TEF TEXT EDITOR AND FORMATTER

TEF TEXT EDITOR AND FORMATTER

By

STEPHEN JAMES MAVEETY, B.Sc.

A Project

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

February 1976

MASTER OF SCIENCE (1976)
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: TEF Text Editor and Formatter

AUTHOR: Stephen James Maveety, B.Sc. (McMaster)

SUPERVISORS: Professor R. Rink and Professor D. Kenworthy

NUMBER OF PAGES: viii, 120

ABSTRACT

A survey of the main features and characteristics of text editors and formatters is given. An implementation of a text editing and formatting system is discussed. The Text Editor and Formatter (TEF) was designed to be easy to learn and use and to allow extensions of the present version with little modification to the existing system. TEF is a content (or context) oriented editing system with line organization and text formatting capabilities.

ACKNOWLEDGEMENTS

I wish to thank my supervisors Dr. R. Rink and Dr. D. Kenworthy for their assistance during the work on this project and for their invaluable assistance during the preparation of this manuscript.

T A B L E O F C O N T E N T S

L I S T O F F I G U R E S

L I S T O F T A B L E S

CHAPTER 1

Discussion of Text Editors And Formatters

1.1 Introduction

Vast amounts of time and money are spent on the collection, preparation and updating of text whether it be computer programs, letters, manuals, user guides, books, reports or manuscripts. Due to the vast amount of information digital computers have become an ideal mechanism to assist in the gathering and manipulation of text. The production of written material such as manuals or course notes, which is subject to frequent change, has the advantage that revisions are easily introduced and immediately available on access to the computer. The production of copies of manuals, user guides, notes, etc. on a high-speed printer is fast and economical.

In many areas, on-line creation and modification of programs and their documentation has become widely accepted as a productive and cost-effective use of the computer.

For manuscript composition, the most obvious advantage offered by computer assisted editing is the tremendous reduction in time required to produce a final or alternate document. Normal editing requires many cycles in which the text must be read and reread, typed and retyped. Text stored

in the computer need never be retyped but only updated. This reduces the possibility of typographical errors with less need for proofreading. Intermediate drafts can be printed out quickly and economically. However, only the final draft need be printed, unless the user wishes to study different formats for his text.

What follows is a description of the design and implementation of an online interactive Text Editor and Formatter (TEF) to facilitate the creation, modification and formatting of text. TEF can manipulate various kinds of text including computer programs and natural language text. TEF can be executed either in time sharing or batch mode and is implemented on a Control Data 6400 computer. At present, TEF operates interactively under INTERCOM version 4.3 which provides time sharing access to the computer.

One of the main objectives of this project was to design a relatively machine independent editing system. This required using a commonly accepted programming language. In this case FORTRAN (FORTRAN Extended Version 4 [CDC 74]) was employed.

An editing system must be convenient to use. This requires a concise, mnemonic command language and a method of text organization to allow the user the ability to work with his text in terms of its structure rather than in a notation dictated by the system.

TEF is a content (or context) oriented text editor with line organization. All text operated on in TEF is grouped into lines. The advantage of content addressing is that it allows the user the ability to select lines of text by their content rather than by line number. Content searching is more natural in addressing natural language text rather than specifying numbers which bear no relationship to the text to be edited. The character strings in the file are not subject to change by the system as relative line numbers are and consequently content indentification provides a more stable method of locating portions of a file.

Some of the major editing features include travelling to the first occurrence of a user-specified pattern in the file, occurring anywhere in a line or only in certain columns of a line, uniform substitution of one pattern for another wherever it occurs or from the current position in the file to the end of the file, character by character editing of a line of text, random access to a pre-defined subsection of text in the file and formatting of the text.

For each line of his text the user may specify format codes that determine margins, headings, running headings, paragraphs, left and/or right margin justification (default), indentations, centering, paging and spacing control and non-formatted output.

Since changes are inevitable in computer software,

systems must be adaptable. TEF was designed to be flexible and can easily be expanded to incorporate new functions, commands or formatting features as required.

The casual user need only learn a few basic editing commands and formatting codes to benefit from TEF's capability. A more frequent and experienced user can obtain maximum power by making use of the additional features of TEF.

1.2 Text Editors and Formatters

1.2.1 Introduction to Text Editors

Current text editors range from simple line editors with changes accomplished only by replacing an entire line, to versatile editors using content searching and string replacement commands. Some of the more versatile editors employ a macro level language [BEN 72] for defining editing functions in terms of existing commands.

A typical structure of a text editor is represented in figure 1.1. The user is seated at a typewriter terminal or a cathode-ray tube display console (CRT) on which is displayed one or more lines of his file. The user enters editing commands via the terminal keyboard to alter the displayed text or advance to another portion of the file to be displayed. The commands are passed by the Terminal Input Handler to the Command Interpreter of the editing system. This routine

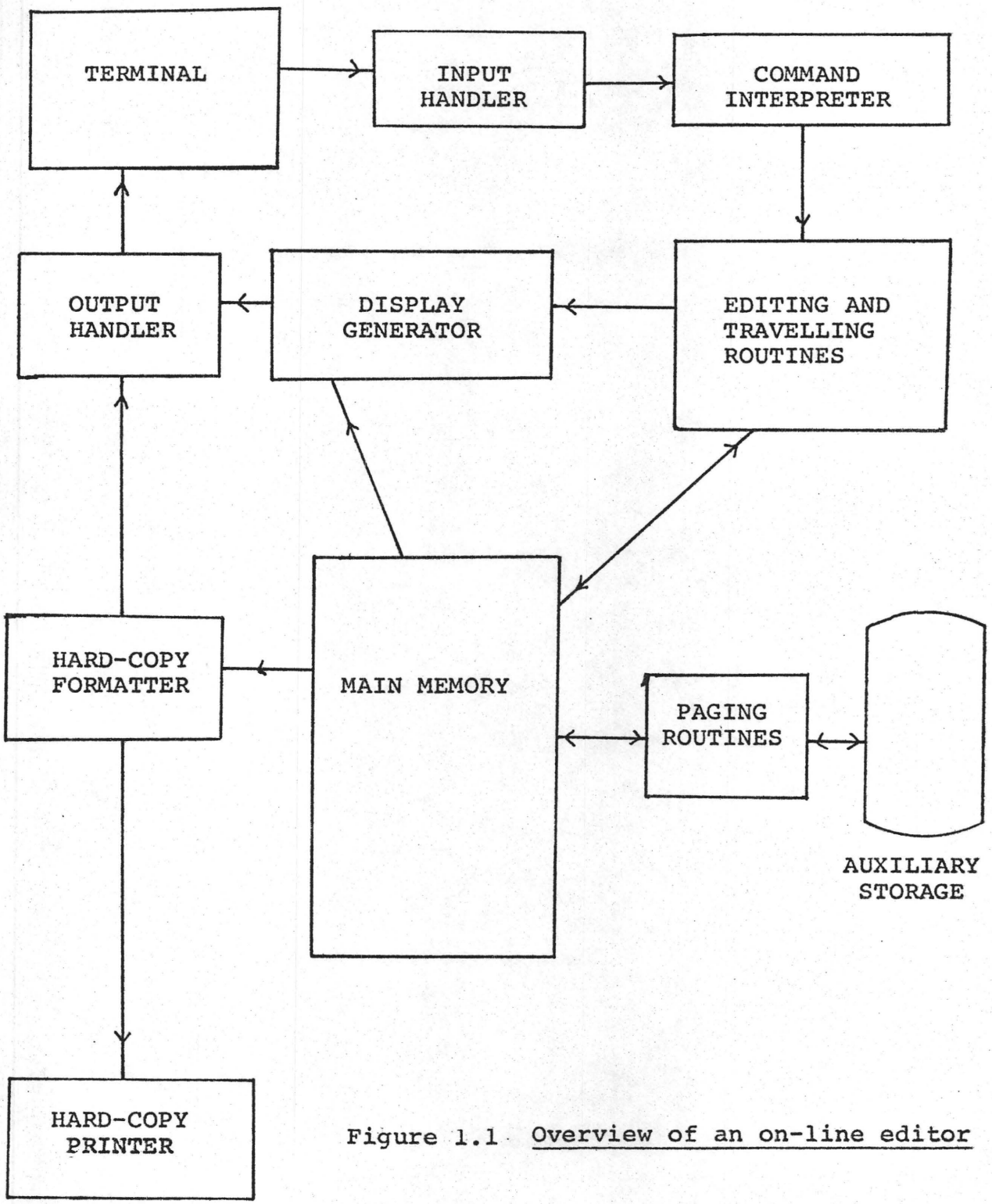


Figure 1.1 Overview of an on-line editor

parses the requested command (e.g. insert, delete, substitute, move, etc.) and the associated data. The associated data refer to positioning information, possibly where in the text to make an insertion, deletion or substitution, or character string data such as the string to be inserted, or searched for.

The user can "travel" through the file to display other areas to be read or edited. The information in the command input would indicate the number of lines to be advanced, or the character string to advance to and the direction (forwards or backwards).

The information extracted from the request is passed to the appropriate editing or travelling routine that performs the specified operation. When an edit is performed on a line of text the internal form of the text is altered and the updated text is reformatted by the Display Generator for feedback to the user. When the files are large and travelling or large edits are invoked, the relevant portion of the file may not be resident in core. In this case, Paging or Mass Storage I/O Routines must be called upon to bring in the data requested from secondary storage.

Superimposed on an editor is usually a time sharing system which supports multiple terminals and supervises sharing of programs, CPU time and core among users, while protecting each user's files from unauthorized access.

The Hard-Copy Formatter is used for free form natural

language text to convert the internal structure of the text to formatted hard copy for output on typewriter terminals, high-speed line printers or typesetting devices. The user may specify formatting commands or codes that determine margins, running headings, paragraphs, left and/or right margin justification, indentations, centering, underscores, type-face changes, etc.

These codes can be stored in-line with the text making them indistinguishable from text for editing purposes or formatting commands can be interspersed between lines of text.

Advanced features implemented on some editors or formatters include foot-note generation [STA 74] , automatic indexing and renumbering of sections or references after changes have been made, spelling checks, typesetting of mathematical symbols [KER --] etc.

The major design goals of an editor from the user's point of view are:

1) convenience for the user, which requires a simple and mnemonic command language;
2) fast response to a large number of terminals;
3) powerful commands performing any functions on a piece of text that can be performed manually;
4) text addressing features which allow the user to quickly "zero in" on a specific piece of text to be edited;

5) hard copy formatted output.

Ideally these goals should be met using as little as possible of such resources as money, implementation time, CPU cycles, core, disk space, etc. However, trade-offs will exist between increased power of the system, user convenience and response time. Increased power in the instruction set implies greater demand on resources and therefore slower response time. Usually the complexity of the editing commands increases for more sophisticated systems (i.e. more parameters, exceptions and forms for each command), possibly reducing the convenience of the system by introducing awkward constructions.

1.2.2 Basic Characteristics of Editors

Generally, there are two types of text to be edited; program text and natural language text. Program text refers to the text of computer language programs while natural language text refers to text such as English or other language text used for communication between people.

Although, any given editor can be used for either natural language or program editing, the design objectives and the resultant capabilities are usually specialized and more convenient for one than the other. To distinguish between these two types of editors, computer program language editors will be called "program editors" while natural language text editors will be called "text editors".

The type of editing performed is different for program editors and text editors. Editing computer programs involves minor substitutions in an existing line of text such as substituting one op-code, variable name or operand address for another, inserting or changing a label, etc. Since many programming languages are designed to be punched onto 80 column cards, it is quite reasonable to store the text line by line in 80 character card image format. Obviously, this is not the most compact method of disk storage since most of the line is normally left blank, but it is certainly the easiest method to implement.

For an editor oriented towards English or natural language text manipulation, one wants to make insertions or deletions of arbitrarily sized character strings at various points in the text. In ordinary English language text each successive line of text is essentially a continuation of the previous line; therefore, the system must deal with line overflowing and contracting from line to line. The unit of storage may be a line or statement of several hundred characters where the text may grow or shrink dynamically.

Any editing task for text stored in a computer requires two inputs: the task to be performed and the identification of the portion of text to which it applies. There are many ways to identify the text to be edited. These include giving a line number, typing an appropriate context string, giving

both a line number and a context string, directly pointing at the desired text using a light pen or indirectly using a pointer symbol or cursor driven by keys (up, down, left, right).

Most program editors use line numbers, either relative to the begining of the file, or absolute line numbers, to identify lines of text. Context searches are more popular for natural language text editors.

Formatting capabilities can be present in both program and text editors. For example, it is useful to be able to present block structure of a procedural language program by suitable line skips and indentations or to format natural language text into paragraphs, sections and pages.

1.2.3 Program Editors

Computer program editors have become increasingly popular in time sharing environments, where a facility for the online preparation and modification of programs is combined with some form of remote job entry.

An online editing system that stores the programs on disk or tape eliminates the need for hand-carrying card decks to and from the machine and enables updates to be made quickly and efficiently. The time required to write, debug and run a program is greatly reduced.

Some program editors provide immediate syntax checking of a language as the statements are entered. Syntax checking

can be an important function of a program editor. However, many systems having this capability can support only a limited number of languages since the syntax of the language is incorporated within the system.

The following are outlines of some of the features of a few of the most common editors.

1) Conversational Context-Directed Editor

The Conversational Context-Directed editor was developed at the IBM Cambridge Scientific Center for the 360/67 CP/CMS operating system and known widely as the CMS editor [IBM 69] .

Although the CMS editor is often used to edit normal text, the editing commands and the interactive teletypewriter terminals are best suited for the simplified text of computer programs.

Text is stored internally depending on the type of text being edited. The text of computer programs is stored in fixed-length 80 character records, corresponding one-for-one with a standard card deck of source code. Files of normal English text are stored as variable-length records (one line/record, with a maximum of 130 characters per record). Variable length records provide substantial savings in the disk space required to store a large online data base.

The line are treated as fixed length in main memory and padded to 130 characters with blanks.

The text file contains lines connected in a two-way

linked list structure for travelling forward or backward in the file. Since the editor was written for a virtual memory machine, the entire file currently being edited is kept core-resident. Response times are normally quite good since no programmed I/O to secondary storage is done by the editing program during editing. However, should the machine or operating system fail as the user is editing, all his work since he last saved the current file on disk will be lost. In addition, the maximum size of a file is limited by the memory size of the virtual machine.

Editing features include adding lines of text to the file, deletion of lines and string replacement within a line. Changes within a line are made by typing the incorrect characters followed by the replacement string; inserts are made by replacing a string of text with the original text included with the text to be inserted. Deletes are done by a null replacement for the string to be deleted. This method of specifying editing changes is referred to as content or context-directed editing.

CMS does not handle line overflows so that if a string replacement causes the line to exceed the maximum length for the particular type of file, the extra characters are truncated. Therefore, a large insertion in the middle of a line must be broken into separate lines to avoid this problem. This is one of the reasons why CMS is not particularly suited for

English text editing.

The CMS editor is line oriented with the line being currently created or modified (current line) in the program determined by a "line pointer" that changes as travelling and editing occur. The current line pointer may move forwards or backwards by one or more lines by following the forwards or backwards pointer chain to scroll through the file.

The user can search for a specified character string occurring either in a fixed line position ("find") or anywhere in a line ("locate"). If a match is found, the line pointer is moved to the line located.

The CMS editor also provides flexible tabbing facilities, useful for column dependent languages such as FORTRAN, or for indenting block structured languages.

2) The WYLBUR Editing System

WYLBUR is a line-oriented text editor developed by the Stanford Computation Centre for use on the IBM 2741 teletypewriter terminals to provide an online editing facility to be used in conjunction with the online and batch services of Stanford's IBM 360/67 [FAJ 73] , [ALL 69] .

A WYLBUR file is divided into lines of text, each containing from 0 to 133 characters. Normally a maximum of 72 characters is used for programs. Each line is stored as an unpadded character string in a variable length record with

an associated length and line number. This method of storage is more compact (usually most of the line is left blank) and each line is addressable. The WYLBUR system uses absolute line numbers that do not change dynamically as editing is performed as opposed to line numbers relative to the top of the file. This provides stable reference points. The line numbers are in the form of decimal numbers which may range from 0 to 9999.999 with at most three digits after the decimal point.

Most of WYLBUR's commands operate on or address a group or range of lines that possess the same distinguishing characteristic. An explicit range is a list of line numbers. A single line is specified by giving its number; a set of contiguous lines by two line numbers separated by a slash (/). Single lines and sets of lines may be specified in any combination. The range of all existing lines may be referred to by ALL, which is the default range for many WYLBUR commands.

Content addressing is accomplished with the associative range. An associative range consists of all lines containing a given string of characters. Explicit and associative ranges can also be combined.

Two very useful and powerful intraline editing commands are available: CHANGE and MODIFY. The CHANGE command allows uniform changes or string replacements to a range of lines. The MODIFY command permits a user to work directly on a copy

of the line, typing deletions, insertions and replacements below. For example, he can delete part of a line by typing a "D" under the first and last characters of the string to be deleted; he can insert a string into a line by typing an "I" under the character before which the insert is to go, followed by the insertion string (total length cannot exceed the maximum line length of the file); and he can replace a same-length string by typing "R" followed by the replacement characters. Replacing strings of different lengths can be performed by using the "D" and "I" forms.

Batch jobs can be submitted from the WYLBUR terminal, batch printed output can be retrieved, and inquiries may be made about both system and job status.

3) Quick Editor (QED)

The original version of QED was implemented at the University of California at Berkeley [DEU 67] and has been revised extensively for commercial use by Com-Share [ARB 67] and also by Bell Laboratories, Murray Hill, New Jersey [KER 72]. An extremely powerful version is running on the Honeywell 6000 computer at Bell Laboratories. What follows is a discussion of this version.

QED stores all the text it is working on in core giving rapid access to all the text but restricting the amount of text which can be edited at one time. Very large files must

be avoided since cost and response time increase with core usage.

All text in QED is stored in buffers. At any time there is a current buffer to which most commands implicitly refer. In each buffer there is a current line which is changed by most editor commands. Text may be entered into a buffer through the terminal or from a file. Text is never filed away automatically, but must be written explicitly to a file.

QED provides a facility for creation and manipulation of several buffers, which are often used to give temporary storage of sets of lines as they are being copied or moved from one part of the text to another. Multiple buffers provide independent areas where text can be kept and operated on conveniently. The pieces can be manipulated as units, or normal editing can be performed on individual lines of any one of them.

The text manipulation facilities approach those of SNOBOL4 [GRI 71] . Pattern matching is performed by regular expressions which are patterns specifying a set of character strings. Regular expressions can be used for matching a pattern at the end or begining of a line, matching any character at a point in the line, matching any of the characters in a specified string and no others, matching any number (including zero) of adjacent occurrences of text matched by regular expressions, for alternate pattern matching (i.e. text which

matches either of two patterns) and other text manipulations. Regular expressions are used for specifying text replacements and for searching for a particular piece of text.

Many features are available for addressing text. Addressing lines of text in the current buffer can be by current line number (relative to the begining of the buffer), by absolute line number, by "." meaning current line, by "$" meaning last line in the current buffer, by context using regular expressions or by additive combinations of the above (e.g. ".+1" meaning the line after the current line).

A useful feature of QED is the provision for executing editing commands from a buffer allowing the execution of user-defined and pre-existing macro command structures. Therefore, a sequence of editing commands can be saved by the system as a normal text file. This set of commands can be re-executed at a later date to re-edit the text. The re-editing facility makes it possible to maintain slightly different versions of a file without having to duplicate the main file many times. The text is stored only once with alternate versions generated by executing a sequence of commands previously saved on a file. This is useful for testing changes to an operational program without actually modifying the original program until the changes have been completely tested and debugged.

1.2.4 Text Editors

Normal manuscript composition and editing requires many cycles where text must be read and reread, typed and retyped. Computer assisted editing provides a tremendous reduction in the time required to produce a final, or alternate document. Text stored in the computer is never retyped but only updated. As a result, the number of typographical errors is reduced requiring less proofreading.

Access to a common data base can be useful for a group of researchers or documenters working in the same area, or for common access to updated project or management information.

Since all of a user's editing can be performed online there is no need to request any intermediate printed copy. However, the convenience and naturalness of hard copy is still important to the "red pencil" school of editing where changes are manually made to a printed copy by a red pencil or similar means and later made online to the corresponding text file. The corrections are therefore performed twice. Transition from hard copy to soft copy techniques will be a gradual one as reliability, user convenience and system availability increase.

The following are outlines of the main features of a few text editors.

1) ASTROTYPE

The ASTROTYPE system [VAN 71] was designed by Information Control Systems and consists of up to four IBM Selectric typewriters and memory units connected to one control unit (the DEC PDP-8). It was intended for preparing forms, manuscripts and input for photo-composition devices.

The text is recorded on magnetic tape as it is typed and can later be modified and printed in final form. Text can be typed in an "unchangeable" mode where it is printed exactly as it was typed, or in an "adjustable" mode, where the control unit prints blocks of text (e.g. paragraphs) according to the current width.

Text is stored in lines numbered sequentially from the top of the file (i.e. relative line numbers). The basic editing command is substitute. Substitutions within a line are made by typing the line number, the old character string plus any additional context text, and the new string including the additional context used above, if applicable.

As in the CMS editor, insertions and deletions within a line are forms of substitution. Thus, to insert within a line, one must specify context to the left or right of the insertion point as the old string and repeat this context plus the text to be inserted as the new string.

Individual lines may also be erased and moved by their line number. Verification is provided by a printout

of the line before the change is actually made.

Printing is done at the typewriter at an average rate of 150 words/minute and various fonts and type sizes may be used by changing the typing ball. The printing can be programmed to stop at any point, so that additional input may be entered manually, and then continue. This is useful when changing a portion of a form letter.

2) System/360 Administrative Terminal System (ATS)

ATS was developed by the IBM Corporation [VAN 71] utilizing the IBM 2741 typewriters as the interactive device. ATS is provided by IBM as a standard package on the 360/370 series computers.

Each time a line is typed in, an internal line is created and a line number is assigned. The length of an internal line may vary from 0 to 130 characters, and a text file may contain up to 9999 internal lines in a linked-list storage structure. The line numbers are relative to the top of the file and change dynamically as lines are inserted or deleted. This can cause problems as with ASTROTYPE since line numbers of text strings may no longer correspond to those of the most recent printed copy. The designers of ATS suggest that editing be performed from the bottom of the file to the top; this eliminates the problem of changing line numbers, but is very inconvenient for the user.

Text can be moved around but not copied. To insert new lines in the middle of a file, the lines must first be typed at the end of the file and then moved to the desired position. Substitutions, deletions, and insertions within a line are made as in ASTROTYPE.

Text can be entered in "formatted" mode, in which case the text can be arranged by the output program to satisfy the specific page format (e.g. line justification), or in "unformatted" mode, in which text is saved and printed exactly as it was typed. As in ASTROTYPE, an online printout can be stopped in the middle, allowing the user to type in additional text.

Because the editing functions are few and the data structure is quite simple, the CPU usage is minimal. However, ATS is far from ideal for general purpose editing.

3) The Hypertext Editing System (HES)

The Hypertext Editing System [VAN 71] is a CRT-based (IBM 2250) system allowing full editing and formatting capabilities. It is oriented toward "typeset" output using a line printer as well as flexible input and online editing. A light-pen and a set of"function keys", under program control, are used to indicate to the system the nature of the edit to be performed. The editing function is selected by pressing the appropriately labeled function key. The portion

of text to which the function applies is then indicated by pointing at the text with the light-pen.

HES provides maximum convenience for the user since no command codes need be remembered and no extra typing is required to indicate a context string.

To delete a portion of the text, the "delete" function key is pressed, after which the two endpoints of the text to be deleted are pointed at with the light-pen. The text is then blanked out on the display for verification. The deletion if correct can be accepted by pressing a control key otherwise it may be cancelled, leaving the original text unchanged. Editing functions include insert, delete, substitute, rearrange and copy. Prompting messages by the system specify the actions available at each step. A maximum of approximately 2500 characters may be deleted or rearranged at a time.

Formatting options are available so that text may be formatted both for online display and hard copy printouts. An off-line computer typesetting program is used for final hard-copy printing on a computer line printer equipped with an upper and lower case printer chain.

The data structure and the editing operations are entirely independent of display or printout lines and pages. Text is externally segmented by the user into arbitrarily long user-designated fragments called "text areas". Each area is a continuous linear string of text, and might be a

chapter, an entire book, or a short footnote. These areas may be interlinked and cross-referenced in any manner so as to form a directed graph of text segments (the vertices of the graph) and their cross references (the edges).

Two types of cross references exist: "branches" and "links". Branches are unconditional jumps between two fragments that the user may encounter in the text forcing him to lightpen a choice in order to proceed. Links are conditional jumps, where the reader may bypass or lightpen. The link is similar to the manuscript footnote principle that allows additional explanations and browsing.

The resultant mobile (i.e. text fragments linked by user controlled jumps) is called a hypertext, "the combination of natural language text with the computer's capacities for interactive, branching, or dynamic display... a nonlinear text... which cannot be printed conveniently... on a conventional page " [NEL 67] , A practical example of a hypertext might be an on-line encyclopedia or a set of programming and systems reference manuals, with each cross-reference lightpen sensitive.

The fragments of text can be examined by tracing linear paths through the hypertext, either for on-line browsing purposes or for printing. The system also remembers the sequences of branches or links that the reader has taken,

and this allows him to reverse his trail. Random access to any point in the text is provided by allowing the author to assign "labels" anywhere in the text and later to "jump" to any of them by lightpenning the appropriate one from a list of choices.

Hypertext areas are stored internally as one or more pairs of variable-sized "pages" (unrelated to a printed page), each pair consisting of an "order code" page and a "text" page. The order code page is an ordered sequence of text pointers specifying displacements into the text page, interspersed with formatting and structure codes. The text page contains the actual text. To edit text, order codes and their pointers are inserted, deleted, or updated while the text strings remain intact. This type of structure involves little character shuffling and recopying but "garbage" strings are rapidly accumulated.

1.2.5. Formatters

Most text formatters are designed to execute in conjunction with a text editor. The input to a formatter is usually the output from a text editing system consisting of text and formatting commands to specify the type of formatting to be performed. The formatting commands can be inserted into the text file by the editor in two ways: each on a separate

line interspersed between lines of text as in the EDIT/HP 3000 Text File Formatter [MAC --] , or stored in-line with the text as done in a Text Formatter developed at the University of Waterloo [STA 74] . In either case, a flag or escape character is used by the formatting system to recognize commands in a file of text.

Typical format commands determine margin widths, headings, running headings, paragraphs, justification, indentations, centering, underscores, type-face changes, unformatted output, etc.

1) The EDIT/3000 Text File Formatter

The EDIT/3000 Text File Formatter is available for the HP 3000 computer systems from Hewlett Packard. The text formatter takes as input an unformatted text file, such as those produced by the HP 3000 editor, formats the text and writes the formatted text onto another file.

The text file is accessed sequentially and is assumed to be a disk file with 80 byte (character) records. The last eight bytes of each record are ignored and may contain sequencing information. The output file is assumed to consist of 72 byte records. The text file and output file characteristic may be altered by system control cards before compilation of the text formatter.

To distinguish commands from the rest of the text, they must be placed on a separate line and the first character

of that line must be a pre-defined control character (usually a period "."). More than one command may be placed on a line separated by semicolons.

Each line of text in the file is treated as a continuation of the previous line until a command which causes a "break" is encountered. At a break all words read are printed with the next text line starting on a new line of output.

The user can control the number of lines per page, number of characters per line, width of margins, page numbering, position of headings, justification, centering text, spacing and paging control.

2) A Text Formatter

This text formatter was developed at the University of Waterloo and is written in a language called SPITBOL [DEW 71], which is a fast compiler version of the SNOBOL4 language. The input text to the formatter is a file created through some file editing system, (e.g. WYLBUR) or by means of some utility.

The input text consists of words and text commands. A text command is a string of characters preceded by the excape character "¢". As in the previously described editor the last eight characters of an input record will always be stripped off to avoid problems when using text editors which introduce line numbers in the records.

The formatter operates in two modes: formatted and unformatted. In formatted mode right justification is usually performed with words delimited by the end of card, delimiting text commands or one or more blanks. In the unformatted mode words are delimited only by the end of card and word delimiting commands. Only one word per line is printed since embedded blanks are considered in the same way as any other character.

There are over 40 formatting commands including such features as special collection modes (e.g. string collection, unformatted mode collection, figure and footnote collection), insertion of a string into a word at the position of the command occurrence (e.g. special print chain characters which cannot be directly input from the terminal (Greek letters) the current date, "¢", footnote reference), overprinting, underscoring, and control of paging and page format.

Figure and footnote generation is an interesting feature of this formatter.

The footnote command automatically inserts the footnote reference marker into the text where the command appeared. Figure and footnote text is given in the place where they are supposed to occur in the output text. Figures will be placed in the page where they fit. If the text for a figure is too large for the remaining part of the page, it is placed in a waiting queue in order to be printed on one of the succeeding pages. The order in which figures occur

in the text is strictly obeyed in the output of text.

If a figure is too large to fit on a complete page, it will be broken into several pages.

Footnotes which are too large are also broken. A footnote must have 3 lines at least in order to be broken. Furthermore, there must be at least as many lines in the remainder of the page as needed to contain the line referring to the footnote and two lines of the footnote, since the reference and footnote text must be on the same page. Footnotes can continue over more than one page when broken.

This formatter is very powerful. Unfortunately, the large number of commands, the lack of mnemonic text commands, the lack of error checking of the text commands and the structure of the commands make it a difficult system to learn and use.

1.3 Outline of Further Chapters

The preceding discussion has outlined the basic characteristics of program editors, text editors and formatters.

In Chapter 2 the text editing system TEF will be described with respect to its design considerations, internal structure and primary features.

Chapter 3 explains the operation of the TEF formatting system and gives some formatting examples.

In Chapter 4 the conclusions and the results of this investigation are discussed.

CHAPTER 2

TEF

2.1 Introduction

TEF is essentially a combination of a text editing system and a text formatter. Editing and formatting can be performed together or independently within the same system. A formatter is a natural companion for a text editor since hard-copy formatting is usually required for all types of text at some time or another.

In combining the two systems the user need only learn to use one system, one set of commands and one type of file structure. All editing and formatting is performed using one system designed for convenience for the user. In a combined system the user does not have to match the characteristics of the editor file to those of the input file for the formatter. The user can cycle between editing and formatting of a file or subsections of the text to experiment with different formatting structures without leaving TEF.

Current editors range from simple and restrictive systems to complex and powerful systems. TEF was an attempt at a system easy and convenient to learn and use, yet providing adequate power for most editing purposes.

The command structure is simple, mnemonic and has at most two alternate forms for a command. The alternate forms usually differ only by the direction of operation (e.g. + for forward, - for backward). The user does not need to learn complex addressing techniques as content addressing is provided as a stable method of text addressing. The commands were chosen to provide features that appeared most useful in the opinion of the author. Obviously, these commands may not always be convenient for every user.

2.2 Considerations in the Design of TEF

The primary objective in the design of TEF was to provide both an editing and formatting system which is easy to learn and use.

One of the most important characteristics for any editor is that it is convenient to use. Therefore, the command structure of TEF was chosen such that the command words have an English meaning which expresses their function. All commands begin with a slash and can be abbreviated to their first three characters. An experienced user would prefer to type as few characters as possible to perform the desired functions. The whole command word may be entered to aid in learning the command structure. The user is free to use any non-alphabetic character or blank as a string delimiter, providing it is not contained in the enclosed string.

Error messages supply information about the type of errors which have occurred and where they were located. A three character program identification is included with each message to indicate the subroutine that detected the error. The user need only be concerned with the message while the source of the error is useful for tracking future program errors when extensions are added or revisions made to the existing system.

When entering text or commands via the CRT, a mistake can be corrected by backspacing and then typing the correct characters. In addition, a complete input line can be deleted using the "control X" function on the CRT terminal.

TEF was designed to process both manuscripts and computer programs. The text editing and formatting systems within TEF are interfaced through the editing command which formats the text file. This provides flexibility for the text editor or formatter to be used separately within the TEF framework.

The method of text organization was chosen to make the user's text easily accessible. One of the basic characteristics of TEF is content addressing. Content addressing allows the user to select lines of text by their content, rather than by line number. In most systems using line numbers, they are defined as displacements of lines relative to the top of the file. Because of this, they are rather arbitrary since they can change dynamically as the file is modified.

Content addressing is a more natural way of addressing natural language text since the user need not be concerned with resequencing of line numbers or specifying numbers that bear no relation to the text to be located or edited. The user is free to address his text by his own labels attached during the construction of the text rather than a fixed notation internal to the system. Since the character strings in the file are not subject to change by the system in the way that line numbers are, content indentification provides a more stable method for locating portions of a file.

The text in TEF is organized into lines containing a maximum of 130 characters. The maximum length of a line of text was chosen to fit easily within a print line file and onto two consecutive lines on the CRT screen. The word size on the CDC 6400 computer is 60 bits allowing the storage of 10 characters per word(with a six bit display code for each character). The length of the lines of text are a multiple of 10 so that the text can be conveniently packed into an integral number of words.

The concept of machine independency and portability is an important consideration which led to programming TEF in FORTRAN. However, machine dependent features exist due to the implementation on the host computer. For example, the mass storage I/O routines are machine dependent.

They provide the interface between FORTRAN and a mass storage device. This does not present any real problem as most systems have similar mechanisms for I/O with mass storage.

Machine independency was traded slightly for efficiency on the host computer. The non-ANSI FORTRAN R-format was chosen for the input to TEF. Using the R-format characters are read and stored right justified in the word with zero fill. This allows a character to be represented by an integer number equivalent to its display code (00-77 Octal). The decoding of the instructions and the manipulation of the characters with this method of storage is very convenient and efficient. This feature does not impose any restriction on machine independency since a routine can easily be written to convert from left justified with blank fill (standard A-format in FORTRAN) to right justified with zero fill, if the machine does not support this feature.

TEF is expandable. The command interpreter was designed to allow addition or deletion of commands to suit the implementor. The interpreter is very simple. It is involved only in deciding what type of command is to be executed. This is performed using a linear scan and a table look-up procedure. The interpreter invokes a subroutine to execute the command. The subroutine in turn processes any parameters or associated data particular to the command. The associated data is easily

processed in a linear scan since there are few alternatives or command forms. To introduce new commands the existing system needs little modification. The command table and the associated branch mechanism need only be extended to include the new function.

TEF consists of 24 commands. The casual user editing a small file can perform most of his desired editing with a small subset of these commands (5 or 6). A frequent and more experienced user can use the full power of the additional features.

Movement through the file can be performed either forward or backward from the current position in the file. Methods for moving through the text file are available to suit the particular type of editing required by the user. When editing or formatting will be applied to most lines of text in the file or the user wishes to scroll through the file, he can move through the file a line at a time with each line displayed for editing. The user can advance to a line of text containing a particular sub-string of text, where only the destination line is displayed for editing. This method is useful for editing text lines which are scattered throughout the file. Random access is provided to position the file at a line of text which was pre-defined as the start of a sub-section of text. This method is useful for editing large files

where most of the editing will be confined to a sub-section of the whole text file. For exanple, when editing a large file over a long period of time most editing will be confined to the end of the text file where the latest text has been added. This is also the fastest method for advancing to a section of text in the file.

A text editor should allow input of text from sources other than the CRT terminal. TEF can input text from a permanent disk file of card images at any time during editing. The input disk file can be divided into sections separated by a card image containing a slash(/) in column one. This allows portions of the input file to be added to the text file at any time or place during editing.

An editing system must be flexible enough to allow editing in both batch and time-sharing modes. Not all types of editing can be conveniently performed interactively. For example, if in a large text file the user made many consistent spelling errors of different words, he could specify commands to position the file at the begining followed by the command to replace every occurrence of one string of characters (the error) by another (the correction) for each spelling error. In interactive mode each line containing an error would be displayed followed by the corrected line. If the response time was slow and the user was not concerned with following

the sequence of corrections this method would be very unattractive to most users. An easier method would be to punch the commands on cards or create a file of these commands and run TEF in batch mode. No output is produced and the file is corrected with minimum time and effort. TEF can be executed in batch mode with the editing commands and text read from the standard input device (card reader) or from a permanent disk file. In batch mode the default for input is the card reader, for output the line printer, and for punched output the card punch.

The user can have replacements and deletions verified before they are made permanent. The lines to be deleted are displayed followed by a prompt to confirm the above deletion of lines. If the deletions are accepted they are deleted permanently. Otherwise, the file remains unchanged. String replacements within a line of text can also be verified. The line is displayed with the replacement made followed by a prompt to confirm or reject the replacement. If the replacement is accepted the new line replaces the original line, otherwise, the original line is unchanged. These prompts can be turned off or on by the user upon entering TEF.

Two useful features in TEF are the ability to locate a line of text in the file containing a specified string (LOCATE) and the ability to replace all occurrences of one

string by another everywhere in the file (AREPLACE). The latter is useful for consistent misspellers and for changing the names of variables or labels within a computer program. The LOCATE command is useful for selective editing of lines of text scattered throughout the file. These two features should be present in all editing systems. The problem was how to efficiently implement them. Both features require the editing system to search for a string of characters within each line of the file. Obviously, searching each line of text character by character could be employed. Unfortunately, this is extremely slow and inefficient. The method chosen is called the substring test technique [HAR 71] [BOO 73].

For each line in the file there is a corresponding hash code or 2-signature which reflects the sequence of two character combinations contained in the line of text. The 2-signature code is computed for the string to search for and compared to the 2-signature of each line. If the hash codes match implying that every two character combination in the string is also in the line of text, then it is possible that this line contains the string. Only in the case of a 2-signature match is the line scanned character by character in search of the string. This method rejects the majority of the incorrect lines depending on the length of the string to be found. However, if the string to be found is broken

between two lines, it will not be recognized by this mechanism. For the lines whose 2-signatures do not match, only one test is necessary to reject the line. This is a considerable improvement over the rigorous character by character scan. A discussion of the sub-string test technique will be given in more detail in section 2.4.

In TEF the format codes are contained in a particular section of the record for each line of text rather than interspersed between lines of text or within the text. There is no need to search through the text to find the format commands as they are displayed on the right of the CRT screen for each line. The text of a file which is being created for input to most text formatters must be "cluttered" with formatting commands or codes between lines of text or within the text. This is very distracting when working with the editor, and are not concerned with the formatting information. In TEF, the format codes are not within the text portion of the file so that the structure of the file (i.e. number of lines) does not change when format codes are added. The text of the file may be written to another file to be used for other purposes without having to remove the formatting commands.

The text file is a random access file in the form of a doubly-linked list structure allowing forward and backward motion. The TEF records are of fixed length and consist

of 20 (60 bit) words.

TEF in conjunction with the SCOPE operating system of the CDC 6400 allows each user complete file protection from unauthorized access. The user can catalog a text file with passwords to restrict access to the file.

2.2.0 Data and File Structure

2.2.1 Random Access Files

The text file is a random access file in the form of a doubly linked list structure. The FORTRAN mass storage input/output subroutines provide the interface between the TEF system and the mass storage device and control the transfer of records between central memory and mass storage.

The mass storage I/O subroutines allow opening (OPENMS) and closing (CLOSMS) of the file, reading (READMS) records from or writing (WRITMS) records into the file and changing between master and sub-indexes (STINDX).

Each record in a random file is uniquely and permanently identified by a record key. The key is used by the mass storage I/O routines and is mapped onto a hardware disk address. When a record is first written the key in the WRITMS call becomes the permanent identifier for that record. The record can be retrieved later by a READMS call that includes the same key and can be updated by a WRITMS call with the same key.

When a random file is in active use the record key information is kept in an array in the user's field length. This array is the directory or index to the file contents. The index is the logical link that enables the mass storage subroutines to associate a user call key with the hardware address of the required record.

If an existing file is reopened, the mass storage subroutines will locate the master index in mass storage transferring it into the index array. When the file is closed, the master index is written from the array to the mass storage device. If a file is opened which does not already exist, its master index is cleared to zero.

There are two types of index key: name and number. The index used for TEF is a number index. This number key must be a positive integer, greater than zero and less than or equal to the length of the index array minus one. A number key is more suitable for this type of application since the

pointers (which are keys) in the doubly linked list can be simple integers. Execution time is faster for a number index since it is not necessary to search the whole index for a matching key entry as with a name key. It also requires less central memory space for a number key.

The file structure of TEF contains a master index and a sub-index. The mass storage routines use the sub-index just as it uses the master index. The sub-index has its own index array for the currently active index; the mass storage routine STINDX allows switching between master and sub-index arrays to access records. The sub-index is read from and written to the file by the standard I/O routines since it is indistinguishable from any other data record.

2.2.2 The TEF Random Access File

The present version of TEF has a master index array of length 21 and a sub-index array length of 51. Since a number key must be greater than zero and less than or equal to the length of the index minus one the file contains 1000 records (i.e. 20 X 50).

By using sub-indexes central memory requirements are reduced since the active index array length can be reduced. For example, if a file of 1000 records was to be created without sub-indexing the length of the master index would be 1001. Additional levels of sub-indexing could be added, limited only

by the amount of central memory space available.

The 20 user accessible master index records are used to store 20 sub-index arrays each capable of addressing 50 different text records. Therefore, each record is uniquely identified by a master index key (integer value 1 to 20) and a sub-index key (integer value 1 to 50).

For example, to read the third record indexed under the second sub-index (i.e. master index 2) the following steps are necessary:

1) make the currently active index the master index array (if it is not already);
2) read the second master index which contains the second sub-index into the sub-index array by specifying 2 as the master number key;
3) change the currently active index to the sub-index array;
4) read the third record in this index by specifying 3 as the number key.

Only step four is performed if the currently active index is the master index for the required record. All the steps will be performed only when the previous read used a different master index. Therefore, it is important that the length of the sub-index be greater than the length of the master index. In this case, the master index would not have to change so often, thus reducing the number of I/O calls.

2.2.3 Doubly Linked List Structure

In a doubly linked list each element of the list (e.g. a line of text) is linked to the previous and following element by backward and forward pointers, allowing motion in either direction in the list. This type of structure was chosen for TEF to allow movement in both directions in the text file. This feature is a necessity in any text editing system to provide access to all of the user's text regardless of the current position in the file. The begining and the end of the file is marked with a zero pointer. That is, the first line has zero for its last or previous line pointer and the last line in the file has zero for its next line pointer.

2.2.4 TEF File Structure

When entering TEF without an existing text file such as in the initial creation run, a file is created with all the records in the file linked together using the next line (forward) pointers. There is a pointer to the begining of this list (next available space pointer) and the last record in the file (list) has a next line pointer of zero indicating the end of the list. This is the available space list (initially all the records are available for use).

All the records in the file are written onto a random

access file and given a unique permanent key to identify them. This method insures that the text file remains the same size as far as the operating system is concerned when text is added to the file. The operating system "sees" all the records both allocated and available. The user only "sees" the allocated records containing his text. If this technique was not used a previously cataloged permanent text file would have to be recataloged or extended each time text was added to it. A text file in this form will occupy more disk space since the presently unused records are stored along with the text records, but the amount of user convenience obtained with this method outweighs this disadvantage. This method allows protection from an operating system failure. If disk space was only allocated when text was added to the file, the master and sub-indexes would change each time a record was allocated. This means that the sub-indexes must be written back onto the disk file each time they change. In addition, the master must be written back onto the disk file before closing the file. Failure to do this will result in the loss of all text added since the file was opened, an invalid available space pointer and the result that some of the forward or backward pointers will be invalid. The user may lose the whole file or may only be able to access a small part of it. In any case, the file will be useless for further

editing. When an operating system failure or time limit occurs the indexes are not written to the random file.

By using an available space list with all the records allocated on disk, the user's file will be protected from a system failure. Since the master and sub-indexes never change throughout editing it is unnecessary that they be written back to the disk file before closing.

However, problems can occur if a system failure occurs before the pointer manipulation required for an addition to the file is completed. For example, the chain of forward or backward pointers may be broken, thus restricting access to part of the text file. In most cases, the text of the file and the formatting information can be recovered by using the random access features of TEF and the output facilities to form a new text file.

For the initial creation of a text file all records are on the available space list except for two. The forward pointer of each record indicates the next record to be used and the last record has zero as its pointer.

The record with master index M and sub-index N will be represented as (M,N). For example, the available space list will initially be in the form:

NEXT AVAILABLE SPACE →(1,1)⟶(1,2)⟶(1,3)... (1,50)→(2,1)...
(10,1) →(10,2) →(10,3)... (20,48)⟶0 .

The records may be stored in any physical position on disk and are not necessarily consecutive.

The last record in each sub-index will point to the first record of the next sub-index (i.e. new master index). For example, record (1,50) points to (2,1).

The last two records in the file are reserved for the TEF system and cannot be accessed by the user. The last record (e.g. (20,50)) is used to store the sub-section markers so that when the file is reopened any previously defined sub-sections will be accessible to the user.

The second last record (i.e. (20,49)) is used to store the pointer to the first line in the file since it is not necessarily going to be the record (1,1). Also, the pointer to the top of the available space list is stored in this record. These two pointers are updated and written to the file each time they change in case a system failure or time limit occurs during editing. These two pointers must be known by the TEF system when an existing text file is reopened. The pointer to the first record in the file allows random access at any time during editing to the first line of text in the file.

The user can therefore access 998 records with the present implementation. For most purposes this will be sufficient. However, it is a simple matter to increase the size of the text file by increasing the size of the master and/or

sub-index arrays.

As text is added to the file, records are allocated and removed from the available space list. When records of text are deleted from the file, they are put back onto the top of the list to be reused.

When TEF is entered with an existing text file the pointers to the beginning of the file and to the top of the available space list are accessed and the file is positioned at the first line of text in the file.

During editing there is a current line at which the file is positioned. This line is displayed on the CRT screen after each command is completed and is the last line displayed when several lines are listed on the screen. The current line can be changed by moving or advancing forward or backward in the file or by direct access to a line of text. The current line is stored in a line buffer on which all the line editing commands operate.

The text file and available space list structure is shown in figure 2.1 . Externally, they can be represented as sequential lists. However, the records can be in any physical position on disk with available space records and text file records intermixed.

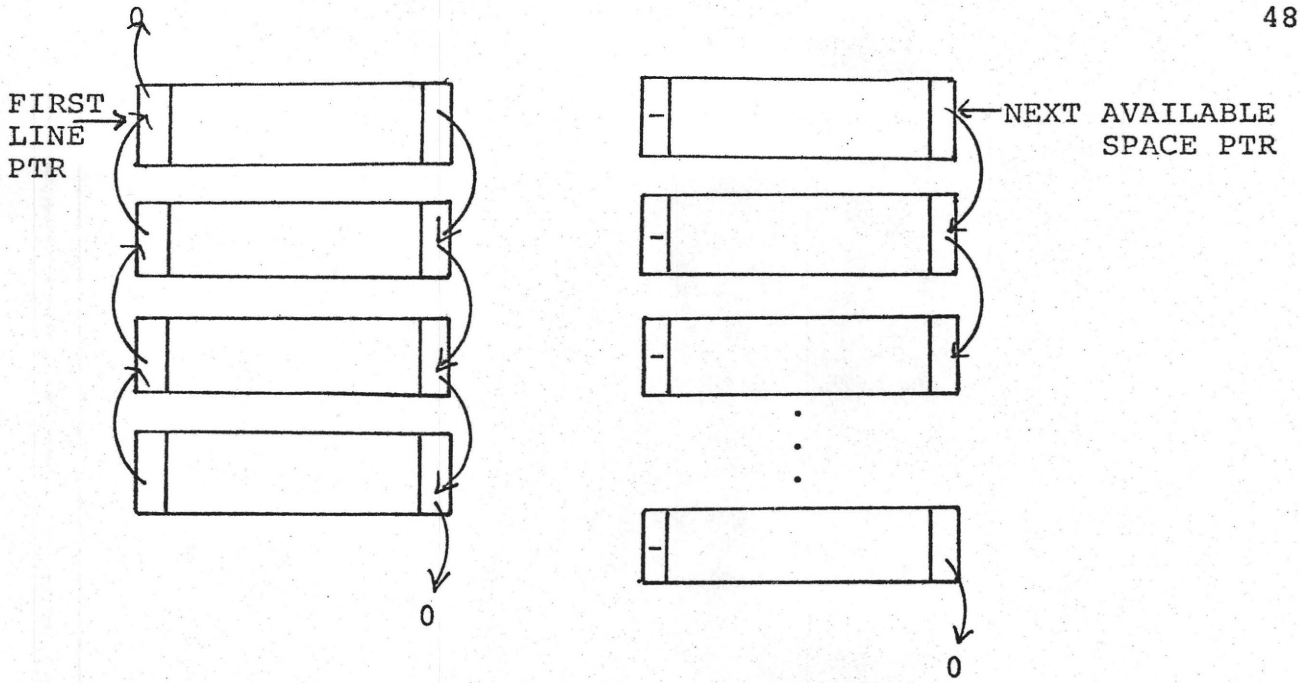
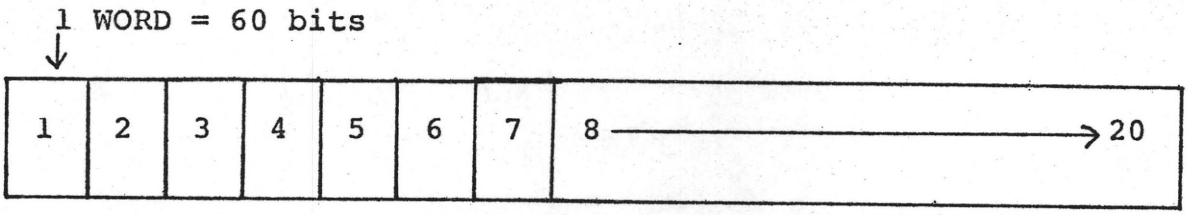


Figure 2.1 File and Available Space List Structure



1. - First word of Hashed 2-signature
2. - Second word of Hashed 2-signature
3. - Master index of Previous Record
4. - Sub-index of Previous Record

} Last line pointer

5. - Master index of Next Record in file
6. - Sub-index of Next Record in file

} Next line pointer

7. - Format Code Word
8. - 20. - Text of the line maximum 130 characters

Figure 2.2 Record Structure

2.2.5 Record Structure

Each record in the text file contains 20 words (60 bits each) of information. The record structure is illustrated in figure 2.2. The first two words (120 bits) contain the hashed 2-signature of the characters of text in the record. The 2-signature is used for content searches using the sub-string text method. This is to be discussed in section 2.4. The third and fourth words contain a pointer to the previous or last line in the file. Word three stores the master key and word four contains the sub-index key of the previous line. The fifth and sixth words contain a pointer to the next line in the text file. The fifth word stores the master key and the sixth word stores the sub-index key. The master key has a value from 1 to 20 and the sub-index key has a value from 1 to 50 when they point to a record in the file. The first record in the file has zero as its last line pointer. This is a special marker for the begining of the file. The last record in the file will have zero for its next line pointer to flag the end of the file.

Word seven contains the format word which stores the formatting codes specified by the user to format the text in the line.

The remaining thirteen words contain the characters of text in the line. For program language text only the

first 80 characters are used. For natural language text the line can contain a maximum of 130 characters of text.

2.3 Addition and Deletion of Lines of Text

2.3.1 Input of Text

Text can be added to the text file from two sources in two different modes. Text can be inputted from a file containing card images (80 character records) or from the CRT terminal. There are two modes of input: program text and natural language text mode. The user selects the mode of input when the file is opened for editing, depending on the type of text to be edited.

In program text mode (i.e. computer programs) the text is stored 80 characters per record or line, corresponding to a card image or an input CRT line.

When editing in natural language text mode the user has control over the maximum percentage (70 to 100 %) of each record (130 characters maximum) to be filled with text from the input source. The remaining space is filled with blanks for future additions to the line. Blanks shifted out of the right end of a line due to additions within the line are ignored. This provides a method to reduce the number of line overflows for small additions to the line (e.g. spelling

corrections, word or small phrase additions). In addition, it reduces the number of lines generated containing only a few characters due to overflow of the previous line. If the input text is likely to be edited or changed frequently the extra space at the end of each line is quite useful. The percentage to be filled can be chosen to suit the particular type of editing to be performed.

Input text is packed into the current record being filled until the maximum number of characters is packed or until the user terminates the addition command. The addition of text is performed such that a word is not split between two lines of text except for the case when a word is longer than the maximum number of characters to be packed per record. This case could occur if the user wishes to enter a very long string of consecutive non-blank characters. An unusual occurrence in most kinds of text.

If the number of characters (excluding trailing blanks) in an input record is less than the maximum record size, all trailing blanks except for one are suppressed at the end of the text and the next input line is added after this blank. This allows the user to end the input card or line at any convenient position, usually after the last word that totally fits on the line, as he naturally would when reaching the end of a typewriter line.

When the input record is full (i.e. no trailing blanks) this is processed as a continuation to the next input line and there are no blanks inserted between this line and the next input line added to the record.

In natural language text mode blank lines on input are ignored. The formatting features control the output of blank lines. Therefore, it is unnecessary that they be stored. Blank lines can be inputted if required by creating a line containing one non-blank character and then delete the character.

An additional method of entering natural language text from a card image file is available for recreating or duplicating text lines. This feature uses a file containing punched output from TEF and recreates the punched lines of text in their original form. Recreation of lines of text must be exact in order for the format codes to operate on the same text as the original. The punched output of TEF maps 130 characters (80 for program text) of each line of text onto 80 character card image records. The above method of entering text performs the reverse map and takes every 130 consecutive characters from a file of 80 character card images and creates a line of text.

This allows storage of a text file on cards or on a sequential disk file (which requires less space) to be

used as a backup or sections of the text file can be duplicated and inserted at arbitrary places in the file. Only the text of the file is recreated with this feature since when duplicating sections or files the user may wish to experiment with different formats for the output of his text. Experimenting with the formatting can be done on a copy of the text file, leaving the original text file intact.

The original formatting information can also be recreated if desired in conjunction with the text. The reformatting will be discussed in Chapter 3.

2.3.2 Record Allocation for input of Text

When text is input a record to store it must be allocated from the available space list and linked onto the doubly linked list structure at the point of insertion. The point of insertion is always after the current line in the file. If the file is empty, the text entered forms the first line in the file. At any given moment there are pointers to the begining of the file (zero if file is empty), top of the available space list (zero if all space is exhausted), current position or line in the file (zero if empty), backward or preceding record or line from the current line (zero if file is empty or positioned at the first line in the file) and forward or succeeding record from the current position (zero

if the file is empty or is positioned at the last line in the file).

The algorithm required to link a new line into the text file is as follows:

1) Save the current line's forward pointer in a temporary location. This is the record following the point of insertion. If the file is empty or the insertion is at the end of the file the forward pointer will be zero.
2) Change the current line's forward pointer to point to the next available record to be used. If the file is empty, there is no current line (i.e. current position pointer is zero) and this step will be omitted.
3) Allocate a record from the available space list and change its backward pointer to point to the current line. If the file is empty the backward pointer is set to zero flagging the begining of the file and the pointer to the begining of the file will be set to this record.
4) Update the current line pointer to point to the newly allocated record. The current line is now the new line just allocated. Pack the text into this record.
5) Update the next available space pointer to point to the forward link of the record just allocated. That is, the the next record in the available space chain will be the next to be used.

6) Change the backward pointer of the record following the point of insertion to point to the last allocated record. If the file was empty or the file was positioned at the end of the file, there is no following record therefore, this step will be omitted.
7) Change the forward pointer in the last allocated record to point to the record following the point of insertion. If the file is empty or the file was positioned at the end of the file, a zero will be inserted to mark the end of the file.

These steps can be better understood by the illustrations in figure 2.3. Figure 2.3 shows the steps required to insert a line (record) between record 2 and record 3. The pointer manipulations are numbered with the corresponding step numbers.

Steps 1 and 2 link the inserted record(s) to any previous records. Steps 3 to 5 allocate a record(s) for the text and steps 6 and 7 link the newly allocated record(s) to any following lines in the file.

If more than one line of text is being added at the same point in the file steps 1 and 2 need only be performed once at the begining of the addition and steps 6 and 7 are done only once at the end of the insertion. Steps 3, 4, and 5 provide the main input loop and are performed for each line of text formed.

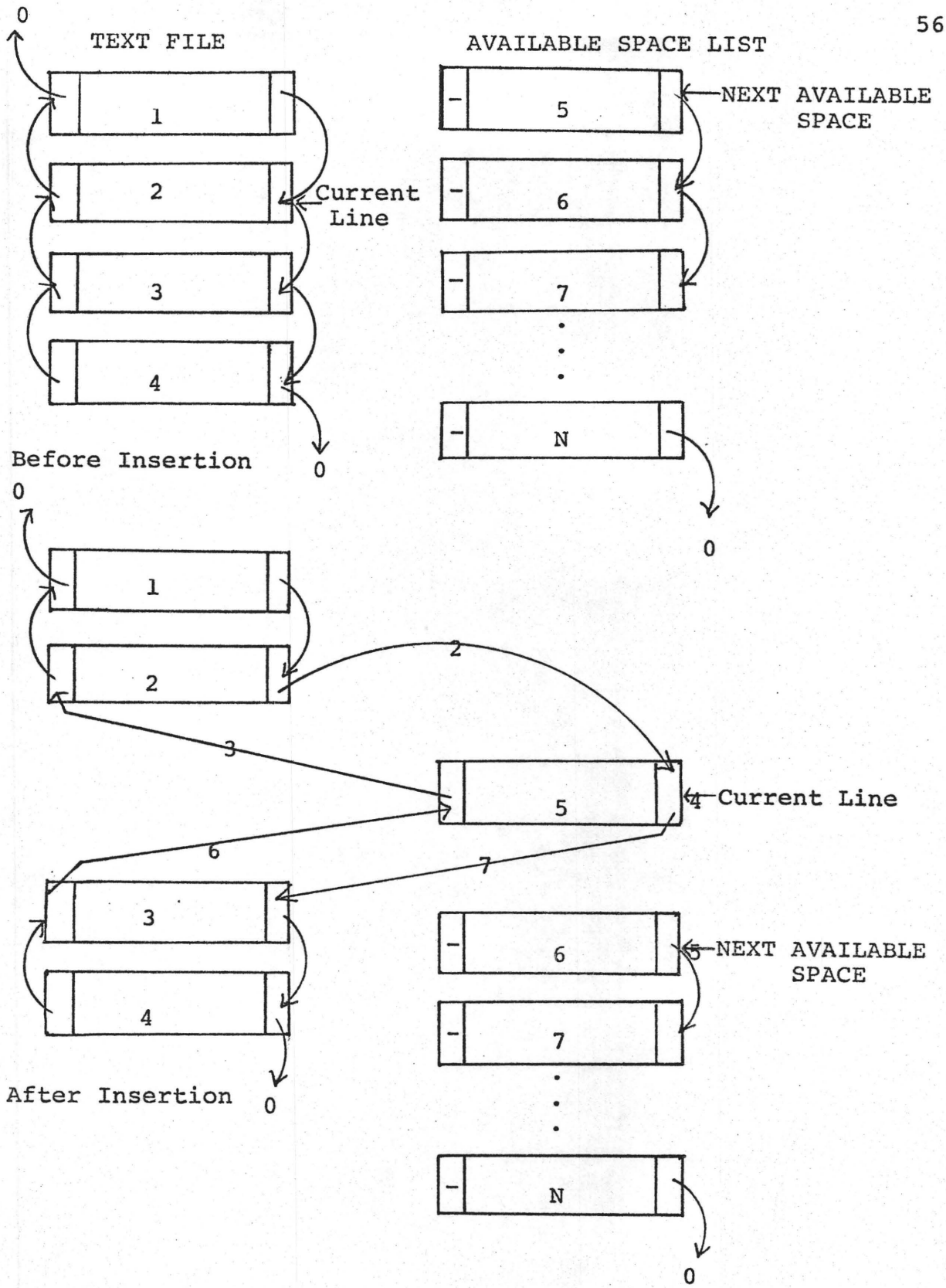


Figure 2.3 Insertion of a Line of Text into the File

A block of records can be taken from the available space list making use of its pointer structure. When a block of several records is being allocated, the forward pointers already point to the required record. For example, if two records (e.g. 6 and 7 in figure 2.3) were to be allocated as a block for input text, the forward pointer of record 6 already points to record 7. During addition of text steps 3 to 5 are repeated for each line inserted and steps 6 and 7 are used to complete the linkage of the block of records to any records following the point of insertion.

This method avoids temporary pointer manipulations when adding more than one line of text to the file. For a large amount of input, such as text read from a permanent file, considerable execution time will be saved. The last added line will become the current line when the addition is complete.

Figure 2.4 illustrates a block of records allocated for text and inserted into the file after the current line.

2.3.3 Deletion of Lines of Text

When a line or lines or text are deleted they must be removed from the file, the file must be joined together again and the deleted record or records added to the available space list. The starting point of a deletion is the current line. Deletions can be performed starting with the

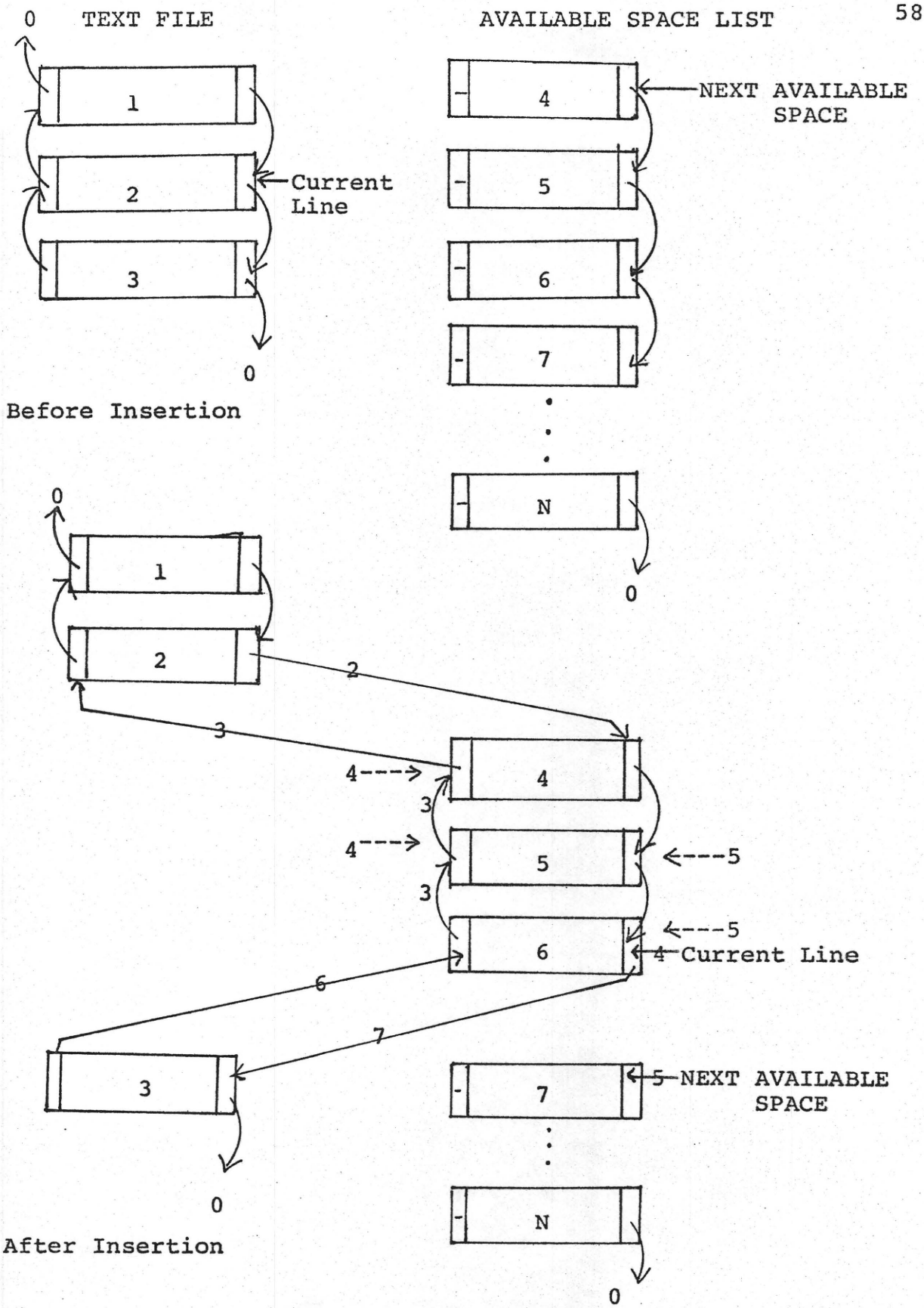


Figure 2.4 Allocation of a Block of Records

current line and moving either forward or backward any number of lines in the file.

There are two types of deletions: backward moving deletions and forward moving deletions. The backward moving deletions start deleting lines with the current line and delete lines moving backwards in the file, making the line before the point of the last deletion the new current line. If the begining of the file is reached during a backward moving deletion the line following the first deleted line will become the current line. If in addition, the deletion started at the end of the file, the whole file is deleted and the file will be empty.

The forward moving deletions start with the current line and delete lines moving forward in the file, making the line after the last deleted line the new current line. If the end of the file is reached on a forward moving deletion the line before the first deleted line becomes the new current line. If in addition, the deletion started from the begining of the file the file will be empty, since all lines will be deleted.

When a deletion causes the whole file to be deleted, the available space list is reorganized into consecutive master and sub-index key order. Frequent additions and deletions to a large file may link together many records with

different master indexes. Changing from one master to another to read or write a record requires 4 mass storage I/O routine calls while only 1 routine call is required if the current master is not changed. Therefore, the fewer times the master must change the better. The ideal situation has each master index and sub-index linked together in consecutive key order, as in the initial file creation run.

Deletion of a line or lines of text requires the following algorithm when moving forward (backward):

1) Save the current position pointer, the backward and forward line pointers.
2) Move forward (backward) to the next (previous) line in the file and make it the new current line. If at the end (beginning) of the file, make the line before (after) the first line deleted the current line. If at the end (beginning) of the file and deletions started at the beginning (end) then the text file is empty.
3) Set the forward pointer of the last (first) line deleted to point to the top of the available space list.
4) Set the next available space pointer to point to first (last) record deleted. The deleted record(s) are now linked onto the top of the available space list.
5) Change the forward (backward) pointer of the line before (after) the first line deleted to point to the current

line.

6) Change the backward (forward) pointer of the current line to point to the line before (after) the first deletion.

The above steps are illustrated in figure 2.5 for a forward deletion of the current line.

Consecutive lines can be deleted in blocks. When more than one line is being deleted, step 2 is repeated for each deleted line and steps 3, 4, 5 and 6 are performed only once at the completion of the deletion. Since the forward pointers of the text file link each record together as in the available space list, the whole block of several deleted records can be put onto the available space list, with only two pointer changes. Steps 3 and 4 add the deleted record(s) to the available space list. Steps 5 and 6 relink the file together where the record(s) were removed.

Figure 2.6 illustrates the forward deletion of several lines at one point.

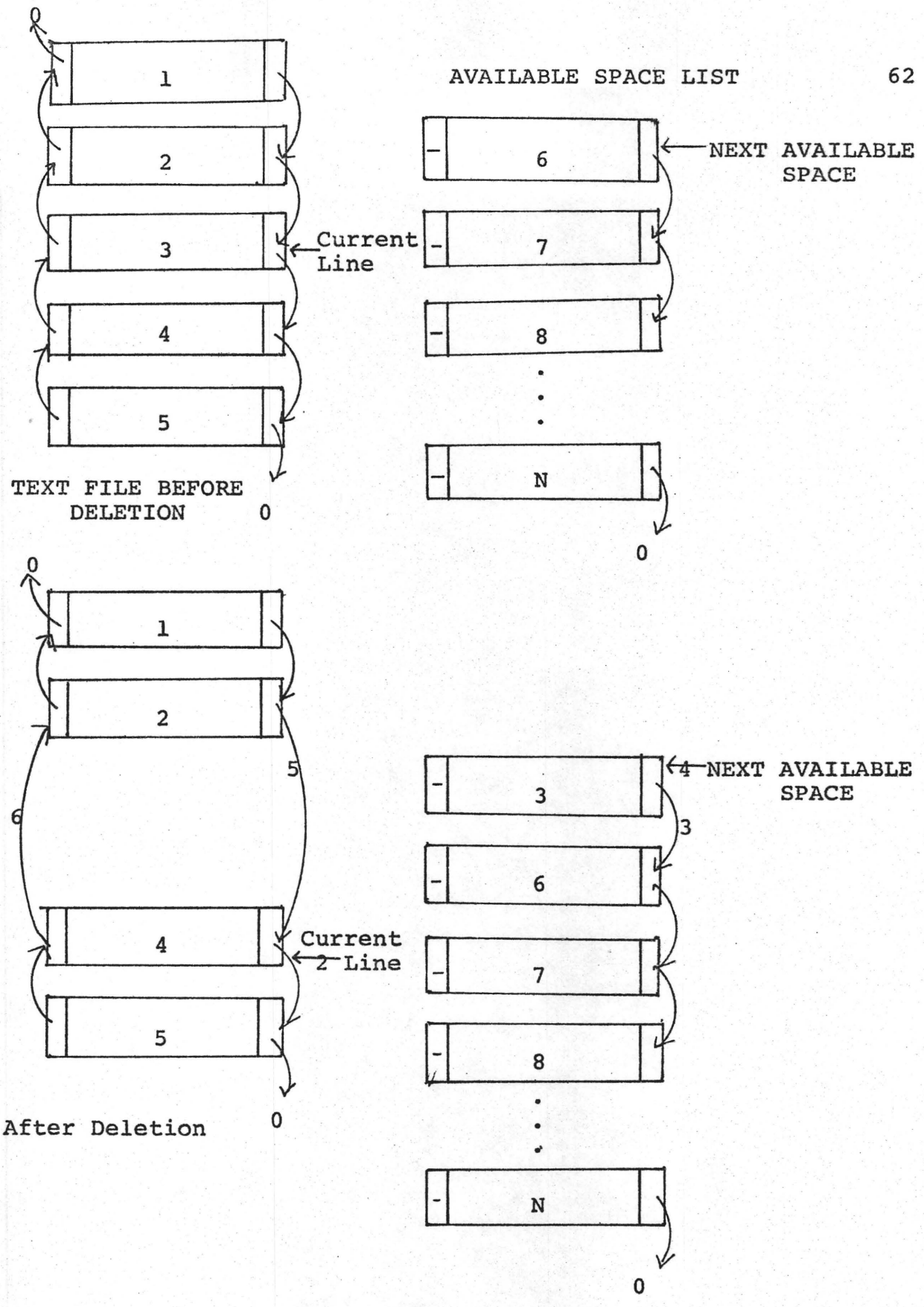


Figure 2.5 Deletion of the Current Line in the File

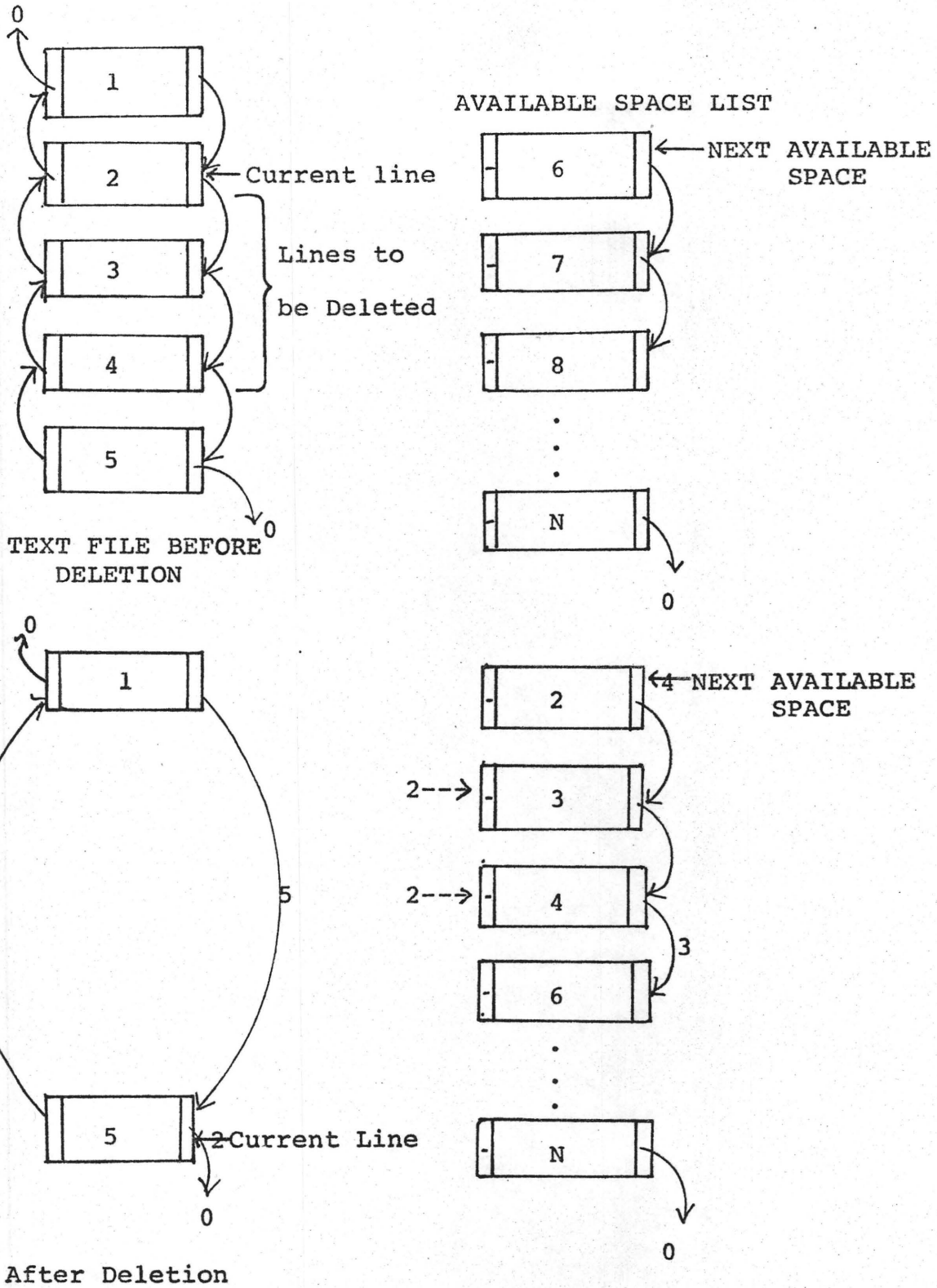


Figure 2.6 Deletion of a Block of Lines (records) (Forward Deletion)

2.4 Implementation of the Substring Test Technique

A content oriented text editor involves a great deal of character searching. Content addressing requires a fast and efficient method of searching for a string of characters within the lines of text in a file. This is extremely important when dealing with large files. Searching each line character by character for the specified string is obviously very slow and inefficient. A method is needed which can determine if one string (search string) is a sub-string of another string (text line) without searching each line. This requirement led to the implementation of the substring test technique which considerably decreases the time required for searching.

2.4.1 The Substring Test Technique

The substring test makes use of a hashing technique and the ability to do operations on the bits of a computer "word". When the same strings are being tested repeatedly, and when the probability of finding the substring is small the substring test technique becomes very useful. It is especially suited to text editing since many lines of text are searched for the occurrence of the same string.

If the search is likely to be unsuccessful, it can usefully be preceded by a computationally faster test for

necessary but not sufficient conditions that the string be found. In some applications, a comparison of the lengths of the strings could be performed. However, this is a weak test for a text editing system because the lines of text and the string to be found will almost always be of different lengths.

A string can be represented by the set of its substrings, and in particular by the set of its substrings of a specific length. In general, such a representation is not unique, but it does preserve the substring property in the sense that, if one string has another string as a substring, the set of substrings of the first will include the set of substrings of the second. Of course, the reverse is not true, because of the lack of uniqueness.

A set S can be represented by a binary string $b_1\ b_2\ b_3\ \ldots\ b_m$ in which a value of one for b_i indicates that S contains at least one element of the set E_i. In general such a representation is not unique unless each E_i contains exactly one element and each possible element is contained in some E_i. However, it preserves the subset property in the sense that, if set S_1 is a subset of the set S_2, the binary string representing S_2 will have ones in all positions where the string representing S_1 has ones. In representing a string in this way a data-object is represented by a simpler data-object which contains less

information but which retains some of the properties of the original.

A string S can be represented by a binary string $b_1\ b_2\ b_3\ \ldots\ b_m$ and is constructed as follows:

1) set all bits b_i to zero
2) for each substring s of fixed length k compute i = HASH(s), and set $b_i = 1$.

The hashing function HASH is assumed to give an integer result in the range 1 to m. A subsequence of k successive elements of a string is referred to as a k-sequence. The resulting binary string, which contains a 1-bit only in positions which correspond to certain k-sequences(E_i) and zero otherwise, is called the hashed k-signature of the string S. A hashed 2-signature is illustrated in the following figure.

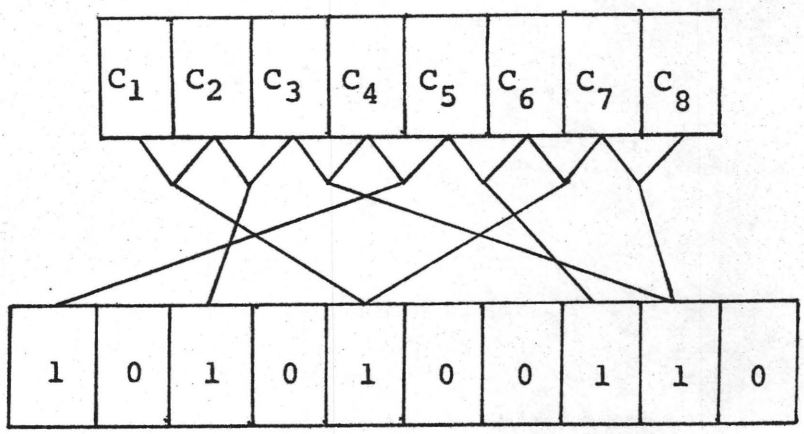


Figure 2.7 10-bit Hashed 2-signature of the string $C_1C_2\ldots C_8$

For any particular hashing function a necessary condition that string S_1 be a substring of string S_2 is that the hashed k-signature of the string S_2 have ones wherever the hashed k-signature of S_1 has ones. It is often convenient to choose m such that the signature fits in a single machine word. In this case the signature test can be implemented in one or two machine instructions. The test can be performed in FORTRAN by:

```
IF ((KSIG1 .AND. (.NOT. KSIG2)) .NE. 0) GOTO 60
IF (.NOT. SUBSTR(S1,S2)) GOTO 60
```

where SUBSTR is the rigorous character by character search of the line and control is transferred to statement 60 if the substring is not found.

The .NOT. operator negates the signature making each 1 bit zero and each zero bit one. Therefore, the expression (.NOT. KSIG2) has ones only in positions which were not set to one by any k-sequence of the search string. The binary string KSIG1 contains ones in all positions which correspond to certain k-sequences that are in the string to be found. A bit by bit logical AND is performed on these two binary strings. If KSIG1 contains a one (k-sequence present) in any position where (.NOT. KSIG2) has a one (k-sequence not present) the result will be non-zero and the text fails. Everyplace KSIG1 has a one bit, (.NOT. KSIG2) must have a

zero in order for the test to succeed, since the result of the AND will be zero.

When implementing a signature test which consists of more than one word (as in TEF) additional tests are required to compare the corresponding word of each signature of the strings. For a two word signature (TEF) the test can be implemented in FORTRAN as:

```
IF ((ISIG1 .AND. (.NOT. KHASH1)) .NE. 0) GOTO 60
IF ((ISIG2 .AND. (.NOT. KHASH2)) .NE. 0) GOTO 60
IF (.NOT. SUBSTR(S1,S2)) GOTO 60
```

where ISIG1 and ISIG2 represent the two words of the signature for the string to search for and KHASH1 and KHASH2 represent the two words of the signature of the string to be searched (i.e. line of text). The majority of cases would be rejected by the first test on the first word of the signature.

2.4.2 Choosing Parameters for the Substring Test

Three parameters must be chosen to implement the substring test technique: the length of the hashed signature (m), the hashing function, and the length of the substrings to be taken (k).

While it is not necessary, it is convenient to choose m such that it is a multiple of the number of bits in a machine word. Using partial words will have no affect on the substring

test since the remaining bits will be set initially to zero and will never change. Clearly the larger m is, the more accurate the results will be since fewer strings will be incorrectly identified as substrings by the signature test. The more time required for the rigorous test, the more worthwhile an improvement in the signature test becomes. For test editing applications where the length and number of lines to search is large, this is an important consideration. In addition, response time in an interactive editing system will suffer considerably as the search time increases.

The amount of accuracy which is worth achieving also depends on the probability that the string is not a substring. It is not economical implementing a signature which makes only 2 percent errors instead of 5 percent, if 30 percent of the strings are in fact substrings. However, if only 1 percent of the strings are substrings, such an improvement could well be beneficial. This probability can only be estimated since in a text editing system the text and search strings can vary widely.

The hashing function should be random and generate an integer result between 1 and m. A suitable random function will spread the k-sequences being hashed evenly over the range 1 to m so that the probability of a particular bit being set by a particular k-sequence is the same for each bit position.

The value of k can also be chosen. Clearly if k is 1, no information is included about the order of the characters. Thus, k should be two or more. If there are N different symbols in the alphabet, there are N^k possible k-sequences. Therefore, it does not make sense to choose k so small that N^k is less than m, since there are only N^k distinct substrings of length k and bits in the signature will be wasted. For text editing applications where N is relatively large (64 to 256) this is not a problem. On the other hand, if k is chosen too large, the number of bits in the signature can become too small, and in fact will be zero if k is larger than the length of the string. When searches for small strings (e.g. 2 or 3 character strings) will be performed k should be chosen small enough to allow a non-zero hashed k-signature. If there are n symbols in the string, there will be n - k + 1, k-sequences. Therefore, to be able to search for a string of 2 characters k must be 2.

Maximum information content corresponds to having about half the bits in the signature zero.

2.4.3 Probability of a False Match

It is possible to develop an expression for the probability of a false match. That is, the probability that a random string of length L_1 will be identified as a substring

of a string of length L_2 by using the signature test.

If we assume that the hashing function is random, and we have random strings, the probability of a particular bit in an m-bit hash signature being set by a particular element of a set is 1/m. The probability that this bit is not set is 1 - 1/m. The probability that this bit is not set by any element of an n-element set is $(1 - 1/m)^n$. Therefore, the probability that a particular bit is set is

$$1 - (1 - 1/m)^n.$$

This is the density of ones in the hash signature which should be approximately 1/2 for maximum information content.

If we are testing to see if the string of characters S_1 is a substring of another string of characters S_2, the probability of the signature test giving an affirmative answer can be estimated. If we assume that S_1 and S_2 are decomposed into L_1 and L_2 segments respectively, the probability $p(L_1,L_2,m)$ that each of the segments of S_1 will hash onto bits already turned on in the hash signature of S_2 can be estimated by

$$p(L_1,L_2,m) = (1 - (1 - 1/m)^{L2})^{L1}$$

As $(1 - 1/m)^m \approx 1/e$, $p(L_1,L_2,m)$ can be approximated by

$$p(L_1,L_2,m) = (1 - e^{-L2/m})^{L1}.$$

Using this approximation Harrison's symmetry relation

[HAR 71] , $p(L_1,L_2,2m) = p^2(L_1/2,L_2/2,m)$ shows the effect of doubling the length of the hashed k-signature. This probability is only an estimation since most text is not entirely random and not all sets of possible substrings correspond to strings (e.g. ;:, =), :., etc. are unlikely combinations in any kind of text). A frequency analysis of different natural language texts or computer languages will show very different trends in the most common character sequences.

The probability of a false match will increase for search strings containing frequently occurring character sequences. The length of the search string can be increased to improve the performance for commonly occurring character combinations.

2.4.4 Implementation of the Substring Test

The substring test is implemented to decrease the search time for content searches in TEF. The first parameter which must be chosen is the length of the hashed signature (m). As the word size on the CDC 6400 computer is 60 bits, a reasonable choice would be multiple of 60 bits. Clearly the larger the value of m the more accurate the results.

The probability estimated in the previous section is tabulated in table 2.1 for m = 60 and in table 2.2 for m = 120. L_1 and L_2 are the number of segments into which

L_2							
192.0	.88563	.78434	.61518	.37845	.14322	.02051	.00042
96.0	.51355	.26374	.06956	.00484	.00002	.00000	.00000
48.0	.16975	.02881	.00083	.00000	.00000	.00000	.00000
24.0	.03657	.00134	.00000	.00000	.00000	.00000	.00000
12.0	.00609	.00004	.00000	.00000	.00000	.00000	.00000
6.0	.00088	.00000	.00000	.00000	.00000	.00000	.00000
3.0	.00012	.00000	.00000	.00000	.00000	.00000	.00000
	3.0	6.0	12.0	24.0	48.0	96.0	192.0 L_1

Table 2.1 The probability that a random string of length $L_1 + K - 1$ will be identified as a substring of a string of length $L_2 + K - 1$ using 60-bit K-signatures

L_2							
192.0	.51095	.26107	.06816	.00465	.00002	.00000	.00000
96.0	.16836	.02834	.00080	.00000	.00000	.00000	.00000
48.0	.03620	.00131	.00000	.00000	.00000	.00000	.00000
24.0	.00602	.00004	.00000	.00000	.00000	.00000	.00000
12.0	.00087	.00000	.00000	.00000	.00000	.00000	.00000
6.0	.00012	.00000	.00000	.00000	.00000	.00000	.00000
3.0	.00002	.00000	.00000	.00000	.00000	.00000	.00000
	3.0	6.0	12.0	24.0	48.0	96.0	192.0 L_1

Table 2.2 The probability that a random string of length $L_1 + K - 1$ will be identified as a substring of a string of length $L_2 + K - 1$ using 120-bit K-signatures.

the strings S_1 and S_2 are decomposed.

Since a string of length n will contain n - k + 1, k-sequences, the tables are the probability that a random string of length L_1 + k - 1 will be identified as a substring of a string of length L_2 + k - 1. Comparing the results shows that when L_1 and L_2 are large there is not much difference in the performance of the two values of m. When L_1 is small (e.g. 3 segments)and L_2 is large (e.g. 96 segments) the results are considerably better for the larger hash signature (i.e. m = 120).

For example, the probability that a 4-character string being identified as a substring of a 97-character string is about 17 percent for m = 120 and about 51 percent using m = 60 with 2-signatures being taken (k = 2). This implies that at least 83 percent of non-substrings will be rejected by the signature test for m = 120 and 49 percent will be rejected for m = 60. An important consideration for TEF is that the signature test function well for small strings (between 2 and 20 characters).

The user should be able to specify small search strings and still obtain fast response time. In TEF the majority of tests will occur with L_1 small (maximum 19) and L_2 large (maximum 129), therefore m = 120 is a worthwhile choice. The two word hash signature involves an additional test to test

both words of the signature. Most text lines will be rejected by the first test of the first word of the signature.

The value for k in TEF is two. This allows search strings as small as two characters and the corresponding hash signatures to contain more information than for larger k.

The hashing function operates on the display code for each character. Since each character is stored right justified with zero fill, each character can be treated as an integer number equivalent to its display code ($00 - 77_8$). The hashing function hashes every two consecutive characters of the string into a bit position (1 - 120) numbered right to left in the word.

The hashing function in FORTRAN is

```
KHASH(ICH1,ICH2,K) = MOD(ICH1*K + ICH2,120) + 1
```

where ICH1 is the first character and ICH2 the second. K is a constant equal to 9. The MOD function returns the remainder when its first argument is divided by its second.

The hashing function returns an integer value in the range 1 to 120. Different values of K were tested for all possible two character combinations of the 26 letters of the alphabet plus blank (729 2-sequences). K = 9 produced the best and most random results mapping between 5 and 7 pairs of characters onto each bit position. Other values for K mapped as many as 26 character combinations onto one bit

position and many bit positions were wasted since no combinations hashed to them.

Within each record of the TEF text file is stored the two word hashed 2-signature corresponding to the text contained in the line. Each time a line is added to the file or altered the hashing function is applied to the text in the line to compute its 2-signature. To decrease the execution time and increase the efficiency the hashed 2-signature is computed partly in assembly language.

Content searching in TEF allows the user to search for a string of characters from 2 to 20 characters long in a file containing lines of 130 characters maximum. The probability that a string of length $L_1 + 1$ will be identified as a substring of a string of length $L_2 + 1$ using 120 bit 2-signatures is tabulated in table 2.3. L_1 ranges from 1 to 19 which corresponds to strings of length 2 to 20 characters (a string of length n decomposes into n - 1, 2-sequences). L_2 ranges from 9 to 129 elements which corresponds to strings of length 10 to 130 characters.

Clearly as the length of L_1 increases the more accurate the results. Taking the extreme cases, the probability of a 20-character string being identified as a substring of a 130-character string is less than .04 percent. This implies that at least 99.96 percent of non-substrings will be rejected by

L_2							
129.0	.66024	.43591	.28780	.19002	.02384	.00299	.00038
109.0	.59834	.35801	.21421	.12817	.00983	.00075	.00006
89.0	.52516	.27579	.14483	.07606	.00304	.00012	.00000
69.0	.43865	.19241	.08440	.03702	.00060	.00001	.00000
49.0	.33638	.11315	.03806	.01280	.00006	.00000	.00000
29.0	.21548	.04643	.01000	.00216	.00000	.00000	.00000
9.0	.07255	.00526	.00038	.00003	.00000	.00000	.00000
	1.0	2.0	3.0	4.0	9.0	14.0	19.0 L_1

Table 2.3 The probability that a random string of length $L_1 + K - 1$ will be identified as a substring of a string of length $L_2 + K - 1$ using 120-bit signatures. L_2 represents the line length in TEF (10 to 130 characters) and L_1 represents the length of the string to be found (2 to 20 characters in TEF).

the signature test. On the other hand, the probability of a 2-character string being identified as a substring of a 130-character string is about 66 percent. This indicates that approximately 34 percent of non-substrings will be rejected by the test. The performance of the substring test will depend on the length of the search string and on the frequency of its character combinations in the text.

When a string L_1 is to be searched in a line of text L_2, the hashed 2-signature for L_1 is computed and compared as outlined in section 2.4.1 with the signature of each line. When the substring test succeeds the line is searched character by character for the occurrence of L_1. If it is found searching stops, otherwise the next line in the file is tested. If the signature test fails, the next line will be tested. The search will terminate when the required string is found or when the end of the file is reached. The user may specify which columns of each line to search.

This technique performs extremely well for this type of application. There are clearly many other similar applications. The hashing function and the representation of an ordered set by its set of k-sequences are information compression functions which preserve as much of the relevant information on a data-object as possible. Computers and man must process increasingly large volumes of data: therefore, information

compression of this type is very important in order to save both computer execution time and user response time.

CHAPTER 3

TEXT FORMATTING

3.1 Introduction

The imaginative use of computers for on-line composition and extensive manipulation of natural language text has expanded rapidly into the area of structuring and formatting of text.

Computer-assisted typesetting, printing and phototypesetting have become popular techniques for producing books, manuals, reports, form letters, etc. quickly and economically.

The quality of the formatted output is limited only by the available hardware devices that can be connected to the computer. Computers can be employed to operate photocomposition devices, typesetting machines [BAR 69] and typewriters.

This chapter will discuss the operation of the TEF formatter with examples of some of its features. At present, the quality of output from TEF is limited due to the available hardware devices at McMaster University. At the time of implementation only line printers were available for output. The character set only supported upper case characters

thus, restricting TEF's use for formal documents requiring upper and lower case alphabetic characters.

Output can be printed on a Versatec Electrostatic printer using 8 1/2 X 11 inch pages or on a standard impact line printer with 8 1/2 X 15 or 11 X 15 inch pages.

3.2 Basic Formatting Concepts

This section will describe the terminology used throughout the chapter.

The formatting information is inserted by the user into the "format word" of a line of text and the text file can be formatted according to these specifications.

The format word consists of 10 character positions (1 computer word) divided into 3 fields as follows:

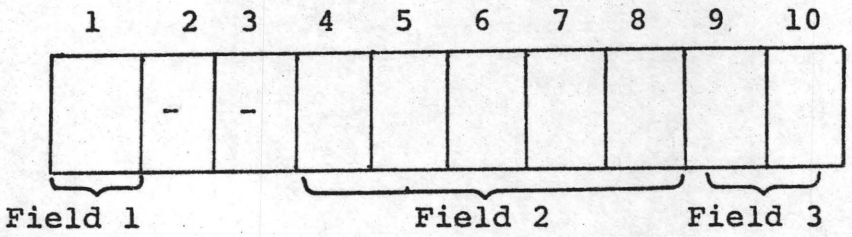


Figure 3.1 The Format Word

Fields 2 and 3 are used for "format codes" and occupy the right most 7 character positions of the format word. Field 1 is not used for formatting, but is used for sub-section markers. It serves as an indication to the user indicating

what lines of his text have been defined as the start of a sub-section in the file.

Character positions 2 and 3 are unused at present and may be used for future extensions to TEF. Each format code consists of two letters. Some format codes are followed by a maximum three digit integer number. Format codes requiring a numeric quantity are inserted into field 2 (5 character positions) while field 3 (2 character positions) contains the two letter codes.

When fields 2 and 3 both contain format codes, field 2 is processed before field 3. Therefore, each line of text can contain two formatting operations to be performed.

The input to the formatter consists of text and format codes. A format code is one of several well defined strings which may be stored in the format word of a line of text. The format codes operate on the text in the line, select or turn off a particular feature, or change formatting parameters. A line which does not contain any formatting information is defaulted to left and right justification since the majority of natural language text will require justification upon output.

The text consists of the lines of text in the file to be formatted. Each line of text is composed of words. Words are strings of characters delimited by one or more

blanks or by the end of the input record (line). The input line may contain from 0 to 130 characters. A blank line is considered as a line containing zero characters. Only the format codes of a blank line are processed. This allows the specification of a format which cannot be achieved by using only one format word.

The formatter within TEF may operate in either of two modes: "formatted" and "non-formatted". In formatted mode words are delimited by the end of the text line and one or more blanks. Words are shifted to fill the output line as much as possible. Right justification is performed by separating words by more than one blank. Additional blanks added between words are spread symmetrically throughout the line. This method avoids blank "streaks" in a page of text.

In non-formatted mode, a word is delimited by the blank following the last non-blank character or the end of the input record. In non-formatted mode blank characters are considered in the same way as any other character. Therefore, only one word is printed per line containing a maximum of 130 characters. Non-formatted mode is used when a format is desired that cannot be achieved in formatted mode, or when insertion of extra blanks will change the intended meaning of the text (i.e. when embedded blanks are significant).

During formatting, output text lines are formed starting at the "left margin". The left margin is the print

column position where the first character of the line is to be placed. The width of the left margin is equal to the left margin minus two. The first character position in an output line is reserved for printer carriage control. A left margin can be in the range 2 to 120.

Each line of text in formatted mode will contain LINWD characters after justification. LINWD is the number of characters to be placed in each line. The right margin is the position where the last character of each line is placed. The width of the right margin is determined by the left margin width plus LINWD characters. The width of the right margin is dependent upon the size of the output page. The maximum size of an output line is 130 characters including the blanks in the left margin.

Each page of output contains a header which occupies the first 7 lines of every page. The header consists of 3 blank lines, the header line, and 3 more blank lines. The header line may contain a running heading and/or a page number. The running heading may be a maximum of 60 characters long. Either the running heading and/or the page number may be omitted. When both are omitted the header line is printed as a blank line. The running heading is initially blank and page numbering is selected if desired before formatting begins.

The page numbering is controlled by the current page

number. It is initially one and is incremented by one after each page header is printed if page numbering is selected. The user may change the value of the current page number and the contents of the current running heading dynamically during formatting.

Following the header is the "text body" which occupies the remainder of the page. The text body consists of lines of text printed in either formatted or non-formatted mode.

The spacing between lines is controlled by the spacing parameter. Single or double spacing may be selected before formatting begins and can be changed dynamically during editing. Additional blank lines may be output using format codes.

The number of lines per page is controlled by the page depth parameter (NLPAG). The page depth counts all lines on the page including the seven header lines. Page depth is selected before formatting begins and causes a new page to be started after NLPAG lines have been printed on a page. A large number chosen for NLPAG will effectively eliminate paging control. The user may override this feature and start a new page at any time during formatting.

3.3 The Formatting of Text

The input to the formatter within TEF consists of the lines of text within the text file and their associated format

codes. The format codes are processed and applied to the line (if applicable) with the formatted output lines written onto an output file.

The default mode of output is the formatted mode with left and right justification for each line. The format codes allow the user the ability to alter the specifications of this mode (e.g. margins, indentations, page format, etc.) or to output a line in non-formatted mode.

In TEF there is an "output line buffer", from where all output text, excluding the header and blank lines is written to the output file. During formatting, the buffer contains the current output line being formed. The source of input to the line buffer is the "current input line". The current input line is the line of the file currently being processed. Depending upon its associated format code, text from the input line will either be added to the current line being formed, or the current line will be printed and the input text will be used to form a new line. The current output line is printed when it is full or when a format code is encountered that causes the current line to be printed before being processed. The line is printed according to the current margins.

The length of the buffer is 131 characters. The first character position is reserved for printer carriage control.

The value of the carriage control is determined by the spacing parameter selected by the user. The length of the buffer allows lines of up to 130 characters to be formed on output. The left margin corresponds to a character position in this buffer and the first character of an output line is placed in this character position. The text can be shifted within the buffer by adjusting the margin width.

In formatted mode, text is added to the buffer and justified so that each line contains the required number of characters per line (LINWD). Each line of text is treated as a continuation of the previous line separated by one blank. Text is added to the current output line from the current input line. The line is printed when it is filled or a "break" occurs. Some format codes ("break codes") when encountered in the input stream, terminate the addition of text to the current output line. When a break occurs the current line is written onto the output file before the format code is executed. A new output line is then started, the format code executed and the text of the line processed.

A line of text can be output in non-formatted mode by specifying the non-formatted code in the format word for the line. In this case, the current line (if any) is printed and the input line of text is printed on the next line exactly as it was read. A new output line is started after the non-

formatted line is printed. A group of lines to be output in non-formatted mode must each contain the non-formatted code.

The following is a discussion of how the format codes operate during formatting. All the format codes in field 2 of the format word are break codes, except for the print blanks after processing the line code (BA).

The format codes in field 2 are of the form XXNNN where XX is a two letter format code and N is a numeric digit. The codes in field 2 provide a means to adjust left and right margins, print blank lines before or after a line of text, start a new page, and change the current page number.

The left margin may be moved left or right, thus decreasing or increasing the left margin width. A shift of the left margin to the left (LL) (right (LR)) code causes the current line formed to be output and then the left margin is moved left (right) NNN character positions. The text of the line is then used to form an output line with the first character placed in the character position of the new left margin. The LL (LR) format code increases (decreases) the length of the output line. These two format codes operate in the same manner as adjusting the left margin on a typewriter.

During formatting the width of the right margin may be changed by moving the right margin left or right. A shift of the right margin left (RL) (right (RR)) code will cause the

current line to be printed and the right margin moved left (right) NNN character positions. When the right margin is moved left (right) the width of each line (i.e. the number of characters per line) is decreased (increased) by NNN characters, effectively increasing (decreasing) the width of the right margin. The text is then processed. The RL and RR format codes operate in the same way as adjusting the right margin on a typewriter.

The LL, LR, RL and RR format codes allow the user the ability to strictly and dynamically control the line width and the margin widths of the output in a manner analogous to manually adjusting the margins on a typewriter.

The output of blank lines is regulated by the format codes for printing blank lines before (BB) or after (BA) processing the text in the line. The BB format code terminates and prints the current line, prints NNN blank lines and then processes the text in the line. The BA code processes and outputs the line of text according to the current mode and prints NNN blank lines after the line is output. These codes may be used for block paragraphing, spacing for diagrams or figures, or for separation of sections of text. They provide a means to output blank lines without storing them in the file and thus, wasting space that could be used for the user's text.

A useful feature of TEF is the margin delay code (MD).

Upon encountering an MD code the current line is printed and a new line is started. The MD code delays moving of the left margin right NNN character positions until one output line has been printed from the input line of text. For example, a frequently used format is of the following form:

MD code - After this line is printed the left margin is moved right NNN character positions or spaces (e.g. 10 spaces here).

Note that at this point after all the required text is printed the left margin must be reset to its original position (i.e. move the left margin left 10 spaces). Figure 3.2 contains a page of output employing the MD feature.

The new page format code (NP) permits the user to control the paging of his text. When the NP format code is encountered the current output line is terminated and output onto the current page and a new page is started. If the number (NNN) following the NP code is non-zero the current page number is set to this number. Else the current page number is used. The header is printed on the top of the page and may be blank. The text of the line is then processed. The user can strictly control the amount of text on each page by overriding the page depth parameter with the NP code.

The NP code is also useful for reformatting selected page numbers or for formatting sections of text with non-

BASIC TERMS AND NOTATION USED

LINE OR RECORD - THESE TWO TERMS WILL BE USED INTERCHANGEABLY DURING THE DISCUSSION AS THEY CAN BE VIEWED FROM THE USERS POINT OF VIEW AS REPRESENTING ONE STRING OF MAXIMUM LENGTH 130.

FORMAT WORD - THE WORD CONTAINED IN EACH RECORD WHICH STORES THE FORMATTING CODES TO SPECIFY THE DESIRED FORMATTING FOR THAT LINE OF TEXT. IT CONSISTS OF 10 CHARACTERS OF TEXT DIVIDED INTO 3 FIELDS. FIELD 1 CONSISTS OF CHARACTER POSITION 1, FIELD 2 CONSISTS OF CHARACTER POSITIONS 4 - 8 AND FIELD 3 CONSISTS OF CHARACTER POSITIONS 9 AND 10, WHERE THE CHARACTERS IN THE WORD ARE NUMBERED FROM LEFT TO RIGHT. CHARACTER POSITIONS 2 AND 3 ARE UNUSED AT PRESENT.

FORMAT OF DISPLAYED LINES - WHEN A LINE OF TEXT IS DISPLAYED ON THE CRT SCREEN THE 130 CHARACTERS OF TEXT ARE SPLIT INTO TWO LINES WITH 70 CHARACTERS ON THE FIRST LINE AND 60 CHARACTERS ON THE SECOND. FOLLOWING THE 60 CHARACTERS ON THE SECOND LINE IS THE 10 CHARACTERS OF THE FORMAT WORD BETWEEN TWO SLASHES.

Figure 3.2 The Margin Delay Feature

consecutive page numbers.

Field 3 contains format codes consisting of 2 letters without a numeric quantity associated with them. The break codes in field 3 are the paragraph indentation code(IN), the centre text code(CT), the non-formatted output code(NF) and the spacing control codes SS (single spacing) and DS (double spacing).

The paragraph indentation code (IN) causes a new paragraph to be started with the text in the line indented five spaces from the left margin. When the IN code is encountered the current output line is output and a new line of text is started with the left margin moved right five spaces for the first output line formed. The left margin is reset after this line is printed and before the remaining text in the line is output.

Text can be centered between the left and right margins for headings and titles. The centre text (CT) format code writes the current output line onto the output file and then processes the text to be centered. Leading and trailing blanks are trimmed from the input line, the resulting text is centered between the left and right margin and written to the output file.

The output of lines of text in non-formatted mode is performed by the non-formatted output code (NF). When the NF

code is applied to an input line of text the current output line is printed and the input line of text is printed in non-formatted mode. The text is output exactly as it was inputted with the first character of the output line starting at the left margin. After printing the line a new output line will be started with the text of the next input line read. This feature is useful for rigidly formatted output such as tables, lists, etc. or for output of character strings where embedded blanks are significant.

Running headings may be defined, changed and cleared dynamically during formatting. The running heading format code (RH) allows the user the ability to define a string of characters as the current running heading to be used in the header line. The RH code trims off leading and trailing blanks from the text of the line and the resulting text string is used as the new running heading. The heading may be a maximum of 60 characters long. The running heading is centred between the left and right margin and is printed at the top of each page.

The clear heading format code (CH) clears the current heading to blanks, effectively deleting the running heading.

Spacing is controlled by the single space (SS) and the double space (DS) format codes. The SS code changes to single spacing and the DS code changes to double spacing after

the current line is output. The text of the line is then processed. Spacing will remain the same until changed by another DS or SS format code. Before formatting begins the user may specify either single or double spacing.

Page numbering is determined by the PO and PY format codes. The PO code turns page numbering off so that page numbers are not printed at the top of each successive page. The PY code selects page numbering for each page and the current page number is incremented by one after each page is output. When page numbering is turned off the current page number remains at the next consecutive page number to be used and does not change until page numbering is turned on again or the user changes it with a NP code. The page number is printed above the right margin three lines from the top of the page.

In the above discussion, I have outlined the operation of each format code in field 2 and 3 acting alone on a line of text. Format codes may be contained in both fields for a line of text. In this case, field 2 is processed before field 3 and then the text is processed.

The user should be aware of the order of execution when specifying format codes. For example, when specifying the new page code and the running heading code together, the new page is started and the header printed before the current heading is changed by the RH code.

The complete text file may be formatted or only sections of it. Formatting will terminate at the end of the file or when a line containing the number 999 in field 2 of its format word is encountered. Formatting may begin at the first line in the file or at the current line.

The remaining pages in this chapter contain examples of formatted output produced by TEF and a section on the reformatting of a text file or sections of it.

TEF USERS MANUAL

INTRODUCTION :

TEF IS AN INTERACTIVE TEXT EDITOR AND FORMATTER FOR CREATING, MODIFYING AND FORMATTING TEXT USING DIRECTIVES PROVIDED BY THE USER AT A TERMINAL. TEF IS IMPLEMENTED ON A CONTROL DATA 6400 COMPUTER AND RUNS INTERACTIVELY UNDER INTERCOM VERSION 4.3, WHICH PROVIDES TIME SHARING ACCESS TO THE COMPUTER, OR IT CAN BE EXECUTED IN BATCH MODE. THE SOURCE LANGUAGE IS MOSTLY FORTRAN WITH A FEW COMPASS ASSEMBLY LANGUAGE ROUTINES.

TEF CAN MANIPULATE VARIOUS KINDS OF TEXT INCLUDING PROGRAMMING LANGUAGE TEXT AND NATURAL LANGUAGE TEXT (E.G. ENGLISH TEXT). HOWEVER, IT WAS DESIGNED PRIMARILY FOR THE MANIPULATION OF NATURAL LANGUAGE TEXT. THE COMMAND AND FILE STRUCTURE WAS CHOSEN TO FACILITATE THIS TYPE OF TEXT MANIPULATION.

THE PRIMARY OBJECTIVE OF THIS PROJECT WAS A TEXT EDITOR AND FORMATTER TO BE USED FOR THE PREPARATION AND PRINTING OF REPORTS, MANUALS, ROUGH DRAFTS OF MANUSCRIPTS, AND OTHER DOCUMENTS IN WHICH MANY REVISIONS ARE NECESSARY.

ONE OF THE BASIC CHARACTERISTICS OF TEF IS CONTENT ADDRESSING. CONTENT ADDRESSING ALLOWS THE USER TO SELECT LINES OF TEXT BY THEIR CONTENT, RATHER THAN BY LINE NUMBER. THIS IS A MORE NATURAL WAY OF ADDRESSING ENGLISH TEXT AND THE USER NEED NOT BE CONCERNED WITH RESEQUENCING OF LINE NUMBERS, OR SPECIFYING

Figure 3.3 Double Spaced Page of Output

BASIC TERMS AND NOTATION USED

+ - DENOTES FORWARD MOTION IN THE TEXT FILE.

- - DENOTES BACKWARD MOTION IN THE TEXT FILE.

* - REPRESENTS A STRING DELIMITER AND CAN BE ANY NON-ALPHABETIC CHARACTER NOT CONTAINED IN THE STRING IT IS ENCLOSING. A SLASH (/) IS THE MOST COMMON DELIMITER USED.

STRING - IS A STRING OF CHARACTERS AVAILABLE IN THE CHARACTER SET OF THE PARTICULAR MACHINE BEING USED.

NNN - A STRING OF FROM 1 - 3 NUMERIC DIGITS.

[ ] - OPTIONAL FORMAT OF COMMAND.

CURRENT LINE - IS THE LINE OF TEXT AT WHICH THE FILE IS POSITIONED. THIS LINE IS DISPLAYED WHENEVER IT CHANGES DURING THE EXECUTION OF AN INSTRUCTION. ALL TEXT EDITING IS PERFORMED ON THE CURRENT LINE.

WORD - IS A CONSECUTIVE STRING OF NON-BLANK CHARACTERS.

LINE OR RECORD - THESE TWO TERMS WILL BE USED INTERCHANGEABLY DURING THE DISCUSSION AS THEY CAN BE VIEWED FROM THE USERS POINT OF VIEW AS REPRESENTING ONE STRING OF MAXIMUM LENGTH 130.

FORMAT WORD - THE WORD CONTAINED IN EACH RECORD WHICH STORES THE FORMATTING CODES TO SPECIFY THE DESIRED FORMATTING FOR THAT LINE OF TEXT. IT CONSISTS OF 10 CHARACTERS OF TEXT DIVIDED INTO 3 FIELDS. FIELD 1 CONSISTS OF CHARACTER POSITION 1, FIELD 2 CONSISTS OF CHARACTER POSITIONS 4 - 8 AND FIELD 3 CONSISTS OF CHARACTER POSITIONS 9 AND 10, WHERE THE CHARACTERS IN THE WORD ARE NUMBERED FROM LEFT TO RIGHT. CHARACTER POSITIONS 2 AND 3 ARE UNUSED AT PRESENT.

FORMAT OF DISPLAYED LINES - WHEN A LINE OF TEXT IS DISPLAYED ON THE CRT SCREEN THE 130 CHARACTERS OF TEXT ARE SPLIT INTO TWO LINES WITH 70 CHARACTERS ON THE FIRST LINE AND 60 CHARACTERS ON THE SECOND. FOLLOWING THE 60 CHARACTERS ON THE SECOND LINE IS THE 10 CHARACTERS OF THE FORMAT WORD BETWEEN TWO SLASHES.

Figure 3.4 Single Spacing Using Margin Delay Feature

THE FORMATTING OF TEXT

SPECIFYING THE FORMATTING INFORMATION DURING EDITING IS PERFORMED USING THE /FORMAT COMMAND AND THE TEXT IS FORMATTED ACCORDING TO THESE SPECIFICATIONS BY THE /FLIST COMMAND. IF A LINE DOES NOT CONTAIN ANY FORMATTING INFORMATION IT IS DEFAULTED TO RIGHT JUSTIFICATION SINCE THE MAJORITY OF TEXT FORMATTING WILL REQUIRE JUSTIFICATION ON OUTPUT.

EACH LINE OF TEXT IS TREATED AS A CONTINUATION OF THE PREVIOUS LINE SEPARATED BY ONE BLANK UNLESS A BREAK OCCURS. CERTAIN FORMAT CODES CAUSE A BREAK TO OCCUR WHEN FILLING AN OUTPUT LINE. WHEN A BREAK OCCURS THE CURRENT OUTPUT LINE BEING FORMED IS ENDED AND OUTPUTTED BEFORE THE FORMAT CODE IS EXECUTED AND A NEW OUTPUT LINE IS STARTED WITH THIS NEW LINE OF TEXT. ALL THE FORMAT CODES IN FIELD TWO OF THE FORMAT WORD ARE BREAK CODES EXCEPT FOR THE PRINT BLANKS AFTER THIS LINE CODE (BA). FOR THE BA CODE THE BREAK OCCURS AFTER PROCESSING THE LINE THAT CONTAINS IT.

THE FORMAT CODES IN FIELD 3 OF THE FORMAT WORD WHICH CAUSE A BREAK ARE PARAGRAPH INDENTATION (IN), CENTRE TEXT (CT), PRINT THE LINE NON-FORMATTED (NF), AND THE SPACING CONTROL CODES SS (SINGLE SPACING) AND DS (DOUBLE SPACING). THE REMAINING FORMAT CODES DO NOT CAUSE A BREAK CONTINUE THE PREVIOUS LINE OF TEXT.

FIELD 1 IS NOT USED FOR FORMATTING INFORMATION BUT IS USED FOR SUB-SECTION MARKERS, AS AN IDENTIFICATION TO THE USER AS TO WHAT LINES OF TEXT ARE DEFINED AS THE START OF A SUB-SECTION IN THE TEXT FILE.

THE FORMAT CODES IN FIELD 2 ARE FOLLOWED BY A MAXIMUM OF 3 NUMERIC DIGITS IN THE FORM XXNNN, WHERE XX IS A FORMAT CODE AND N IS A NUMERIC DIGIT. THESE CODES PROVIDE A MEANS TO ADJUST LEFT AND RIGHT MARGINS DURING FORMATTING, PRINT BLANK LINES BEFORE OR AFTER A LINE OF TEXT, TO START A NEW PAGE AND TO CHANGE THE CURRENT PAGE NUMBER.

WHEN THE LEFT MARGIN IS SHIFTED LEFT (LL) (RIGHT (LR)) THE TEXT IN EACH LINE IS SHIFTED LEFT (RIGHT) NNN SPACES IN THE OUTPUT LINE. WHEN THE RIGHT MARGIN IS SHIFTED LEFT (RL) (RIGHT (RR)) THE WIDTH OF EACH LINE (E.I. NUMBER OF CHARACTERS PER LINE) IS DECREASED (INCREASED) BY NNN CHARACTERS, EFFECTIVELY INCREASING (DECREASING) THE WIDTH OF THE RIGHT MARGIN.

THE MARGIN DELAY CODE (MD) DELAYS MOVING OF THE LEFT MARGIN RIGHT UNTIL ONE OUTPUT LINE IS PRINTED FROM THIS LINE OF TEXT. THIS IS USEFUL FOR FORMATTING TEXT OF THE FORM,

MD CODE - AFTER THIS FIRST LINE IS PRINTED THE LEFT MARGIN IS MOVED THE SPECIFIED NUMBER OF SPACES RIGHT.(E.G. 11 SPACES HERE)

NOTE THAT AT THIS POINT AFTER ALL THE REQUIRED TEXT IS PRINTED THE LEFT MARGIN MUST BE RESET TO ITS ORIGINAL POSITION (I.E. MOVE LEFT MARGIN LEFT 11 SPACES).

Figure 3.5 Single Spaced Page of Output

5) MOVE COMMAND

PURPOSE : TO MOVE FORWARD OR BACKWARD FORM THE CURRENT LINE IN THE FILE.

FORMAT 1 : /MOVE+*STRING* OR /MOVE-*STRING*

DESCRIPTION : FORMAT 1 OF THE MOVE INSTRUCTION WILL MOVE FORWARD (OR BACKWARD) IN THE TEXT FILE FROM THE CURRENT LINE TO A LINE OF TEXT CONTAINING STRING AS ITS FIRST NON-BLANK CHARACTERS. IF THERE IS NO LINE IN THE FILE WHICH MATCHES THIS STRING THE END (OR BEGINING) OF THE TEXT FILE WILL BE REACHED AND A MESSAGE WILL BE OUTPUT. CARE SHOULD BE TAKEN IN SPECIFYING THE STRING EXACTLY INCLUDING EMBEDDED BLANKS OTHERWISE THE WHOLE FILE WILL BE SEARCHED AND DISPLAYED WITH NO SUCCESS AND CONSIDERABLE EXECUTION TIME WASTED. FOR EXAMPLE, IF THE COMMAND

/MOVE+/THIS IS /

WAS ENTERED AND THE FILE CONTAINED THE LINES,

THE CURRENT LINE
THIS ISN T IT.
THIS IS IT.

THE FILE WILL BE POSITIONED AT THE THIRD LINE AND DISPLAY IT. NOTE THAT IF THE COMMAND WAS /MOVE+/THIS IS/ THE FILE WOULD BE POSITIONED AT THE SECOND LINE.

Figure 3.6 Formatted and Non-Formatted Output

THE INPUT OF TEXT

TEXT CAN BE ENTERED INTO THE TEXT FILE FROM TWO SOURCES IN TWO DIFFERENT MODES. TEXT CAN BE INPUT FROM A FILE CONTAINING CARD IMAGES (80 CHARACTER RECORDS) OR FROM THE CRT TERMINAL. A FILE CONTAINING CARD IMAGES CAN BE CREATED AND USED AS INPUT BY UTILIZING THE /TEXTFILE COMMAND OR TEXT CAN BE ENTERED FROM THE CRT TERMINAL USING THE /ADD COMMAND. THERE ARE TWO MODES OF INPUT : PROGRAM TEXT MODE AND ENGLISH TEXT MODE. PROGRAM MODE REFERS TO THE TEXT OF COMPUTER LANGUAGE PROGRAMS, WHILE ENGLISH TEXT REFERS TO ANY NATURAL LANGUAGE TEXT SUCH AS REPORTS, MANUALS, MANUSCRIPTS ETC. THE CHOICE OF THESE TWO MODES IS MADE WHEN THE FILE IS FIRST OPENED FOR EDITING. THE USER IS PROMPTED TO SELECT THE TYPE OF TEXT TO EDIT BY THE LINE,

--TYPE OF TEXT TO EDIT, ENTER N - NATURAL LANGUAGE TEXT

ENTER P - PROGRAMMING LANGUAGE TEXT

THE USER SELECTS PROGRAM TEXT MODE BY ENTERING THE LETTER P AND ENGLISH MODE BY ENTERING THE LETTER N (NATURAL LANGUAGE TEXT). WHEN TEXT IS ENTERED IN PROGRAM MODE, THE TEXT IS STORED 80 CHARACTERS PER RECORD OR LINE. WHEN USING THE /TEXTFILE COMMAND EACH CARD IMAGE READ IN OCCUPIES ONE LINE(OR RECORD) OF TEXT. WHEN ENTERING TEXT FROM THE CRT THE MAXIMUM NUMBER OF CHARACTERS ENTERED PER LINE IS 72. THEREFORE, THE LAST 8 CHARACTERS IN THE RECORD WILL BE BLANKS.

WHEN ENGLISH TEXT MODE IS SELECTED, THE USER HAS CONTROL OVER THE PERCENTAGE OF EACH RECORD TO BE FILLED (70 - 100 % OF 130 CHARACTERS MAXIMUM). THE PROMPT,

--ENTER PERCENTAGE OF RECORD TO BE FILLED...

IS DISPLAYED TO SELECT THE PERCENTAGE TO BE FILLED. THE REMAINING SPACE IS FILLED WITH BLANKS FOR FUTURE ADDITIONS TO

Figure 3.7 **Single and Double Spaced Output**

INTRA-LINE EDITING COMMANDS

THE INTRA-LINE EDITING COMMANDS ALLOW CHARACTER BY CHARACTER EDITING OF A LINE. THEY PROVIDE A MEANS OF FORMING A NEW LINE FROM PARTS OF AN EXISTING LINE ALONG WITH INSERTIONS OF NEW TEXT. THESE COMMANDS CAUSE CHARACTERS TO BE COPIED FROM THE OLD LINE INTO A NEW ONE BEING FORMED, SKIPPED OVER IN THE OLD LINE WITHOUT BEING COPIED, AND INSERTED INTO THE NEW LINE. WHEN THE EDIT IS FINISHED THE NEW LINE REPLACES THE OLD ONE AND ADDITIONAL NEW LINES ARE GENERATED WHERE NECESSARY IF OVERFLOW OF THE OLD LINE OCCURS. EACH INTRA-LINE EDIT TO BE STORED MUST BE TERMINATED BY THE /FINISH COMMAND. IF WHILE PERFORMING AN INTRA-LINE EDIT, A NON-INTRA-LINE EDIT COMMAND IS ENTERED THE EDIT IN PROGRESS IS CANCELLED AND THE OLD LINE IS NOT CHANGED.

THE CURRENT POSITION IN THE OLD AND THE NEW LINE IS DETERMINED BY THE COMMANDS PERFORMED. THE CURRENT POSITION IN EITHER LINE IS THE NEXT CHARACTER TO BE PROCESSED. FOR EXAMPLE, AFTER A COPY THE CURRENT

Figure 3.8 Margin and Line Width Changes, line width of 45 characters, left margin width of 15 spaces.

INTRA-LINE EDITING COMMANDS

THE INTRA-LINE EDITING COMMANDS ALLOW CHARACTER BY CHARACTER EDITING OF A LINE. THEY PROVIDE A MEANS OF FORMING A NEW LINE FROM PARTS OF AN EXISTING LINE ALONG WITH INSERTIONS OF NEW TEXT. THESE COMMANDS CAUSE CHARACTERS TO BE COPIED FROM THE OLD LINE INTO A NEW ONE BEING FORMED, SKIPPED OVER IN THE OLD LINE WITHOUT BEING COPIED, AND INSERTED INTO THE NEW LINE. WHEN THE EDIT IS FINISHED THE NEW LINE REPLACES THE OLD ONE AND ADDITIONAL NEW LINES ARE GENERATED WHERE NECESSARY IF OVERFLOW OF THE OLD LINE OCCURS. EACH INTRA-LINE EDIT TO BE STORED MUST BE TERMINATED BY THE /FINISH COMMAND. IF WHILE PERFORMING AN INTRA-LINE EDIT, A NON-INTRA-LINE EDIT COMMAND IS ENTERED THE EDIT IN PROGRESS IS CANCELLED AND THE OLD LINE IS NOT CHANGED.

THE CURRENT POSITION IN THE OLD AND THE NEW LINE IS DETERMINED BY THE COMMANDS PERFORMED. THE CURRENT POSITION

Figure 3.9 Margin and Line Width Changes, line width of 40 characters, left margin width of 20 spaces.

PAGE NO.

Figure 3.10 Page of Non-Formatted Output

3.4 Reformatting of a Text file

The text of a file may be punched on cards or stored as card images on a permanent file. Storage of a text file as a sequential file requires much less storage than if it is stored as a random access file. A text file that is used only occasionally can be stored more compactly as a sequential card image file and the text file can be formed when needed from this text. For example, a user's manual can be stored as a sequential file and a random access file need only be formed when extra copies are required or for updates. The TEF punch command produces two output files. One contains the actual text of the file and the other contains the commands necessary to insert the original formatting information into the newly created text file. Therefore, a text file or any part of it can be recreated when needed using these two files. The text lines are recreated by reading from a file containing the punched text. The formatting information is inserted by using TEF in batch mode with the input commands read from the file containing the reformatting commands. The user may recreate just the lines of text and add new formatting information if desired.

Several different formatted versions of a text file can be maintained and only one copy of the text need be stored. The different versions correspond to different formatting

command files. The user recreates the text of the file and then uses the specific formatting command file for the version required. This can be useful when deciding on a suitable format for a section of text or for output on devices requiring different page formats. Alternate formatted versions can be easily produced and compared.

CHAPTER 4

CONCLUSIONS

In Chapters 2 and 3, I have discussed the TEF text editing and formatting system. In pursuing this project, I have achieved the following initial objectives: to produce a flexible text editor and formatting system which is easy to access, learn and use.

The command structure is very simple and hopefully will be convenient for most editing purposes. I have investigated a number of text editors and formatters and decided upon a set of commands which would be helpful in editing the user's text.

A simple and natural addressing scheme is used. Addressing lines of text by their content is a concept which all users can easily understand.

The majority of the formatting features are very primitive. This allows the user to achieve almost any format for his text by using combinations of these basic commands. Many formatting features are analogous to the operations of a typewriter.

TEF requires about 37K words of central memory storage.

Additions will require little modification of the existing system. TEF is written in a commonly accepted language (FORTRAN) and is relatively machine independent.

Response times vary widely depending on the system load of the computer and on the type of instructions being executed. For example, a typical response time for advancing in the file one line is less than a second or two, while formatting of a large file may take 2 or 3 minutes.

The CPU time required to format an 8 1/2 X 11 inch page double spaced with 65 characters per line and 58 lines per page is about .12 seconds. The TEF users manual consists of about 700 text file lines. The formatting of this file takes about 5.5 CPU seconds and produces 51 pages of output (8 1/2 X 11 inch pages).

In summary, I have implemented and tested a text editing and formatting system which is a useful tool for the preparation of written material subject to frequent revisions.

4.1 Improvements and Additions to TEF

This section will discuss some features which could prove to be useful extensions to TEF. Because of time restrictions they were not implemented in the present version of TEF and will be discussed only briefly here.

4.1.1 Text Buffer Areas

A useful feature of some text editors allows the storing of text in a text buffer and this buffer of text can then be inserted at any place in the file and as many times as required. This feature is useful for programming language editors when placing common documentation into many subroutines or for frequently changing structures such as common blocks or dimensioning in FORTRAN.

Another use for the text buffer areas is in macro definitions [KER 72] . The editor can be directed to take its editing commands from a buffer instead of from the CRT. The creation of a complex editing procedure in a buffer, and then repeated application of the buffer to some text, would permit a great saving in effort. Having multiple buffers with names associated with them or the ability to define a macro by a name would allow execution of a long sequence of commands by specifying the name of the buffer or macro.

4.1.2 Text Compression

The "information explosion" noted in recent years makes it essential that storage requirements be kept to a minimum. Large text files can be compressed into a more compact and condensed form to greatly reduce the storage requirements. Other advantages of text compression are reduced

data transmission time and the security of the encoded compressed text.

Most data compression methods exploit the non-randomness of useful information at the character level, by storing the more frequent combinations of characters in condensed form. The searching processes of information retrieval can be carried out on the compressed text, involving fewer matching operations.

Mayne [MAY 75] and Marron [MAR 67] discuss automatic data compression techniques which choose codes based on the commonest sequences of characters in a given piece of text. Some methods scan all of the original data to discover the most frequent character sequences or strings while others scan a small sample of the text.

Lesk [LES 70] investigates several algorithms for compressing English and FORTRAN text.

In one typical method the characters which occur most frequently are recoded so that fewer bits are needed to represent them than in the original. Similarily, the less frequent characters may require more bits to be represented than in the standard machine representation.

Another method involves the recoding of common character strings. A sequence of characters forming a group of frequent occurrence is recoded as a single entity.

Clearly, if a number of different sequences of several characters occur frequently in the text, there is the possibility of a considerable compression by recoding such strings.

For large text files a compression scheme could be valuable for a text editing system.

4.1.3 Additional Formatting Features

Figure and footnote commands would be useful additions to TEF. Many authors rely heavily on footnote references. TEF could be extended to generate a footnote and a footnote reference marker at specific places in a user's text. The figure command would collect text for a figure and "fit" it onto a page or split it over several pages when necessary.

The length of the running heading could be altered to allow a running heading of several lines.

The specification of where the page number is to be printed could prove useful. Some users might prefer the page number either at the bottom of the page or centered at the top of the page.

Additional printer chains could provide special characters, underscores, lower case characters and indexing notation.

Automatic indexing and renumbering of sections or references after changes have been made to a file would also be a valuable addition.

Notation for Appendices

STRING - a string of characters with or without embedded blanks.

\+ - denotes forward motion in the file.

\- - denotes backward motion in the file.

NNN or MMM - a maximum three digit integer number.

[] - optional part of command.

CH - a single character.

d - a delimiter, any non-alphabetic character including blank.

APPENDIX A

Summary of Editing Commands

Command	Description
1) /TEXTFILE	Adds text to the text file from a card image file.
2) /ADD	Adds text to the text file from the CRT input.
3) /RECREATE	Reads input from a card image file of TEF punched output and recreates the original lines which were punched.
4) /BEGIN	Positions the text file at its first line of text.
5) /MOVE±dSTRINGd	Move forward(+) or backward(-) from the current line in the file to a line whose begining non-blank characters match STRING.
/MOVE[±NNN]	Move forward(+) or backward(-) in the text file NNN lines from the current line. If ±NNN is omitted move ahead one line.
6) /LOCATE±dSTRINGd[NNN,MMM]	Locate a line of text in the file moving forward(+) or backward(-) from the current line that contains STRING between character positions NNN and MMM. If NNN,MMM is omitted locate STRING anywhere in a line of text.
7) /CREPLACEdSTRING1dSTRING2d	Replace the first occurrence of STRING1 by STRING2 in the current line.

8) /AREPLACEdSTRING1dSTRING2d — Replace every occurrence of STRING1 by STRING2 starting with the current line and moving forward in the file.

9) /DELETE±dSTRINGd — Starting with the current line move forward(+) or backward(-) in the file and delete lines until a line is reached whose begining non-blank characters match STRING.

/DELETE [±NNN] — Delete NNN lines of text starting with the current line and moving forward(+) or backward(-) in the text file. If ±NNN is omitted the current line is deleted.

10) /SECTIONdCHd — Position the file at the pre-defined sub-section CH.

11) /LIST [dNNNd] — List NNN lines of text starting with the current line with their corresponding format codes on the printer. If dNNNd is omitted the whole file is listed.

12) /PUNCH [dNNNd] — Punch NNN lines of the text file starting with the current line and generate the commands necessary to recreate the formatting information they contain. If dNNNd is omitted the whole file is punched.

13) /SCRATCH — Scratch any output from TEF accumulated so far. (TEF Output File)

14) /CONTINUE — Repeat the previously entered command.

15) Intra-line Editing Commands

a) /KOPYdCHd and /XKOPYdCHd — Copy characters from the old line to the new line being formed up to and including(KOPY) or excluding (XKOPY) the next occurrence of the character CH.

b) /SKIPdCHd and /XSKIPdCHd	Skip characters in the old line up to and including(SKIP) or excluding(XSKIP) the next occurrence of the character CH.
c) /INSERTdSTRINGd	Insert STRING into the new line being formed.
d) /FINISH	Copy the rest of the old line into the new line being formed and replace the old line by the new one in the text file.
16) /ENDFILE	Make the current line in the file the last line in the file.
17) /EXIT	Exit from TEF and return to INTERCOM control.
18) /FORMATdFORMAT CODEd	Add the FORMAT CODE into the format word of the current line. If the FORMAT CODE is a single letter A - J mark the current line as the start of a sub-section in the file. If the FORMAT CODE is an X clear out all the formatting information to the default right justification. See APPENDIX B for the FORMAT CODES.
19) /FLIST [dCd]	Format the text file starting at the current line using the formatting information in the format words of the lines of text. If dCd is omitted format the whole file.

APPENDIX B

Summary of Format Codes

Format Code	Description
1) LLNNN	Move the left margin left NNN character positions.
2) LRNNN	Move the left margin right NNN character positions.
3) RLNNN	Move the right margin left NNN character positions.
4) RRNNN	Move the right margin right NNN character positions.
5) MDNNN	Delay the moving of the left margin right NNN character positions until one line of text is output from this line.
6) BBNNN	Print NNN blank lines before this line of text.
7) BANNN	Print NNN blank lines after this line of text.
8) NPNNN	Start a new page, If NNN is non-zero set current page number to NNN.
9) IN	Paragraph indentation of 5 spaces from the left margin.
10) CT	Centre the text in this line and print it.
11) NF	Print this line exactly as it was input.
12) RH	Use this text as the current running heading.

13) CH	Clear the running heading to blanks.
14) SS	Change the inter-line spacing to single spacing.
15) DS	Change the inter-line spacing to double spacing.
16) PO	Turn page numbering off.
17) PY	Turn page numbering on.

Blank is default right justification.

REFERENCES

[ALL 69] Allen, L., Borgelt, J., Fajman, R. et al.,
WYLBUR Reference Manual,
Third Edition 1969,
Stanford Computation Center.

[ARB 67] Arbor, Ann,
QED Reference Manual,
Com-Share, Michigan,
Reference No. 9004-4, Jan. 1967.

[BAR 65] Barnett, Michael P.,
Computer Typesetting, Experiments and Prospects,
The M.I.T. Press,
Cambridge, Massachusetts.

[BEN 72] Benjamin, Arthur J.,
"An Extensible Editor for a Small Machine with Disk Storage",
Comm. ACM, 15,8, (AUG. 1972), 742-747.

[BOO 73] Bookstein, Abraham,
"On Harrison's Substring Testing Technique",
Comm. ACM, 16, 3, (March 1973), 180-181.

[CDC 74] Control Data Corporation,
6000 Computer Systems,
FORTRAN Extended Version 4 Reference Manual,
Sunnyvale, California 94086.

[DEU 67] Deutsch, L. Peter and Butler W. Lampson,
"An Online Editor",
Comm. ACM, 10, 12, (Dec. 1967), 793-799.

[DEW 71] Dewar, Robert B. K.,
"SPITBOL Version 2.0",
SNOBOL4 Project Document S4D23,
Illinois Institute of Technology,
Chicago, Ill. February 1971.

[FAJ 73] Fajman, Roger and John Borgelt,
"WYLBUR: An Interactive Text Editing and Remote Job Entry System",
Comm. ACM, 16, 5, (May 1973), 314-322.

[GRI 71] Griswold, R.E., Poage, J.F., Polonsky, I.P.,
The SNOBOL4 Programming Language, 2nd ed.,
Bell Telephone Laboratories, Inc.
Prentice-Hall, Inc.

[HAR 71] Harrison, Malcolm C.,
"Implementation of the Substring Test by Hashing",
Comm. ACM, 14, 12, (Dec. 1971), 777-779.

[IBM 69] IBM Cambridge Scientific Center Report,
"A Conversational Context-Directed Editor",
Form No. 320-2041,
Cambridge, Mass., March 1969.

[KER --] Kernighan, B. W. and L. L. Cherry,
"A System for Typesetting Mathematics",
Computing Science Technical Report No. 17,
Bell Laboratories, Murray Hill, N.J.

[KER 72] Kernighan, B. W., D. M. Ritchie and K. L. Thompson,
"QED Text Editor",
Computing Science Technical Report No. 5,
Bell Laboratories, Murray Hill, N.J.

[LES 70] Lesk, M. E.,
"Text Compression",
Bell Laboratories, Murray Hill, N.J.,
November 1970.

[MAC --] MacDonald, Jack,
Program Documentation for the EDIT/3000 Text
File Formatter,
Hewlett-Packard, Data Systems.

[MAR 67] Marron, B. A. and P. A. D. De Maine,
"Automatic Data Compression",
Comm. ACM, 10, 11, (Nov. 1967), 711-715.

[MAY 75] Mayne, A. and E. B. James,
"Information Compression by Factorising
Common Strings",
The Computer Journal, 18, 2, (May 1975), 157-160.

[NEL 67] Nelson, Theodor H.,
"Getting it out of our System",
In Information Retrieval: Critical View,
George Schecter (Ed.), Thompson Books,
Washington, D.C., 1967.

[STA 74] Staa, Arndt von,
"A Text Formatter",
Research Report CS-74-21,
Department of Computer Science,
University of Waterloo, Nov. 1974.

[VAN 71] Van Dam, Andries and David E. Rice, "On-line Text Editing: A Survey", Computing Surveys, 3, 3, (Sept. 1971), 93-114.