ESD-TR-75-119

82154

1 of 2 cys.

Technical Note 1975-13

LO – A Text Formatting Program

A. Evans, Jr.

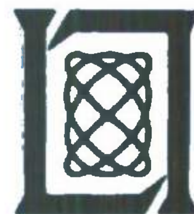
21 February 1975

Prepared for the Advanced Research Projects Agency
under Electronic Systems Division Contract F19628-73-C-0002 by

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LEXINGTON, MASSACHUSETTS



Approved for public release; distribution unlimited.

ADA007824

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. This work was sponsored by the Advanced Research Projects Agency of the Department of Defense under Air Force Contract F19628-73-C-0002 (ARPA Order 2006).

This report may be reproduced to satisfy needs of U.S. Government agencies.

The views and conclusions contained in this document are those of the contractor and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency of the United States Government.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

Eugene C. Raabe, Lt. Col., USAF
Chief, ESD Lincoln Laboratory Project Office

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LINCOLN LABORATORY

LO – A TEXT FORMATTING PROGRAM

A. EVANS, JR.
Group 44

TECHNICAL NOTE 1975-13

21 FEBRUARY 1975

Approved for public release; distribution unlimited.

LEXINGTON MASSACHUSETTS

Abstract

LO is a text formatting program used to prepare documents such as this report. It reads an input text file and creates an output file. The input file contains text to be printed, interspersed with commands to LO that direct its operation. Commands are expressed in a language of considerable sophistication. The command language is described, the presentation being designed to be suitable both as a primer to the learner as well as a reference manual for the expert.

LO currently runs on the TX-2 Computer at Lincoln Laboratory, and details are provided on its use in that environment. There is a brief discussion of the steps involved in implementing it on a different computer system.

Table of Contents

Abstract iii

Ch 1: Introduction 1

- 1.1: Text Processing 3
- 1.2: Variables and Data Types 4
- 1.3: Expressions 7
- 1.4: Commands 9
- 1.5: Errors and Warnings 11
- 1.6: Output Devices 11

Ch 2: Basic LO Capabilities 13

- 2.1: Page Layout 13
- 2.2: Text Control 16
- 2.3: Substitution 18
- 2.4: Text Output 20
- 2.5: Headers and Footers 22
- 2.6: Page Numbering 23
- 2.7: Buffers 24
- 2.8: User-Defined Variables 25
- 2.9: Conditionals 27

Ch 3: Advanced LO Topics 29

- 3.1: Tabs 29
- 3.2: Sentence Ending 30
- 3.3: Hyphenation 31
- 3.4: Footnotes 32
- 3.5: Flags 34
- 3.6: Macros 35
- 3.7: Traps 38
- 3.8: Input/Output 40
- 3.9: The Command ".overstrike" 41

3.10: The Command ".set" 42
3.11: The Command ".expand" 45
3.12: Variable Width Fonts 46

Ch 4: Built-In Variables 47

Ch 5: Command Descriptions 53

Ch 6: Examples 63

6.1: Date and Time 63
6.2: Miscellaneous Examples 65
6.3: Chapter and Section 68
6.4: Recursive Macros 70

Ch 7: LO on TX-2 73

7.1: How to Use LO 73
7.2: On-Line Documentation 74
7.3: LO on Another Computer 75

Ch 8: Summary Tables 77

8.1: Table of Variables 77
8.2: Table of Commands 78
8.3: Options for ".set" 80

Index 83

LO — A TEXT FORMATTING PROGRAM

Chapter 1: Introduction

LO is a text formatting program used to prepare documents such as this report. The user is provided with a command language of considerable sophistication in which to express his wants. The design philosophy in specifying the command language has been to favor generality - problems have been solved by inventing general tools rather than specific solutions. The result has immense flexibility, but is perhaps not as easy for the beginner to learn as it might be. LO's command language has much the flavor of a computer programming language, and I think it safe to predict that programmers will find it easier to learn than will non-programmers. LO would have turned out much differently had I considered a different audience, but I wanted a tool that my colleagues and I would find pleasant to use, and we are all programmers.

In writing LO, I took ideas from wherever I found them. LO owes a great deal to the RUNOFF programs on TENEX and MULTICS, although much of it is new. The current LO on TX-2 is descended from an earlier TX-2 version written by D. Austin Henderson, Jr. I have assumed responsibility for LO and have performed a complete rewrite of both the program and this description. Many useful suggestions for improving LO were made by Louis N. Gross, Harry C. Forsdick and Alan G. Nemeth; these are gratefully acknowledged.

LO reads from an input text file and creates an output file. Lines are read from the input file one at a time and processed. Each line must be less than 300 characters long. If the line starts with any character other than "." it is treated as a string of words of text to be added to the output, with line breaks and pagination as described hereafter. If the line starts with "." it is treated as a command; following the "." is the command name, followed by space,

followed by zero or more parameters.

This report is organized so as to be useful both as a primer and as a reference manual. The rest of Chapter 1 provides introductory information about various aspects of LO's processing. Chapter 2 presents LO's basic capabilities, each section addressing some aspect of LO, describing how it works and the commands associated with it. Chapter 3 is organized similarly but addresses topics of less interest to the beginner. The next two chapters contain all the details, in reference manual form. In Chapter 4, each of LO's predefined variables is described, and Chapter 5 contains descriptions of all the commands. Next, Chapter 6 contains some examples of the use of LO's abilities. In Chapter 7, the implementation of LO under the APEX Time Sharing system on the TX-2 computer at Lincoln Laboratory is discussed. Included is a brief discussion of some of the issues involved in implementing LO on another computer. Finally, Chapter 8 contains tabular listings of all the predefined variables and all the commands. The beginner is advised to read carefully this chapter and the next, and then scan the remainder of the document so he will know where to look for help as he becomes interested in LO's more advanced features.

This document has been designed to make it as easy as possible to find things in it. An index of terms appears at the end, with references to the page and line number where each term is discussed. Internal references in this document are usually by section number, and the lower outside corner of each page contains the inclusive section numbers that appear on that page. Further, the lower inside corner shows the title of the current chapter, and the lower center contains the title of the current section. All these things are done, not surprisingly, using LO features. In fact, this document pushes fairly hard in most (although not all) of the directions that LO lets one go.

In writing this document, it has proved convenient to assume that LO's output is a printed document, while in fact it is a text file[1].

---

[1] This is not unreasonable, since the user will probably print the file as soon as LO has created it for him.

Thus there are references such as "upspacing the paper," "page eject," "output page," etc. This metaphor simplifies the writing.

This document has been prepared on TX-2 and describes the TX-2 implementation of LO; it therefore uses the TX-2 character set. A few facts about this set may help the reader. The set includes the following characters:

α β γ λ	Greek letters
|	vertical bar
‖	double bar
×	multiplication cross

Note the difference between vertical bar "|", letter "I" and number "1". Note also that "×" is used for multiplication, "*" serving a different purpose. The underscore character "_" is nonspacing, so that it appears in the same print position as the character that immediately follows it. Also, the set includes backspace, as well as superscript, subscript and normal.

LO is written entirely in BCPL, and the source code is readily available and well commented. Section 7.3 discusses briefly the issues involved in implementing LO on another computer; please see me for further information. I will also be glad to help users to make the best use of this tool. I will listen receptively to suggestions for changes and improvements to LO, but I do not promise to act on them. Of course, I will be responsive to any reports of bugs - either in the LO program or in this document.

1.1 - Text Processing: In processing text, any combination of spaces, tabs and carriage returns is treated as a single space and regarded as a word separator. This "word" is thus a concatenation of consecutive printing characters.

In LO's usual mode of operation, words are added to the output line separated by a single space until one is found which goes beyond the right margin. The line is then added to the output, and the next line is started with the word which overflowed the previous line[2]. If the user so requests (see Section 3.3), LO will attempt to hyphe-

---

[2] There is no check of overflow on this first word in the line. This prevents endless looping trying to output a word which is longer than a line.

nate this word.

There are two concepts relevant to storing text: adjust mode and fill mode. The description just given was of fill mode, in which LO puts as many words as there are room for on each line, starting a new line only when necessary (or when directed to do so by a command). The transition in the input from one line to the next is irrelevant. Independent of fill mode is adjust mode, which concerns alignment of the right margin (as in this document). When fill mode is on and a line has been completed and adjust mode is on, LO then inserts spaces as needed between words so that the line ends exactly on the right margin[3]. When fill mode is off, each non-command line (including blank lines) is copied into the output exactly as it appears in the input. The setting of the adjust switch is irrelevant if fill is off. The adjust switch and the fill switch each may be set independently using the ".set" command, which is described in Section 3.10.

The character "*h*" is translated as a non-separating space[4]. The "*h*" is treated as a non-space in deciding where to end a line (in fill mode) and in adding spaces in right-margin adjustment; it is replaced by a space at the very last moment of LO's processing. "*h*" is also given special treatment in the insertion of hyphens.

Tabs in ".plain" lines and in lines printed while fill mode is off and under control of ".unproc" are processed specially. See Section 3.1.

LO inserts an extra space between sentences. Briefly, LO assumes a sentence break when one word ends with "." and the next word starts with an upper-case letter. The actual algorithm is more complex, and the user is provided control over its action. Details are provided in Section 3.2.

1.2 – <u>Variables and Data Types</u>: LO maintains a table of variables for the benefit of the user. Some variables are built in with predefined meanings and values, and the user is free to define others of his

---

(3) The spaces are inserted randomly into the gaps between words.

(4) A different character may be selected using the command ".set spacer", as described in Section 3.10.

choosing. Syntactically, a variable name is made up of upper- and lower-case letters, digits, and the period. The first character must be a letter, either upper- or lower-case. Variable names used in this document are enclosed in "#"s. Thus #lspacing# is used to refer to the variable named "lspacing".

Each variable has a data type, which is one of INT (integer), LIN (line spacing), STR (string) or BUF (buffer). An INT is an integer, positive or negative, whose magnitude is less than 2^{34}. A STR is a character string of (essentially) unlimited length made up of any characters except carriage return. A BUF is a buffer used to hold lines of text either for later insertion into LO's output or for rescanning as input to LO. Buffers are discussed further in Sections 2.7 and 3.6.

A LIN denotes an amount of vertical spacing on the output page. In general, a LIN is an integer which represents a whole number of lines of vertical spacing. However, Lincoln Writers as well as certain LDX character sets permit half-line spacing as a possibility[5]. If the user is not using such a character set, he may ignore the next paragraph and assume that all references to LIN quantities are to integers. The default conversions have been arranged so that everything works correctly.

Each full line of vertical spacing is divided into a fixed number of part lines. In the present implementation of LO this number is fixed at two, and the (read-only) variable #partsperline# has this value to reflect that fact[6]. A vertical spacing is written as two integers separated by the character "*", where the first integer specifies the number of whole lines and the second the number of part lines[7]. Thus, given that #partsperline# is two, a spacing of "1*1" represents one-and-a-half spacing; "0*3" would work as well. All

---

[5] None of the APEX LDX sets - 1 to 7 - has this property. This document was printed with a set which does have it.

[6] Someday it may be possible for the user to vary this quantity, but I am unlikely to work on this until there is hardware to support the added flexibility.

[7] A parenthesized expression may be used instead of either of these integers. The parentheses are required by the precedence rules of the expression scanner.

quantities involving vertical spacing are expressed this way, and certain commands (including mostly those that set these variables) expect a parameter of type LIN.

Certain variables are initially declared with default values. Many of these values, such as the current page number in #pageno#, are updated automatically by LO as processing proceeds. Special commands are provided for convenience to change many of these variables (for example, the command ".lspacing" to set the variable #lspacing#), but most of them may also be changed by ".store"[8]. Setting a variable has the obvious effect on the future operation of LO. For example, setting #rmarg# using either ".store" or ".rmarg" changes the right margin for all successive output.

There are over 50 variables initially defined in LO, many of them of quite specialized use. All are described completely in Chapter 5 and summarized in tabular form in Chapter 8; a few of interest to the beginner are now listed. The value shown is the default value. In this table and throughout this document, "N" stands for an INTeger value and "L" for a LINe value.

bline	66*0	L	Number of lines on the page.
lmarg	0	N	Current left margin.
lspacing	2*0	L	Vertical spacing between lines.
nextpage	2	N	Page number to be assigned to the next page.
pageno	1	N	Page number of the current page.
rmarg	71	N	Number of characters per output line.
year		N	The current year - 1974, 1975, etc.
month		N	The current month - 1, ..., 12.
day		N	The day of the month - 1, ..., 31.
weekday		N	Day of the week - 0 = Sunday, 6 = Saturday.
hour		N	Hour of the day - 0, ..., 23.
minute		N	Minute of the hour - 0, ..., 59.
second		N	Second of the minute - 0, ..., 59.

The last seven variables listed are set by LO to the date and time when it begins its work; they are not updated during LO's operation.

A variable may be used in either of two kinds of context: In an expression it is treated as an integer or as a LIN; if it is a string

---

[8] Some variables are read only and cannot be changed by the ".store" command. This permits LO to check that their value is reasonable, for otherwise much confusion might follow. For example, it would be disastrous were #lspacing# to become zero or negative.

it is scanned and converted to either an INTeger or a LINe, depending on its syntax, with an error message if its syntax is not that of either. It is a detected error if the value is a buffer or is undefined. The variable is NOT to be enclosed in "#"s in an expression. Expressions are described in Section 1.3.

The other type of context requires that the variable be substituted for, as a string. This occurs in ".subst" and ".triple" commands and in headers and footers. Here the variable must be enclosed in "#"s. If the variable is an integer or a LIN it is converted to a string (with a "*" for LINs), and if it is a buffer the value is the widest line in the buffer[9]. If there is more than one line of that width, the first such is given. Here "width" refers to print positions on the page. Substitution is discussed in Section 2.3.

1.3 — <u>Expressions</u>: Many of the command lines require parameters of type INT or LIN. These may be either absolute (like "2" or "1*1"), or they may be expressions. These expressions are evaluated when the command line is encountered. The LIN operator "*" takes precedence over "×" and "/", which in turn take precedence over "+" and "-"; otherwise evaluation is from left to right. Parentheses may be used to change the order of evaluation as usual.

An INTeger is represented internally in LO as a signed quantity whose magnitude is less than 2^{34}. A LINe is represented as a whole number of part-lines whose magnitude is less than 2^{34}. Built-in LINe-valued variables are not permitted to have negative values, although user-defined variables are not so restricted. The table below shows the result of various operations on INTs and LINs. The arithmetic is done using TX-2 hardware instructions, which perform, for the most part, in the expected manner. If both p and q are positive INTegers, then the value of the expression

$$p - q \times (p/q)$$

is the remainder on dividing p by q. It is harder to predict the effect if p or q is negative.

---

(9) This is seldom useful, but results from the processing of "λ" for a buffer. This latter IS useful.

	+ -	×	/	*
N op N	N	N	N →2	L
N op L	L →1	L →3	err	err
L op N	L →1	L →3	L →3	err
L op L	L	err	err	err

Notes:

→1 N is converted to L by assuming it to be a number of full lines. That is, ˜3˜ in LINe context is interpreted as ˜3*0˜.

→2 Division always truncates its result towards zero. It is a detected error to attempt to divide by zero.

→3 Take L as a whole number of part lines, perform the operation (with truncation for ˜/˜), and interpret the result as a whole number of part lines. It is a detected error to attempt to divide by zero.

err These situations are detected errors.

Operators in LO Expressions

Some examples now follow, in which it is assumed that there are two part lines in each full line. (That is, 1*0 denotes the same spacing as does 0*2.)

```
1*1   + 2*1  → 4*0
1*1   + 2    → 3*1
1*1   × 3    → 4*1
8     / 3    → 2
8*0   / 3    → 2*1
```

An easy way to see this last result is to think of it as

```
0*16 / 3    → 0*5
```

which is, in fact, how it is done in LO.

LO converts from INT to LIN when needed. If an INT is used in a LIN context, it is interpreted as a whole number of lines. Thus,

```
.leave 1
```

has precisely the same effect as does

```
.leave 1*0
```

It is a detected error to use a LIN value in an INT context. ∝ may be used for this conversion if it is needed. ∝ preceding a LINe-valued expression evaluates to that INTeger representing the number of part lines. Thus, the value of ˜∝ 2*1˜ is the INTeger ˜5˜.

β preceding a buffer name evaluates to that INTeger which is the number of lines in the named buffer. The count is made of lines stored and does not include space after each line (as derived from #lspacing# when the buffer was created). γ preceding a buffer name evaluates to that LINe which is the amount of vertical spacing in the buffer (including the space lines). It is just the amount of vertical spacing that would be used were the buffer ".insert"ed. See Section 2.7.

λ preceding a variable name in an expression evaluates to that INTeger which is the width of the string that would be substituted for that variable, were it to appear in a context subject to substitution. λ preceding a buffer name evaluates to an INTeger which is the width of the widest line in the buffer. Note that this is the width (in print positions) and not necessarily the number of characters in the line.

Variables are understood by the expression evaluator. If a variable has an INT or LIN value, then that is used. If it has a string value then that value is scanned and interpreted as either an INTeger or a LINe, depending on its syntax. A variable appearing in an expression is not to be enclosed in "#"s - these are to be used only in ".subst" or ".triple" command lines, and in headers or footers.

The character "|" (vertical bar) terminates an expression and it and all remaining characters on the line are ignored, so arbitrary comments may be inserted to its right.

1.4 - Commands: LO has a repertoire of over 65 commands, some of which are rather specialized and of little interest to the beginner. A complete description of all the commands in alphabetic order appears in Chapter 5, and a tabular summary appears in Chapter 8. This section defines the notation that is used in the command descriptions, and many of the commands of interest to the beginner are described in Chapter 2.

A command line starts with a ".", followed immediately by the command name, followed either by the end of line (if the command takes no parameters) or by space. Optional parameters follow, usually each

followed by a comma.

Multiple commands may appear on a line, separated by the character "‖"; all spaces after the "‖" are ignored. Comments may appear after the character "|" on most lines which are interpreted as a command. Note that the command syntax requires a space after the command name, so that there must be a space before "‖" or "|" if there are no arguments. The last "‖" may be followed by text. If one or more commands on a line are followed by text, the text may not begin with "|" or "‖" or ".". Of course, text appearing on a line without commands may start with "|" or "‖", or with space followed by ".".

As discussed in Section 1.1, LO's usual mode of operation is to store words one after another into the output, putting as many as there is room for on each line. The phrase "causes a break" is used in the command descriptions to indicate terminating the present output line so that the next word of input text goes on the next output line. A command does not cause a break unless its description states explicitly that it does. Some commands are not meaningful when LO's output is to a buffer.

Some commands (such as ".subst" or ".if") take as final argument "the rest of the line". This "rest" is operated on by the command, and the result is processed in all ways as though that line had appeared in the input text in place of the command line. The "rest" may be either a command line or text.

The following abbreviations are used in the command descriptions:

N	Any expression whose value is an INTeger.
L	Any expression whose value is a LINe spacing.
B	A buffer-valued variable.
V	Any variable.
<V-list>	A list of variables, separated by commas.
<triple>	A three-part string, as described in Section 2.5. The first non-blank character is taken as the break character.
<text>	Any text at all. The text extends to the end of the line and is not scanned for "|" or "‖".

Other conventions used only once are defined as needed.

1.5 – Errors and Warnings: If LO detects an error while processing a command line, a suitable message and the line at fault are typed to the user, and HELP is called[10]. If the user resumes from this HELP call, the line in error is either ignored totally or a default value is used, and processing continues. An obvious mark in the error message shows how far the scan has gotten when the error was detected.

Only one error is reported (usually) on a line. If an error is detected in a line in which substitution has already been performed (by ".subst"), the line printed will be the result of the substitution. If LO is processing a trap or a header or footer line when the error is detected, that fact is mentioned in the error message.

If LO detects more than 25 errors, it assumes that the problem is serious enough that continuing is not worth while. It comments to this effect on the console (even if error output is to a file) and then terminates the run.

If the user has requested that his output be LDXed (i.e., be printed by TX-2's on-line LDX printer), this LDXing is suppressed if any errors are detected, and LO comments on the console about this suppression. Further, in this case LO peels to BT with negative epsilon (i.e., returns to the operating system reporting that the run was unsuccessful).

LO issues certain warning messages that are not fatal and do not, for example, inhibit LDXing the output file or cause peeling to BT with negative epsilon. Further, certain situations produce optional warnings, in that LO does not usually comment on them but will do so if the user so requests. Details are provided in the description of "warn" option of the command ".set", in Section 3.10.

1.6 – Output Devices: The following data may be of use in preparing output to be printed on the LDX printer at TX-2. There are 67 lines on each page. Wide character sets, such as that used in printing this document, permit 81 characters on a line. (This leaves about a half

---

(10) This HELP call may be suppressed by the optional argument "n" in invoking LO from APEX. If argument "e" is used the error output is to a file rather than to the console and there is no HELP call. See Section 7.1.

inch of left margin and no right margin at all.) Narrow characters (such as sets 6 and 7) permit 109 characters per line, again with no right margin. This document is printed with #rmarg# set to 70, with #bline# at 62*0, and with #lspacing# at 1*1. The windows are 1 on the even-numbered pages and 5 on the odd pages. The value of #topwindow# is 2*0.

There is a typeout program TO available which can be used to type LO's output on the console 6 typewriter with any printing element (i.e., "golf ball") of the user's choice. TO permits multiple balls to realize a larger character set than is available on a single ball. Documentation for TO may be found in the Semi-Public Program notebook in the TX-2 room.

Chapter 2: Basic LO Capabilities

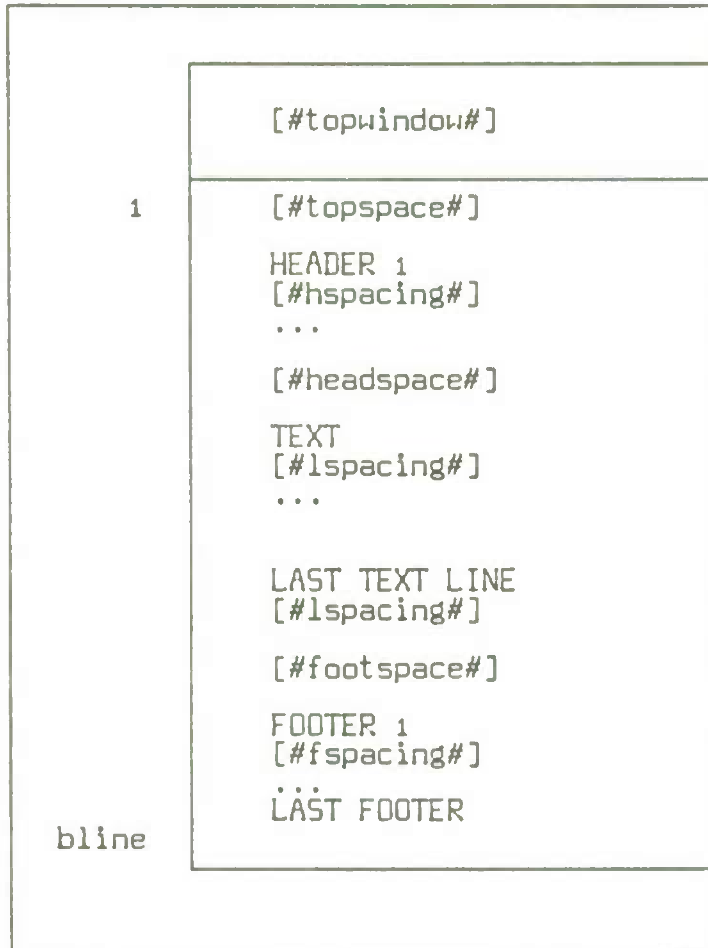
This chapter provides a somewhat tutorial introduction to some of LO's basic capabilities. Each section addresses some aspect of the processing that LO does, describing the facility and discussing the commands that relate to it. The abbreviations that are used in the command descriptions are described in Section 1.4.

For most commands, a two- or three-letter abbreviation is also available with equivalent meaning. These abbreviations are not given here but are presented in the complete description of all the commands which appears in Chapter 5, as well as in the table in Section 8.2.

As previously mentioned, any line starting with ".", is interpreted as a command. There are three characters treated specially if they appear immediately after this ".": Any line starting with ".*" is treated as a comment, the remaining part of the line being ignored by LO. Any line starting with ".." has the first dot stripped off, and the rest of the line is copied verbatim into the output. (This is useful for inserting commands into a buffer intended as a macro, as discussed in Section 3.6.) Finally, a line starting with ".+" is interpreted as a built-in LO command, even though there might be a buffer with the same name. This also is discussed in Section 3.6.

2.1 – <u>Page Layout</u>: LO assumes a page whose top line is numbered one and whose bottom line is numbered #bline#, so there may be #bline# lines per page. A page consists of #topspace# blank lines, followed by as many header lines as specified (from zero to 10, inclusive) each followed by #hspacing# vertical space, followed by #headspace# blanks, followed by lines of text each followed by #lspacing# space, followed by #footspace# blanks, followed by (from zero to 10) footer lines each followed by #fspacing# vertical space. The total vertical spacing used by the above is #bline#. In addition, #topwindow# blank lines are left at the top of each page, but this amount is <u>not</u> taken from #bline#. Thus, using a nonzero value of #topwindow# moves the whole display down on the page, while using a nonzero value of #topspace#

leaves blank space at the top of the page but does not move the last footer. All this is best shown in the figure below. The [] brackets enclose vertical spacings. Note the "1" in the left margin to



Vertical Layout on the Page

indicate line number 1, and "bline" to mark the bottom line on the page.

The left margin is just after column #lmarg#, so with #lmarg# at zero (the usual case) the first column stored into is column one. The right margin is just past column #rmarg#, so there are

rmarg - lmarg

character positions per line. With the default values of 0 for

#lmarg# and #71# for #rmarg#, there are 71 characters per line. The user may set any of these variables as he chooses to adjust the page layout.

The layout of footnotes on the page is described in Section 3.4.

The following commands affect the layout of the text on the page. The command

.lmarg N

causes the left margin to be set to be just after column N, and the command

.rmarg N

causes the right margin to be after column N. If N is missing in ~.lmarg~, the default of zero is taken instead.

The length of the page may be set by

.bline L

which specifies that the total spacing on the page is L. Default provides for 66 lines on the page.

The commands ~.hspacing~, ~.headspace~, ~.footspace~, and ~.fspacing~ may be used to change the variables with the same names. These variables control vertical spacing on the page, as described earlier in this section. The value of #topspace# may be set with ~.store~, and #topwindow# may be set either with ~.store~ or with ~.set window~, as described below and in Section 3.10.

It is convenient if the #lmarg# is always at zero for the bulk of a document, even though the user may want all the printing to be moved to the right on the page. LO therefore provides a separate mechanism for indenting all of a document: the window. Each line printed is preceded by a number of spaces (default zero) that can be set by the user. LO may be used to prepare a document that is to be printed on both sides of the page. For such text, it is desirable to have a large left margin on odd-numbered pages and a large right margin on even-numbered pages. Thus, two window settings are available: one for even-numbered pages and one for odd-numbered pages. As each line is output, it is preceded by #window# spaces. Just before starting each even-numbered page, #window# is set to #ewindow#; and it is set to #owindow# just before each odd-numbered page. The variables #ewindow# and #owindow#, as well as #topwindow#, may be set with ~.store~ or

with the ~.set~ command. For example, the command

    .set window both 5

sets both #ewindow# and #owindow# to 5, and

    .set window even 1, odd 5

sets #ewindow# to 1 and #owindow# to 5. #topwindow# may also be set:

    .set window odd 5, top 1*1, even 1

does the above and also sets #topwindow# to 1*1. Of course, any expression, as described in Section 1.3, may be used in place of the constants in these examples.

2.2 - Text Control: These commands control the appearance of text on the page.

After each line is printed the paper is upspaced by the current value of the variable #lspacing#; the command

    .lspacing L

may be used to set that variable. The default value is two (actually, 2*0), which provides double spacing. This document uses a value of 1*1.

To leave extra vertical spacing L after a line, use

    .leave L

If L is missing it is taken as one (i.e., 1*0), which results in leaving one full line of extra space. There are two restrictions to the amount of space left: If leaving space causes the bottom of the page to be reached, the remaining space is lost and is not left at the top of the next page. Also, this command will not leave any space at the top of a page(11). The command ~.need~ (described just below) can be used to insure that a given amount of space all appears on a single page, if that is what is wanted. ~.leave~ causes a break.

To force a line break with no other effect, use

    .break

Text collected so far for the next line is output, and a new line is started.

---

(11) A convenient way to leave space L at the top of a page is to follow either ~.plain~ or ~.triple ////~ by the command

    .leave L - lspacing

The first command causes upspacing of #lspacing#, so that the total space left is L.

The command

.indent N

forces a break and causes the next line (and only that line) to be indented by N spaces from the current value of #lmarg#. If N is negative the next line is "undented" (i.e., starts to the left of the left margin), but it is a detected error if N + #lmarg# is negative.

Sometimes it is necessary that all of a certain item, such as a figure, appear on the same page. For this purpose, the command

.need L

may be used to ensure that there is at least L vertical spacing still remaining on the page. If there is, the command has no effect; if not, it causes a page eject. On the other hand, the command never causes an eject if the paper is already at the top of the page(12). The command may be used to insure that a certain piece of text, such as a figure, all goes on the same page.

An obvious use of the concepts of ".leave", ".need" and ".indent" is in starting a new paragraph: One wants some space between paragraphs (as in this document); it is unaesthetic for there to be a single line of the beginning of a new paragraph at the bottom of a page; and a new paragraph is usually indented. For this reason the command

.para

is provided. It is equivalent to the three commands

```
.leave paraspacing - lspacing
.need paraneed
.indent paraind
```

This leaves some extra blank space, ejects the paper if there is not enough space remaining on the page, and causes the next line to be indented. The default values for #paraspacing#, #paraneed#, and #paraind# are 3*0, 3*0, and 5, respectively; the respective values for this document are 2*0, 3*0, and 4. Commands of the same name are provided to set these three variables.

Sometimes it is desirable for a given field to start in a given position on the page, it being understood that it is to go on the next

---

(12) This prevents LO from looping trying to find enough space for an excessively large need.

line if that print position has already been reached. Examples in this document are the table of variables in Chapter 4 and the complete list of commands in Chapter 5. For this purpose the command

.charpos N

is provided. If at least one space remains before reaching column N, this command causes extra spaces to be inserted so that the next text character stored will be in column N; while if position N-1 has already been passed a new line is started and the effect of ".indent N" is simulated. In either case, the next character stored will be into column N. The effect is something like a tab to column N, with the proviso that a new line is to be started first if column N-1 has already been reached. The column number is taken with respect to the current value of #lmarg#.

2.3 – Substitution: An important LO idea is the substitution into text of the value of a variable. There are three contexts in which a variable is replaced by the string representation of its value: in the ".subst" command, in ".triple", and in headers and footers. In all three cases, the substitution algorithm is the same. The text in which substitution is to be performed is scanned for the appearance of variable names enclosed in "#"s, each such being replaced by the value of the variable. To oversimplify the algorithm, each instance such as "#foo#" is replaced by the (string representation of the) value of variable #foo#, whether #foo# is built-in or user-defined. If the variable is of type STRing, its value is used directly; if it is of type INTeger, its value is converted to a string (with a leading "-" if necessary) and used; if it is LINe, the form "F*P" is used, where "F" is an integer representing the number of full lines and "P" the number of part lines, with "P" less than #partsperline#(13); and it is a detected error if the variable is of type BUFfer.

Sometimes it is desirable to have a "#" in the line after substitution. For this reason, N or more successive "#"s where N > 1 are replaced by N-1 of them, with no substitution. This seems clear, but the algorithm is not quite that simple. The string to be substituted

---

(13) In the present implementation in which #partsperline# is fixed at two, "P" is either zero or one.

in is scanned from left to right, looking for "#"s, all other characters being copied into the output. If two or more "#"s are encountered, all but one are copied into the output and the scan continues. If exactly one "#" is encountered, the "#" and all characters up to the following "#" are scanned. The characters between the "#"s are looked up as a name(14), the string representation of that name is stored into the output, and the scan continues.

Consider the effect of this method on the text

    #abc##def#

in which variables "#abc#" and "#def#" appear. Note that the two adjacent "#"s in the middle of the line are NOT replaced by a single "#", as suggested by the first description.

The user may, if he desires, specify a variable delimiter other than "#". For example, the command

    .set variabledelimiter ?

or the abbreviated version

    .set vd ?

may be used to set it to "?", as described in Section 3.10. Note that changing the variable delimiter will impact on substitution in headers and footers.

As previously mentioned, one use of the substitution algorithm just discussed is in the command ".subst". LO interprets the command

    .subst <text>

by first applying the substitution algorithm to the <text> and then using the result as a line of input to LO. This line may be either text or commands; indeed, the result may also be a ".subst" command. For example, the command

    .rmarg rmarg - 10

may be used to set the right margin to a value 10 less than its previous value. The variable #rmarg# is accessed in expression context and is not to be enclosed in "#" symbols. The effect of

    .subst .rmarg #rmarg# - 10

---

(14) These characters may instead be an integer to represent a parameter to a macro, as discussed in Section 3.6.

would be the same, although LO would achieve the effect differently. If the previous value of #rmarg# were (say) 60, then applying the substitution algorithm to the above would yield

    .rmarg 60 - 10

which would have the proper effect.

2.4 – Text Output: The commands discussed in this section actually store certain text into LO's output.

To cause the text <text> to be displayed on a line by itself, centered between the margins, use the command

    .center <text>

This first causes a break and then displays <text> centered. A previous ".indent" is taken cognizance of in determing where to put the text. It is a detected error if the text is too wide for the current margins.

The command

    .triple '<text1>'<text2>'<text3>'

first forces a break. Next, substitution is performed in each of <text1>, <text2>, and <text3>, in precisely the manner discussed in Section 2.3. Finally, <text1> (as it exists after the substitution) is stored next to the left margin, <text2> centered on the page with respect to the current margins, and <text3> next to the right margin. (If there is a preceding ".indent" command, it has its usual effect.) The quote character need not be the single quote shown – it is the first non-blank character after the space following ".triple". It is a detected error if there is not enough space on the line for all of this. The parameter to ".triple" is a <triple>, and this term is used in the remainder of this document.

One purpose for which this command is convenient is numbered equations. For example, the command

    .triple //f(x) = x^2 + 2x + 1/(4.7)/

causes the equation to be displayed centered and the equation number to be right-justified, like this:

$$f(x) = x^2 + 2x + 1 \qquad (4.7)$$

There is a special feature to facilitate preparation of displays such as the Table of Contents of this document. As mentioned, it is a

detected error if the total width of the three parts of the triple exceeds the space available on the line. However, a special check is made after the excessive width is detected and before the error message is given. If, after substitution, the leftmost character of the center part is "#"[15], then just enough characters are stripped from the left end of the center part to make it small enough to fit. If not, the right end is similarly checked, with characters being removed from the right end. If neither end is a "#", then the error is reported. Thus the line

```
.triple 'Ch 2: Basic LO Capabilities'## . . . . . '13'
```

(assuming that there were enough dots in the center to make the whole triple too wide) could be used to store the entry for this chapter in the Table of Contents. The two "#"s are needed so that one will be left after the substitution. This example is considered in further detail in Chapter 6.

Sometimes it is desirable that text appear in the output just as it appears in the input. That is, the exact spacing in the input is to be copied into the output. (For example, this is useful in tables, such as the one on page 8.) LO provides several techniques to achieve this effect. If only one verbatim line is to be stored, the command

```
.plain <text>
```

may be used. This forces a break and then puts <text> on a line by itself, exactly as it appears in the input. A leading "." in <text> is not interpreted as introducing a command.

Sometimes it is useful to include a lengthy block of text in the output exactly as it appears in the input. One method of doing this is to turn off "fill" mode, as discussed above in Section 1.1. Another method is to use the command

```
.unproc
```

After this command appears, LO copies into its output all text, until it encounters the command

```
.proc
```

Note that in this mode all commands (except of course ".proc") are just copied but not obeyed, while turning off "fill" permits all

---

<15> It is the current variable delimiter that is used.

commands to be processed as usual. The effect of ".unproc" is terminated by the end of a buffer or the end of an input file, while the effect of fill mode is independent of these.

Normally, as mentioned earlier, any sequence of spaces, tabs, and carriage returns in the input is replaced by a single space. This is not the case in the three contexts just discussed. Not only are spaces copied verbatim, but tabs also may be given special effect. Details of the tab processing are provided in Section 3.1.

2.5 – Headers and Footers: LO may be directed to print header information at the top of each page, as well as footer information at the bottom. Up to 10 lines of headers and 10 lines of footers are allowed. They may be different on even and odd pages, to facilitate the preparation of documents to be printed on both sides of the sheet of paper. The header commands (".header", ".eheader", ".oheader") and footer commands (".footer", ".efooter", ".ofooter") all operate similarly, so it should be understood that the following discussion of ".header" applies equally to all of them.

The command

    .header N, <triple>

specifies header N, where $1 \le N \le 10$, for both even and odd pages. A <triple> is a three-part string, as discussed in Section 2.4 under the command ".triple". The <triple> and the current values of #lmarg# and #rmarg# are saved as header N. Each time a new page is started, each header currently saved is processed as for ".triple", first performing substitution and then storing each part of the <triple>. In so doing, the margins used are the values of #lmarg# and #rmarg# that were stored with the header line, and not (necessarily) those in effect at the time the new page is started. Note particularly that the substitution is performed when the page is started, and NOT when the ".header" line is encountered, so that variables whose value changes will print differently. For example, page numbers are requested by placing "#pageno#" in the <triple> at the desired place, so the command

    .header 1, ''DRAFT'Page #pageno#'

causes a header line on each page with the text "DRAFT" centered and "Page xx" at the right edge, with as many print positions as required

for the current value of ~xx~. Keep in mind that the variable delimiter (default ~#~) used is the one that happens to be in effect at the time the header is printed, so great care should be used in changing it.

Note the difference between a null header and a header with a <triple> consisting of only the four break characters: The first produces no header line; the second produces an empty header line. To specify a null header, use no comma on the line.

Initially, all header and footer lines are null.

2.6 – Page Numbering: LO provides several facilities to let the user control the numbering of output pages. The command

.page N

causes first a break and then a page eject. If N is present, the next page is numbered N (and the next N+1...); if not, the page numbering continues unaltered.

At any moment while storing text into the output, the variable #pageno# holds the number of the current page, and #nextpage# holds the value to be assigned for the next page. The proper way to alter the page numbering sequence is to change #nextpage#, either with

.nextpage N

or

.page N

or by ~.store~ing into #nextpage#. Changing #pageno# is seldom useful, for two reasons. First, if there is a footer line involving #pageno#, the page being output probably wants the old page number rather than the new one. To see the second problem, it is first necessary to know how LO processes page numbers. When a page eject takes place, either because the page is full of text or because of the ~.page~ command, the footers are printed with the current value of #pageno# and #pageno# is then set to #nextpage#. Just before the first line of text is stored into the next page, #pageno# is again set to #nextpage# and then #nextpage# is incremented by one. Then the headers are printed and then the text. Thus, setting #pageno# by ~.store~ing into it is less than useful, since it has no permanent effect. Note that footers are printed on page eject, while headers are not printed until the first text line is stored on the new page.

2.7 – Buffers: The program maintains buffers for accumulating lines of text for later insertion into the output file or for rescanning as input to LO. Comments in this document about "storing lines into the output" refer to adding lines to either the currently active buffer or the output file. The command

.buffer B

is used to switch output to buffer B. As a further effect, the variable #actbuf# is set to the name of the buffer. If no buffer is given in the command, output again is aimed at the output file, and #actbuf# is set to the empty string. Switching to a buffer causes data to be appended to that buffer.

When switching from the output file to a buffer, there is NOT a line break. That is, partial output to the file is held while output is to a buffer, and restored when output returns to the file. On the other hand, switching output away from a buffer forces a line break in that buffer.

To clear the contents of buffer B, give the command

.clear B

If no buffer is specified, the buffer currently active to receive output is cleared. It is a detected error if this command is given when output is to the output file. It is also a detected error if buffer B is currently active, at any level, for input. This last point is addressed at further length in Section 3.6 in which the use of buffers as macros is discussed.

β may be used as described in Section 1.3 to determine the number of lines in a buffer, and γ may be used to determine the amount of vertical spacing the buffer would take up were it ".insert"ed.

All buffers are initialized to be empty when they are first declared.

It is a detected error to attempt to use the same buffer for both input and output.

A buffer may either be inserted verbatim into the output, or it may be rescanned as input to LO. The command

.insert B

permits inserting the contents of buffer B into the current output

(either buffer or output file). In this mode the text is set up as lines when it is stored into the buffer, with vertical spacing controlled by the value #lspacing# had when the buffer was created and the line lengths, margins and indentations being similarly controlled. Page spacing is irrelevant to the creation of a buffer, page breaks being inserted only as the text is finally output to the output file.

Usually the spacer "ƀ" is replaced by space only when the text is finally output to the file. Sometimes it is desirable that this substitution be done as the buffer is being created, since it is possible that a different spacer character may be in effect when the buffer is finally output. At such times, deletion of the hyphenator character should also be done as the buffer is being created rather than when it is inserted. To achieve these effects, a buffer may be specified as an "insert buffer" by issuing the command

    .set on insert

when the buffer is active for output. The footnote buffer #fnbuf# is an insert buffer, while all user-defined buffers are initialized to be non-insert buffers.

The command

    .rescan B

causes buffer B to be rescanned as input to LO. Arguments may be passed to this rescan, as discussed under Macros in Section 3.6.

2.8 – User-Defined Variables: There has been much discussion of the use of variables in various contexts, with most of the examples using one of the many variables "built-in" to LO. In addition, LO lets the user declare his own variables, specifying for each the data type for the possible values of that variable(16).

A variable must be declared before it can be used, just as in many programming languages. Four declaration commands are available:

---

(16) The following discussion presents the idea that a data type is specified for each variable at the time of its declaration, and that this type is forever associated with that variable. This is not, in fact, the way LO works; it is the way I wish it worked. It's too late to change it now, since there is too much existing LO input text for me to make this non-upward-compatible change. I recommend that all new LO text be prepared using the method about to be described.

```
.declarebuf <V-list>
.declareinteger <V-list>
.declareline <V-list>
.declarestring <V-list>
```

Here <V-list> is a list of variables, separated by commas. Each variable is declared to be of the type specified and initialized to a default value. For BUFfers the default is an empty buffer; for INTegers it is zero; for LINes it is 0*0; and for strings it is the empty string. (These defaults may be changed using

```
.set default ...
```

as described in Section 3.10.)

Once a variable has been declared, either by one of the above commands or by virtue of its being built-in, it may be accessed in either expression or substitution context, as already described. Further, except for those built-in variables that are read only, its value may be changed by the command

```
.store V,<value>
```

This changes the value of variable V to a new value of the same type that V previously held. If V was type INTeger, <value> is scanned as an integer; while if V was of type LINe, it is scanned as a line. If V was of type STRing, all characters from just after the comma to the end of the line are taken as the new value of V. It is a detected error if V was of type BUFfer.

When a variable is no longer needed it may be discarded: The command

```
.undeclare <V-list>
```

causes each variable in the list to be dropped from LO's table of variables. If the variable was a buffer, the buffer is first cleared. (It is a detected error to attempt to clear a buffer currently active for input or output.) Undeclaring a built-in variable is permissible. The effect is that the value of the variable is no longer available to the user, although LO still knows about it. For example, undeclaring #lmarg# makes the variable #lmarg# unavailable for expressions or substitutions, but the user may still change the left margin using the command ".lmarg". However, he no longer has a way to determine what value is in effect for the left margin.

If a variable already declared is redeclared, the old value is "pushed down" and is no longer accessible. Undeclaring it later causes the old value to be "popped up" so as again to be in effect. There is no limit to the number of "pushes" that may be in effect at any moment. If the user wishes, he may direct LO to give him a warning message if he redeclares a variable, using

```
.set on warn redeclare
```

to elicit a warning message on declaring user-defined variables, or

```
.set on warn redeclareperm
```

to elicit a message on redeclaring built-in variables, as described in Section 3.10. The latter command also causes a warning on undeclaring any built-in variable.

2.9 – Conditionals: There is a conditional command, to let the continued operation of LO be dependent on the values of variables. The command

.if V_1, <rel>, V_2, <text>

causes value V_1 to be compared with value V_2, where <rel> is one of the relations "ge", "gr", "eq", "ne", "ls" or "le". If the relation holds, the <text> is processed as input to LO; otherwise, it is ignored. The <text> may be either commands or text.

The values V_1 and V_2 are computed as if they appeared in LINe context, with the exception that negative values are permitted. The default conversions result in proper results if the expressions are INTegers. Both of

```
.if lspacing, eq, 2, ...
.if lspacing, eq, 2*0, ...
```

have the same effect. It is also true that both of

```
.if day, le, 9, ...
.if day, le, 9*0, ...
```

have the same effect, although one would be unlikely to write the second one since #day# is INTeger-valued.

The command ".if" lets one control the execution of only one line (although that line may have several commands on it). To control more complex actions, the command

```
.skip N
```

may be used. The effect is to skip over and ignore the next N lines of LO's input. The obvious use, of course, is

```
.if ... , .skip 5
```

For those who do not like to count lines, the command

```
.skipto <label>
```

may be used. This causes LO to ignore all lines until it encounters

```
.label <label>
```

in which the <label>s match[17]. This latter command is treated as a comment if it is encountered in LO's processing. Thus one might have

```
.if ..., .skipto aaa
stuff...
.label aaa
```

for a one-armed conditional, or

```
.if ..., .skipto aaa
stuff...
.skipto bbb
.label aaa
more stuff...
.label bbb
```

for a two-armed conditional. The effect of either ".skip" or ".skipto" is terminated by the end of an input source, either a buffer or a file, with a warning message to the user.

<17> Two labels match if the first three characters are the same and they are of the same length. Thus "foobar" matches "footle", but neither matches "foolish".

Chapter 3: Advanced LO Topics

This chapter continues the discussion of Chapter 2, but it addresses topics less likely to be of interest to the beginner. For that reason, the discussion is slightly more terse, as well as more complete.

3.1 – Tabs: As discussed in Section 1.1, LO normally treats as a single space any concatenation of spaces, tabs, and carriage-returns that appears in the input. Also mentioned there is that special processing of tabs is possible in text in ".plain" commands and in text printed when fill mode is off and under the control of ".unproc". (See also Section 2.4.) The processing is that each tab encountered in such text is replaced by an appropriate number of spaces before any further processing. To use this feature, one must tell LO where it is to assume tabstops in the input, using the command

.tabsin N_1, N_2, ...

Here the N_k are INTeger expressions, positive and each greater than the preceding one, that specify the positions of the tabs in the input medium. In scanning the input text, LO keeps track of the current print position of the input. (Proper account is taken of backspace as well as nonspacing characters such as underscore.) Then each tab character is replaced by the number of spaces necessary to reach the next tab stop as defined by the last ".tabsin" command.

Note that this complete discussion is with respect to the input text only. Parameters such as #lmarg# and #indent# and #window# do not impact on this processing.

For tabs uniformly set across the carriage, the usage

.tabsin × N

may be used to set tabs every N positions. That is, the two commands

```
.tabsin × 60
.tabsin 60, 120, 180, 240, 300
```

are equivalent in effect. (The maximum line width is 300, so the last stop is at position 300.)

In the following examples, the notation ť is used to stand for a tab character and ƀ for backspace. Following the command

```
          .tabsin 4, 8, 12, 20, 25
```

the following pairs of lines (with "ħ" used in the second line for space) are equivalent in effect:

```
          .plain abťcdťefťghťij
          .plain abħħcdħħefħħghħħħħħħij

          .plain abcdeťfg
          .plain abcdeħħħfg

          .plain abcdeťfg
          .plain abcdeħħħfg

          .plain |ƀ-ťz
          .plain |ƀ-ħħħz
```

This last prints as if the user had typed

```
          .plain +   z
```

The default setting when LO starts processing is no tabs at all. This setting may be restored by using ".tabsin" with no arguments.

The command ".tabsout" is proposed for specifying tab stops in LO's output. It is not yet implemented, and is unlikely to be unless a desire is expressed.

3.2 – Sentence Ending: LO inserts an extra space between sentences. LO assumes a sentence end when one word ends with period (or any other sentence ENDER) and the next word starts with an upper-case letter (or other sentence STARTER). Further, a right parenthesis (or other RPAR) appearing after an ENDER also indicates a sentence end. Finally, a word only one or two characters long (including the ENDER) is not interpreted as the last one in a sentence, as described below. The set of ENDERs is initially ".", "?" and ":"[18], and ")" is the only member of RPAR. The members of STARTER are initially the upper-case letters and "(". Membership in each of these three sets is controlled by the ".set" command. The command

```
          .set starter <char> <char> ...
```

causes each of the <char>s to be added to the set of the STARTERs, while

---

(18) The colon is the TX-2 character red hand.

.set off starter <char> <char> ...
causes the listed <char>s to be removed from the set of STARTERs. Similar options are provided for controlling membership in ENDER and RPAR. Details are provided in Section 3.10. In this document, the STARTERs consist of the upper-case letters and "(" (default), along with "α", "β", "γ", "λ", "#", "<" and the open-quote character. The only extra ENDER is the TX-2 character NORMAL.

A special case is made of a "word" exactly two characters long, the second of which is an ENDER. In typing a person's name with an initial, LO would normally detect a sentence end after the initial (since the next name would start with an upper case letter). Since an extra space is not wanted in this case, LO does not assume such a short word to be a sentence ender. The user may override this default by the command

.set on wide

See Section 3.10 for further details.

3.3 – Hyphenation: Sometimes fill mode results in a rather short line if the next word to be stored is rather long. If adjust mode is also on, an unaesthetically large amount of blank space may be left between words. In this situation the usual solution (as in books and newspapers) is to hyphenate the offending long word. It would be pleasant if LO had an effective algorithm for deciding where to insert hyphens in English words, but I know of no such(19).

Since LO is not intelligent enough to determine where to put hyphens, provision has been made for the user to tell LO where hyphens are permissible. If fill mode results in more than four spaces being left at the end of a line, an attempt is made to hyphenate the next word. (Hyphenation is attempted only if the user has specified a hyphenation character. Default is no hyphenation; the remainder of this discussion assumes that "~" has been set as the hyphenator.) The word is examined for the presence of the character "~", and each is assumed to appear before a syllable break. LO picks the longest prefix that will fit, inserts a hyphen after it, and prints the rest of the word on the next line. Further, each instance of "~" in the

(19) I will welcome suggestions from users.

input file is removed from the output. Finally, if the user does not like "~", he may specify some other hyphenation character, using the command

    .set hyphenator □
or the abbreviated version
    .set hy □
to set the hyphenator to "□". Obviously, this part of this document uses a hyphenator other than "~". It is convenient to use one of TX-2's nonspacing characters for hyphenation.

There is interaction between hyphenation and the use of the spacer character "ʎ" (default). If the "~" appears before or after an "ʎ", the "ʎ" (as well as any "ʎ"s that immediately precede or follow it) is deleted and no hyphen is inserted. This allows the user, by inserting an "ʎ", to specify that spaces are not to be inserted during adjustment, while still permitting LO to break the line at that point. Consider, for example, a name such as "A. B. Smith". This looks ugly if LO inserts adjustment spaces after an initial, like this: "A. B. Smith". If the user types "A.ʎB.ʎSmith" no spaces will be inserted, but LO will perforce put the entire name on a single line. Typing instead "A.~ʎB.~ʎSmith" will result in no spaces being inserted during adjustment, but will permit LO to break the line after either initial. "A.ʎ~B.ʎ~Smith" would work equally well. This feature is also useful in equations that appear in text.

There is one other special case in hyphenation: If the last character of a syllable is "-", LO does not insert an extra hyphen. This lets the user permit a line break at a place that already contains a hyphen. For example, if LO breaks the typed word "upper-~case" it will put "upper-" on the first line and "case" on the second, not inserting the obviously unwanted hyphen.

3.4 – Footnotes: A convenient mechanism is provided for setting up footnotes. The sequence

```
.footnote
<text> (perhaps many lines)
...
.footnote end
```

causes the <text> appearing between the two ".footnote" commands to be set up as a footnote. The footnote is saved (in the buffer #fnbuf#)

and is inserted into the output at the bottom of the page. This sequence does not cause a line break in the main body of the text. All the above sequence must be in the same input source, unless the <text> contains completely nested inclusions. That is, it is not possible to have a macro one of whose effects is to initiate a footnote.

It is the user's responsibility to store the reference to the footnote – there is nothing automatic about this. The user must be careful to insure that the footnote reference and the footnote appear on the same page. Suppose that LO encounters the text

```
I am[1] some text containing a footnote reference.
.footnote
1. I am the text of the footnote.
.footnote end
```

It could happen that the end of a page could occur, say, after the word ~containing~, in which case the footnote would be on the next page. To be guaranteed safety, the footnote should appear immediately after the reference, like this:

```
I am[1]
.footnote
1. I am the text of the footnote.
.footnote end
some text containing a footnote reference.
```

Section 3.6 shows the macro used to store footnotes in this document.

The usual printer's conventions regarding footnotes are followed. If the footnote occurs on the last line of a page, it is held till the next page. If a footnote is too long to fit on the page, the excess is continued on the next page. A break line is printed (with an #lmarg# of zero) after the last text line and before the footnote. The default footnote break is ~--------------------~, a string of 20 minus signs[20]. The user may change the footnote break by storing into the built-in STRing variable #fnbreak#, with ~.store~. The amount of space between the last text line and the break is given by the (LIN-valued) variable #fnsp1#. #fnsp2# specifies the spacing between the break and the beginning of the footnote text, and #fnsp3#

<20> This document uses a character set in which red minus prints as ~–~ (as opposed to ~-~ for the usual minus) and the footnote break is 30 red minus signs.

specifies the space between successive footnotes. Defaults are 2*0, 1*0 and 2*0 for #fnsp1#, #fnsp2# and #fnsp3#, respectively; for this document, their respective values are 1*0, 0*1 and 0*1. They may be set using ".store".

Before setting up a footnote, LO sets #lmarg#, #rmarg#, and #lspacing# to #fnlmarg#, #fnrmarg#, and #fnlspacing#, respectively. (Their respective defaults are 0, 71, and 1*0.) On completion of setting the footnote, they revert to their old values. They may be set using ".store". If any flags are pending when the footnote is encountered (see Section 3.5) they are held until the footnote is complete. On the other hand, flags may be set up as usual in a footnote if the ".flag" command occurs after the ".footnote" command.

It is sometimes desirable to suppress the printing of footnotes, such as while outputting a figure. The command

```
.footnote push
```

causes all pending footnote text to be held in abeyance until the command

```
.footnote pop
```

appears.

3.5 – Flags: The user may have a flag stored in the right margin, as shown. The command *

```
.flag N
```

causes the next N lines to be flagged, where if N is missing it is taken as one. Each ".flag" command supersedes any previous such command, so that a large block of text may all be flagged by the sequence

```
.flag 9999
<text>, extending over many lines
...
.break
.flag 0
```

The ".break" command is needed to insure that the last line is flagged.

The user may select a flagger other than "*". The command

```
.set flagger xxxx
```

sets the flagger to "xxxx". If an empty flagger is specified, no flags will be printed, even if ".flag" commands appear. The variable

#flagger# holds the flag currently in use. #flagger# may be changed only with the ".set" command just shown and not with ".store", since it is read-only.

The default position for the flag is in column 74 - three to the right of rmarg. The user may set a different value by ".store"ing into #flagcol#. If an adequately wide window is being used (see the discussion of windows in Section 2.1) the flag may be stored to the left of the text by specifying a negative value of #flagcol#.

A flagged line going into a buffer for later ".insert"ion is marked in the buffer as flagged and is printed with the flag when the buffer is ".insert"ed. The flag and flag column used for printing are those in effect when the line is stored into the buffer, and not (necessarily) those in effect at the time of printing.

See Section 3.4 for special treatment of flags with respect to footnotes.

3.6 - <u>Macros</u>: Rescanning a buffer provides a macro-like ability in LO, and this facility is augmented by permitting the user to pass parameters to the rescan. For example, the command

```
.rescan foo zilch barf
```

causes the contents of buffer "foo" to be rescanned as input to LO. During this scanning any instance of "#1#" to be substituted for (as in the ".subst" command - see Section 2.3) is replaced by "zilch", and "#2#" is replaced by "barf". Further, during this rescanning the variable #params# has the value two, indicating that two parameters were passed in invoking the macro. The command word "rescan" need not be typed; the line

```
.foo zilch barf
```

is equivalent to the above. This lets the user use macros with syntax similar to that of built-in commands.

The command syntax permits three types of arguments to be passed: If the first character is an upper- or lower-case letter, a digit, or the character ".", a sequence of such characters is read and passed as the argument. The actual parameter is the characters read. If the first character is "→", the expression up to the next comma or end of line is evaluated as an integer and the parameter is the string

representation of that integer. If the first character is any other character, it is taken as a quote character and the parameter is all text up to (but not including) the next occurrence of that quote character. A comma may be used optionally after the first or third types of arguments; it is required after the second type unless that argument ends the line.

Invoking a macro causes the old values of #params# and of the parameters to be "pushed down" before the values associated with the call are stored; returning from a macro causes these to be restored. Thus a macro may continue to access its own parameters with no trouble after invoking another macro.

The arguments passed to LO from the BT when LO is invoked may be accessed with this parameter mechanism. #1# refers to the name of the input file specified to BT and #2# to the name of the output file. (It is ".lo" if no output file is specified.) Any other arguments specified are similarly available. This fact is alluded to in Section 7.1. #params# holds the number of parameters passed through the BT.

A macro overrides a built-in name, so that, for example, existence of a buffer named "para" keeps the user from accessing the ".para" command. (Of course he may still access it through its abbreviation ".pr".) This overriding may be suppressed by using "+" just after the ".", so that

```
.+para
```

will invoke ".para" even in the above case. A variable named "para" with type other than BUFfer has no effect on the accessibility of ".para".

Since a macro usually contains command lines starting with ".", a convenient mechanism is provided to store such lines into LO's output. Any line starting with ".." has the first "." stripped off, and the rest of the line is stored into the output verbatim. The values of #lmarg# and #indent# are ignored.

A special case is made when no arguments are passed. In this case, the body of the macro may access the parameters that were available in the caller's environment. Also, #params# has the value it previously had. Sometimes it is desirable to pass zero arguments explicitly, with #params# equal to zero. For this reason, the command

```
        .rescan foo
```

invokes buffer "foo" with the parameters that were in force in the calling environment, while the command

```
        .foo
```

invokes "foo" with #params# equal to zero. If parameters are passed the two syntaxes are identical.

The command

```
        .end
```

appearing in the body of a macro causes expansion of that macro to terminate and control to return to the caller on completion of processing the line containing this command. This is useful, for example, in conditionals. (See Section 2.9.) Normally it is a detected error if a buffer attempts to ".clear" itself. There are two exceptions to this restriction: if the ".clear" command occurs on the last line of the buffer, or if it occurs on any line after the appearance of ".end" on that line.

Recursive macro calls are permitted in which a macro invokes itself. Of course a conditional is needed somewhere to prevent an infinite loop. There are two limitations to the maximum recursion depth. LO maintains a stack of input sources with one entry for each ".include" file and one entry for each macro currently in force. The maximum depth of this stack is currently 40 (although this could be changed easily if it turns out to be a restriction). The second limit is the BCPL runtime stack. Care in coding can minimize the impact of these two restrictions. If the recursive call is on the last line of a buffer, or if it is preceded on its line by ".end", then no new entry is made in the input stack. This fact can easily be taken advantage of to keep 40 inputs from being a significant limit. To minimize the use of BCPL stack, be sure the recursive call is not on a line with ".subst". Recursive use of ".include" will work but is not recommended; it is exceedingly inefficient. The limitation in this case is virtual address space required.

A simple example of a macro is the one used in this document for storing footnotes. Text substantially like the following appears as part of this document:

```
.declareinteger n‖ .store n, 1    |  footnote counter
.declarebuf fn‖ .buffer fn
..subst #1#<#n#>#2#
..footnote
..subst <#n#>hh#3#
..footnote end
..store n, n + 1
.buffer
```

This defines a macro that takes three parameters, as indicated by the fact that the largest integer between "#"s is three. Now, note the footnote that appears at the bottom of this page[21]. This appears as a result of the following lines of input to LO:

```
Now, note the footnote that appears at the bottom of this
.fn 'page' '.' 'I am a sample footnote.'
This appears as a result of...
```

The footnote reference occurs between the first and second parameters to ".fn" without any added spaces. The variable *#n#* is an integer that counts footnotes. This example will result in footnotes being numbered consecutively throughout the document; a modification to reset the counter on each new page is shown in the next section.

3.7 – <u>Traps</u>: LO permits the user to specify any of three traps. A trap in LO, as in many programming languages, permits the user to specify that a certain action is to be performed any time a certain event takes place[22]. Three events may be trapped in LO: the end of a line of output, before printing headers at the top of a new page, and before printing any text at the top of a new page (but after printing the headers). The command

```
.trap <event> <action>
```

is used to set a trap. Here <event> is any of "endofline" (or "eol"), "newpage" (or "np"), or "pagetop" (or "pt"), matching the three events mentioned above, and "<action>" is any line of text suitable for input to LO. The effect of this command is that <action> is stored as the value of the string-valued variable named <event>[23]. When any of the three trappable events occurs, the appropriate variable is examined. If its value is other than the empty string, the following

---

<21> I am a sample footnote.

<22> A trap is similar to the "on condition ..." facility in PL/I.

<23> For example, setting a trap for "endofline" causes <action> to be stored as the value of the variable #endofline#.

actions are performed: First, the value of the variable is saved; then the variable is set to the empty string; and finally the saved value is processed as a line of input to LO. With the exception mentioned below, this may involve any of LO's capabilities, including storing text or giving command lines. If the user desires an effect too complex to be described conveniently on one line, the action may be to invoke a macro which does the work.

Normally, a trap is executed only once and then cleared, as just described. However, if the command

    .reset

is encountered while processing the <action> line, then that trap is reset on completion of its processing.

There is one exception to the ability to do anything while processing a trap: The "newpage" trap takes place at the top of a page before the headers are stored. LO is unprepared at this time to store text into the output, so no attempt should be made to do so. Text to appear at the top of the next page should be stored with the "pagetop" trap. The column headers for the three tables in Chapter 8 are stored this way.

A simple example may be of interest. Suppose page numbering in roman numerals is desired. Although LO provides for conversion of an integer to roman numeral form, no automatic mechanism is provided for storing roman page numbers; it must be programmed. Normally, page numbering is achieved by referring to the variable #pageno# in a header or footer. Suppose the user uses instead the variable #page.roman# in his header, and includes in his input to LO the text

    .trap newpage .storeroman page.roman, pageno‖ .reset

This means that at the beginning of each page, after setting #pageno# but before printing headers, the variable #page.roman# is set to the string representing the roman numeral equivalent of #pageno#, and the trap is reset. (See the description of the ".storeroman" command in Chapter 5.)

As another example, consider the footnote macro shown on page 38. To reset the footnote counter on each new page, the command

    .trap newpage .store n, 1‖ .reset

might be used.

A trap may be cleared by storing the empty string into the associated variable, either with ".store" or

.trap <event>

A "pagetop" trap still pending at the end of the input text is processed. However, any ".reset" encountered while processing this trap is ignored. Thus if a "pagetop" trap with reset is being used to print column headings, there will be one page at the end of the output with column headings only, but there will be no more of them.

To invoke a trap previously set without waiting for the associated event, use

.trap do <event>

This causes the trap stored for <event> to be performed and then cleared (unless it ".reset"s itself). It works even if there is no trap currently set. To determine whether a trap, say "endofline", is currently set, the code

.if λ endofline, gr, 0, ...

is convenient.

3.8 – <u>Input/Output</u>: LO provides various ways to communicate with the world around it while it is running: APEX files may be read or written; commands may be directed to APEX; and there may be communication with the user's console. These abilities are discussed in this section.

To include a file from the current APEX directory, use

.include <file>

where <file> is the name of an APEX file. This command is replaced by the contents of the named file. Note that certain types of processing, such as ".unproc", footnotes, ".skip", etc., may not extend over the end of a file. If these are started in the file, they must also end in that file. No direct mechanism is provided for including a file from another directory. However, the ".bt" command described later in this section may be used to SGET a copy of the file into a dot name, and that may then be ".include"ed.

The command

.end

causes the rest of the input file to be ignored. It is useful in

conditionals - see Section 2.9. Note that this command also may be used to terminate a macro, as described in Section 3.6.

The command

.output B, <file>

causes the contents of buffer B to be written as an APEX file named <file>. (The comma is optional.) Each line is copied directly followed by a single new-line character. Format concepts such as pagination, headers and footers, and the value of #lspacing# are irrelevant in this process.

Any BT commands may be invoked from LO. The command

.bt <text>

causes the <text> (with an appended carriage return) to be passed to 5BTF on the next map.

The command

.console <text>

causes the <text> to be written on the user's console. If there is no <text>, then the user is given a prompt to invite him to type a line, and LO then processes that line as input. (The prompt is <newline> "?" <tab>.) NO and DELETE may be used as usual under APEX for line delete and character erase, respectively. The output appears on the console even if the "e" option was used in invoking LO.

One use of ".console" is in debugging complex macros. For example, to see what value variable #zilch# has at a certain point in the processing, include the line

.subst .console the value of zilch is #zilch#

3.9 - The Command ".overstrike": LO provides a command to facilitate the preparation of overstruck text, usually for underlining. The command is

.overstrike <quoted word> <text>

The first non-blank character after the space after "overstrike" is taken as a quote character, and all characters up to the next occurrence of that character are taken as the <quoted word>. The <text> (which starts immediately after the second occurrence of the quote character) is scanned and each overstrikeable character is preceded by the characters in <quoted word>. The result is then

processed as usual by LO.

Underscore is frequently used for the <quoted word>, as in

    .overstrike '_' stuff to be printed

which has the same effect as if the user had typed

    stuff to be printed

Recall that the character underscore on TX-2 is a nonspacing character.

The overstrikeable characters are initially the letters and digits; this set may be altered by the command

    .set overstrike <char> <char> ...

to mark the indicated characters as being overstrikeable, as described in Section 3.10. In this document, the period ".” has been marked overstrikeable, as may be seen in the chapter and section headings.

3.10 – The Command ".set": The command line ".set" may be used to control certain of LO's internal parameters. It turns on or off certain switches, as determined by SWITCH. Normally SWITCH is ON; it may be turned off for the duration of a particular ".set" line by the control "off", as described below.

This command causes a break only for the "fill" option and not for any of its other options.

".set" is followed optionally by either "on" or "off", followed by a control word which, in turn, is followed by optional parameters. All of the options are described in the remainder of this section, and a tabular summary of the options appears in Section 8.3. Note that abbreviations are shown for some of the options.

adjust — The adjust switch is set to the value of SWITCH. Right margins are lined up (as in this document) when the switch is ON. The default value is ON.

default <type> <value> — The default value for newly declared variables is set. If <type> is "int" or "INT", <value> is read as an INTeger value and that value is hereafter used as the default value for variables declared with ".declareinteger". Similarly, if <type> is "lin" or "LIN", <value> as a LINe is used for ".declareline"; and if <type> is "str" or "STR" the rest of the line is taken as the <value> and is used for ".declarestring". If the line is empty after the word "default", the usual defaults are reset: 0 for INT, 0*0 for LIN, and the

empty string for STR.

ender <char> <char> ...
: Each character listed is either added to or deleted from the set of sentence enders, as SWITCH is ON or OFF. This set is used to decide when to insert an extra space at the end of a sentence. If no character is given, all graphical characters are set. The default enders are ".", "?", and ":"(24). See Section 3.2.

fill
: The fill switch is set to the value of SWITCH. When it is ON (default), as many words as possible are added to each output line so as to fill it; while otherwise text lines from the input are copied into the output. This sub-command causes a break. See the discussion in Section 1.1.

flagger <word>
: The characters in <word> are used hereafter instead of "*" (default) for flags controlled by ".flag". If <word> is missing, no flags will be printed. See Section 3.5.

hyphenator <char>
hy <char>
: The hyphenation indicator is set to <char>. If <char> is missing no hyphenation is attempted. The default is no hyphenation. See Section 3.3.

insert
: If output is to a buffer (the command is otherwise illegal), that buffer's insert switch is set to SWITCH. The default is OFF for all user-declared buffers; it is ON for the built-in buffer #fnbuf# used for storing footnotes. When a line is stored into an insert buffer, any spacer is replaced by a space character and any hyphenator is deleted. Further, when an insert buffer is ".insert"ed, no processing is done for any spacer or hyphenator. See Section 2.7.

off
: SWITCH is turned OFF for the rest of the line. It is initially ON at the beginning of processing each ".set" command.

on
: SWITCH is turned ON for the rest of the line. Since it is ON by default this option is not really needed, but it provides a symmetry that seems pleasing.

overstrike <char> <char> ...
ov <char> <char> ...
: Each character listed is either added to or removed from the set of overstrikeable characters, as SWITCH is ON or OFF. (The overstrikeable characters are those used by the ".overstrike" command, described in Section 3.9.) The appearance of any lower- (upper-) case letter controls all the lower- (upper-) case letters, just as the appearance of any digit controls all the digits. If no character is given, all printable characters are set. The default set of overstrikeable characters is the upper- and lower-case letters and the digits. It is not possible to specify space as being overstrikeable, although the effect may be achieved

---

(24) The colon is red hand.

with judicious use of "h".

rpar <char> <char> ...
This controls membership in the right parenthesis set for extra space insertion at the end of a sentence. Operation is as for "ender", above. The default set contains only the right parenthesis ")". See Section 3.2.

spacer <char>
The character <char> is used hereafter instead of "h" (default) as the nonbreaking space.

starter <char> <char> ...
This controls membership in the set of sentence starters used to determine when to insert a space between sentences. Operation is as for "ender" above. The default set is the upper case letters and the left parenthesis "(". See Section 3.2.

variabledelimiter <char>
vd <char> The character <char> is used hereafter instead of "#" (default) to delimit variables in ".subst" commands and in triples, including headers and footers. Because headers can be printed at any time and the substitution in them uses the current variabledelimiter, it is best if this is changed only at the beginning of a job, or not at all. See Section 2.3.

vwf This is to be used if the output is for a device with a variable width type font. See Section 3.12.

warn <option> <option> ...
The user may, if he chooses, request LO to warn him about certain situations. If <option> is "redeclare" (or "rd"), any redeclaration of a previously declared user variable will produce a warning message. If <option> is "redeclareperm" (or "rdp"), any redeclaration of a built-in variable will produce a warning, as will any attempt to undeclare such a variable. Finally, if <option> is "conversion" (or "cv"), LO will warn the user of any conversion between INT and LIN, and will warn about the use of any of the following commands: ".declare", ".storeinteger", ".storeline", or ".storestring". Use of this command turns the relevant warning option ON or OFF, depending on SWITCH. "on" or "off" may be inserted among the options to alter the sense of SWITCH. Warnings are discussed in Section 1.5.

wide A switch W is turned ON or OFF, depending on SWITCH. When W is ON, a word one or two characters in length may end a sentence, while if W is OFF such a word is never assumed to end a sentence, even if ends with an ENDER. See Section 3.2 for further discussion. Default is OFF.

width N, <char> <char> ...
The named characters are treated hereafter by LO as if their printed width were N. This option may be used only if a variable width font has been specified. See Section 3.12.

window <option> V, <option> V, ...

This command controls the page window. If <option> is "even", "odd" or "both", then V must be an INTeger valued expression. Depending on <option>, #ewindow#, #owindow# or both of them are set to V. On each even-numbered page, #ewindow# spaces are left before each line; #owindow# spaces are left on each odd-numbered page. If <option> is "top", V must be LINe valued, and #topwindow# is set to that value. Thereafter that much vertical space is left at the top of each page, the space not being part of #bline#. All three windows are default zero. See Section 2.1.

3.11 – <u>The Command ".expand"</u>: A mechanism is provided to iterate through a sequence of arguments. The command

.expand <command>, N, A_1, A_2, ...

causes <command>, which may be either a built-in LO command or a user-defined macro, to be invoked repeatedly, each time with N (or fewer) arguments, until all the arguments to ".expand" are used up. That is, the line

.expand foo, N, A_1, A_2, A_3, ...

is equivalent to the lines

.foo A_1, A_2, ..., A_N
.foo A_{N+1}, A_{N+2}, ..., A_{2N}
...

If the number of arguments supplied to ".expand" is not a multiple of N, there will be fewer than N arguments to the last call to "foo". The syntax of parameters to this command is the same as that for macros, as described on page 35. For example, the command

.expand store, 2, lspacing, '1*1', lmarg, 0, fnbreak, '----'

is equivalent to the commands

```
.store lspacing,1*1
.store lmarg,0
.store fnbreak,----
```

A convenient use of this command is to set switches in LO from the BT. As described in Sections 7.1 and 3.6, the variable #params# holds the number of parameters supplied to LO from the BT. If one types the command

5do art lo stuff - BEGIN lspacing 1 rmarg 60 WORD-EXAM

to the BT[25], then when LO starts its work #1# will hold "stuff", #2#

---

(25) Here "BEGIN" and "WORD-EXAM" refer to the TX-2 characters used by 5BT as open and close quotes, respectively.

will hold ".lo", #3# will hold "lspacing 1 rmarg 60", and #params will hold 3. If the command

```
.if, params, ge, 3, .subst expand, store, 2, #3#
```

appears near the beginning of file "stuff", the effect will be to set single spacing and a right margin of 60. The ".if" lets it be optional to pass the third parameter.

3.12 – Variable Width Fonts: Most typewriters and most computer output devices print all characters equally wide. The term "variable width font" suggests a type font in which, for example, the "m" is wider than the "i". A mechanism is contemplated for LO which will facilitate preparation of documents for such devices. It is not yet implemented, and until it is the "vwf" and "width" options to ".set" are treated as erroneous. Please see the author for further information or to offer suggestions.

Chapter 4: Built-In Variables

A table of variables is maintained by LO. Some initial variables are declared by the program and are updated as the program proceeds with the text processing. Others are declared by the user, as described in Sections 1.2 and 2.8. This chapter lists all the built-in variables and describes their use in LO.

The initially declared variables with their default values are listed in alphabetic order on the next few pages. In the column headed "Type", N indicates INTeger, S indicates STRing, L indicates LINe number, and B indicates buffer. A preceding "*" indicates that the variable is read only and may not be altered by a ".store" command.

The column headed "Val" shows each variable's default value - the value to which it is initialized when LO starts its work. For type LIN the value is the number of whole lines. An entry of "→" means that the value is stated in the "Description" to the right. An entry of "~" means that the initial value is irrelevent. An entry of "--" indicates that the initial value is not the same each time LO is invoked, as follows: The integer variables #year#, #month#, #day#, #weekday#, #hour#, #minute# and #second# are initialized to the current date and time each time LO is invoked; they are not updated during LO's processing. The variable #loginname# is initialized to the user's login name as derived from APEX, with a preceding script "q" if quasi. The variable #params# is set to the number of parameters supplied to LO when it is invoked from the BT, as described in Sections 7.1 and 3.6.

Any of these variables (with a few exceptions) may be assigned a different value to impact on the operation of LO in the obvious way. The read-only variables may be changed only with special commands appropriate for each, while the others may be changed with special commands or with ".store".

Declaring a new variable with the same name or undeclaring one of these variables causes it to lose its special ability to affect LO's operation. No error message is given in this case(26). It is a detected error to attempt to change the type of a system variable with a ".store..." command.

Section 8.1 lists all of LO's predefined variables, along with a reference to the section in which each is discussed.

Name	Val	Type	Description
actbuf	→	*S	The name of the buffer currently receiving output, or the empty string if output is to the file. This variable is set by ".buffer"; it is initialized to the empty string.
bline	66	L	Total vertical spacing on each page, exclusive of that used by #topwindow#. It is set by ".bline".
day	--	N	Day of the month - 1, 2, ..., 31.
efooter	0	*N	Number of even footer lines currently in force.
eheader	0	*N	Number of even header lines currently in force.
endofline	→	S	Trap action to be obeyed after outputting the next complete text line. It is set by ".trap"; it is initialized to the empty string.
ewindow	0	N	Spaces on the left of each even-numbered page. It is set by ".set window ...".
flagcol	74	N	The column in which to start storing flags. (See Section 3.5.) A negative value may be used to store a flag in the left window.
flagger	→	*S	Character(s) used to flag lines under the control of ".flag". This variable is set by ".set flagger ..."; it is initialized to "*".
flagsw	0	N	Number of lines yet to be flagged. If at the end of a line this variable is positive, a flag is printed on the line and the variable is decremented by one.
fnbreak	→	S	Characters used to separate footnotes from the preceding text. It is initialized to a string of 20 minus signs.
fnbuf	→	*B	The buffer used to hold footnotes waiting to be processed. It is initialized to be an empty buffer

(26) An optional warning message will be given if asked for; see the "warn" option for ".set" in Section 3.10.

Name	Val	Type	Description
			marked as an insert buffer.
fnlmarg	0	N	Left margin to be used in storing footnotes.
fnlspacing	1	L	Line spacing to be used in storing footnotes.
fnrmarg	71	N	Right margin to be used in storing footnotes.
fnsp1	2	L	In storing a footnote, the vertical spacing after the last text line and before the footnote break.
fnsp2	1	L	In storing a footnote, the vertical spacing after the footnote break and before the first footnote line.
fnsp3	2	L	In storing footnotes, the vertical spacing between footnotes.
footspace	3	L	Space after the last text or footnote line and before the footer. It is set by ".footspace".
fspacing	1	L	Vertical spacing left after each footer line. It is set by ".fspacing".
hour	--	N	Hour of the day - 0, 1, ..., 23.
headspace	3	L	Space after the last header line before the first text line. The total space is headspace + hspacing - 1*0 It is set by ".headspace".
hspacing	1	L	Vertical spacing left after each header line. It is set by ".hspacing".
indent	0	N	Indentation for the next line, in addition to #lmarg#. This is set by ".indent" and is reset to zero after starting each line.
lastcol	~	*N	The last print position on the line into which output has been stored. This is compared with #rmarg# to know when the line is full. It is set to #lmarg# + #indent# on each new line.
linecount	~	N	Number of non-blank lines printed so far on the page, not including headers. This variable is set to one immediately after printing headers on each page and is incremented by one after completing each text line. This is useful for a subsequent text reference to a line number on a previous page. The variable is not used by LO.
lineno	~	*L	Line number on the page. This is compared with #maxline# to decide when the page is full. When an upspacing of L is to be performed for any reason, #lineno#+L is first compared with #maxline#. If it is less the spacing is done, while otherwise the current page is ejected and a new one started.

Name	Val	Type	Description
lmarg	0	N	Character position of the left margin. The first character stored on each line is just to the right of #lmarg#. It is set by ˜.lmarg˜.
loginname	--	S	User's name, from Apex. It is preceded by a script ˜q˜ for a quasi name.
lspacing	2	*L	Line spacing: The paper is upspaced by this amount after each line, so that #lspacing#-1*0 blank vertical space appears after each output line. It is set by ˜.lspacing˜.
maxline	~	*L	The maximum value for #lineno# on the current page. It is set after printing headers on each page, taking footspace, footers, etc., into account. It is decremented when a footnote is stored. See the description above for #lineno#.
minute	--	N	Minute of the hour - 0, 1, ..., 59.
month	--	N	Month number - 1, 2, ..., 12.
newpage	→	S	Trap action to be obeyed before printing the headers of the next page. It is set by ˜.trap˜; it is initialized to the empty string.
nextpage	1	N	Page number to be given to the next page. It is set by ˜.nextpage˜ and by ˜.page˜, and it is incremented on each page eject.
ofooter	0	*N	The number of odd footer lines currently in force.
oheader	0	*N	The number of odd header lines currently in force.
owindow	0	N	Spaces to be inserted on the left of each odd page. It is set by ˜.set window ...˜.
pageno	0	N	Number of the current page. This should be changed by changing #nextpage#, which see. See Section 2.6.
pagetop	→	S	Trap action to be obeyed at the top of the next page, after printing headers. It is set by ˜.trap˜; it is initialized to the empty string.
paraind	5	N	Indentation for each paragraph, for ˜.para˜.
params	--	*N	Number of parameters supplied to the current macro, or the number of parameters supplied to LO through 5BT when outside of any macro.
paraneed	3	L	Need for each paragraph, for ˜.para˜.
paraspacing	3	L	Space for each paragraph, for ˜.para˜.
partsperline	2	*N	Number of partial lines per full line of vertical spacing. This value is permanently fixed in the current implementation.
rmarg	71	N	Right margin. The last character on the line is in this print position. It is set by ˜.rmarg˜.

Name	Val	Type	Description
second	--	N	Number of seconds after the minute - 0, 1, ..., 59.
spacer	→	*S	The character to be replaced by space. It is set by ".set spacer <char>"; it is initialized to "ƀ".
topspace	0	L	Space at the top of each page before the first header. This space is included in #bline#.
topwindow	0	L	Space at the top of each page. This space is not included in #bline#. It is set by ".set window top L".
totalchars	0	N	The total number of characters printed so far by LO. This is not used by LO but may be by users.
weekday	--	N	Day of the week - Sun = 0, ..., Sat = 6.
window	0	N	The number of spaces being inserted on the left of each line on the current page. It is set before printing headers on each page to either #ewindow# or #owindow#, depending on whether the new value of #pageno# is even or odd.
year	--	N	Year - 1974, 1975, ...

Chapter 5: Command Descriptions

This Chapter contains a description of each of LO's commands, in alphabetic order. Two- or three-letter abbreviations are provided for the more common commands. The following conventions are used in these descriptions:

N	Any expression whose value is an integer.
L	Any expression whose value is a line spacing.
B	A buffer-valued variable.
V	A variable.
<V-list>	A list of variables, separated by commas.
<triple>	A three-part string, as described in Section 1.4. The first non-blank character is taken as the quote character.
<text>	Any text at all. The text extends to the end of the line.
<file>	An APEX file name.

Other conventions used only once are defined as needed. Recall that a variable name is a sequence of upper- and lower-case letters, digits, and the character ".", with the first character being an upper- or lower-case letter.

Since the commands are in alphabetic order by the full name of the command, some of the abbreviations are slightly out of their proper alphabetical position.

All the commands are tabulated in summary form in Section 8.2, along with a reference for each to the section in which it is discussed.

.bline L
.bl L — The variable #bline# is set to L, so that the bottom line on the page is hereafter assumed to be vertical spacing L from the top of the page. Default is 66=0. This command is not meaningful when output is to a buffer. See Section 2.1.

.break
.br — This command causes a line break, the current line being completed and the next text printed starting on a new line.

.bt <text>

The <text>, with a carriage-return appended, is sent to 5BTF on the next map.

.buffer B
.bu B

This command switches LO's output to buffer B and sets #actbuf# to B. If B is missing, output is set to the output file and #actbuf# is set to the empty string. Buffer B is not cleared. If this command switches output away from the file, it does not force a line break; while otherwise it does. See Section 2.7 for a discussion of buffers, and Section 3.6 for their use as macros.

.center <text>
.ce <text>

This command forces a break and then centers <text> between the left margin (appropriately allowing for any preceding ".indent" command) and the right margin. See Section 2.4.

.charpos N

If the last character on the output line is one or more spaces to the left of character position N with respect to the current value of #lmarg#, enough spaces are stored so that the next character stored goes into position N. Otherwise, a new line is started and the effect of ".indent N" is simulated. In either case, the next character stored is into column N. If this command appears before storing any text on a line (for example, just after ".break"), it acts as does ".indent N". See Section 2.2.

.clear B
.cl B

The contents of buffer B are cleared. If B is missing the buffer currently active to receive output is cleared. It is a detected error to attempt to clear a buffer from which input is currently being taken - unless the ".clear" command is on the last line of that buffer, or a ".end" precedes the ".clear" command on that line. See Section 2.7.

.comment <text>
.co <text>

This is a comment line and is completely ignored. The ".*" convention may equivalently be used.

.console <text>

If <text> is non-empty, the <text> is merely typed on the user's console and there is no other action. If <text> is empty, the user is given a prompt[27] after which he is to type a line followed by carriage return. (The NO and DELETE keys may be used as usual under APEX for line delete and character erase, respectively.) This line is then processed as a line of input to LO; it may be either text or commands. See Section 3.8.

<27> The prompt is <newline> "?" <tab>.

.declare <V-list>
.dcl <V-list>
.de <V-list>

This line declares each variable in the V-list, giving it a value of type undefined so that it cannot be stored into with ".store". The abbreviation "dcl" is provided for the convenience of PL/I programmers.

.declarebuf <V-list>
.db <V-list>

Each of the variables in the <V-list> is declared and initialized to an empty buffer marked as not being an insert buffer. See the discussion of declarations in Section 1.2.

.declareinteger <V-list>
.di <V-list>

Each of the variables in the <V-list> is declared and initialized to the integer value zero (or to the default set by ".set default int"). See the discussion of declarations in Section 1.2.

.declareline <V-list>
.dl <V-list>

Each of the variables in the <V-list> is declared and initialized to the LIN value 0*0 (or to the default set by ".set default lin"). See the discussion of declarations in Section 1.2.

.declarestring <V-list>
.ds <V-list>

Each of the variables in the <V-list> is declared and initialized to the empty string (or to the default set by ".set default str"). See the discussion of declarations in Section 1.2.

.efooter N, <triple>

This command sets up the Nth footer line for the even pages. See Section 2.5. This command is not meaningful when output is to a buffer.

.eheader N, <triple>

This command sets up the Nth header line for the even pages. See Section 2.5. This command is not meaningful when output is to a buffer.

.end

The current input source is marked as being at its "end of file", whether it is the main file, a file fetched by ".include", or a buffer. On completion of the processing of the line containing this command (there may be other commands after it on the line), the current input source is terminated. See Section 3.8 for use in ".include" files, and Section 3.6 for use in macros.

.endofline <text>
.eol <text>

This command is obsolete and has been replaced by ".trap", which should be used.

.expand <command>, N, A_1, A_2, ...
The command <command>, which may be either a built-in LO command or a user-defined macro, is invoked repeatedly, each time with N (or fewer) arguments, until all the arguments to ~.expand~ are used up. See Section 3.11 for details.

.flag N
.fl N
The next N output lines are flagged with an asterisk (default) in the column specified by #flagcol#, as shown. *
If N is missing it is taken as one. See Section 3.5.

.footer N, <triple>
This command sets up the Nth footer line for both the even and odd pages. See Section 2.5. This command is not meaningful when output is to a buffer.

.footnote This command is used for setting up footnotes and is described in detail in Section 3.4. This command is not meaningful when output is to a buffer.

.footspace L
The variable #footspace# is set to L. Thereafter, L vertical spacing is left between the last text line and the first footer line on each page. The default is 3*0. This command is not meaningful when output is to a buffer.

.fspacing L
The variable #fspacing# is set to L. Thereafter, L-1 extra vertical spacing is left between successive footer lines. The default value is 1*0. This command is not meaningful when output is to a buffer.

.header N, <triple>
This command sets up the Nth header line for both the even and odd pages. See Section 2.5. This command is not meaningful when output is to a buffer.

.headspace L
The variable #headspace# is set to L. Thereafter, L vertical spacing is left between the last header line and the first text line on each page. The default value is 3*0. This command is not meaningful when output is to a buffer.

.help <text>
This is a programmed HELP call, with the <text> printed as part of the message. If error output is to a file, so also is the output from this command – unless the ~d~ option is used. The command is of most use to the author of LO in debugging the program.

.hspacing L
The variable #hspacing# is set to L. Thereafter, L-1 extra vertical spacing is left between successive header lines. The default value is 1*0. This command is not meaningful when output is to a buffer.

.if V_1, <rel>, V_2,<text>
This is a simple conditional command. <rel> may be any of the strings ~eq~, ~ne~, ~ls~, ~le~, ~gr~ or ~ge~; and V_1 and V_2 are expressions. If the values V_1 and V_2 do not satisfy the given relation, the rest of the line is ignored;

otherwise, <text> is processed as a line of input to LO. The <text> starts with the first non-blank character after the third comma. See Section 2.9.

.include <file>
.inc <file>
The named file from the user's APEX directory is included in the input in place of this line. To include a file from another directory, use ".bt" to SGET the file into a dot name, and then ".include" that. The file name must be followed by a space or the end of the line, so a space is needed before any "||" or "|".

.indent N
.ind N
The variable #indent# is set to N, causing the next line to be indented by N characters from the current left margin. This command causes a break. If N is negative, the next line extends to the left of the present left margin, but it is a detected error if #lmarg# + #indent# is negative.

.insert B
.ins B
This command inserts a verbatim copy of the contents of buffer B into the output (buffer or file). If output is to the file, pagination continues while the lines are added, so many-printed-page buffers may be assembled. Line spacing is that which was in effect at the time buffer B was prepared, as indeed are all other parameters of the layout. This command causes a break. Compare this command with ".rescan".

.label <word>
This command terminates the effect of the ".skipto" command, which see. If encountered otherwise it is ignored. See Section 2.9.

.leave L
.le L
Vertical spacing of L is inserted into the text being created. If the bottom of the page is encountered while leaving lines, the rest of the lines are not left. If all the lines would fall at the top of a page this command leaves no lines. This command causes a break. If L is missing, the default is one full line.

.lmarg N
.lm N
The variable #lmarg# is set to N, so that the left margin is just to the right of the Nth position on the line. If N is missing, it is taken as zero.

.lspacing L
.ls L
The variable #lspacing# is set to L, so that upspacing of L takes place after each output line created hereafter. The default value of #lspacing# is 2*0, for double spacing.

.need L
.ne L
If there is room on the page being created for at least L vertical spacing, this command has no effect. If this test appears when at the top of a page, the test succeeds no matter how large L may be. If there is not L vertical space remaining, a new page is started. This command is not meaningful when output is to a buffer.

.newpage <text>
This command is obsolete and has been replaced by ".trap", which should be used. This command is not meaningful when output is to a buffer.

.nextpage N
The variable #nextpage# is set to N, so that the next page printed will be numbered N and succeeding pages N+1, etc. #pageno# is not changed until after footers are printed on the current page. See Section 2.6. This command is not meaningful when output is to a buffer.

.ofooter N, <triple>
This command sets up the Nth footer line for the odd pages. See Section 2.5. This command is not meaningful when output is to a buffer.

.oheader N, <triple>
This command sets up the Nth header line for the odd pages. See Section 2.5. This command is not meaningful when output is to a buffer.

.output B, <file>
The contents of buffer B is copied into the APEX file named <file>. Each line is copied exactly as it appears in the buffer, with a new-line character after each line but with no pagination or headers. See Section 3.8.

.overstrike <quoted word> <text>
.ov <quoted word> <text>
The first non-blank character after the space after "overstrike" (or "ov") is taken as a quote character, and all characters up to the next occurrence of that character are taken as the <quoted word>. The <text> (which starts immediately after the second occurrence of the quote character) is scanned and each overstrikeable character is preceded by the characters in <quoted word>. The result is then processed as usual by LO. See Section 3.9.

.page N
.pa N
A new page is started. If N is given, the new page is numbered N and succeeding pages N+1, etc.; if not, the page numbering is unaffected. If this command appears at the top of a page, it may change the page numbering but does not result in an extra blank page. Thus ".page" repeated does not leave two pages.[28] This command causes a break; it is not meaningful when output is to a buffer.

.pageno N The variable #pageno# is set to N and #nextpage# to N+1, so that subsequent pages will be numbered consecutively thereafter. The command ".nextpage" is usually more useful. See the complete discussion in Section 2.6. This command is not meaningful when output is to a buffer.

<28> Something must be placed on a blank page; it may be a line with a few blanks. See footnote 11 on page 16.

.pagetop <text>
This command is obsolete and has been replaced by ~.trap~, which should be used. This command is not meaningful when output is to a buffer.

.para
.pr This line is equivalent to the following three lines:
.leave paraspacing - lspacing
.need paraneed
.indent paraind
The three variables may be set by ~.paraspacing~, ~.paraneed~ and ~.paraind~, respectively, or by ~.store~. This command causes a break.

.paraind N
.pri N The variable #paraind# is set to N, so that indentation on the first line of a paragraph is set to be N positions. See ~.para~.

.paraneed L
.prn L The variable #paraneed# is set to L, so that L lines will be ~.need~ed before each paragraph. See ~.para~.

.paraspacing L
.prs L The variable #paraspacing# is set to L, so that L-lspacing extra lines are left before each paragraph. It is a detected error to attempt to set #paraspacing# to be less than #lspacing#. See ~.para~.

.plain <text>
.pl <text>
The <text> is placed on a line by itself exactly as it appears, starting at the current indented left margin. This command causes a break. See Section 2.4 for a further discussion of ~.plain~, and Section 3.1 for a discussion of tabs in such lines.

.proc This command causes reversion to normal formatting following the use of the ~.unproc~ command. It is legal only after an ~.unproc~ line.

.rescan B <arguments>
.rs B <arguments>
Buffer B is rescanned as input to LO, with processing of control lines as usual. If arguments are supplied, they may be accessed in B as described in Section 3.6. Compare this command with ~.insert~.

.reset This command is defined only if it is encountered while processing a trap action, and resets that action to be still in effect. See Section 3.7.

.rmarg N
.rm N The variable #rmarg# is set to N, setting the right margin to the Nth position on the line. If this results in 0 ≤ flagcol ≤ rmarg, then #flagcol# is set to #rmarg#+3. See Section 1.1.

.set <parameters>
: This control line permits the user to override certain of LO's conventions. Details are provided in Section 3.10.

.skip N
: N lines of the input are skipped. If N is missing it is taken as one. This command will not cause skipping past the end of a file (see ".include") or past the end of an input buffer (see ".rescan"). See Section 2.9.

.skipto <word>
: Input text is skipped until a line starting with

    .label <word>

  is found with a matching <word>. This command will not cause skipping past the end of the file (see ".include") or past the end of an input buffer (see ".rescan"). The <word> may contain any characters other than space or tab. See Section 2.9.

.store V,<text>
.st V,<text>
: Variable V is updated, the nature of the updating being dependent on V's previous value. If the previous value of V was type INTeger, <text> is evaluated as an integer and the result is stored into V; if it was type LINe, <text> is evaluated as a line; and if of type STRing, the remainder of the line is stored into V. Note that in storing into a STRing the remainder of the line is not scanned for "||" or "|". It is a detected error if V is undeclared, if V is declared but has no value, or if V is a buffer-valued variable. Note that a variable declared by ".declare-integer", ".declareline" or ".declarestring" may be ".stored"ed into immediately. Built-in variables may be changed with the expected effect. See Section 1.2.

.storeinteger V, N
.si V, N
: The value of variable V is set to the integer N, regardless of its previous type. It is a detected error to attempt to change the type of a built-in variable.

.storeline V, L
.sl V, L
: The variable V is given the line-valued value L, regardless of its previous type. It is a detected error to attempt to change the type of a built-in variable.

.storeroman V, N
.sr V, N
: The integer N is converted to roman numeral form using lower-case letters and stored (as by ".storestring") into the variable V. V may then be used as may any string-valued variable. N must be positive and less than 4000.

.storeromanupper V, N
.sru V, N
: The effect is as for ".storeroman", but upper-case letters are used for the roman numerals.

.storestring V,<text>
.ss V,<text>
: The value of variable V is set to the string <text>. It is a detected error to attempt to change the type of a built-in variable.

.subst <text>
.su <text>

This command causes the substitution algorithm described in Section 2.3 to be applied to the <text>, and the result then to be used as input to LO. This command does not itself cause a break, but the resulting line could be a command that does.

.tabsin <tablist>

The <tablist> must be a list of positive integers, separated by commas, in ascending order. Tabs appearing in the input in ~.plain~ lines, when fill mode is off, and under the control of ~.unproc~, are replaced by spaces as if the tabs on the input device were as given in the <tab list>. See Section 3.1.

.tabsout <tablist >

This command is not yet implemented. See Section 3.1.

.trap <event> <text>

The trap <event> is set, where <event> must be one of ~endofline~ (or ~eol~), ~nextpage~ (or ~np~), or ~pagetop~ (or ~pt~). When the specified event takes place, the text <text> is interpreted as a line of input to LO. The ~endofline~ event occurs on completion of processing the current output line, before paper upspacing and possible page ejection. The ~nextpage~ event takes place after page ejection but before printing headers at the top of the next page. (No text should be emitted in the action for this event.) The ~pagetop~ event takes place after printing headers and before printing any text on the new page. Setting a ~newpage~ or ~pagetop~ trap is not meaningful when output is to a buffer. See the complete discussion of traps in Section 3.7. If <event> is ~do~, than <text> should name a trap. The action stored for that trap is then performed as if the event had occurred. It is permissible to ~.trap do~ an event for which no action is currently stored.

.triple <triple>
.tr <triple>

The <triple> is a three-part string. Substitution is first performed in the triple (as described in Section 2.3) and then the first part is printed left justified, the second part centered between the margins, and the third part right justified. The first non-blank character after ~triple~ (or ~tr~) is taken as the quote character. This command causes a break. See Section 2.4.

.undeclare <V-list>
.und <V-list>

This command causes each variable on the <V-list> to be undeclared. If it had been declared before the matching ~.declare~, then that declaration is restored. (See the discussion of declarations in Section 1.2.) It is a detected error to attempt to undeclare an undeclared variable. It is also a detected error to undeclare the name of a buffer while outputting to it, or inputting from it — but see the discussion under ~.clear~ and in Section 3.6 for

an exception. Undeclaring a built-in variable is permissible, and a warning message (see Section 1.5) will be given for this if desired.

.unproc
.unp

All text following this command, and before the next ".proc" command (including all command lines other than ".proc") is inserted in the text being created exactly as it appears in the input textfile, with the following considerations: The spacing between output lines and the left margin are those used in the layout at that point. If output is to the file, page breaks with footers and headers continue to be inserted. This command causes a break. The effect of ".unproc" is terminated by the end of an input file (see ".include") or the end of an input buffer (see ".rescan"). See also ".tabsin", for special processing of tabs in the controlled text. Also, compare with fill mode, discussed in Section 1.1.

Chapter 6: Examples

This chapter contains some examples of LO text, mostly having to do with macros. Most of the examples are taken from the source text for this document, since the reader has in front of him the results of the operation of these macros. The numbers printed to the left of the macro are not part of the LO text; instead they are printed to facilitate reference to individual lines in the discussion that follows. Some of the examples from this document are simplified slightly for the sake of expositional efficiency. Abbreviations are used in a few places when a line would otherwise be too long for the page; each such use is commented on.

6.1 – Date and Time: In the first example, the problem is to start with the LO variables #hour# and #minute#(29) and store into variable #time# a string like this:

hour	minute	time
0	5	'12:05 AM'
3	10	' 3:10 AM'
12	20	'12:20 PM'
19	3	' 7:03 PM'
23	55	'11:55 PM'

There are several problems to be solved. If the value of #minute# is less than 10, an extra "0" must be stored after the colon; similarly, if the final time is between one and nine, an extra leading space is needed. Note also that 15 minutes past midnight is referred to as 12:15 AM. Now note the following LO text:

```
1     .declarestring time
2     .store time,AM
3     .if hour, gr, 12, .store hour, hour - 12‖ .store time,PM
4     .if hour, eq, 12, .store time,PM
5     .if hour, eq, 0, .store hour, 12
6     .subst .store time,#minute# #time#
7     .if minute, ls, 10, .subst .store time,0#time#
8     .subst .store time,#hour#:#time#
9     .if hour, ls, 10, .subst .store time, #time#
```

---

(29) Recall that these integer-valued variables give the time when LO starts its work.

This sequence stores the proper value[30]. In line 1 #time# is declared, and it is initialized to the string value ˜AM˜ in line 2. No more than one of the three tests in lines 3, 4 and 5 will succeed; sometimes none of them will. The reader should satisfy himself that they do the right processing. In line 6 the minute is stored and the space before ˜AM˜ or ˜PM˜. In line 7 a zero is stored if the minute is only one column wide. Lines 8 and 9 store the hour, with a leading blank if the hour is less than ten. In general, spaces are used in this example (as well as in the remaining ones) to improve readability. For example, there is usually a space after each comma. However, it is important that the last comma on lines 2, 3 and 4 not be followed by a space. In these lines the store is into the string-valued variable ˜time˜, so all characters after the comma are used in determining the value to be stored. The space after the last comma in line 9 is important, since the reason for that line is to store a leading space into the value of ˜time˜.

Further examples of time conversion are found in the file dt·lo in directory ART. One example from that file is of interest. The last task in that file is to store into the STRing-valued variable #wday# the day of the week. Recall that the built-in INTeger variable #weekday# is 0 for Sunday, 1 for Monday, ... The following code is used:

```
1     .declarestring wday
2     .skip weekday
3     .end || .store wday,Sunday
4     .end || .store wday,Monday
5     .end || .store wday,Tuesday
6     .end || .store wday,Wednesday
7     .end || .store wday,Thursday
8     .end || .store wday,Friday
9     .end || .store wday,Saturday
```

The ˜.skip˜ command causes between 0 and 6 lines to be skipped. Each following line starts with ˜.end˜ to terminate processing of the file on completion of processing that line, followed by a ˜.store˜ of the desired value. The ˜.end˜ must come first, since the value to be stored into #wday# extends to the end of the line.

<30> A programmer might claim that it is bad code because it sometimes alters its input arguments. This fact, while true, does not detract from its value as an example of LO usage.

6.2 - Miscellaneous Examples: The next example is the macro used in this document to store one or more lines of indented text. The macro stored in buffer "pix" takes an arbitrary number of parameters, up to a maximum of nine. Each parameter is printed on a separate line, exactly as it is typed. For example, the line

```
.pix 1 'now is the time' <last line<
```

would produce the display

```
1
now is the time
last line
```

(Note the use of "<" as the quote character for the third parameter.) Similarly, the line

```
.pix /first line/ /second line/ /.../ /last line/
```

would produce

```
first line
second line
...
last line
```

The LO text used to define the macro "pix" now follows.

```
 1    .declarebuf pix|| .buffer pix
 2    ..lspacing lspacing - 0*1
 3    ..leave 0*1
 4    ..need params × lspacing
 5    ..lmarg lmarg + 10
 6    ..subst .plain #1#
 7    ..if params, ge, 2, .subst .plain #2#
 8    ..if params, ge, 3, .subst .plain #3#
 9    ..if params, ge, 4, .subst .plain #4#
10    ..if params, ge, 5, .subst .plain #5#
11    ..if params, ge, 6, .subst .plain #6#
12    ..if params, ge, 7, .subst .plain #7#
13    ..if params, ge, 8, .subst .plain #8#
14    ..if params, ge, 9, .subst .plain #9#
15    ..if params, gr, 9, .subst .help params = #params#, > 9
16    ..lmarg lmarg - 10|| .lspacing lspacing + 0*1
17    ..leave 0*1
18    .bu
```

Line 1 declares "pix" as a buffer, and then switches LO's output to that buffer. Lines 2 through 17 start with "..", so each line is copied into the output with the first "." removed. In other words, buffer "pix" contains just the text shown in those lines, but with one leading "." removed from each line. Finally, line 18 switches LO's output back to the file. Note that #lspacing# is decreased by 0*1 in line 2 and incremented by that amount in line 16, so the final value is the same as the initial value but the lines stored are closer

together. Similarly, #lmarg# is changed in lines 5 and 16 so as to leave it unchanged but store the lines with a different value. The macro fails to work with more than 9 parameters, and the ".help" in line 15 is its way of complaining.

Another example is the macro used to produce the command descriptions in Chapter 5. The macro requires one, two, three or four parameters. The first is the command name; the second is the argument prototype; the third is the command abbreviation; and the fourth is an alternate abbreviation. Only one argument need be supplied if the command takes no argument and has no abbreviation; but, if it has an abbreviation but takes no argument, the second argument to ".foo" must explicitly be the empty string. The macro is defined using the text

```
 1     .buffer foo‖ .clear
 2     ..leave 0*1‖ .need (params + 2) × lspacing
 3     ..indent -10‖ .subst  .#1#
 4     ..if params, ge, 2,.subst #2#
 5     ..if params, ge, 3,.indent -10‖ .subst  .#3# #2#
 6     ..if params, ge, 4,.indent -10‖ .subst  .#4# #2#
 7     ..if params, gr, 4,.help params > 4
 8     ..charpos 0
 9     .buffer
10     .lmarg 10
```

This stores into buffer "foo" (which was previously declared) lines 2 through 8, with a "." deleted from each line. Line 10 sets #lmarg# to 10 after storing the buffer. When the buffer is expanded, line 3 causes the text of the first parameter to be stored into column 1. Note the ".need" in line 2 for two more lines than the number of parameters. Note also that there are two spaces after ".subst" in lines 3, 5 and 6. This keeps the result of the substitution from being interpreted by LO as a command, which would be disastrous since it is intended as text to be stored. Use of this macro is as follows. Compare with the text actually produced in Chapter 5.

```
 1     .foo bline L bl
 2     The variable #bline# is...
 3     .foo break // br
 4     This command...
 5     .foo bt /<text>/
 6     The <text>, with...
 7     ...
 8     .foo declare /<V-list>/ dcl de
 9     This line declares...
10     ...
11     .foo end
12     The current input source...
```

Note that the second argument in line 3 is the empty string, and in line 8 the second argument is a string delimited by "/" as a quote character.

The next problem is somewhat interesting, and the reader is advised to try to solve it himself before looking at the solution presented here. In its present form it appears rather artificial, but the technique is one I have made use of in practice. It is desired to define a macro "foo", so that the effect of the command

    Z-1   .foo flum

is that the variable #flum#, assumed to be of type STRing, has the current page number appended to it. That is, the above command should result in LO's obeying the command

    Z-2   .subst .store flum,#flum#-#pageno#

If the previous value of #flum# were the string "2-5" and this was executed on page 11, the new value of #flum# would be "2-5-11". Something like this is done in preparing the index, although a special case is made in storing the first entry. This problem is harder than it looks at first glance, and the reader should now try to solve it.

It is clear that the line to be obeyed eventually is Z-2, but the problem is what came just before this. That is, a line such as

    Z-3   .subst .subst .store #1#,XXXX

must have been obeyed, but what can XXXX be so that after performing substitution on it the result is "#flum#-#pageno#"? The obvious possibility of "##1##-##pageno##" is clearly incorrect, since the effect of the substitution algorithm on "##" is to leave "#". It is possible to solve this problem somewhat painfully by changing the variable delimiter, but the following seems to be more elegant. It uses a variable named #splat# whose value is "#"(31). We have

    Z-4   .declarestring splat|| .store splat,#

With this variable available, it is clear that the XXXX in Z-3 can be

        #splat##1##splat#-##pageno##

since applying the substitution algorithm to this yields the desired value. Thus, the macro "foo" contains the single line

    Z-5   .subst .subst .store #1#,#splat##1##splat#-#pageno#

---

(31) In the TX-2 community, the mark "#" is usually pronounced "splat".

With only one "#" on each end of "pageno", it gets substituted for on the first ".subst" rather than the second, but the final result is the same.

6.3 – Chapter and Section: The next series of examples concerns chapter and section processing in this document. This interacts with page headers and footers, as well as with the Table of Contents. The variables needed are declared by the commands

```
D-1    .declarestring  title    |  chapter name
D-2    .declarestring  sbt      |  section name
D-3    .declarestring  SS       |  starting section
D-4    .declarestring  SE       |  ending section
D-5    .declareinteger ch       |  chapter number
D-6    .declareinteger se       |  section number
D-7    .declarestring  cs1      |  for Table of Contents
D-8    .declarestring  cs2      |  ditto
D-9    .declarestring  dots     |  ditto
D-10   .store dots,## . . . . . . . . . . . . . . . .
D-11   .store cs2,h
```

Note the use of the vertical bar "|" to introduce comments. To see the use of the first four variables, note that the footer line is produced by the command

```
          .footer 1, '#title#'#sbt#'#SS##SE#'
```

As will be seen, before printing the header on each page, #SS# is set to the current section number and #SE# is set to the empty string. #SE# is changed when each section is started.

We now look at the macros that process chapter and section beginnings. The present chapter starts with the line

```
          .chapter 6, 'Examples'
```

Following is the piece of text that defines the macro ".chapter".

```
C-1    .buffer chapter    |  chap.number, title
C-2    ..subst .store ch, #1#    |  chapter number
C-3    ..store se, 0    |  section number
C-4    ..subst .store title,#2#
C-5    ..store sbt,
C-6    ..subst .store SE, - #ch#.0
C-7    ..page
C-8    ..plain
C-9    ..leave 1*1
C-10   ..subst .center Chapter #1#:  #2#
C-11   ..leave 2*0
C-12   ..para
C-13   ..if pageno, ge, 10, .store cs2,
C-14   ..buffer contents
C-15   ...leave 1*0
C-16   ...need 4*0
```

```
C-17  ..subst ..triple 'Ch #ch#: #title# '#dots#'#cs2##pageno#'
C-18  ...leave 0*1
C-19  ..buffer
C-20  ..store cs1,hhh
C-21  .buffer
```

Since a new page is started in line C-7, the ".plain" on line C-8 is needed so that the ".leave" in line C-9 will actually leave some space. (Recall that a ".leave" given at the top of a page leaves no space, as described in Section 2.2.) The actual chapter title is stored in line C-10.

Each section is started by calling ".section" with the section number and section title as parameters. This macro is defined by the text

```
S-1   .buffer section    |  sect.number, title
S-2   ..leave 1*0‖ .need 3*0
S-3   ..subst .store se, #1#    |  section number
S-4   ..subst .store sbt,#2#
S-5   ..subst .store SE, - #ch#.#se#
S-6   ..subst .overstrike '_ #ch#.#se#h-h#sbt#:
S-7   ..if se, eq, 10, .store cs1,hh
S-8   ..if pageno, ge, 10, .store cs2,
S-9   ..buffer contents
S-10  ..su ..tr '       #ch#.#se#:#cs1##sbt# '#dots#'#cs2##pageno#'
S-11  ..buffer
S-12  .buffer
```

Line S-6 stores the section title into the text, underlined. Next we have

```
N-1   .buffer newpagework    |  This is invoked on each newpage.
N-2   ..subst .store SS,#ch#.#se#
N-3   ..store SE,
N-4   .buffer
N-5   .trap newpage .newpagework ‖ .reset
```

Lines N-2 and N-3 go into buffer #newpagework#, and line N-5 sets a newpage trap that invokes this buffer and resets the trap. This provides for the initialization of #SS# and #SE# mentioned earlier. #SE# is updated also in line S-5.

The Table of Contents is the last item printed when the LO document is created. It is achieved by storing information into buffer #contents# at the beginning of each chapter and each section. This buffer is then rescanned at the end, after the page numbering and headers and footers are set as appropriate. The lines of interest are C-13 to C-20 and S-7 to S-11. The line stored for a new chapter is in C-17 and for a new section in S-10. (Note abbreviations for ".subst"

and ".triple" in line S-10.) Each line is a ".triple" command, with the chapter or section number and name in the left part, the page number in the right part, and the variable #dots# in the center. Substitution is performed as the line is stored into the buffer so that #pageno# will be replaced by the value it has when the line is stored. The value of #cs1# is set to "*hhh*" in C-20 at the beginning of each chapter; it is changed to "*hh*" in S-7 as soon as the section number reaches 10. It serves to keep the section name lined up vertically when the section number goes to two digits. Variable #cs2# serves a similar function for the page number. It is initialized to "*h*" in line D-11 and is cleared to the empty string in either C-13 or S-8, whichever is first executed after the page number reaches 10. The value of #dots#, set in D-10, is two "#"s followed by many dots. The variable is replaced by its value when the substitution is done in C-17 or S-10 as the ".triple" line is stored into buffer #contents#, and "##" becomes "#" when substitution is done for the ".triple", so there is one "#" when the triple is expanded during the printing of the Table of Contents. As explained in Section 2.4, a "#" at the left end of the center string of a triple causes characters to be discarded from that end of that string in the event that the triple is too wide to fit on the line. The value of #dots# has been chosen to be wide enough so that this is the case for all ".triple" lines in the buffer. The effect is the array of dots seen in the Table of Contents as it is printed.

6.4 – <u>Recursive Macros</u>: It is a maxim that no manual for a programming language that permits recursion is complete until a factorial example is included. LO is enough of a programming language that I would not dare to disobey this maxim. We define a macro "fact" so that

```
.fact N, 6
```

causes the factorial of six to be stored into variable N, presumably previously defined. An extra buffer "fact1" is used, as well as INTeger variables "f1" and "f2". The following text should appear in the input:

```
F-1   .declarebuf fact, fact1
F-2   .declareinteger f1, f2
F-3   .buffer fact
F-4   ..store f1, 1
```

```
F-5    ..subst .fact1 #2#
F-6    ..subst .store #1#,#f1#
F-7    .buffer fact1
F-8    ..subst .store f2, #1#
F-9    ..if f2, ls, 2, .end
F-10   ..store f1, f1 × f2
F-11   ..fact1 → f2-1
F-12   .buffer
```

Suppose the call for #fact# is as above. Buffer #fact# initializes #f1# to one and then calls #fact2# with six as parameter. #fact2# stores its parameter into #f2#. If the parameter is less than two it is done, while otherwise it replaces #f1# by ˜f1 × f2˜ and then calls itself recursively with an argument one smaller. The arrow on line F-11 causes ˜5˜ to be passed instead of ˜6-1˜. It is not really needed, since the ˜.store˜ in line F-8 would get the right value anyway.

Strictly speaking, variable #f2# is not needed, since lines F-9, F-10 and F-11 could be started with ˜.subst˜ and ˜f2˜ replaced by ˜#1#˜. But this would put the recursive call on a line with ˜.subst˜, which (as suggested in Section 3.6) results in inefficient use of storage for recursion.

As another example of recursion, consider a test program for #fact# as just defined. Suppose that lines F-1 to F-12 are followed by the following lines:

```
F-13   .declarebuf foo‖ .buffer foo
F-14   ..subst .store m, #1#
F-15   ..fact n, →m
F-16   ..subst .console #m#   #n#
F-17   ..foo → m + 1
F-18   .buffer
F-19
F-20   .declareinteger m, n
F-21   .foo 0
```

When LO reads this file, it will print successive lines with n and n! on each. After 13! is printed, a partial result becomes too large and an error message is given.

Chapter 7: LO on TX-2

This chapter discusses LO as it is implemented on TX-2. A few comments are given on what would be involved in moving it to another computer.

7.1 – <u>How to Use LO</u>: To use LO, log in to the APEX Time-Sharing System on TX-2 and prepare (using any convenient text editor) a text file – say, called ˜inputfile˜. Then type to BT the line

    5do art lo inputfile -outputfile- -args- -options-

The effect of this command is to read ˜inputfile˜ and process it, producing an output file. Only the first argument is required: If the second argument is not given, output is into the text file ˜·lo˜. If the output exceeds one book, a text group will be made. (Such text groups can be LDXed directly by 5ldx.)

If any errors are detected during LO's operation, LO reports this fact to the operating system by peeling to the BT with negative epsilon. This is the same condition used in determining whether or not to LDX the output (if requested).

Extra arguments and options may be specified, referred to above as ˜args˜ and ˜options˜. The arguments are supplied to the run as parameters, available through the macro parameter mechanism described in Section 3.6. The options set switches in LO, and are as follows:

- b BCPL comment convention. Any line in any input file starting with ˜//˜ in the first two columns has those two characters removed, and the line is then processed as usual. This provides a mechanism for using LO to format long comments in BCPL programs.
- c Check width for the LDX. An error message is given if any output line is too wide for the LDX. The check takes proper account of underlines. This option is independent of the ˜l˜ option.
- d Set debug mode. This is intended for debugging LO and is not of interest to the average user. It affects the ˜.help˜ command and the printing of error messages; it may also have other effects and so is best avoided unless you know what you are doing. In particular, it may invoke an experimental feature that I am in the process of debugging.

e Error output to a file. This causes all error messages from LO to be stored into a file rather than being printed on the user's console. If any errors are detected, the user is informed of that fact and given the name of that file at the end of the run.

l LDX the output. The output file (which is nonetheless created) is printed on TX-2's on-line LDX printer, using the default character set in 6ldx. This LDXing is suppressed if any errors are detected.

n No help calls. This suppresses the 5HELP call that is normally part of each error message. It is implied by option ˜e˜.

p Prompt. This causes LO to print each page number on the console as it completes outputting that text page. For long files it provides the user with reassurance that LO is still there and that APEX has not crashed. This appears on the console even if error output has been directed to a file.

If any ˜args˜ are to be given, the output file must be specified.[32] The ˜args˜ and ˜options˜ are separated by an argument of ˜/˜. For example, to process input file ˜foo˜ into the default output file with macro arguments ˜a˜ and ˜qwert˜, and with prompt mode and error output to the file ˜·lo·err·foo˜, type

```
5do art lo foo - a qwert / p e
```

If any of ˜args˜ is to contain characters other than those that are permitted in an APEX file name, it may be enclosed in APEX's BEGIN WORD-EXAM quotes, so that

```
5do art lo foo - BEGIN a b c WORD-EXAM
```

will supply ˜a b c˜ as the third macro argument.

Any special character (as the term is defined by 5BT) may be used where ˜-˜ or ˜/˜ is mentioned above.

7.2 – On-Line Documentation: It is possible for one in the TX-2 room to obtain LO's documentation on line from APEX. (Obtaining the documentation via the ARPAnet is not practical, since it depends on a particular LDX character set. Further, a tape mounting is required.) It is not recommended, however, since the computing takes well over five minutes on full machine and not quite forever under time sharing. Also, LDXing takes about nine minutes. (The output is well over four

---

[32] If the second argument is ˜-˜, the default ˜·lo˜ is used for ·output.

books long.) Nonetheless, it can be done, like this. Login to any free name, and type

```
5get art lo·lo
5s lo·lo
```

This puts on the scope instructions for tape mounting to read the relevant files. Follow these instructions, and then type

```
5do art lo lo·tn - - p
```

This LOs the document with prompt mode, which is reassuring. Then type

```
5get art ch2·file
5ldx ·lo ch2·file
```

WAIT UNTIL THE LDXING IS COMPLETE, and then (and only then) type

```
5quit
```

The output of LO is a text group; typing 5quit too soon may cause some of its books to be dropped. (Only the one actually being LDXed and the next one are frozen into core.) The last few pages include the Abstract and Table of Contents; reorder as needed.

7.3 – LO on Another Computer: Since LO is written entirely in BCPL, it should not be too difficult to move it to another computer. (I assume, of course, that there already is a BCPL.) That part of LO that accepts parameters from the console would, of course, have to be rewritten, but this is all in one place. Also, such commands as ˜.bt˜ and ˜.console˜ would have to be rethought, and ˜.include˜ and ˜.output˜ might require a different syntax for file names.

Another problem is the use of certain TX-2 characters that are not likely to exist elswehere. For an ASCII character set, I suggest using ˜;˜ for ˜‖˜, ˜*˜ for ˜×˜, and ˜$˜ for ˜*˜. Ampersand-digit could be used for the Greek letters α, β, γ and λ.

LO uses two packages written in TX-2 machine code (TAP): a hash-code lookup package used to store variable names, and a free-space package. Also used is a random number generator.

Undoubtedly other issues would appear as soon as a transfer project started. I will be glad to cooperate and to supply the BCPL source code, as well as the text files that make up this document.

Chapter 8: Summary Tables

This chapter contains tables that summarize all of LO's predefined variables and commands, as well as the options to ".set".

8.1 - Table of Variables: The table in this section summarizes all the variables predefined in LO. A more complete description of each of the variables is found in Chapter 4. The column headings are:

Name	The name of the variable.
Val	The default value. For strings, the value is shown within 'quotes'. See Chapter 4 or the reference for values marked "--", which cannot be described here in the available space.
RO	Read only. A "x" in this column indicates that the variable cannot be changed with ".store".
Type	Letters N, L, S and B refer to INTeger, LINe, STRing and BUFfer, respectively.
Ref	Reference. This is the section number in which the variable is discussed. If none is given, the only discussion is in Chapter 4.

Name	Val	RO	Type	Ref	Description
actbuf	''	x	S	2.7	buffer currently receiving output
bline	66*0		L	2.1	page length
day	--		N	1.2	day of the month
efooter	0	x	N	2.5	number of even footers
eheader	0	x	N	2.5	number of even headers
endofline	''		S	3.7	trap action before next line
ewindow	0		N	2.1	window on even pages
flagcol	74		N	3.5	column for flags
flagger	'*'	x	S	3.5	flag
flagsw	0		N	3.5	number of lines yet to be flagged
fnbreak	--		S	3.4	break before footnotes
fnbuf	--	x	B	3.4	buffer to hold footnotes
fnlmarg	0		N	3.4	left margin for footnotes
fnlspacing	1*0		L	3.4	spacing for footnotes
fnrmarg	71		N	3.4	right margin for footnotes
fnsp1	2*0		L	3.4	space before footnote break
fnsp2	1*0		L	3.4	space after footnote break
fnsp3	2*0		L	3.4	space between footnotes
footspace	3*0		L	2.1	space before footer
fspacing	1*0		L	2.1	space between footers

Name	Val	RO	Type	Ref	Description
hour	--		N	1.2	hour of the day
headspace	3*0		L	2.1	space after header
hspacing	1*0		L	2.1	space between headers
indent	0		N	2.2	indent for next line
lastcol	--	×	N		last used column
linecount	--		N		count printed lines on the page
lineno	0*0	×	L		current vertical position
lmarg	0		N	2.1	left margin
loginname	--		S	1.2	user's login name
lspacing	2*0	×	L	2.2	space after each line
maxline	--	×	L		max value for lineno
minute	--		N	1.2	minute of the hour
month	--		N	1.2	month of the year
newpage	""		S	3.7	trap before next page
nextpage	1		N	2.6	number of next page
ofooter	0	×	N	2.5	number of odd footers
oheader	0	×	N	2.5	number of odd headers
owindow	0		N	2.1	window on odd pages
pageno	0		N	2.6	number of current page
pagetop	""		S	3.7	trap after next page header
paraind	5		N	2.2	indent for .para
params	--	×	N	3.6	number of parameters
paraneed	3*0		L	2.2	need for .para
paraspacing	3*0		L	2.2	spacing for .para
partsperline	2	×	N	1.2	part lines per whole line
rmarg	71		N	2.1	right margin
second	--		N	1.2	second of the minute
spacer	'␢'	×	S	1.1	non-separating space
topspace	0*0		L	2.1	space at the top of each page
topwindow	0*0		L	2.1	window at top of each page
totalchars	0		N		total characters output so far
weekday	--		N	1.2	day of the week
window	0		N	2.1	window on current page
year	--		N	1.2	year

8.2 – <u>Table of Commands</u>: The table in this section summarizes all of LO's commands. The column headings are:

- **Command** Command name.
- **Abv** Abbreviation for the command, if one exists.
- **Brk** The entry is "yes" or "no" as the command does or does not cause a break. An entry of "-" means that it sometimes does – see the reference. An entry of "?" means that the command itself does not, but an effect may include a break. (For example, ".subst" does not itself cause a break, but the result of the substitution may be a break-causing command.)

Buf — The entry is "yes" or "no" as the command is or is not meaningful when output is to a buffer. An entry of "-" means that it sometimes is - see the reference.

Ref — Reference. This is the section number in which the command is discussed. If no reference is given, the command is discussed only in Chapter 5.

Args — The argument types are listed. A value in "(...)" indicates the default value used if the argument is missing. An entry of "--" means the argument is too complex to summarize in the available space - see the reference.

Command	Abv	Brk	Buf	Ref	Args	Description
.bline	.bl	no	no	2.1	L	set page length
.break	.br	yes	yes	2.2		break output text
.bt		no	yes	3.8	<text>	send <text> to 5BTF
.buffer	.bu	-	yes	2.7	B	output to buffer B
.center	.ce	yes	yes	2.4	<text>	<text> is centered
.charpos		no	yes	2.2	N	"tab" to position N
.clear	.cl	no	yes	2.7	B (actbuf)	clear buffer B
.comment	.co	no	yes		<text>	comment - ignored
.console		?	yes	3.8	<text>	I/O to console
.declare	.de	no	yes		<V-list>	declare variables
.declarebuf	.db	no	yes	2.8	<V-list>	declare buffers
.declareinteger	.di	no	yes	2.8	<V-list>	declare integers
.declareline	.dl	no	yes	2.8	<V-list>	declare LINes
.declarestring	.ds	no	yes	2.8	<V-list>	declare strings
.efooter		no	no	2.5	N, <triple>	store even footer
.eheader		no	no	2.5	N, <triple>	store even header
.end		no	yes	3.6		end an input source
.expand		?	yes	3.11	--	
.flag	.fl	no	yes	3.5	N (1)	flag next N lines
.footer		no	no	2.5	N, <triple>	store a footer
.footnote		no	no	3.4	--	store a footnote
.footspace		no	no	2.1	L	set #footspace#
.fspacing		no	no	2.1	L	set #fspacing#
.header		no	no	2.5	N, <triple>	store a header
.headspace		no	no	2.1	L	set #headspace#
.help		no	yes		<text>	call 5HELP
.hspacing		no	no	2.1	L	set #hspacing#
.if		?	yes	2.9	--	conditional command
.include	.inc	no	yes	3.8	<file>	include a text file
.indent	.ind	yes	yes	2.2	N	indent next by N
.insert	.ins	yes	yes	2.7	B	insert buffer B
.label		no	yes	2.9	<word>	terminate .skipto
.leave	.le	yes	yes	2.2	L (1*0)	leave spacing L
.lmarg	.lm	no	yes	2.1	N (0)	set the left margin
.lspacing	.ls	no	yes	2.2	L	set spacing
.need	.ne	no	no	2.2	L	require L spacing

Command	Abv	Brk	Buf	Ref	Args	Description
.nextpage		no	no	2.6	N	number next page
.ofooter		no	no	2.5	N, <triple>	store odd footer
.oheader		no	no	2.5	N, <triple>	store odd header
.output		no	yes	3.8	B, <file>	output buffer B
.overstrike	.ov	no	yes	3.9	<w> <text>	overstrike
.page	.pa	yes	no	2.6	N (nextpage)	eject the page
.pageno		no	no	2.6	N	set #pageno#
.para	.pr	yes	yes	2.2		start new paragraph
.paraind	.pri	no	yes	2.2	N	indent for .para
.paraneed	.prn	no	yes	2.2	L	need for .para
.paraspacing	.prs	no	yes	2.2	L	spacing for .para
.plain	.pl	yes	yes	2.4	<text>	store text verbatim
.proc		yes	yes	2.4		end .unproc
.rescan	.rs	no	yes	3.6	B	rescan a buffer
.reset		no	yes	3.7		reset a trap
.rmarg	.rm	no	yes	2.1	N	set right margin
.set		-	yes	3.10	--	set switches
.skip		no	yes	2.9	N (1)	skip N input lines
.skipto		no	yes	2.9	<word>	skip lines
.store	.st	no	yes	2.8	V,<text>	store into V
.storeinteger	.si	no	yes		V, N	store N into V
.storeline	.sl	no	yes		V, L	store L into V
.storeroman	.sr	no	yes		V, N	roman numeral
.storeromanupper	.sru	no	yes		V, N	ROMAN numeral
.storestring	.ss	no	yes		V,<text>	store <text> into V
.subst	.su	?	yes	2.3	<text>	substitute
.tabsin		no	yes	3.1	N, N, ...	set input tabs
.tabsout		no	yes	3.1	N, N, ...	set output tabs
.trap		no	-	3.7	<ev> <text>	set a trap
.triple	.tr	yes	yes	2.4	<triple>	store a triple
.undeclare	.und	no	yes	2.8	<V-list>	undeclare variables
.unproc	.unp	yes	yes	2.4		store text verbatim

8.3 — Options for ~.set~: The following table summarizes the options for ~.set~. This command is discussed in detail in Section 3.10, and most of the options are discussed in that part of this document in which they are relevant. Many of the options use an internal switch SW. This is initially ON when starting to scan a ~.set~ line; it is controlled by the options ~on~ and ~off~. The column headings in the table are:

Option The option.
Abv The abbreviation, if one exists.
Params Parameters. If none is shown, none is expected. The

	notation "..." means that the parameter is repeatable. For the meaning of abbreviations see the reference.
SW	The entry is "yes" or "no" as the option is or is not dependent on the setting of the switch SW. If it is "yes", then "on" or "off" may may precede the option.
Ref	This is the section in which the option is discussed.
Sets	This is the LO feature set by this option. A value in {...} is the default value.

Option	Abv	Params	SW	Ref	Sets
adjust			yes	1.1	right-adjust {ON}
default		<type><val>...	no	2.8	declaration default
ender		<char> ...	yes	3.2	sentence end chars
fill			yes	1.1	fill mode {ON}
flagger		<word>	no	3.5	flagger {*}
hyphenator	hy	<char>	no	3.3	hyphenator {}
insert			yes	2.7	buf insert sw {OFF}
off			-	3.10	turn SW off
on			-	3.10	turn SW on
overstrike	ov	<char>	yes	3.9	overstrikeable chars
rpar		<char> ...	yes	3.2	sentence end - paren
spacer		<char>	no	1.1	spacer {h}
starter		<char> ...	yes	3.2	sentence start chars
variabledelimiter	vd	<char>	no	2.3	var delimiter {#}
warn		<option> ...	yes	1.5	warning switches
wide			yes	3.2	sentence end {OFF}
window		<opt> V, ...	no	2.1	windows {0}

Index

Following is an index of various terms used in this document. All variables are described in Chapter 4, and all commands in Chapter 5. Neither commands nor variables are listed in this index, since Chapter 8 contains tables listing each predefined variable and each command, along with a reference for each to the section in which it is described.

The references appear here in the same order that they appear in the document. This is not necessarily the order of importance.

The notation P:L refers to line L on page P, where the line number starts at one with the first text line (not counting the header)[33]. For example, a reference to this point in the text would appear as "82:13".

adjust mode 4:3, 42:27
BCPL 73:23
buffers 24:4

chopping 3:31
comments 9:24, 10:4, 13:15, 68:9
conversion 7:3, 7:10, 8:32, 27:23

data types 5:7
errors 11:3, 74:1
expressions 7:17

fill mode 4:3, 21:27, 43:10
flags 34:10, 34:22, 43:16
footers (see headers)

footnotes 32:32, 37:35
headers 13:25, 22:13, 68:23
help call 11:3, 65:35, 74:10

hyphenation 31:18, 43:20
h (script h) 4:16, 25:9, 32:12
include files 40:24

LDX 11:19, 11:29, 74:6
macros 35:17
margins 14:46, 22:26, 34:6

---

(33) It is derived from the LO variable #linecount#.

page numbering	23:11
parameters	35:31
prompt	74:13
sentence end	30:22
substitution	18:15
tabs	4:22, 29:8
TO	12:7
traps	38:22, 69:33
triple	20:17
variables	4:30, 25:24, 47:4, 77:6
warnings	11:23, 27:6, 44:28
windows	15:24, 35:7, 45:3
α	8:38
β	9:2
γ	9:6
λ	9:11
‖	10:3
→	35:34
..	13:17, 36:29, 65:22
.+	13:20, 36:22
.*	13:15

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-75-119	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) LO – A Text Formatting Program		5. TYPE OF REPORT & PERIOD COVERED Technical Note
		6. PERFORMING ORG. REPORT NUMBER Technical Note 1975-13
7. AUTHOR(s) Evans, Arthur, Jr.		8. CONTRACT OR GRANT NUMBER(s) F19628-73-C-0002
9. PERFORMING ORGANIZATION NAME AND ADDRESS Lincoln Laboratory, M.I.T. P.O. Box 73 Lexington, MA 02173		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order 2006
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE 21 February 1975
		13. NUMBER OF PAGES 92
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Electronic Systems Division Hanscom AFB Bedford, MA 01731		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

speech understanding systems
text formatting program
TX-2 Computer

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

LO is a text formatting program used to prepare documents such as this report. It reads an input text file and creates an output file. The input file contains text to be printed, interspersed with commands to LO that direct its operation. Commands are expressed in a language of considerable sophistication. The command language is described, the presentation being designed to be suitable both as a primer to the learner as well as a reference manual for the expert.

LO currently runs on the TX-2 Computer at Lincoln Laboratory, and details are provided on its use in that environment. There is a brief discussion of the steps involved in implementing it on a different computer system.