself. {xber} Long live the king! {xfra} Bernardo? {xber} He.

Roff Style (Nroff, Proff, Runoff, Script similar)

```
.ce
.bf
ACT 1, Scene 1
.sp
.ul 4
Elsinore.
A platform before the castle.
.br
Francisco on his post.
Enter to him Bernardo.
.sp
.at(xb)
.br
.ul
Bernardo:
.en(xb)
.at(xf)
.br
.ul
Francisco:
.en(xf)
.xb
Who's there?
.xf
Nay, answer me:
stand, and unfold yourself.
.xb
Long live the king!
.xf
Bernardo?
.xb
He.
```

□ □ □

EZ27 Style

```
/*cent *font 3 "ACT 1, Scene 1" *space *font 2 "Elsinore.
A platform before the castle.  /*break/ Francisco on his
post.  Enter to him Bernardo." *space *set xber "/*break
*font 2 "Bernardo:"/" *set xfra "/*break *font 2
"Francisco:"/" *xber/ Who's there?  /*xfra/ Nay, answer me:
stand, and unfold yourself.  /*xber/ Long live the king!
/*xfra/ Bernardo?  /*xber/ He.
```

BIBLIOGRAPHY

Some references are not cited in the text: [de Tollenaere 1977; Lesk 1977; Vasdi 1978] describe various experiences with computerized formatting; [Custom 1978; Friedlander 1968; News 1978; US 1977] relate the impact of computerization on newspaper and magazine publishing; [Andersson 1970] lists various kinds of equipment; [Martin 1974; Slater 1972; Terrant 1975; Terrant 1980] are documents of general scope covering many aspects of computers in publishing; and [UNIX 1979] contains several of the other references.

□ □ □

[APS 1977]

"APS Tests Computer System for Publishing Operations", Physics Today **30**, 12 (December 1977), page 75.

American Physical Society (member of AIP) is using UNIX with some success.

[Alt 1973]

Franz L Alt, Judith Yuni Kirk: "Computer Photocomposition of Technical Text", Communications of the ACM **16**, 6 (June 1973), pages 386-391 (7 refs).

[Andersson 1970]

P L Andersson: "Phototypesetting--A Quiet Revolution", Datamation **16**, 16 (December 1, 1970), pages 22-27.

Includes a list of 41 phototypesetters from 12 manufacturers.

[Asimov 1957]

Isaac Asimov: "Galley Slave", Galaxy Science Fiction **15**, 2 (December 1957), pages 8-41.

A short story. Reprinted in: The Rest of the Robots. Doubleday, Garden City, NY, 1964, pages 127-162. Republished: Panther Books, London, England, 1968, pages 177-223.

[Barry 1977]

Michael W Barry: "Computing in the Printing Industry", Australian Computer Journal **9**, 1 (March 1977), pages 39-41 (2 refs).

[Beach 1976]

Richard J Beach: Photon ROFF Text Formatter (PROFF). University of Waterloo Computer Science Research Report CS-76-08, Waterloo, Ont., April 1976.

[Beach 1977]

Richard J Beach: Computerized Typesetting of Technical Documents. University of Waterloo Computer Science Research Report CS-77-38, Waterloo, Ont., 1977.

Describes some experience with Proff.

[Beatty 1979]

John C Beatty, Janet S Chin, Henry F Moll: "An Interactive Documentation System", Computer Graphics **13**, 2: Proceedings, SIGGRAPH '79 (August 1979), pages 71-82 (18 refs).

Describes REDPP and related items.

[Bemer 1973]

Robert W Bemer, A Richard Shriver: "Integrating Computer Text Processing with Photocomposition", IEEE Transactions on Professional Communications **PC-16**, 3 (September 1973), pages 92-96 (2 refs).

Describes system used to produce Honeywell Computer Journal. Reprinted in: Honeywell Computer Journal **7**, 4 (December 1973), pages 261-267.

And (in slightly different form, by Bemer only) as: "The Role of a Computer in the Publication of a Primary Journal", AFIPS National Computer Conference **42** Part 2: Methods and Applications (1973), pages M16-M20 (1 ref).

[**Berenyi 1977**]

Ivan Berenyi: "Computing's Youngest Grandchild--An Accident", Data Processing **19**, 4 (April 1977), pages 40-43.

A brief history of computerized word processing.

[**Berns 1969**]

Gerald M Berns: "Description of FORMAT, A Text-Processing Program", Communications of the ACM **12**, 3 (March 1969), pages 141-146 (13 refs).

[**Black 1980**]

J P Black, D J Taylor, D E Morgan: A Reliable B-Tree Implementation. University of Waterloo Computer Science Research Report CS-80-15, Waterloo, Ont., March 1980.

[**Boehm 1976**]

Peter J Boehm: "Software and Hardware Considerations for a Technical Typesetting System", IEEE Transactions on Professional Communications **PC-19**, 1 (March 1976), pages 15-19.

Describes the system at Computype.

[Boissavy 1973]

Michel Boissavy, Raymond Lointier: "Définition d'un système conversationnel de mise en page", Metra **XII**, 4 (December 1973), pages 561-577.

Detailed article, in French, about page make-up with graphics terminals.

[Brader 1979]

Mark S Brader: Photon/532/Set: A Text Formatter. University of Waterloo Computer Science Research Report CS-79-33, Waterloo, Ont., 1979.

[Brooks 1975]

Frederick P Brooks Jr: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, Reading, MA, 1975.

About why large software projects rarely run smoothly.

[Buccino 1980]

Joseph H Buccino (appendix by Charles H Forsyth): A Reliable Typesetting System for Waterloo University of Waterloo Computer Science Research Report CS-80-20, Waterloo, Ont., April 1980.

Describes all the software and hardware related to the Mathematics Faculty Computing Facility's Photon 737 Econosetter.

[Card 1979]

Charles Card: "Standardizing Languages for Word and Text Processing", CIPS Review **3**, 6 (December 1979), pages 26-27.

[Chaundy 1957]

T W Chaundy, P R Barrett, Charles Bates: The Printing of Mathematics: aids for authors and editors and rules for compositors and readers at the University Press, Oxford (revised edition). Oxford University Press, London, 1957.

[Cole 1976]

A J Cole: Macro Processors. Cambridge University Press, Cambridge, England, 1976.

[Coulouris 1976]

G F Coulouris, I Durham, J R Hutchinson, M H Patel, T Reeves, D G Winderbank: "The Design and Implementation of an Interactive Document Editor", Software--Practice and Experience 6, 2 (April-June 1976), pages 271-279 (4 refs).

Describes the Quids editor-formatter system.

[Custom 1978]

"Custom System Suits Papers' Unique Needs", Computerworld 12, 13 (March 27, 1978), special report on data communications terminals, page S/27.

Describes success of system at Philadelphia Inquirer and Daily News.

[de Tollenaere 1977]

Felicien de Tollenaere: "Experiences with Computerized Photocomposition", Sprache und Datenverarbeitung 1, 2 (July 1977), pages 156-159 (6 refs).

The vicissitudes of a novice's encounter with phototypesetting.

[Edelson 1977]

D Edelson: "Computer Aided Chemical Documentation--A Text Processor for Chemical Equations", Computers and Chemistry 1, 4 (1977), page 265 (4 refs).

The text processor is Eqn.

[Friedlander 1968]

Gordon D Friedlander: "Automation Comes to the Printing and Publishing Industry", IEEE Spectrum 5, 4 (April 1968), pages 48-62 (2 refs).

[Gardner 1981]

J A Gardner: The FRED Text Editor Reference Manual. On-line documentation, University of Waterloo Mathematics Faculty Computing Facility TSS, Waterloo, Ont., January 13, 1981, edition.

[Glut 1980]

Donald F Glut: The Empire Strikes Back. Ballantine, New York, NY, 1980.

A novel based on a film.

[Hamblin 1977]

Dora Jane Hamblin: That Was The LIFE. W W Norton, New York, NY, and George J McLeod, Toronto, Ont., 1977.

About Life magazine; chapter 7 describes fitting type to layout.

[Honeywell 1972]

Honeywell Time-Sharing System Pocket Guide: Series 600/6000 Software. Honeywell Information Systems publication BS12, Waltham, MA, 1972.

Pages 31-34 describe the TSS editor's formatter.

[**Hutt 1967**]

G Allen Hutt: Newspaper design (second edition). Oxford University Press, London, 1967.

[**Justus 1972**]

Paul E Justus: "There is More to Typesetting than Setting Type", IEEE Transactions on Professional Communications PC-15, 1 (May 1972), pages 13-16.

Examples of bugbears of hyphenation and justification.

[**Kernighan 1974**]

Brian W Kernighan, P J Plauger: The Elements of Programming Style. McGraw-Hill, New York, NY, 1974.

A book of principles to help one program better.

[**Kernighan 1976-A**]

Brian W Kernighan, Lorinda L Cherry: "A System for Typesetting Mathematics", undated, in [UNIX 1979].

Describes Eqn. Minor revision of: Communications of the ACM 18, 3 (March 1975), pages 151-156 (7 refs).

[**Kernighan 1976-S**]

Brian W Kernighan, P J Plauger: Software Tools. Addison-Wesley, Reading, MA, 1976.

About broadly useful programs; chapter 7 describes a simple Roff-like formatter.

[**Kernighan 1978-A**]

Brian W Kernighan: "A TROFF Tutorial", August 4, 1978, in [UNIX 1979].

[**Kernighan 1978-D**]

B W Kernighan, M E Lesk, J F Cssanna Jr: "Document Preparation", Bell System Technical Journal 57, 6, (July-August 1978), pages 2115-2135 (20 refs).

On UNIX.

[Kernighan 1978-T]

Brian W Kernighan, Lorinda L Cherry: "Typesetting Mathematics--User's Guide" (second edition), August 15, 1978, in [UNIX 1979].

Manual for Eqn.

[Kimura 1978]

Izumi Kimura: "On Teaching the Art of Compromising in the Development of External Specifications", Journal of Information Processing 1, 1 (April 1978), pages 33-41 (12 refs)

Discusses the teaching of programming in the context of an example problem in automatic-recognition text formatting. A preliminary version (in Japanese) appeared in: Proceedings, Programming Symposium 18 (January 1977), pages 161-168.

[Knuth 1973-F]

Donald E Knuth: The Art of Computer Programming 1: Fundamental Algorithms (second edition). Addison-Wesley, Reading, MA, 1973.

Chapter 2 is all about list structures, dynamic storage allocation, and so on.

[Knuth 1973-S]

Donald E Knuth: The Art of Computer Programming 3: Sorting and Searching. Addison-Wesley, Reading, MA, 1973.

Section 6.2.4 concerns B-trees, among other things.

[Knuth 1978]

D E Knuth: Mathematical Typography. American Mathematical Society Josiah Willard Gibbs lecture, January 4, 1978; printed as Stanford University Computer Science Report STAN-CS-78-648, Palo Alto, CA, 1978.

Describes Tex and Metafont, and background philosophy and some math. Reprinted in: Bulletin (New Series) of the American Mathematical Society **1**, 2 (March 1979), pages 337-372 (51 refs).

[Knuth 1979]

D E Knuth: Tau Epsilon Chi, a System for Technical Text. American Mathematical Society, Providence, RI, 1979.

Revision of: Stanford University Computer Science Report STAN-CS-78-675, Palo Alto, CA, September 1978.

[Korbuly 1975]

Dorothy K Korbuly: "A New Approach to Coding Displayed Mathematics for Photocomposition", IEEE Transactions on Professional Communications **PC-18**, 3 (September 1975), pages 283-287 (2 refs).

System involves manual character counting.

[Kuney 1966]

J H Kuney, B G Lazorchak, S W Walcavich, D Sherman: "Computerized Typesetting of Complex Scientific Material", AFIPS Fall Joint Computer Conference **29** (1966), pages 149-156 (6 refs).

Description of ACS system.

[Kuney 1969]

J H Kuney: "Processing Manuscripts for Input to a Computerized Typesetting System for Scientific Journals", IEEE Transactions on Engineering Writing and Speech **EWS-12**, 2 (August 1969), pages 49-52 (3 refs).

Description of ACS system.

[**Kunzel 1966**]

George Z Kunzel: "A Computer-Assisted Page Composing System, Featuring Hyphenless Justification", AFIPS Fall Joint Computer Conference **29** (1966), pages 157-167 (2 refs).

Describes system at the CIA.

[**Landau 1971**]

Robert M Landau (editor): Proceedings of the ASIS Workshop on Computer Composition. American Society for Information Science, Washington, 1971.

Contains several articles listed separately in this bibliography, and various discussions, as well as a list of problem areas on pages 135-140.

[**Lesk 1977**]

M E Lesk, B W Kernighan: "Computer Typesetting of Technical Journals on UNIX", AFIPS National Computer Conference **46** (1977), pages 879-888 (11 refs).

Describes some experience using Troff, Tbl, and Eqn.

[**Lesk 1978**]

M E Lesk: "Typing Documents on the UNIX System: Using the -ms Macros with Troff and Nroff", November 13, 1978, in [UNIX 1979].

[**Lesk 1979**]

M E Lesk: "Tbl--A Program to Format Tables", January 16, 1979, in [UNIX 1979].

[**MacDonald 1978**]

Alan MacDonald: "Word Processors to Dominate as Business Communicators", Data Management **16**, 2 (February 1978), pages 18-22.

[Mack 1975]

Paul F Mack: "Lower Composition Costs through Optical Scanning and Photocomposition", IEEE Transactions on Professional Communications **PC-18**, 3 (September 1975), pages 279-282.

Description of Mack system.

[Makris 1966]

Constantine J Makris: "A Special Purpose Computer for High-Speed Page Composition", AFIPS Fall Joint Computer Conference **29** (1966), pages 137-148.

[Martin 1974]

J Sperling Martin: "Editorial Processing Centers: A Study to Determine Economic and Technical Feasibility: Annex Part II: A Review of Relevant Technology to the Publication of Scientific and Technical Journals", Information, Part 2, Reports, Bibliographies **3**, 6 (1974), pages 1-14 (37 refs).

[Mashey 1976]

J R Mashey, D W Smith: "Documentation Tools and Techniques", IEEE International Conference on Software Engineering **2** (1976), pages 177-181 (46 refs).

Describes text processing experience on PWB UNIX.

[McCreight 1977]

Edward W McCreight: "Pagination of B*-Trees with Variable-Length Records", Communications of the ACM **20**, 9 (September 1977), pages 670-674 (6 refs).

[Messina 1970]

Carla G Messina, Joseph Hilsenrath: Edit-Insertion Programs for Automatic Typesetting of Computer Printout. US Department of Commerce National Bureau of Standards Technical Note 500, US Government Printing Office, Washington, DC, April 1970.

[Mitchell 1977]

Robert T Mitchell: "Word Processing in the Office of Today and Tomorrow", _IEEE National Telecommunications Conference_ III (1977), pages 40:4-1 to 40:4-6 (3 refs).

A history of word processing, from early typewriters on.

[Moitra 1979]

Abha Moitra, S P Mudur, A W Narwekar: "Design and Analysis of a Hyphenation Procedure", _Software--Practice and Experience_ 9, 4 (April 1979), pages 325-337 (6 refs).

Describes the hyphenator in Dip.

[Mooers 1965]

C N Mooers, L P Deutsch: "TRAC, A Text Handling Language", _ACM National Conference_ 20 (1965), pages 229-246 (8 refs).

Describes the TRAC macro-processor.

[Moore 1970]

C G Moore, R P Mann: "CypherText: An Extensible Composing and Typesetting Language", _AFIPS Fall Joint Computer Conference_ 37 (1970), pages 555-561 (6 refs).

[Mudur 1979]

S P Mudur, A W Narwekar, Abha Moitra: "Design of Software for Text Composition", _Software--Practice and Experience_ 9, 4 (April 1979), pages 313-323 (25 refs).

Description of Dip.

[Muir 1972]

G Muir, K Preston: "Typesetting Programs", _in_ [Slater 1972] pages 29-42.

Article includes a follow-up discussion.

[News 1978]

"News Magazine's System Expedites Production", Computerworld **12**, 11 (March 13, 1978), page 6.

Similar to [US 1977], but a bit less detailed.

[Newspapers 1977]

"Newspapers Take a Big Step in Automation", Business Week 2490 (July 4, 1977), pages 58-60.

Describes interactive page make-up at the New York Daily News.

[Nudds 1977]

D Nudds: "The Design of the MAX Macroprocessor", The Computer Journal **20**, 1 (February 1977), pages 30-36 (8 refs).

[Ocker 1971]

Wolfgang A Ocker: "A Program to Hyphenate English Words", IEEE Transactions on Engineering Writing and Speech **EWS-14**, 2 (June 1971), pages 53-59 (7 refs).

Details hyphenator at RCA, omitting only some tables. Reprinted in:
IEEE Transactions on Professional Communications **PC-18**, 2 (June 1975), pages 78-84 (7 refs).

[Ossanna 1977]

Joseph F Ossanna: "NROFF/TROFF User's Manual", May 15, 1977, in [UNIX 1979].

[Perry 1966]

John H Perry Jr: "Integrated Automation in Newspaper and Book Production", AFIPS Fall Joint Computer Conference **29** (1966), pages 125-136.

Describes system at Perry Publications.

[Pierson 1971]

John L Pierson: "A Computer Program for Electronic Typesetting", IEEE Transactions on Engineering Writing and Speech **EWS-14**, 2 (June 1971), pages 46-52 (3 refs).

Describes Page-1.

[Pierson 1972]

John Pierson: Computer Composition Using PAGE-1. Wiley, Interscience, New York, 1972.

Manual for Page-1.

[QED 1980]

QED Text Editor Reference Manual. Cn-line documentation, University of Waterloo Mathematics Faculty Computing Facility TSS, Waterloo, Ont., September 18, 1980, edition.

[Reed 1977]

Lynda Reed: "Automated Text Processing Gains Ground", Telesis **5**, 3 (June 1977), pages 80-85 (1 ref).

Rationale behind the Ted text editor at Bell Northern Research.

[Reid 1980]

Brian K Reid: "A High-Level Approach to Computer Document Formatting", ACM Symposium on Principles of Programming Languages **7** (1980), pages 24-31 (13 refs).

Describes Scribe.

[Rich 1965]

R P Rich, A G Stone: "Method for Hyphenating at the End of a Printed Line", Communications of the ACM **8**, 7 (July 1965), pages 444-445.

A simple algorithm.

[Right 1978]

"Right to the Source... on WP", Data Management 16, 2 (February 1978), pages 28-29.

A list of 74 word processing equipment makers.

[Roff 1978]

Roff Tutorial Guide and Reference Manual. On-line documentation, University of Waterloo Mathematics Faculty Computing Facility TSS, Waterloo, Ont., January 1978 edition.

[Roistacher 1974]

Richard C Roistacher: "On-Line Computer Text Processing: A Tutorial", Behavior Research Methods and Instrumentation 6, 2 (March 1974), pages 159-166 (9 refs).

Describes how to choose a text processing system, for novices.

[Saltzer 1965]

J Saltzer: "A Right-Justifying Type Out Program", in The Compatible Time-Sharing System: A Programmer's Guide (P A Crisman, editor.) The Massachusetts Institute of Technology Press, Cambridge, MA, 1965, section AH.9.01, pages 9-13.

The manual for Runoff.

[Schneider 1974]

Ben Ross Schneider Jr: Travels in Computerland; or, Incompatibilities and Interfaces: A Full and True Account of the Implementation of the London Stage Information Bank. Addison-Wesley, Reading, MA, 1974.

In which a Professor of English encounters the World of Computers.

[Seybold 1971]

John W Seybold: "Software Interfaces and System Aspects", _in_ [Landau 1971] pages 87-98.

[Shakespeare 1604]

William Shakespeare (or Shakspere): _Hamlet_ (from second quarto and first folio editions). Longmans Canada, Toronto, Ont., 1961.

A tragedy.

[Shatzkin 1971]

Leonard Shatzkin: "Book Publishing Needs for Computer Photocomposition", _in_ [Landau 1971] pages 67-71.

[Slater 1972]

J F Slater (editor): _Computer-Aided Typesetting_: Proceedings of an International Conference held in London. Transcripta Books, London, 1972.

Includes an article listed separately in this bibliography, and others covering such things as equipment and keyboards. One appendix lists 36 models of typesetters from 12 manufacturers.

[Stuckey 1969]

R Dwight Stuckey: "AutoSCRIPT--An Automated Publications System", _IEEE Transactions on Engineering Writing and Speech_ **EWS-12**, 2 (August 1969), pages 29-33.

[Stuckey 1973]

R Dwight Stuckey: "AutoSCRIPT--A System for Computerized Document Preparation", _Western Electric Engineer_ **XVII**, 1 (January 1973), pages 38-41.

[Terrant 1975]

Seldon W Terrant: "The Computer and Publishing", ASIS Annual Review of Information Science and Technology (Carlos A Caudra, editor) **10** (1975), pages 273-301 (86 refs).

General summary of all aspects of the field, with emphasis on references.

[Terrant 1980]

Seldon W Terrant: "Computers in Publishing", ASIS Annual Review of Information Science and Technology (Martha E Williams, editor) **15** (1980), pages 191-219 (155 refs).

An updating of [Terrant 1975].

[Troff 1980]

Troff/Nroff Documentation. On-line documentation, University of Waterloo Mathematics Faculty Computing Facility TSS, December 2, 1980, edition.

Describes the TSS variant of Nroff.

[Tunnicliffe 1971]

William W Tunnicliffe: "System X--A User Need", in [Landau 1971] pages 17-65.

Transcript of a talk about what a composition system should be like.

[Type 1979]

Type Manual. On-line documentation, University of Waterloo Mathematics Faculty Computing Facility TSS, Waterloo, Ont., March 7, 1979, edition.

[UNIX 1979]

UNIX Programmer's Manual 2: Supplementary Documents (7th edition). Bell Laboratories, Murray Hill, NJ, 1979.

[US 1977]

"U. S. News Data Terminals Compress Deadline Schedule", Communication News (October 1977), page 45.

Describes results of using system at that magazine. Reprinted in: IEEE Transactions on Professional Communications **PC-21**, 1 (March 1978), pages 23-24.

[Varley 1977]

H Leslie Varley: "Composition from Publisher-Prepared, Machine-Readable Input", IEEE Transactions on Professional Communications **PC-20**, 2 (September 1977), pages 65-67 (1 ref).

About the Mack system.

[Vasdi 1978]

Peter Vasdi: "How an Author Learned to Program and Found a New Freedom", Canadian Datasystems **10**, 11 (November 1978), pages 28-32.

User's experience learning and using Script for 2000-page document.

[Vergès 1972]

Jeanne-Claire Vergès: "L'Edition de Textes Basée sur la Reconnaissance Automatique de Leurs Structures", Canadian Information Processing Society Canadian Computer Conference (1972), pages 324401-324412 (14 refs).

An article in French, favoring the automatic recognition approach.

[Vroff 1976]

Vroff Reference Manual and Tutorial Guide. On-line documentation, University of Waterloo Mathematics Faculty Computing Facility UNIX, Waterloo, Ont., 1976.

[Walter 1969]

Gerard O Walter: "Typesetting", Scientific American **220**, 5 (May 1969), pages 60-69 (2 refs).

History of typesetting with emphasis on equipment, from origins to CRTs.

[Waterloo 1978]

Waterloo SCRIPT Reference Manual. On-line documentation, University of Waterloo Department of Computing Services, Waterloo, Ont., January 13, 1978, edition.

[Wetherell 1978]

Charles Wetherell: Etudes for Programmers. Prentice-Hall, Englewood Cliffs, NJ, 1978.

A book of exercises; chapter 4 is about writing a text formatter.

[Wohl 1977]

Amy D Wohl: "What's Happening in Word Processing", Datamation **23**, 4 (April 1977), pages 65-72.

Survey of office WP equipment; lists 80 manufacturers.

[Ziegler 1969]

J C Ziegler: "Text/360 from a User's Point of View", IEEE Transactions on Engineering Writing and Speech **EWS-12**, 2 (August 1969), pages 33-35