HONEYWELL

MULTICS WORDPRO REFERENCE MANUAL

SOFTWARE

MULTICS WORDPRO REFERENCE MANUAL

SUBJECT

Detailed Description of the Multics Word Processing System (WORDPRO), Including Comprehensive Discussions of the Various Facilities and Related Commands

SPECIAL INSTRUCTIONS

This manual supersedes AZ98, Revision 1, dated November 1978, and its addendum AZ98-01A, dated August 1979. Refer to the PREFACE for "Significant Changes."

The manual has been extensively revised and reorganized. Throughout the manual, change bars in the margins indicate technical additions and asterisks denote deletions. Sections 1, 2, 3, A, B, C, and D are either new or completely rewritten and do not contain change bars.

SOFTWARE SUPPORTED

Multics Software Release 10.1

ORDER NUMBER

AZ98-02

July 1983

Honeywell

PREFACE

This manual describes the capabilities of **WORDPRO**, the Multics word processing system.

WORDPRO is comprised of a set of standard Multics commands that perform various word processing tasks. Readers of this manual should have a working knowledge of the Multics command environment. *New Users' Introduction to Multics - Part I* and *Part II*, (Order No. CH24 and CH25), respectively, are very useful as they provide programmers and other users with a basic introduction to Multics.

This manual contains references to the *Multics Commands and Active Functions*, (Order No. AG92), referred to as the *Multics Commands*, *Multics Programmer's Reference Manual*, (Order No. AG91), referred to as the *Multics Reference Manual*, and the *Multics Subroutines and Input/Output Module*, (Order No. AG93), referred to as the *Multics Subroutines*.

Often, user-typed lines and lines displayed by Multics are shown together in the same examples. To differentiate between these lines, an exclamation point (!) precedes user-typed text. This is done only to distinguish user text from system-generated text, it is not to be included as part of the input line. Also, a "carriage return" is implied at the end of every user-typed line.

Note: Because of page constraints in this document, certain character strings of data used in examples may not match exactly the information as seen on a user's terminal. That is, the character strings in examples may be folded (contained on several lines), whereas the actual interactive (live) session may display the same information on a single line or multiple lines with different line breaks than shown.

File No.: 1L13 AZ98-02

Significant Changes in AZ98-02

The following commands and subroutine have either changed, been deleted, or are new in this version.

REVISED	NEW	DELETED
add_symbols	append_list	qedx
change_symbols	compdv	
compose	compose_index	
process_list	convert_runoff	
sort_list	describe_list	
	display_comp_dsm	
	display_list	
	expand_device_writer	
	expand_list	
	format_document	
	hyphenate_word_	
	modify_list	
	process_compout	

Many of the formatting controls in Section 2 were revised and are not identified in the following list. Only deleted and new controls are included. The deleted controls were undocumented in this release, but will remain fully supported for an indefinite period.

DELETED			NEW			
.~	.fl	.spc	.+	.cfl	.ett	.pfl
.bb	.fla	.tb	..	.chl	.fnt	.phl
.bbc	.fle	.tbb	.*	.ecf	.frf	.spt
.bbe	.flo	.tbe	.bbt	.ech	.fth	.tac
.bbi	.hl	.tlc	.bcf	.else	.ftr	.tcl
.be	.hla	.tlh	.bch	.elseif	.gl	.then
.bec	.hle	.tre	.bet	.endif	.hit	.trn
.bee	.hlo	.trf	.bpf	.epf	.if	.ttl
.br	.ps	.unn	.bph	.eph	.ift	.unb
.cbn	.sp	.wi	.btc	.eqc	.indctl	.unh
			.btt	.etc		

The section on "Electronic Mail" was deleted; refer to *Mail System Users' Guide*, (Order No. CH23).

CONTENTS

SECTION 1

GENERAL INFORMATION

WORDPRO is the Multics word processing system. It consists of a set of related Multics commands that assist Multics users in the input, update, and maintenance of high-quality documents. Applications of WORDPRO range from simple form letters to complex technical manuals. Because the WORDPRO system is integrated into the Multics operating software, users can develop and maintain all types of documents at the same time they are performing other data processing activities. Document preparation is accomplished rapidly, increasing productivity and producing characteristically superior results. Additionally, WORDPRO provides Multics security, ease of use, and document management tools that are not available in other systems.

TEXT EDITING

The ted text editor is the suggested editor to be used for WORDPRO application because of its many powerful features that can simplify the task of maintaining large, complex documents. (Refer to *Multics Text Editor (TED) Reference Manual*, (Order No. CP50) for information about this command.) The ted text editor is but one of several editors available on Multics (others are edm, emacs, qedx, and teco).

WORDPRO FORMATTER

The WORDPRO system offers two methods of text formatting by utilizing the format_document command (the elementary text formatter) or the compose command (the full-blown WORDPRO text formatter). To decide which command to use, you must first decide what you want the text formatter to do. The format_document command is easy to learn and performs quite quickly when compared to either compose or runoff (see note below), but it performs only a small number of special actions. (Refer to Section 4, format_document command, for a complete description of elementary text formatting, including the supported format controls, examples, etc.) The compose command, on the other hand, is more difficult for a new user to understand because it offers many more features (refer to Section 2 for a description of the formatting controls, and to Section 4 for a description of the compose command). In other words, if you are preparing a simple memo or business letter, you may choose to use format_document, but if you require more complex items such as footnotes, change bars, or multicolumn text, you *must* use compose and its extended group of formatting controls. Keep in mind that what you learn about format_document also applies to compose (i.e., format_document utilizes a small, compatible subset of compose controls), so you may wish to first learn about format_document and then switch to compose. You can use compose to process a document with format_document controls.

Note: The runoff command, described in the MULTICS COMMANDS, is a forerunner to compose and is capable of performing only a limited number of compose functions.

In addition to the formatting choices noted above, WORDPRO provides the following outstanding features:

- A graphic function for generating simple artwork.
- An online dictionary for detecting spelling errors.
- A Speedtype function for improving productivity.
- A list-processing function for creating and maintaining data records.

SECTION 2

WORDPRO TEXT FORMATTER

The WORDPRO system uses the **compose Text Formatter** (referred to here simply as the *Formatter*) to format text. The input to the Formatter is a file (or group of files) containing plain text and embedded formatting directives (or controls).

Control arguments supported by the compose command (described in Section 3) allow the user to "tailor" an invocation of the Formatter by modifying a wide variety of default values and initial parameter values. Most importantly, the desired output device is given with a control argument, thus allowing input files to be completely device-independent.

Formatted output may be directed back to the user's terminal or to a file for eventual transcription to another online device (e.g., a lineprinter or typesetter) or medium (such as magnetic tape) to be transported to an offline device. If the output is directed back to the user's terminal, it may be printed page by page to allow positioning of forms. Pages damaged during printing (e.g., by a paper jam or ribbon failure) may be reprinted without having to restart the entire document.

GENERAL SYNTAX

A formatting control is a delimited character string having the form *.XXX variable-field*, where *XXX* is a one- to six-character keyword token and *variable-field* contains parameter values and/or keywords that are interpreted during processing of the control. A single blank (ASCII SP) is required between the keyword token and the variable field. The delimiter may be either of the delimiters described under input line or symbol delimiter (both discussed under "Formatting Terminology" below). The interpretation of the variable field for each control is discussed later in the descriptions of the controls. If .XXX is the first non-blank string in an input line, then the line is processed as a control line; otherwise, it is processed as a line of text (also refer to "Notes" under Formatting Controls later in this section).

In the control descriptions, if a space is shown in a symbolic variable field, then at least one blank must appear in that position. Any other space given (outside quoted strings) is ignored.

The symbols that appear repeatedly in variables fields are:

#	an unsigned number. If an integer is needed, the given value is truncated to next smaller integer.
n	a signed or unsigned number. If an integer is needed, the given value is truncated to next smaller integer.
expr	a numeric or string expression.
a	any single character.
ab	any character pair.

name — a name string up to 32 characters beginning with an alphabetic and containing only alphanumerics and the underscore (_).

string — an arbitrary character string.

title — a three-part title of the form |part1|part2|part3|.

SECTION ORGANIZATION

The remainder of this section is subdivided as follows:

- Formatting Terminology (description of technical terms)
- Formatting Features (description of features that make up the Formatter)
- Built-in Variables (description of variables)
- Creating Artwork (description of the artwork features)
- Formatting Controls (description of Formatter controls, alphabetically organized)
- Comprehensive Control Summary (grouping of all controls with associated page reference to the control description)

The Formatter controls can be divided into three groups:

basic — features with which users can process most prose documents having straightforward formatting requirements.

intermediate — features with which users can process sophisticated technical documents having complex formatting requirements.

advanced — features with which users can create "macro" procedures that are context- or device-dependent, or provide for automatic generation of Tables of Contents, Glossaries, Bibliographies, and Cross Reference Indexes.

Refer to the "Comprehensive Control Summary" located at the end of this section for a GROUPING of all controls. Using this summary, a working unit can tailor responsibilities for individuals or work groups based upon the complexity required for the job or the individual. That is, the Formatter can be learned in steps of complexity: basic, intermediate, and advanced.

FORMATTING TERMINOLOGY

The terminology needed for understanding of the formatting features and controls are discussed here. In addition, any terms appearing below in italics are discussed separately.

back page
: the page to the left when a book is open for reading. It normally has an even page number.

blank space
: some amount of empty space intentionally left in the output line. When *filling* text, it is preserved when it appears at the left margin otherwise, it is

compressed to a single blank (ASCII space). When not *filling*, it appears in the output as given.

bottom margin
: the amount of *white space* appearing between the *page footer* and the bottom of the page.

built-in variables
: a collection of data items maintained by the Formatter whose values are available to the user by means of *substitution of variables*.

center
: the midpoint between the *left margin* and the *right margin*.

change bar
: a mark in the page margin space showing that text has been added, deleted, or modified.

column
: a vertical area of text on the page, located by its left and right margins which determine its *width*. By convention, "column 0" refers to the entire page and is always defined.

control
: a formatting control as discussed in "General Syntax" above.

control line
: a *control* and all its parameters.

counter variable
: a *numeric variable* whose value is incremented by some specified amount every time it is referenced during *substitution of variables*.

display mode
: the number system in which numeric values are shown in *substitution of variables*. The available modes are Arabic, binary, octal, hexadecimal, uppercase and lowercase Roman, and uppercase and lowercase alphabetic.

equation block
: an inline *text block* containing *title* lines.

escape
: a technique for ensuring the literal appearance of certain characters that have special meaning to the Formatter or whose occurrence causes some special action.

expression
: a construction made up of *symbolic references*, *special references*, literal constants, arithmetic, relational, or logical (Boolean) operators, and conforming to the normal rules of algebra. An expression may be a *numeric expression*, a *string expression*, a *relational expression*, or a *logical expression*.

expression evaluation
: the procedure of reducing an *expression* by *substitution of variables* and performing the actions of the operators. The order of precedence in evaluating mixed expressions is:

    numeric expressions
    string expressions
    relational expressions
    logical expressions

filling
: the procedure of accumulating text from input lines so as to fit as much as possible into the output line. When not filling, each input text line generates an output line; overlength lines are not truncated and may extend past the right margin.

flush both (justified)
: text aligned to both margins by adjustment of the *wordspace* and/or *letterspace*.

flush left
: text aligned to the left margin.

flush right
: text aligned to the right margin.

font
: the design of a typeface supported by the output device. Examples are "ascii", "Helvetica", "Times Roman", and "Italian Gothic". Fonts may have several intensities such as light, medium, bold, and ultra bold, and may be available in both italic and upright form. A font may also be the name of a collection of type having no filial association such as "News Commercial Pi" or "Universal Greek and Math". The fonts supported by an output device are given in its device description table.

footer margin
: the minimum amount of *white space* appearing between the last *text block* (or *footnote*) and the *page footer*.

footnote
: a specially formatted *text block* containing additional or descriptive information that normally appears at the bottom of a page (see footnote description under "Formatting Features" later in this section).

formatted tables
: the formatting style in which the *columns* are further subdivided into "table columns". The table columns are independent of each other and each has its own set of formatting parameters. In this style, the *text block* is a "table entry" that may contain text for each of the table columns. The *text block* is not finished until processing reverts back to the containing *column*. Text from one table column does not spill over into the next. When producing output in this style, the Formatter is said to be in "table mode".

front page
: the page to the right when a book is open for reading. It normally has an odd page number.

gutter
: when more than one *column* appears on a page, the amount of space between the right margin of one *column* and the left margin of the next. If the first *column* does not lie at the page left margin, the page is said to have a "left *gutter*".

header margin
: the amount of *white space* appearing between the *page header* and the first *text block*.

indention
: a local adjustment to the */left margin* or */right margin* value. It does not change the margin value, but does control the position of the first and last characters in an output line.

input line
: the unit of information from the input file processed by the Formatter. */control lines* are delimited by un*protected* semicolon (;) characters or NL (ASCII code 012) characters; *text blocks* are delimited by NL characters only. It may contain any combination of text and/or *control*s. After all *control*s in the line are processed, any remaining text is accumulated into a *text block* for eventual placement on the output page.

lead
: the amount of space appearing between output lines. For example, a */linespace* value of 1 produces output with zero lead while a */linespace* value of 1.5 produces output with a half-line of lead.

leader
: a character or short character string that is replicated to fill the blank space between any *tabulation* stop position or *formatted table* column and the end of text in the preceding one (or the preceding margin). The leader string is formatted flush right in the blank space.

left margin
: the left-most position on the page or column in which text is allowed to appear.

letterspace
: the amount of space appearing between individual characters in the output line.

line art
: a simple graphic "picture" constructed from *rule*s and other special symbols supported internally by the Formatter (see "Creating Artwork" below).

linespace
: the amount of automatic vertical advance when proceeding from one output line to the next.

logical expression
: an *expression* consisting of any number of *relational expression*s and logical (Boolean) operators. The order of precedence of the logical operators is:

&	Boolean AND
^|	Boolean EXCLUSIVE OR
|	Boolean OR

math symbol
: one of a subset of the symbols of mathematics that the Formatter can fabricate to span multiple lines. The subset is limited to those symbols normally needed for the syntax definitions of various meta-languages (particularly COBOL) and the dyadic (or infix) operators (see "Creating Artwork" later in this section).

numeric expression
: an algebraic *expression* consisting of *symbolic reference*s to *numeric variable*s and *counter variable*s, literal numeric constants, other *numeric expression*s, arithmetic operators, and parentheses. The order of precedence of the arithmetic operators is:

-	arithmetic negative sign

+	arithmetic positive sign
*	multiplication
/	division
\	modulus (remainder)
+	addition
-	subtraction

numeric variable
: a variable whose value may only be a decimal number.

page footer
: an optional special text block that appears at the bottom of every page. It may contain both *title* lines and normally-formatted text.

page header
: an optional special block that appears at the top of every page. It may contain both *title* lines and normally-formatted text.

page mode
: the state of the Formatter when it is processing text according to page formatting parameters, not constructing a special text block, and not in any of the intermediate or advanced special processing modes.

picture block
: a form of keep (see text block (intermediate) later in this section) that allows following text to be inserted ahead of it (also known as a "float").

plain text
: an ordinary text paragraph.

pointsize
: the size of type in the current font given in typographic points (1 point = 1/72 inch).

protected character
: a character whose literal presence is ensured. One of the design philosophies of the Formatter is that its control syntax does not require "blind" (or nonprinting) characters in the input files. Therefore, there must be a mechanism for signalling the literal occurrence of the several graphic characters that have special syntactic meaning. The mechanism chosen is flagging the literal occurrence by preceding it with the asterisk (*) (also called "escaping" the literal occurrence).

relational expression
: an *expression* consisting of two *numeric expressions* or *string expressions* and a relational operator. The order of precedence of the relational operators is:

=	equal
^=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

right margin
: the right-most position on the page or column in which text is allowed to appear.

rule
a horizontal, vertical, or slant line of some given thickness and length.

running multicolumn
the formatting style in which parallel, identically formatted *columns* of text are placed side by side on the page. Text from the bottom of one *column* spills over to the top of the next *column*. When producing output in this style, the Formatter is said to be in "multicolumn mode".

special reference
a form of *symbolic reference* in which the referred object (that is, the string contained within the symbol delimiters) is not a variable name but has one of the following forms:

- any Formatter control
- an active function (see *Multics Commands*) also contained within brackets ([])
- an *expression evaluation* also contained within braces ({})

string expression
an *expression* consisting of *symbolic references* to *string variables*, literal string constants, and *substring expressions*. There are no "string operators"; the only operation is concatenation, that being implied by the positions of the given strings. A *string expression* must begin with a double quote (").

string variable
a variable whose value may be any character string.

substitution of variables
the procedure by which *symbolic references* to *built-in variables* and *user variables* are replaced with their current values.

substring expression
an *expression* consisting of a *symbolic reference* to a *string variable* or a literal string constant, followed by one or two *numeric expressions* given in parentheses, and having the form:

"string" ({i-expr} {,k_expr})

where i_expr has the value i and k_expr has the value k. Note that k_expr may not be given unless i_expr is also given. The substring extracted depends on the values of the *numeric expressions* as follows:

i	the substring begins with the *i*th character of *string* and continues through its end.
-i	the substring begins with the *i*th character *from the end of string* and continues through its end.
i, k	the substring begins with the *i*th character of *string* and consists of the next *k* characters.
-i, k	the substring begins with the *i*th character *from the end of string* and consists of the next *k* characters.
i, -k	the substring begins with the *i*th character of *string* and continues through the *k*th character *from its end*.
-i, -k	the substring begins with the *i*th character *from the end of string* and continues through the *k*th character *from its end*.

In all cases, the length of the substring must be positive (or 0) and neither *i* nor *k* may reference a character outside the range of *string*.

symbol delimiter
: the graphic character used to delimit a reference to a *symbolic variable*. (see "delimiter, symbol" under "Formatting Features" later in this section).

symbolic reference
: a reference to the current value of a variable, made by enclosing its name in symbol delimiter characters.

symbolic variable
: the reference name of a built-in or user-created variable.

tabulation
: the technique of presetting specific horizontal positions on the page and then signalling an advance to the next such position with a single keystroke. This technique is directly analogous to "tab stops" on a typewriter; however, the concept is extended by the ability to provide a *leader*.

text block (basic)
: (also see "intermediate" description below) the unit of output produced by the Formatter. A *text block* is constructed by accumulating output text from the input file until a *text break* is encountered. A text block may contain up to 1000 output lines.

  The *text block*s supported by the Formatter are *plain text* paragraphs, *white space*, *page header*s, *page footer*s, *footnote*s, *text header*s, and *text caption*s.

  Text blocks are classed according to the way they are placed on the page (primary type) and their content or usage (secondary type). When a *text block* is placed on the page, it may be subjected to *widow* processing.

  Primary blocks are physically separate entities and may not intersect or overlap. Two primary block types are defined:

  inline — *text block*s that are placed on the page immediately upon completion.

  special — *text block*s that are handled according to special considerations determined by their secondary type.

  Secondary blocks may overlap and may be contained within each other and within primary blocks. Several secondary block types are defined; however, for basic formatting, the only one of interest is:

  title — blocks that may contain *title* lines.

  Under certain conditions, the processing of a *text block* may be suspended. When a block is suspended, certain parameters and variable values associated with the block are set aside (or pushed) for resumption of processing in that block. When the block is resumed, those items set aside are restored to an active state (or popped) and processing of the block continues as though it had never been interrupted.

text block (intermediate)
: (also see "basic" description above) the remaining secondary block types are:

  art — blocks that may contain conventional artwork constructs which are converted to symbols or line art.

  keep — blocks that are not subject to widowing.

literal — blocks that may contain lines that appear to be control lines.

text break (basic)
(also see "intermediate" description below) an interruption of normal processing in the current *text block*. Three different basic breaks are defined:

- format — finish processing the current output line (i.e., finish formatting any pending text as a short line), then begin a new output line in the current *text block*.
- block — finish processing the current *text block* by finishing the current output line as above, adding any pending *text caption*, then begin a new *text block*.
- page — finish processing the current page by finishing the current block as above, then finish the page by adding any pending footnotes and page footer, and eject the page. Any text deferred by *widowing* is picked up at the top of the next page.

text break (intermediate)
(also see "basic" description above) one additional text break is defined:

- column — finish the current *column* by finishing the current *text block* and filling out the *column* with nontrimmable white space, then begin processing in the next *column* (or the next page, if the *column* is the last on the page).

text caption
an optional special block that is inseparably attached to the end of an output *text block*. It may contain both *title* lines and normally formatted text.

text header
an optional special block that is inseparably attached to the beginning of an output *text block*. It may contain both *title* lines and normally formatted text.

title
an output line that is formatted in three parts: a *left margin* part, a *center* part, and a *right margin* part. In *page header*s and *page footer*s, a distinction is made between "blank" titles which appear (as *white space* lines) in the output and "null" titles which do not appear in the output.

title delimiter
the graphic character used to delimit the parts of a *title* line.

top margin
the amount of *white space* appearing between the top of the page and the *page header*.

user variables
a collection of data items defined and maintained by the user whose values may be retrieved by means of *substitution of variables*.

white space
some amount of empty space intentionally left on the page; e.g., interparagraph space appearing at the top of a page or column (discarded), or space left for later "paste up" of hand drawn artwork (left as-is). It may be trimmed, discarded, or left as-is.

widow
: some (usually small) fragment of text that is the least amount that may be split away from its containing block in the event of a page or column overflow. The default fragment size is two lines but may be changed by the user.

width
: the amount of space in a *column* or the page available for text.

wordspace
: the amount of space appearing between words of justified text in the output line.

FORMATTING FEATURES

The formatting features are described below.

Change Bars

Text revision marks may be used on a line-by-line basis to show addition, deletion, or modification of text material. The marks may be associated with a change level character and the Formatter allows generation of marks based on these characters. By default, the marks are a vertical line (|) for additions and modifications, and an asterisk (*) for deletions, and they appear in the outside page margin. See the .cba, .cbd, .cbf, and .cbm controls below, and the -change_bars and -change_bars_art control arguments in the description of the compose command in Section 3.

Character Translation

In rare instances it is necessary to translate certain characters in the input file into other characters in the output, due primarily to unavoidable conflicts among various characters that have special meaning. For example, it is not possible (because of the nature of the Multics command processor) to evaluate an active function expression that contains parentheses or brackets. Such characters must be processed as other characters without special meaning and translated to the desired characters when the processing is complete. See the .trn control.

Note: Due to the complexity of this translation feature and the number of times it must be performed during file processing, it is quite expensive and users are advised not to use it indiscriminately. It should be disabled as soon as the need for it has passed.

Comments

Comments not affecting the formatted output may be placed as desired in the input files. See the .* control.

Default Conditions

The Formatter initializes the values of the formatting parameters at the beginning of each input file given in the command line so as to produce the following default output:

- 8.5 x 11 inch page (85 columns x 66 lines)

- 1 inch margins; top, bottom, left, and right (6 lines top and bottom; 10 columns left and right)
- 6.5 inch single-column text; filled, justified, and unhyphenated (65 columns)
- no page headers or footers
- initial font and pointsize as given in the device description table for the output device
- output written to the user_output I/O switch (normally connected to the user's terminal)

Certain of these default values may be changed with control arguments given in the command line that invokes the Formatter (see the description of the compose command later in this section). All formatting parameter values may be changed locally with formatting controls in the input file(s).

Delimiter, Symbol

The default symbol delimiter character is percent (%). Although isolated literal appearances of the character may be protected with the escape convention, certain input files may require extensive use of the character in lines that are subject to evaluation. To relieve the user from the necessity of escaping all the literal appearances, the symbol delimiter character may be changed to any other character whenever desired. See the .csd control.

Delimiter, Title

The default title delimiter character is the vertical line (|). Although isolated literal appearances of the character in title lines may be protected with the escape convention, certain input files may require extensive use of the character. To relieve the user from the necessity of escaping all the literal appearances, the title delimiter character may be changed to any other character whenever desired. See the .ctd control.

Document Indexing

Many technical documents require a cross reference index showing those pages whereon certain keywords and phrases are mentioned. The formatter provides the ability to gather data for such indexes during the processing of the input files. The feature may also be used to gather data for bibliographies and glossaries. Data for as many as ten indexes may be gathered simultaneously. See the .hit control.

HIT STRINGS

A hit string is a line of text made up of keywords or phrases and various delimiters. There are no restrictions on the number or location of hit strings in a document. When the formatter encounters a .hit control, it prepends the given hit string(s) with the input line number, a constant string (used for synchronization), the delimiter change string (if any), and the hit type character, then appends the end delimiter and page number, and emits it to one of ten raw data files. The receiving raw data file is specified by an index number given with the .hit control.

Delimiters

Three characters are reserved for use as delimiters in the hit line. Their definitions and default values are shown below. If any of them are needed in the text of a hit string, the delimiters must be changed to allow their use as text. See the .hit control.

hit "|" the hit delimiter is used to signal the beginning of a hit string. It appears only once in a hit string, but a hit line may contain multiple hit strings, each with its own hit delimiter.

key "~" the key delimiter is used to separate individual keys in the hit string. It may appear as often as needed.

end ";" the end delimiter separates the keys from the page number. This delimiter is not given in the hit line but is appended (with the page number) by the formatter.

Hit Types

The following type of hits are available. The desired type is specified by giving its hit type character.

N a null key. The hit string does not cause an entry in the index and may be of any arbitrary form, only the hit delimiter is required. This hit type is useful for annotating the raw data file as desired (e.g., with the names of the sections of the document).

K specific keys. A primary keyword or phrase followed by one or more subordinate keys. Any number of subordinate keys may be given (depending on the number allowed the hit data processing program; see the description of the compose_index command) and they are treated as primary, secondary, etc., in the order given. For example:

```
K|commands~command lines
K|commands~arguments
K|commands~control arguments
```

S a "see" reference. As for the K type above except that there must be at least one subordinate key and the last key must make reference to some other primary key. No page number is shown in the index for a hit of this type. For example:

```
S|control arguments~see commands
```

U permuted uppercase keys. This type may have a primary key only. The primary key is expanded into the equivalent of a set of specific keys by extracting each word (or protected phrase; see the description of the compose_index command), translating it to all uppercase, and appending the given primary key as a secondary key. For example:

```
U|command control arguments
```

expands into the equivalent of:

K|COMMAND~command control arguments
K|CONTROL~command control arguments
K|ARGUMENTS~command control arguments

L — permuted lowercase keys. As for the U type above except that the new primary keys are forced to all lowercase.

I — permuted initial caps keys. As for the U type above except that the new primary keys are forced to initial caps only.

A — permuted as-is keys. As for the U type above except that the new primary keys are left as-is.

Error Messages

The Formatter issues error messages for all situations it determines to be errors, whether internal program errors, implementation restrictions, errors in the user's input file, or explicitly requested messages (see the .err control). The form of these error messages is:

```
filename; linenumber:  {filename; linenumber: ...}
     Error message(s)
     Source line
```

The first line shows the history of inserted files at the time of the error beginning with the command line input file. The second line shows any system error message that may have been returned to the Formatter and a message from the Formatter. The third line shows the offending input line (this line is not shown for messages requested with the .err control).

If formatted output is being written to an output file or the -check control argument (see the description of the compose command) was given with the invocation of the Formatter, then error messages are written directly to the error_output I/O switch.

If formatted output is being written back to the user's terminal, then error messages are accumulated in a list in the process directory. If the invocation of the Formatter is allowed to complete normally, the list is printed at termination. If the user QUITs the Formatter invocation, then the error list is printed in response to the program_interrupt command (see *Multics Commands*); otherwise, it is discarded when the Formatter invocation is released.

The Formatter also supports the severity active function (see *Multics Commands*) with the following severity schedule:

0 - No errors
2 - User errors (undefined variables, misspellings, invalid control parameters, etc.) that prevent specific actions.
3 - Missing/inaccessible insert files.
4 - Program errors/limitations and/or internal inconsistencies that may cause a formatter abort.
5 - Command line errors that prevent any execution.

Note, however, that error severity is NOT shown in the error messages.

Escaping Characters

Certain applications require the literal use of characters that have special meaning to the Formatter or are treated in some special way. Among such characters are the symbol delimiter, the title delimiter, and the ASCII motion characters (HT, SP, LF, FF, etc.). To subvert any special meaning or action, the characters may be protected by preceding them with an asterisk (*). Escaping of characters is part of expression evaluation (see "Expression Evaluation" below), therefore, lines containing escaped characters must be evaluated.

In general, any character may be escaped; however, there is a special set that is replaced with something other than the escaped character. This set is:

*`	left double quote (")
*'	right double quote (")
*-	an EN dash (useful in phototypeset documents where proportional spacing tends to nearly obliterate the hyphen -- use for the unary negation operator in text or the first character of control arguments)
*N	an EN space (single SP for ascii device)
*M	an EM space (double SP for ascii device)
*b	backspace (ASCII 010)
*n	newline (ASCII 012)
*s	wordspace (ASCII 040)
*t	horizontal tab (ASCII 011)
*f	formfeed (ASCII 014)
*c*ddd*	character whose *decimal* rank in the ASCII collating sequence is *ddd*.
*C*ddd*	
*c#*ooo*	character whose *octal* rank in the ASCII collating sequence is *ooo*.
*C#*ooo*	

Escaping from the Formatter

In some applications it is necessary to perform actions that are not supported by the Formatter but are supported by the Multics command processor. To perform such actions, it is possible to escape from the Formatter to the command processsor. See the .exc control.

Expression Evaluation

Expression evaluation takes place for:

- all specified expressions in the variable fields of controls.
- the substring expression of string expressions (see "Formatting Terminology" above).
- explicit evaluation constructs of the form "%{*expr*}%".
- all title line parts when they are inserted into the output.

The action is performed by scanning the expression *once* from left to right for symbol delimiters. When a delimiter is found, the action proceeds according to the character following the delimiter.

- If it is a symbol delimiter, then the pair is reduced to a single delimiter and scanning continues with the character following the pair.

- If it is an opening bracket ([), then the expression is scanned for a matching closing bracket (]) and the string thus contained is processed as an active function.
- If it is an opening brace ({), then the expression is scanned for a matching closing brace (}) and the string thus contained is processed (recursively) as an expression.
- If it is a dot (.), then the expression is scanned for the matching closing symbol delimiter and the string thus contained is processed as a formatting control.
- Otherwise, the expression is scanned for the next symbol delimiter and the string thus contained is processed as a symbolic reference.

File Insertion

Any Formatter input file may be inserted into the text at any point. This inserted file is a normal input file in all respects and may contain any valid combination of input text and formatting controls. Inserted files are located by means of the Formatter's search list (see "Search List" in the compose command description). See the .ifi and .rt controls.

Note: Once inserted, a file should remain in existence throughout the invocation of the Formatter. It is an error to attempt to reinsert a file that has ceased to exist after the first insertion.

Files, Auxiliary Output

In many applications (particularly in the preparation of Tables of Contents and cross reference Indexes), it is necessary to write text-dependent information to a file other than the normal output file. The Formatter supports the ability to write information to up to ten such auxiliary files. See the .wrt control.

Fonts and Type Sizes, Changing

The typeface or pointsize of the output may be changed at any time. See the .fnt control.

Footnotes

Footnotes are text blocks containing additional or descriptive material pertaining to some item in the material on the page and possibly referenced at one or more points on the page. Footnotes are placed at the bottom of the page, just ahead of the page footer or bottom margin. They may be formatted differently than main body text and are separated from the main body by a special footnote header line. When the Formatter is processing a footnote, it it is said to be in *footnote mode*.

Footnotes may be referenced or unreferenced, either globally throughout the document or locally for individual notes. Unreferenced footnotes may be defined anywhere on the page: referenced footnotes are defined at the point of their first references. Referenced footnotes are assigned sequential reference numbers that advance automatically as they are used on a page and resets to "1" at the top of each new page.

The amount of page space needed for unreferenced footnotes is deducted from the available text space.

The number of, and space for, referenced footnotes are attached to the text blocks containing their first references on a line-by-line basis, and are accounted for during widow processing. When output lines are spilled into the next column or page, any attached footnotes accompany them. Any footnote references in output lines spilled onto the next page are resequenced beginning with "1" at the top of the new page. If an output line and some of its footnotes (but not all) fit, then the line and those footnotes that fit are retained and the remaining footnotes are spilled. In this case, reference numbers for footnotes spilled onto the next page are *not* resequenced and output lines continue to be placed on the page until a true overflow condition occurs. See the .bbf, .bef, .frf, .ftp, .ftu, and .hlf controls.

Hyphenation

Hyphenation may be enabled and disabled either globally for the entire document or locally on a line-by-line basis.

Note: The Formatter does not support local discretionary hyphenation, so there is no mechanism to resolve the hyphenation of homographs such as "record" (rec-ord or re-cord) whose hyphenation is context-dependent.

The algorithm for hyphenation is not internal to the Formatter but is supplied as a free-standing subroutine in the Multics WORDPRO system. Because of this, the user is free to replace the Honeywell-supplied hyphenation routine with any other routine implementing a different algorithm as long as the interface requirements are met.

See the .hy, .hyf, and .hyn controls, the -hyphenate control argument in the description of the compose command in Section 3, and the description of the hyphenate_word_ subroutine in Section 4.

Indention

The position of the first and last characters in an output line may be adjusted with respect to the margins by giving left and right indention values, respectively. Positive indention values move the affected characters away from the margin (toward the center of the page) and negative values move them toward the margin. See the .in, .inl, .inr, and .inb controls.

Input Line Continuation

In some applications it is necessary to give very long input control lines, particularly lines having logical expressions containing other mixed expressions. Since such lines quickly exceed the line length for some terminals, an input line continuation feature is provided to improve input file readability. See the .+ control.

Linespacing

The amount of automatic vertical advance given with each output text line can be set to any desired value. See the .ls control.

Page Definition

The user may specify the dimensions of the output page for:

left and right page margins
top margin
header margin
footer margin
bottom margin

and the number of, widths, depth offsets, and gutters for columns on the page.

Values are given in terms of 10-pitch terminal characters and lines, and may have up to three decimal places (for example, "1.333" or "2.75"). It is an error to give any values that do not conform to the maxima or minima for the output device as given in the device description table. See the .pd, .pdl, .pdw, .vm, .vmt, .vmh, .vmf, and .vmb controls and the -indent control argument in the description of the compose command in Section 3.

In addition to the basic page definition parameters, the user may also specify the number of, widths, gutters, and initial page depths for, columns on the page. See the .pdc control.

Page Headers and Footers

A page header is a block placed at the top of each page; a page footer is a block placed at the bottom of each page. Both may contain title lines and normally formatted text. Any title lines may be assigned an index number so that they may be individually managed without affecting the rest of the lines in the block. Indexed lines in a page header are numbered from the top down; indexed lines in a page footer are numbered from the bottom up. Page headers and page footers may be specified the same for all pages or separately for front and back pages. The margin space is the same for all pages but the header and footer text may be specified the same or separately for odd and even pages. See the .phl, .bph, .eph, .pfl, .bpf, and .epf controls.

Running column header/footer capability is also available. This feature can be used to generate header lines BETWEEN the page header and first text (e.g., "Section XXX (cont.)") and BETWEEN the last text and the page footer (e.g., "Continued on page xxx").

The column header/footer lines are independent of any page header/footers and any local text titles/captions. The syntax for their use is (almost) the same as that for page headers and footers. See .chl, .cfl, .bch, .ech, .bcf, and .ecf controls.

Page Numbers

The Formatter counts pages as they are produced. The value of the page counter may be inserted into any title line part by enclosing its symbolic name in symbol delimiter characters as:

```
%PageNo%
```

The page counter is initialized to "1" at the beginning of each input file, but it can be reset to any desired value. Its value may be displayed as Roman or Arabic numerals, or as an alphabetic; the default is Arabic. See the .brp control. (Also, see "Page Numbers, Structured" below).

Page Numbers, Structured

The internal page counter maintained by the Formatter is not a simple integer counter as implied in "Page Numbers" above, but is an array of 20 counters in which counting is done in the last element. The counters hold the numeric values to be used in a "structured" page number of the form, for example, "2-6.3(b)14-V". The counters are initialized by giving the desired initial page number structure as the parameter of the .brp control.

Note: Presently, the entire structure must be repeated with the desired values for every structure change.

The page number structure is given as an ordered set of counter values and field separators. The counter values are given as they are to appear in the display of the structure in the formatted output and the separators may be chosen from ".", "-", "(", and ")".

Note that the "appearance" of a counter value may be ambiguous. For example, the desired values of the first five counters in the structure cited above are obvious but the value of the sixth is not. It could be either the uppercase roman representation of "5" or the alphabetic representation of "22". In order to resolve these ambiguities, a second parameter is supported by the .brp control. Ambiguities not explicitly resolved by the second parameter are resolved to Roman, hexadecimal, or alphabetic, in that order. See the .brp control.

Picture Blocks

A picture block is a special block that is placed on the page on a space-available basis. If a picture block does not fit, it is deferred to the next column or page and following text continues to be placed on the page until an overflow occurs. Picture blocks are stacked and placed on the page in the order they are given. See the .bbp, and .bep controls.

Printwheel Changing

For incremental plotting terminals that employ a "daisywheel" (or similar interchangeable typeface element), the Formatter supports the production of pages that require the use of more than one such printwheel.

This feature is accomplished as follows:

1. In the device description table for the terminal, the printwheels are assigned index numbers.

   Note: . In the device support modules available from Honeywell as Priced Software Products, the printwheel index numbers are:

1	10-pitch PICA
2	10-pitch APL
3	12-pitch ELITE

   These assignments are merely suggested; authors of device description tables for other devices are free to make any convenient assignments.

2. Each font defined for the terminal gives the required printwheel index number.
3. As a page is formatted, the index numbers of all required printwheels are recorded for later use.

4. When the page is printed (either directly to the terminal by the Formatter or from an output file by the process_compout command), the printwheel index number for the first unprinted text is compared to the index number of the mounted printwheel. (If the page is the first printed page for the invocation of the Formatter (or process_compout), the mounted printwheel is assumed to be #1; otherwise, it is the printwheel left mounted from the previous printed page, which may well be this same page with a different printwheel.)
5. If the printwheel index numbers differ and printwheel *n* is needed, then *n* BEL/HT sequences are transmitted, *leaving the print head at column 10n.* The user then changes the printwheel and types a single NL (ASCII code 012). Then, one more BEL/HT sequence is transmitted and *the print head returns to the left margin.* The user then repositions the page as necessary and types another NL.
6. If the printwheel index numbers are the same, then all text on the page for the mounted printwheel is printed in its required position with empty space left for text requiring other printwheels.
7. At the end of the page, if any text for other printwheels remains unprinted, the next printwheel index is taken from the recorded list and processing returns to step 4 above.

Note: Terminals that allow the simultaneous mounting of multiple printwheels are not supported.

Programming Features

The essence of programmability is the control of processing according to dynamic conditions and communication with the user during processing. The Formatter supports these abilities but the experienced user should note that the level of sophistication is about that of the BASIC programming language, except that conditional groups may be nested to any convenient depth. See the .ts, .go, .la, .if, .elseif, .then, .else, .endif, .ty, .rd, and .wt controls.

Note that it is an error to leave a group with the .go or .rt controls.

Punctuation Space

The Formatter recognizes sentence termination punctuation normally used in written English. When such punctuation is encountered in filled text, extra blank space may be inserted to separate the sentences. The amount of space inserted is given in the device description table for the output device.

Tables, Formatted

A formatted table is a form of output in which the text column is subdivided into vertical areas known as table columns, and each table column is given its own limited set of formatting parameters. The formatting parameters of each table column are independent of all the others and independent of the parameters of the containing text column.

When the Formatter is producing output in this form, it is in table mode. In table mode, the unit of output is not a simple text block, but is a table entry that may (and usually does) contain text from all the table columns. The completion of a table entry

is signalled by an event that returns the Formatter to the formatting parameters of the containing text column.

The Table of Contents of this manual is a formatted table with the Section numbers in the first table column, the titles in the second, and the page numbers in the last.

Additional capabilities provided by the table mode feature are:

1. Alignment of a given text string to the specified column. This can be used for applications such as aligning a column of prices at the decimal point. The TOC feature also uses this feature to align page numbers at the "-" separator. See ASTR as described in the .tab control.
2. Positioning of the text in a table column at the top, center, or bottom of the table entry. This can be used for applications such as aligning the price column (in a catalog) at the bottom line of the item description. The TOC feature also uses this feature to align the page number at the last line of the title. See V as described in the .tab control.

See the .tab, .taf, and .tan controls.

Tabulation

Up to 20 typewriter-like horizontal tabulation stop patterns may be defined. Each pattern may have up to 20 stop positions and an optional leader string may be given for each stop position. All defined tabulation patterns may be active at the same time, each with its own assigned "tab character". When tabulation is active, text filling is temporarily disabled and text alignment is forced to align-left. See the .htd, .htn, and .htf controls.

Text Alignment

Text may be formatted as flush left, flush right, flush both (justified), centered, or flush to the inside (binding edge) or outside. See the .all, .alr, .alb, .alc, .ali, and .alo controls.

Text Blocks, Secondary

Secondary text blocks are subblocks that may overlap and intersect and may be contained in primary text blocks. The secondary block types defined are:

art	blocks that may contain conventional artwork. See the .bba and .bea controls.
keep	blocks that may not be split between columns and/or pages. See the .bbk and .bek controls.
literal	blocks that may contain lines that appear to be control lines or have certain special characters in them. See the .bbl and .bel controls.
title	blocks that may contain title lines. See the .bbt and .bet controls.

Text Breaks

Text breaks are used to control the appearance of paragraphs and their placement on the page. See the .brb, .brf, .brn, and .brp controls.

Text Filling

Filling of text may be enabled or disabled at any point. See the .fi, .fif, and .fin controls.

Text Headers and Captions

A text header is a specially formatted subblock attached to the beginning of a text block; a text caption is a specially formatted subblock attached to the end of the text block. Both may contain title lines and normally formatted text. The attachment is inseparable; that is, the subblock is not split away from the text block if the text block must be split between pages. See the .tcl, .ttl, .btt, and .ett controls.

Title Lines

A title line is an output line that has three parts: a left margin part, a centered part, and a right margin part. The parts are given (where allowed) by enclosing them in title delimiter characters as follows:

```
|left part|center part|right part|
```

The default title delimiter character is the vertical line (|) but may be changed to any other character. (See the .ctd control).

Delimiters for empty parts to the right of the last desired part may be omitted. If two successive delimiters are given, the corresponding part is set to an empty (zero-length) string. If the title consists only of two or more occurrences of the delimiter, then all parts are empty and the affected line becomes a white space line.

The parts may contain symbolic references to program built-in and user variables; however, in basic formatting, usually only the page number is wanted. The references are retained in symbolic form and substitution is done with the current values of the variables when the title line is inserted into the document.

Undention

Reverses the sense of direction of indention and is effective for only the next output line. See the .un, .unl, .unh, .unr, and .unb controls.

Variables, Built-in

The Formatter supports an extensive set of built-in variables that allow the user to retrieve and use the values of various formatting parameters, device description parameters, formatting mode switches, and command line values. See "Built-in Variables" described below.

Variables, Substitution of

During expression evaluation, all symbolic references are replaced with the current value of the referenced variable. String values are used as they exist. Numeric values are converted to strings according to the current display mode for the referenced variable. See the .srm and .ur controls.

Variables, User

User variables may be defined and their values assigned or changed whenever desired. The names of variables are constructed with alphanumeric characters and underscores (_) with a maximum length of 32 characters. The first character of a variable name must be alphabetic.

Variables may be assigned numeric values or string values. Given variable values are evaluated before they are assigned (see "Expression Evaluation" above).

If a variable value is given as a quoted string, the evaluated string value is assigned to the variable. The maximum string length is 1092 characters.

If a variable value is *not* given as a quoted string, it may contain only numeric digits and a decimal point (.) with a possible leading sign character. The evaluated numeric value is assigned to the variable. The maximum absolute value of a numeric variable is 2,097,151 and it may have up to three decimal places. See the .srv and .src controls.

Note: When %[]% and %{}% contain variables, the user must double the %s. For example:

```
.ur .if %%[active_fnc_name %Variable% XX]%%

.ur .ur xxxx %%{%variable%-1}%% xxxx
```

White Space (Extra Lead)

Any arbitrary amount of white space may be added to improve readability, either as an independent text block or within a text block. See the .spb and .spf controls.

In some instances, the user may want to advance to some specific page depth (such as the extra space usually found at the top of the first page of a chapter or section), or to ensure that some amount of "protected" (nontrimmable) white space appears at some place on the page, or to ensure than no less than some given amount appears without having sure knowledge of how much has already been given (this case occurs frequently in "macro" formatting packages). See the .spd and .spt controls.

BUILT-IN VARIABLES

Descriptions of the built-in variables of the Formatter are described here. Each description begins with a title line of the form:

Name :: type :: default value :: controls

where:

Name — the name of the variable as it is used in symbolic references.

type — the storage type of the variable. The possible values are numeric, counter, string, or logical.

default value — the initial value of the variable and the value assigned if no value is given with a control or control argument that affects its value.

controls — any controls and control arguments that affect the value.

The double colon (::) has no meaning other than a field separator in the title line.

AlignMode :: string :: "both" :: .alb .alc .ali .all .alo .alr
the current text alignment mode. The possible values are "both", "left", "right", "inside", "outside", and "center".

ArgCount :: numeric :: 0 :: .ifi .. .rt
the number of arguments passed to an inserted file.

Arg*i* :: string :: "" :: .ifi .. .rt
the *i*th argument passed to an inserted file. *i* may have the values 1, 2, ... ArgCount.

ArtMode :: logical :: 0 :: .bba .bea
true when in an artwork subblock; otherwise, false.

CallingFileName :: string :: "" :: .ifi .. .rt
the "current reference name" (no suffix) of the caller of an inserted file.

CallingLineNo :: numeric :: 0 :: .ifi .. .rt
the current input line number in the caller of an inserted file.

CommandArgCount :: numeric :: 0 :: -arguments
the number of values given with the -arguments control argument in the command line.

CommandArg*i* :: string :: "" :: -arguments
the *i*th argument given with the -arguments control argument. *i* may have the columns 1,2,...CommandArgCount.

Date :: string :: date() :: none
the current date in the form mm/dd/yy (month/day/year).

Device :: string :: "ascii" :: -device
the "current reference name" of the output device as given with the -device control argument in the command line. This "current reference name" usually implies some specific configuration or operating mode. There must be a device description table with this "current reference name".

DeviceClass :: string :: "typewriter" :: -device
the class of the output device as given in the device description table. The class of a device generally implies the technology used in its design and typical values are "typewriter", "printer", and "typesetter".

DeviceName :: string :: "ascii" :: -device
the generic "current reference name" of the output device as given in the device description table. The generic "current reference name" of a device usually implies some set of major features or the existence of multiple device description tables for different configurations and typical values are "ascii", "diablo", "Dymo", and "VIP".

Eqcnt :: counter :: 1 :: .eqc
the equation reference counter.

ExtraMargin :: numeric :: 0 :: -indent
the amount of extra left margin space to be added.

FileName :: string :: entry in input file list :: none
the "current reference name" of the current command line input file.

FillMode :: logical :: -1 :: .fi .fif .fin -nofill
true when text filling is enabled; otherwise, false.

FontName :: string :: device dependent :: .fnt -device
the name of the current font. The initial value for the variable is the initial font as given in the device description table.

Footcnt :: counter :: 1 :: .bef .ftp .ftu
the footnote counter.

FootnoteMode :: logical :: 0 :: .bbf .bef
true when processing a footnote; otherwise, false.

FootReset :: string :: "paged" :: .ftp .ftu
the footnote numbering mode. Its value is "paged" when footnotes are being inserted and numbered for each page, and "unref" when unreferenced footnotes are being used.

From :: numeric :: 1 :: -from
the number of the first output page to print as given with the -from control argument.

FrontPage :: logical :: -1 :: .brn .brp .brs
true when the current page is a front or facing page, that is, the left edge is the binding edge; otherwise, false.

Galley :: logical :: 0 :: -galley .gl
true when the -galley control argument or the .gl control are given; otherwise, false.

HeadSpace :: numeric :: 0 :: .spb .spd .spt .tcl .ttl .vmh .vmt
the amount of white space on the page immediately preceding the current page depth given in terms of 10-pitch lines.

Hyphenating :: logical :: 0 :: .hy .hyn .hyf -hyphenate
true when hyphenation is enabled; otherwise, false.

Indent :: numeric :: 0 :: .inb .inl
the current value of left margin indention given in terms of 10-pitch characters.

IndentRight :: numeric :: 0 :: .inb .inr
the current value of right margin indention given in terms of 10-pitch characters.

InputDirName :: string :: "" :: .ifi .. .rt
the "current reference pathname" of the directory containing InputFileName.

InputFileName :: string :: "" :: .ifi .. .rt
the "current reference name" of the current input file.

InputLineNo :: numeric :: 0 :: none
the current line number in InputFileName.

KeepMode :: logical :: 0 :: .bbk .bek
true when processing a keep block; otherwise, false.

LineInput :: string :: "" :: none
the contents of the next line in the caller of an inserted file, that is, the line whose input line number is one greater than the value of CallingLineNo. Use of this built-in also advances the value of CallingLineNo.

LinesLeft :: numeric :: 54 :: .bbf .bef .bpf .efp .pfl .pd .pdl .vm .vmf .vmb
the amount of space left on the page available for text, given in terms of 10-pitch lines.

LineSpace :: numeric :: 1 :: .ls -linespace
the current linespacing value given in terms of 10-pitch lines.

Measure(*string*) :: numeric :: 0 :: none
the visual width of the *string* as measured in the current font and pointsize, and given in terms of 10-pitch characters.

NextPageNo :: string :: "2" :: .brn .brp
the page number of the next page to be printed.

OutputFileOpt :: logical :: 0 :: -output_file
true when the -output_file control argument is given; otherwise, false.

PageLength :: numeric :: 66 :: .pd .pdl -device
the current page length given in terms of 10-pitch lines. The default may be less than the value shown here due to device restrictions in the device description table.

PageLine :: numeric :: 0 :: none
the depth of the current output line on the page given in terms of 10-pitch lines.

PageNo :: string :: "1" :: .brn .brp
the page number of the current page.

PageWidth :: numeric :: 65 :: .pd .pdw -device
the current page width given in terms of 10-pitch characters. The default may be less than the value shown here due to device restrictions in the device description table.

Parameter :: string :: "" :: .ifi .. -parameter
the entire list of arguments passed to an inserted file given as a single string, or (before any file is inserted) the value given with the -parameter control argument.

ParamPresent :: logical :: 0 :: .ifi ..
true if any arguments are passed to an inserted file; otherwise, false.

Pass :: numeric :: 1 :: -passes
the number of processing passes remaining to be performed (including the current pass). Output is produced only when the value is 1.

PointSize :: numeric :: 7.2 :: .fnt -device
the current type size given in typographic points (72 points = 1 inch). The default value is the size of 10-pitch characters.

Print :: logical :: -1 :: -galley -from -to -pages -passes
true when the current line is to be printed; otherwise, false.

StopOpt :: logical :: 0 :: -stop
true when the -stop control argument is given; otherwise, false.

SymbolDelimiter :: string :: "%" :: .csd
the character being used to delimit symbolic references.

TableMode :: logical :: 0 :: .tan .taf
true when in table mode; otherwise, false.

Time :: string :: time() :: none
the time of day at command invocation in the form hhmm.m (hours minutes tenths of minutes).

TitleDelimiter :: string :: "|" :: .ctd
the character being used to delimit title parts.

To :: numeric :: -1 :: -to
the page number of the last page to be printed as given with the -to control argument. The default value implies the end of the input file.

Undent :: numeric :: 0 :: .unl .unh
the value of any pending left undention.

UndentRight :: numeric :: 0 :: .unr
the value of any pending right undention.

UserInput :: string :: "" :: none
one line as typed in by the user on the user_input I/O switch.

VMargBottom :: numeric : 4 : .vm .vmb
the bottom margin, that is, the amount of white space between the page footer and the bottom of the page given in terms of 10-pitch lines.

VMargFooter :: numeric : 2 : .vm .vmf
the footer margin, that is, the amount of white space between the last text line and the page footer given in terms of 10-pitch lines.

VMargHeader :: numeric : 2 : .vm .vmh
the header margin, that is, the amount of white space between the page header and the first text line given in terms of 10-pitch lines.

VMargTop :: numeric : 4 : .vm .vmt
the top margin, that is, the amount of white space between the top of the page and the page header given in terms of 10-pitch lines.

WaitOpt :: logical :: 0 : -wait
true when the -wait control argument is given; otherwise, false.

Widow :: numeric :: 2 : .wit
the current text widow size given in terms of 10-pitch lines.

CREATING ARTWORK

The artwork feature of the Formatter permits the user to insert certain conventional overstruck character patterns into an input file and have them displayed as fabricated symbols and line art diagrams. The detection and expansion of the artwork constructs described here is controlled with the block-begin-art and block-end-art formatting controls described below.

Note: Overstriking of characters is inappropriate for video terminals.

For the purposes of discussion here, artwork constructs are shown with dollar signs representing the backspace character as in "X$X".

The replacement strings for the artwork symbols are contained in the device description table for the output device (see Section 4) and are the closest approximation to the desired output symbol possible for the device.

The occurrence of an artwork construct in an input line implies an output symbol (or symbol fragment) with some definite position, width, and height. The values are either given in the device description table or are part of the input artwork construct and are used by the Formatter when placing output symbols and text on the page.

In this section, the word "rule" refers to a typographic rule, that is, a line of given length, thickness, and orientation.

Artwork Conventions

Two subsets of the 95-character ASCII graphic set are defined: the "line art" set and the "math symbol" set. Members of the line art set are syntactically significant if they are overstruck with another character from the set. Members of the math symbol set are syntactically significant if they are overstruck with a valid size character (see the discussion of size characters following). The characters in a construct may be given in any order.

Line art	Math Symbol	Meaning
-		element of a horizontal rule
|	|	element of a vertical rule or a vertical bar, depending on overstrike pattern
/	/	element of a +45 degree slant rule or a division sign, depending on overstrike pattern
\		element of a -45 degree slant rule
(	(	left semicircle or left parenthesis, depending on overstrike pattern
)	)	right semicircle or right parenthesis, depending on overstrike pattern
^	^	up arrow, diamond top vertex, or upward movement, depending on overstrike pattern
v	v	down arrow, diamond bottom vertex, or downward movement, depending on overstrike pattern
<	<	left arrow, diamond left vertex, or leftward movement, depending on overstrike pattern
>	>	right arrow, diamond right vertex, or rightward movement, depending on overstrike pattern
	[	left bracket
	]	right bracket
	{	left brace
	}	right brace
	X	multiplication sign (one-high math symbol only)

~		vertical or slant rule terminator
*	*	horizontal rule terminator or text deletion symbol, depending on overstrike pattern
" '		replicator character showing overstrike but having no pictorial meaning
H h		half-line control, up or down, depending on overstrike pattern
S s		superscript/subscript control, depending on overstrike pattern
	=	double vertical bar concatenate symbol
	c	text modification change bar (one-high math symbol only)
	o	bullet (one-high math symbol only)

If any of the characters in the math symbol set is overstruck with a numeric or alphabetic character it is considered a math symbol and the overstrike character is interpreted as the symbol size as follows:

1 - 0	1 through 10
a - z	11 through 36
A - Z	31 through 56

Examination of the two subsets reveals that there are ambiguous cases (the most obvious of which is "v") that may be either line art, a math symbol, or a size character depending on usage. These ambiguities may be resolved in favor of line art by adding a replicator character (" or ') to the input artwork construct.

The four movement constructs perform "micropositioning" and the size character represent the count of increment to be moved. The amount of space for an increment is given in the device description table.

Artwork Syntax

The syntax for artwork construction is as follows:

1. Text filling should be off to preserve element position in an artwork diagram. This does not prevent the use of artwork in filled text, however.
2. Line art may be contained within math symbols and vice versa.
3. All rules continue through intersections unless they are specifically terminated by an appropriate terminator character.
4. Unterminated horizontal rules generate a reported syntax error at the right margin.
5. Unterminated slant rules generate a reported syntax error at either margin or continue to the end of the artwork block.
6. Unterminated vertical rules continue to the end of the artwork block.
7. Positioning of plain text is the responsibility of the user. The movement constructs are provided for this purpose.

Artwork Constructs

Boxes:

Boxes are created by defining their corners with the constructs below. The corners appear on the page at the position of the first character of the construct.

-$|	upper left corner
*$|	upper right corner
-$~	lower left corner
*$~	lower right corner

Artwork constructs are shown with dollar signs representing the backspace character.

Superscripts and Subscripts:

Superscripts and subscripts are created by placing the letter "s" (either uppercase or lowercase) and one of the two vertical motion characters at the point where the baseline change is to occur. Alternatively, "S" for superscript or "s" for subscript and a replicator character may be used. When using superscripts and subscripts, you must remember to return to the normal text baseline. For example, the chemical symbol for water could be given as:

```
Hs$v2^$s0
```

The baseline offset for superscripts and subscripts is one third of the current linespace value but the feature is effective only for those devices capable of fractional linespacing. The superscript and subscript constructs are:

S$^	superscript
s$^	superscript
S$"	superscript
S$'	superscript
S$v	subscript
s$v	subscript
s$"	subscript
s$'	subscript

Half-Lines:

Half-lines are created by placing the letter "h" (either uppercase or lowercase) and one of the two vertical motion characters at the point where the baseline change is to occur. Alternatively, "H" for half-line up or "h" for half-line down and a replicator character may be used. When using half lines, you must remember to return to the normal text baseline.

The baseline offset for half-lines is one half of the current linespace value but the feature is effective only for those devices capable of fractional linespacing. The half-line constructs are:

H$^	half-line up
h$^	half-line up
H$"	half-line up
H$'	half-line up
H$v	half-line down
h$v	half-line down
h$"	half-line down
h$'	half-line down

Math Symbols:

Multi-line math symbols are created by "stacking" the appropriate math symbol construct in the same input column for the desired number of lines. The constructs are given as one of the math symbol selector characters below and the alphanumeric character representing the desired size. The maximum symbol size is 56. The size character and the number of input lines given should be the same.

Note: It is critical that text filling be off and that alignment be flus left for these constructs so that the positioning of the symbol elements are maintained.

The math symbol selector characters are:

[	opening bracket
]	closing bracket
{	opening brace
}	closing brace
(	opening parenthesis
)	closing parenthesis
|	single line (Boolean OR)
=	double line (concatenate)

In addition, the following symbols are available as "one-high" only (these symbols may also be used in filled text):

o$1	bullet
/$1	division sign
X$1	multiplication sign
$1	asterisk
c$1	heavy vertical line (change bar mark)

Diamonds:

Diamonds are created by defining their vertices with the constructs below. The vertices appear on the page at the position of the first character of the construct. The left and right vertices must be in the mid line of the diamond and the top and bottom vertices must be in the center column of the diamond.

Diamonds must always be an *odd* number of lines high and an *odd* number of columns wide. The number of lines and columns must be the same. If only the left and right vertices are given, the resulting diamond is one line high. If only the top and bottom vertices are given, the resulting diamond is a half-line high and is centered between the two lines giving the vertices (if the output device is capable of fractional linespacing; otherwise, the vertices are separated by the value of linespacing). Symbols for diamond vertices may *not* appear in column 1. The diamond vertex constructs are:

^$"	top vertex
v$"	bottom vertex
<$"	left vertex
>$"	right vertex

Lozenges:

Lozenges (flattened diamonds) are created by defining their corners and vertices with the constructs below. The corners and vertices appear on the page at the position of the first character of the construct. The left and right vertices

must be in the mid line of the lozenge. The left pair of corners must be in the same column and the right pair of corners must be in the same column.

Lozenges must always be an *odd* number of lines high but may be any number of columns wide. The minimum lozenge height is three lines. If all four corners are given in the same column, the top and bottom lines of the resulting diamond are one column wide. Symbols for lozenges may *not* appear in column 1. The lozenge constructs are:

-$/	upper left corner
*$\	upper right corner
-$~	lower left corner
*$~	lower right corner
\$~$/	left or right vertex

Rules:

Typographical rules are created by giving their starting and ending points with the constructs below. Rules are started by giving a rule selector character and any other line art character in an artwork construct and are ended by giving the appropriate terminator character with any other line art character in an artwork construct.

Unterminated rules that attempt to cross either page margin result in error messages. Unterminated rules that reach the end of the artwork block are terminated gracefully. The rule selector and terminator characters are:

-	horizontal rule
|	vertical rule
\	left slant rule
/	right slant rule
*	horizontal terminator
~	vertical and slant terminator

Circles and Rounded Boxes:

A circle may be created by giving a left and a right parenthesis, each with a replicator character, in artwork constructs separated by exactly one column. The separating column may be used for a single character. The resulting circle is three lines high; no other circle size is provided.

A rounded box may be created by opening up the space between the parenthesis constructs and giving horizontal rules in the line before and the line after the one containing the parentheses. The extra space may be used for any text.

The circle constructs are:

($"	left semicircle
)$"	right semicircle

FORMATTING CONTROLS

Each of the formatting controls will be described in detail. The presentation is in alphabetical order of the control tokens. An index style *summary* is included immediately following these descriptions.

Each description begins with a line showing the control token and its variable field, ending with the optional input line delimiter (;), then gives the name of the control

and which break type it implies, if any. When a break is indicated, the break is executed before the action of the control.

Note: The use of the semicolon in these descriptions does not imply that its use in actual practice is required. It is used here merely for clarity of presentation. The NL character (ASCII code 012) is the usual input line delimiter.

Formatting controls need not be entered individually in a control line of an input file. They may be entered as:

```
.pdl 66
.pdw 71
.vm 3,1,1,3
.sr fw "325"
```

or as:

```
.pdl 66:.pdw 71;.vm 3,1,1,3;.sr fw "325"
```

using the semicolon (;) as the control delimiter and the newline (NL) as the input line delimiter. One exception to this type of construction (i.e., multiple formatting controls on a single entry line) is with the controls: .if, .then, .else, and .elseif. These unique controls accept either a semicolon or space character as the delimiter. For example:

```
.if<conditional>
.then;.srv<whatever>
          .
          .
          .
```

or,

```
.if<conditional>
.then<SP>.srv<whatever>
          .
          .
          .
```

Any parameters in the variable fields shown in braces ({}) are optional and their default values are used if they are not given. The left slash (\) is used to show that exactly one of the values so separated may be given. An ellipsis (...) indicates continuation of a parameter string to the extent given in the explanation.

.* *any text*; comment, no break
a comment line having no effect on any output.

.+*string*; continue, no break
string is appended directly onto the previous input line. That is, the two lines are processed together as a single line as though there had been no intervening input line delimiter character. Note that, in this single exceptional control, no SP character following the control token is needed. If one is given, it becomes part of the input line.

.. *path* {*arg1, arg2, ...*}; insert-file, no break
suspend processing of the current input file and begin reading input from the given file. (This form is supported as being less "programmatic" than the ".ifi" form.)

If any arguments are given, the built-in variable "ParamPresent" is set to "-1" and the entire argument list is copied into the built-in variable "Parameter", destroying any existing value. If no arguments are given, "ParamPresent" is set to "0" and the contents of "Parameter" are not changed. (See "Built-in Variables" above.)

The individual arguments (if any) are copied into the built-in variables "Arg1", "Arg2", ... and the number of arguments given is assigned to the built-in variable "ArgCount". Any existing values of these built-in variables are saved in a push-down/pop-up stack and are restored when processing returns to the suspended input file. If any of the arguments contain blank space, they must be given as quoted strings.

The depth to which files may be inserted is not limited.

.alb; align-both, format break
align the text at both the left and right margins as adjusted by the indention values. Filling must be enabled for this alignment to operate. If filling is disabled, .alb has the effect of .all. This is the default alignment mode.

.alc; align-center, format break
center the text between the left and right margins as adjusted by the indention values. Filling has no effect on this alignment.

.ali; align-inside, format break
align the text at the inside margin (binding edge) as adjusted by the appropriate indention value. Filling has no effect on this alignment.

.all; align-left, format break
align the text on the left margin as adjusted by the left indention value. Filling has no effect on this alignment.

.alo; align-outside, format break
align the text at the outside margin (away from binding edge) as adjusted by the appropriate indention value. Filling has no effect on this alignment.

.alr; align-right, format break
align the text at the right margin as adjusted by the right indention value. Filling has no effect on this alignment.

.bba {#}; block_begin_art, no break
begin flagging output text lines as artwork lines to be processed by the artwork expander. The parameter is given as the number of input lines following whose generated text should be flagged in the output. (See "Creating Artwork" above.) If the parameter is not given, then the flagging of output continues until the occurrence of a .bea control and may span blocks, columns, and pages.

.bbf {u} {.c\p}; block-begin-footnote, no break
suspend processing of the current text block and begin processing a footnote.

The first parameter may only be "u" to indicate that the footnote is to be unreferenced. If the footnote is to be referenced, then a footnote reference string is constructed according to the current value of the footnote counter and the style and procedure for the output device (see Appendix C) and is placed as a hanging undent on the first line of the footnote.

If the second parameter is not given, then the footnote formatting parameters are carried forward from the previous footnote, or set from the default footnote formatting parameters if this is the first footnote. If the second parameter is given, then the footnote formatting parameters are set according to its value. Any formatting parameters changed while processing the footnote are carried forward to subsequent footnotes but do not affect main body text.

If the second parameter is given and the first is omitted, the separating comma (,) must still be given. The allowed parameter values are:

c — format the footnote according to the current column formatting parameters. This is the default when the Formatter is in multicolumn or table modes.

p — format the footnote according to the current page formatting parameters. This is the default when the Formatter is in page mode.

.bbk {#}; block-begin-keep, format break

finish the current line and begin flagging output text lines as exempt from being split across column or page boundaries. The parameter is given as the number of following input lines whose generated text should be flagged in the output. If the parameter is not given, then the flagging of output continues until the occurrence of a .bek control. When this flagging is active, all block, column, and page breaks are inhibited. If the size of a keep block exceeds the maximum page space available it is treated as a normal text block and split across pages.

.bbl {#}; block-begin-literal, no break

process input lines as text lines even though they may appear to be control lines or contain apparent escape sequences. The parameter is given as the number of following input lines to process. If it is not given, then continue until the occurrence of a .bel control.

.bbp {#}; block-begin-picture, no break

if # is given, then define an unbreakable picture block of exactly # lines of vertical white space: the parameter is given as an unsigned number. If # is not given, then accumulate output lines into an unbreakable picture block until the occurrence of a block-end-all or block-end-picture control. Text headings and/or captions given while in picture mode (# not given) pertain to the picture and not to a possible containing text block. A picture block is vertical white space or a formatted block that is inserted on a space-available basis. If, at the completion of a picture block, sufficient space remains on the current page, it is inserted immediately. If the picture block does not fit on the current page, inline text is inserted from behind to ahead of the picture and the picture is placed at the top of the next page. If the size of a picture block exceeds the maximum text space available on a page as determined by the vertical margins and any headers and footers, an error diagnostic message is produced and the block is broken into full and partial pages. Multiple picture blocks are queued, not merged into a single block. Up to ten picture blocks may be queued. Queued picture blocks are inserted in the order in which they were defined.

.bbt {#}; block-begin-title, format break

begin accepting three-part title lines as input. The parameter is given as the number of input lines to accept. If it is not given, the title lines are accepted until the occurrence of a .bet control.

.bcf {*n*}; begin-column-footer, no break
cancel the column footer block and begin a new formatted column footer block. The formatting of the block proceeds as for any inline text block except that none of the "special block" features (e.g., page header lines, footnotes) may be used and title lines are allowed. The variable field parameter is:

n the initial indention value for this footer block.

If it is omitted or given as "0", then the block is aligned at the left column margin. If it is given as any other value, then the block is aligned at the given position in the column.

.bch {*n*}; begin-column-header, no break
cancel the column header block and begin a new formatted column header block. The formatting of the block proceeds as for any inline text block except that none of the "special block" features (e.g., page footer lines, footnotes) may be used and title lines are allowed. The variable field parameter is:

n the initial indention value for this header block.

If it is omitted or given as "0", then the block is aligned at the left column margin. If it is given as any other value, then the block is aligned at the given positiin in the column.

.bea; block-end-art, no break
stop flagging output lines for artwork conversion.

.bef; block-end-footnote, no break
if the Formatter is in footnote mode, then save the current formatting parameters for use in the next footnote, then finish the footnote and resume processing the suspended text block (if any), leaving footnote mode.

If there is no suspended block, then attach the footnote to the page header as an "orphan;" otherwise, attach the footnote to the current output line of the suspended block.

If the footnote is referenced, then insert the footnote reference string (see the .bbf control above) into the output line, and advance the footnote reference counter.

.bek; block-end-keep, no break
stop flagging output lines as exempt from block splitting and reactivate the block, column, and page breaks.

.bel; block-end-literal, no break
stop ignoring control lines in the input.

.bep; block-end-picture, no parameters, no break, no substitution
Stop accumulating output lines into the current picture block and revert to inline block processing. If the picture will fit in the space remaining on the current page, then insert it immediately; otherwise, queue the picture block for insertion on a space-available basis. If the picture mode is not in effect, ignore the control.

.bet; block-end-title, no break
stop accepting title lines as input.

.bpf {*n*} {e\o\a}; begin-page-footer, no break
cancel the page footer block of the type specified by the second parameter and begin a new formatted page footer block of the same type. The formatting of

the block proceeds as for any inline text block except that none of the "special block" features (e.g., page header lines, footnotes) may be used and title lines are allowed. The variable field parameters are:

n — the initial indention value for this footer block.

If it is omitted or given as "0", then the block is aligned at the left page margin. If it is given as any other value, then the block is aligned at the given position on the page.

e\o\a — the footer block to be created.

The meaning of the three allowed parameter values are:

e even page footer block only
o odd page foot block only
a all page footer blocks (Default)

.bph {*n*} {e\o\a}; begin-page-header, no break

cancel the page header block of the type specified by the second parameter and begin a new formatted page header block of the same type. The formatting of the block proceeds as for any inline text block except that none of the "special block" features (e.g., page footer lines, foot) may be used and title lines are allowed. The variable field parameters are:

n — the initial indention value for this header block.

If it is omitted or given as "0", then the block is aligned at the left page margin. If it is given as any other value, then the block is aligned at the given position on the page.

e\o\a — the header block to be created.

The meaning of the three allowed parameter values are:

e even page header block only
o odd page head block only
a all page header blocks (Default)

.brb; break-block, block break (see Note)

finish the current output line as for the .brf control following and then finish processing the current block as appropriate for the block type.

Note: For certain special blocks and formatting modes (e.g., page headers, and keeps) that require some other control to signal completion, this control causes only a format break, that is, completion of the block is inhibited until the expected control is encountered.

.brc {#}; break-column, column break

finish the current block (if any), fill the remainder of the current column with white space, and then change to the column indicated by the parameter. The parameter is given as an unsigned number.

If the parameter is not given, then advance to the next column on the current page or, if the next column is not defined, finish the current page and begin with the first column on the next page. The appearance of this control on a page causes the page to be formatted as an unbalanced page.

If the parameter is given as zero, finish the current block (if any), balance *all* the columns at the current page depth, and revert to the formatting parameters of the full page until the occurrence of a .brc control that selects a defined column.

If the parameter is given any nonzero value, finish the current block (if any), fill the remainder of the current column with white space, and then change to the given column, filling all intervening columns with white space. If the given column is less than the current, then the current page is also finished (without balancing) and processing begins in the given column on the next page. It is an error to give a column number that is not defined for the page.

It is an error to give this control when not running multicolumn mode.

.brf; break-format, format break
finish the current output line by formatting any pending text as a short line.

.brn {#}; break-need, no break (see Note)
if the parameter is greater than the amount of text space available, advance immediately to the next page or column, as appropriate. The default value is 1.

Note: This control performs all the actions of the page or column breaks except that the current output line is not finished, that is, any pending text is carried forward to the next page or column. Therefore, it is considered as not causing a break.

.brp {e\o*n*}; break-page, page break (see Note)
(basic)
finish the current page and set the page counter according to the parameter given. "Finishing" the page involves finishing the current block (and possibly balancing the columns, if in multicolumn mode) and ejecting the page, inserting any footnotes and/or the page footer. If the parameter is omitted, the page counter is advanced by 1. If the page is empty when this control is given, it is not ejected and the automatic page counter advance does not take place, however the page counter is still set from a given parameter. (Also refer to the "intermediate description" of the .brp control below.)

Note: For certain special blocks and formatting modes (e.g., page headers and keeps) that require some other control to signal completion,this control is ignored.

The allowed values of the parameter are:

e	set the page counter to the next even value.
o	set the page counter to the next odd value.
n	new value for the page counter, given as an integer. If given as an unsigned number, set the page counter to the value given. If given as a signed number, change the page counter by the amount given.

.brp {e\o*page-number*} {*mode-string*}; break-page, page break (see Note)
(intermediate)
finish the current page and set the page number structure according to the parameters given. "Finishing" the page involves finishing the current block (and possibly balancing the columns, if in multicolumn mode) and ejecting the page, inserting any footnotes and/or the page footer. If no parameters are given, the last page number counter in the existing structure is advanced by 1. If the page is empty when this control is given, it is not ejected and the automatic page counter advance does not take place, however the page number structure is still set from any given parameters. (Also see the "basic description" of the .brp control above.)

Note: For certain special blocks and formatting modes (e.g., page headers and keeps) that require some other control to signal completion, this control is ignored.

The variable field parameters are:

e — set the last page number counter to the next even value.

o — set the last page number counter to the next odd value.

page-number — if given as a signed number, change the last page number counter by the amount given.

Otherwise, a new value for the page number structure is given as a character string consisting of counter values and separators. Counter values are given as they are to appear in the output (with ambiguities resolved by *mode-string* below). Separators may be chosen from ".", "-", "(", ")", and "|" where "|" represents a null separator. For example, the page number A2 would be given as "A|2".

mode-string — a string made up of keywords and commas where the keywords are in one-to-one correspondence with the counter values in the given *page-number* and the commas are in one-to-one correspondence with the separators. Any of the keywords may be omitted or chosen from the list below. All commas except those following the last desired keyword must be given.

Key	Value displayed as
ar	Arabic
bi	binary
hx	hexadecimal
oc	octal
al	lowercase alphabetic
au	uppercase alphabetic
rl	lowercase Roman
ru	uppercase Roman

A keyword given for a counter value overrides the apparent value given in the *page-number*. For example:

Page	.brp parameters
i	"i rl" or "1 rl"
I-i	"I-i au" or "9-1 au,rl"

(Also see the .srm control.)

.brs {#} {"*text*"} {"*header*"} {"*footer*"}; break-skip, page break

finish the current page, and then create blank pages according to the given parameters. The pages created are not assigned page numbers and the page counter is not advanced. Any parameters given apply only to the blank pages created by the current use on the control; they do not carry forward. Note that all parameters must be given as quoted strings regardless of whether they contain blanks or not. The variable field parameters are:

\# — the number of blank pages desired. The default value is 1.

text — a line of text to appear centered on the blank page(s) created.

header — a single page header line for the blank pages created. It must be given as a title line. If a header is wanted, then a text line

must also be given, but the text may be given as a blank line ("").

footer a single page footer line for the blank pages created. It must be given as a title line. If a footer is wanted, then a header must also be given, but the header may be given as a null header ("").

.btc {*n*}; begin-text-caption, no break
suspend processing of the current block (if any) and begin processing a formatted text caption block. The formatting of the block proceeds as for any inline text block except that none of the "special block" features (e.g., page footer lines and footnotes) may be used and that title lines are allowed.

If a pending text caption already exists, then add the output generated to it; otherwise, use the output generated as the caption.

The variable field parameter is:

n the initial indention value for this caption.

If it is omitted, then the caption is aligned according to the text left indention.

If it is given as an unsigned number, then the caption is aligned at the value given relative to current column or page left margin.

If it is given as a signed number, then it is used as a local adjustment to the text left indention.

.btt {*n*}; begin-text-title, no break
suspend processing of the current block (if any) and begin processing a formatted text title block. The formatting of the block proceeds as for any inline text block except that none of the "special block" features (e.g., page footer lines and footnotes) may be used and that title lines are allowed.

If a pending text title already exists, then add the output generated to it; otherwise, use the output generated as the title.

The variable field parameters are:

n the initial indention value for this title.

If it is omitted, then the title is aligned according the text left indention.

If it is given as an unsigned number, then the title is aligned at the value given relative to current column or page left margin.

If it is given as a signed number, then it is used as a local adjustment to the text left indention.

.cba {*c*}; change-bar-add, no break
if change bars are enabled by giving the -change_bars or -change_bars_art control argument in the compose command line, and the parameter is less than or equal to the change level character (in the ASCII collating sequence sense; 0,1,...9,A,B,...Z,a,b,...z) given with the control argument, then flag output lines

containing the text *and white space* following with text modification marks. If neither control argument is given, then ignore the control.

The parameter is given as a single character and is the change level (Revision number or Addendum letter) for the text change. The default character is a blank (ASCII SP).

By default, the mark is a vertical line (|) and it is placed in the outside page margin, separated from the text by one column. The mark and placement may be changed with parameters given with the control arguments.

.cbd {*c*}; change-bar-delete, no break
if change bars are enabled by giving the -change_bars or -change_bars_art control argument in the compose command line, and the parameter is less than or equal to the change level character (in the ASCII collating sequence sense; 0,1,...9,A,B,...Z,a,b,...z) given with the control argument, then flag the next output line only with a text deletion mark. If neither control argument is given, then ignore the control.

The parameter is given as a single character and is the change level (Revision number or Addendum letter) for the text change. The default character is a blank (ASCII SP).

By default, the mark is an asterisk (*) and it is placed in the outside page margin, separated from the text by one column. The mark and placement may be changed with parameters given with the control arguments.

.cbf {*c*}; change-bar-off, no break
if change bars are activated by a preceding .cba or .cbm control and the parameter is equal to the currently active change level, then stop flagging output lines with text modification marks. If change bars are not active for the given change level then ignore the control.

The parameter is given as a single character and is the change level (Revision number or Addendum letter) for the text change. The default character is a blank (ASCII SP).

.cbm {*c*}; change-bar-modify, no break
if change bars are enabled by giving the -change_bars or -change_bars_art control argument in the compose command line, and the parameter is less than or equal to the change level character (in the ASCII collating sequence sense; 0,1,...9,A,B,...Z,a,b,...z) given with the control argument, then flag output lines containing the text *only* following with text modification marks. If neither control argument is given, then ignore the control.

The parameter is given as a single character and is the change level (Revision number or Addendum letter) for the text change. The default character is a blank (ASCII SP).

By default, the mark is a vertical line (|) and it is placed in the outside page margin, separated from the text by one column. The mark and placement may be changed with parameters given with the control arguments.

.cfl {#} {*n*} {*title*}; column-footer-line, no-break
define column footer lines according to the values given in the variable field. If no parameters are given, then all column footers are cancelled. The variable field parameters are:

the index value for the line.

If it is omitted or given as "0", then the current column footer block is cancelled and this line becomes line 1 of a new column footer block.

It it is less than or equal to the highest index number in the block, then title replaces that line in the block. If no title is given, then the line is replaced with a null line.

If it is greater than the highest index line number in the block, then title becomes the indexed footer line with the given number in the block and any intervening indexed lines become null lines. If no title is given, then the control is ignored since all lines involved would be null lines.

n the indention value for this footer line. Note the *n* may not be given unless *#* is also given since they both appear as simple, unsigned numbers.

If it is omitted or given as "0", then title is aligned at the left page margin. If it is given as any other value, then title is aligned at the given position on the page.

title the three-part title used as the footer line. Any references to symbolic variables in the title are evaluated when the line is placed on the page.

.chl {#} {*n*} {*title*]; column-header-line, no-break

define column header lines according to the values given in the variable field. If no parameters are given, then all column headers are cancelled. The variable field parameters are:

the index value for the line.

If it is omitted or given as "0", then the current column header block is cancelled and this line becomes line 1 of a new column header block.

It it is less than or equal to the highest index number in the block, then title replaces that line in the block. If no title is given, then the line is replaced with a null line.

If it is greater than the highest index line number in the block, then title becomes the indexed header line with the given number in the block and any intervening indexed lines become null lines. If no title is given, then the control is ignored since all lines involved would be null lines.

n the indention value for this header line. Note the *n* may not be given unless *#* is also given since they both appear as simple, unsigned numbers.

If it is omitted or given as "0", then title is aligned at the left column margin. If it is given as any other value, then title is aligned at the given position on the page.

title — the three-part title used as the header line. Any references to symbolic variables in the title are evaluated when the line is placed on the page.

.csd {*c*}; change-symbol-delimiter, no break
change the symbol delimiter to the given character. The default value for the parameter is percent (%).

.ctd {*c*}; change-title-delimiter, no break
change the title delimiter to the given character. The default value for the parameter is the vertical line (|).

.ecf; end-column-footer, no break
finish the column footer block begun with the preceding .bcf control and use the block as the appropriate column footer.

.ech; end-column-header, no break
finish the column header block begun with the preceding .bch control and use the block as the appropriate column header.

.else; conditional-else, no-break
begin the conditional execution group clause that is executed when the expr of the preceding .elseif or .if control is false. The end of the clause is marked by a .endif control and the eventual occurrence of that control is required. (See "Notes" under Formatting Controls above.)

.elseif *expr*; conditional-elseif, no break
begin the conditional execution group clause that is executed when the expr of the preceding .elseif or .if control is false, evaluate expr and proceed according to the result. The end of the clause is marked by a .elseif, .else, or a .endif control and the eventual occurrence of one of them is required. (See "Notes" under Formatting Controls above.)

If expr is given as a logical expression, then evaluate it as given. If it is given as a string expression, then evaluate it as though it were expr ^= "". If it is given as a numeric expression, then evaluate it as though it were expr ^= 0.

If the result of evaluation is true or expr is not given, then execute the clause; otherwise, skip the clause.

.endif; conditional-end, no break
marks the end of a conditional execution group.

.epf; end-page-footer, no break
finish the page footer block(s) begun with the preceding .bpf control and use the block as the appropriate page footers.

.eph; end-page-header, no break
finish the page header block(s) begun with the preceding .bph control and use the block as the appropriate page headers.

.eqc {*n*}; equation-count, no break
if the parameter is given, it must be an unsigned number and is the value to be assigned to the internal equation counter. If the parameter is not given, then the internal equation counter is advanced by 1.

.err *string*; error, no break
generate a Formatter error message using *string* as the text of the message but do not show this control line.

.etc; end-text-caption, no break
finish the caption block begun with the preceding .btc control.

.ett; end-text-title, no break
finish the title block begun with the preceding .btt control, resume the suspended block (if any).

If there is no suspended block, then use the title block as the title for the next text block.

If there is a suspended block and it is untitled, then prepend the title block onto the suspended block as a text title.

If there is a suspended block and it is already titled, then insert the title block between the existing text title and the first line in the suspended block.

.exc *string*; execute-command, no break
string is passed to the Multics command processor for execution as a command line.

.fi; fill-default, format break
enable or disable filling according to the default given for the invocation of the Formatter. See the -nofill control argument in the description of the compose command. If the -nofill control argument is not given, filling is enabled by default.

.fif; fill-off, format break
disable filling.

.fin; fill-on, format break
enable filling.

.fnt {*name*} {*/member*} {*size*}; font, no break
change to the given font.

If any parameters are given, then push the current font and pointsize onto a 20-element push-down/pop-up stack. If no parameters are given, then use the top element of the stack as the given font and "pop" the stack. It is an error to attempt to "pop" an empty stack. The variable field parameters are:

name the name (or alias) of the desired font. It must be supported by the output device and registered in the device description table. If no name is given, then the name of the current font is used. It is an error to give the name of an unsupported font.

/member the name (or alias) of the desired family member of the given font. It must be supported by the output device and registered in the device description table. If no name is given, then the current member name is used. It is an error to give a member name not in the given font family. (Refer to "Device Table Compiler" in Appendix C for a discussion of font naming.)

size the desired type size, given in typographic points. It must be within the range of sizes supported for the given font as registered in the device description table. If no size is given, then the current pointsize value is used. It is an error to give

a size value outside the allowed range; however, if the device supports only one size (as for an ASCII terminal or lineprinter), then this apparent error is ignored.

.frf {#}; footnote-reference, no break
prepare a footnote reference string (see the .bbf control above) for the #'th previous footnote according to the style and procedure for the output device (see Appendix C) and insert it into the current output line. The default value for the parameter is "1", that is, the immediately prior footnote. It is an error to give this control when not in a text block or to refer to a footnote that has already been placed on the current or some earlier page.

.fth; footnote-hold, no parameters, no break, no substitution
do not insert footnotes on the page of their reference, but hold them aside for insertion by the user with the insert-footnote control or, by default, at the end of the document. The footnote counter runs continuously until reset by another footnote control.

.ftp; footnote-paged; no break
format all following footnotes as "paged" and referenced.

.ftr{#}; footnote-running, no break, no substitution
same as paged in .fth above except runs continuously until reset.

.ftu; footnote-unreferenced, no break
format all following footnotes as unreferenced.

.gl; galley; no break
format the file in galley mode. This control must be the first control line read by the formatter; it is an error for it to appear elsewhere.

.go *label_name*; go-to, no break
reposition the input file to the first occurrence of a .la control defining *label_name* and continue processing with that line. *label_name* must be unique in the file for correct operation, however uniqueness is not required. It is an error to make reference to an undefined *label_name*. If *label_name* is undefined, processing continues with the input line following the .go control.

.hit {*n*} {*=ABC*} *KSTR* {*STR*}; hit-line, no break
emits a line of data to one of several auxiliary files collecting data to be processed as cross-reference indexes, bibliographies, etc. (see "Document Indexing" above).

The variable field parameters are:

n — an unsigned number showing which data file is to be used. The allowable values are 0 through 9 with 0 being the default.

=ABC — the three hit string delimiters to be used for this hit line only. The default delimiters are the triplet |~;, however, if any of those characters are needed in the text of a hit string, then the delimiters must be changed to allow their use as text. For example:

```
=!~;S!segno|offset~see addressing
=|~+K|Line Terminators~;+
```

KSTR — the hit type character.

STR a hit string.

.hlf {*title*}; header-line-foonote, no break
use the given title as the footnote header line. The default footnote header is a full page or column width horizontal rule.

.htd {*name*} {*#s,#s,#s,...*}; horizontal-tab-define, no break
define a horizontal tab stop pattern according to the ordered set of parameters. If no parameters are given, then all tab stop patterns are cancelled. No more than 20 tab stop patterns may be defined but all may be simultaneously active.

The variable field parameters are:

name the name of the pattern.

#s,#s,#s,... the position/leader string pairs for the 1st, 2nd, 3rd, ..., tab stop positions. Up to 20 positions may be given. If no positions are given, the pattern is cancelled.

The position values are given as unsigned, nonzero numbers in increasing order from left to right across the page.

The leader strings may be any character strings, however, if any contains a blank (ASCII SP), it must be given as a quoted string.

.htf {*aa...*}; horizontal-tab-off, format break
disable horizontal tabulation for the given set of characters. If no tab stop pattern is associated with any of the characters, then the control is ignored. If no characters are given, then horizontal tabulation is disabled for all patterns.

.htn *a name*\ *#s,#s,#s,...*; horizontal-tab-on, format break
enable horizontal tabulation according to the parameters given. Text filling is suspended if enabled, and text alignment is forced to flush left. Both parameters are required and are:

a the character to be assigned a temporary role of horizontal tabulation character and associated with the given tab stop pattern. The character is not available for use in text during this use.

name the name of the tab stop pattern to be enabled.

#s,#s,#s,... position/leader string pairs as for .htd above. Patterns defined by this mechanism are not retained as they are for named patterns with the .htd control, but are discarded upon the next occurrence of .htf or .htn.

.hy; hyphenate-default, no break
enable hyphenation according to the initial default and syllable size as set with the -hyphenate control argument for this invocation of the Formatter. If no syllable size is given with the control argument, then the default size is used. If the control argument is not given, then the initial default is off. The default syllable size is 2.

.hyf; hyphenate-off, no break
disable hyphenation.

.hyn {#}; hyphenate-on, no break
enable hyphenation and set the syllable size according to the given parameter. The parameter is given as an unsigned integer and specifies the number of

characters in the smallest allowed hyphenated syllable. If the parameter is not given, then the default syllable size is used.

The default syllable size is the size given with the -hyphenate control argument for this invocation of the Formatter. If no size is given with the control argument or the control argument is not given, then the default syllable size is 2.

.if *expr*; conditional-if, no break
begin a conditional execution group, evaluate *expr* and proceed according to the result. (See "Notes" under Formatting Controls above.)

If *expr* is given as a logical *expr*ession, then evaluate it as given. If it is given as a string expression, then evaluate it as though it were *expr* ^= "". If it is given as a numeric expression, then evaluate it as though it were *expr* ^= 0.

If the result of evaluation is true or *expr* is not given, then execute the "then" group (if any) following; if it is false, then execute "else" group (if any) following. (See .then and .else controls.)

The end of the conditional execution group created by this control is marked with the .endif control and the eventual occurrence of that control is required.

.ifi *path* {*arg1, arg2, ...*}; insert-file, no break
refer to ".." control above for a description of this format control.

.ift; insert-footnotes, no parameters, no break, no substitution
Insert all pending footnotes and reset the footnote counter to 1.

.in {*n*}; indent-left, format break
refer to .inl control below for a description of this format control.

.inb {*n*}; indent-both, format break
set the indention for both left and right margins according to the value of the parameter. The default value is 0.

If it is given as an unsigned number, set the indention to the value given.

If it is given as a signed number, change the indention by the amount given.

.indctl {*state*}; indent-controls, no break
where state is "on" or "off". The existing state is remembered in a circular (20 entries) queue and the state is set as specified. If the state is not given, the queue is popped.

The program no longer requires that a control line have the control starting in column 1, but merely that the control (.XXX) be the first non-blank string in the line (refer to "General Syntax" above). This feature allows indention of control lines and makes macros more readable. Note however, that this is an incompatible change since certain applications may rely on the now-ignored leading white space to "protect" certain formatting controls. Such protection will now have to be done with a .bbl control.

.inl {*n*}; indent-left, format break
set the indention for the left margin according to the value of the parameter. The default value is 0.

If it is given as an unsigned number, set the indention to the value given.

If it is given as a signed number, change the indention by the amount given.

.inr {*n*}; indent-right, format break
set the indention for the right margin according to the value of the parameter. The default value is 0.

If it is given as an unsigned number, set the indention to the value given.

If it is given as a signed number, change the indention by the amount given.

.la *label_name*; label, no break
record *label_name* as a target for .go controls. The Formatter supports up to 2000 labels for each input file.

.ls {*n*}; linespace, no break
set the linespace value according to the parameter given.

If *n* is omitted, then set the linespace to "1"; however, if -linespace was given in the invocation of the Formatter, then use that value.

If *n* is given as an unsigned number, then set the linespace to the value given.

If *n* is given as a signed number, then change the linespace by the amount given.

.pd {*l*} {,*w*} {,*c* { *(d)* }} {,*g,c* { *(d)* }... } {,b\u}; page-define-all, column break (see Note)
define the page according to the ordered set of parameters. If a value is not given for a parameter (i.e., its field is blank or null), then its default value is used; however, if following values are given, the comma for the skipped value must still be given.

Note: This control causes a column break only if the column structure changes; otherwise, no break occurs.

The variable field parameters are:

l	the length of the page given as a number of 10-pitch lines. The default value is 66. If it is given as an unsigned number, then the length is set to the value given. If it is given as a signed number, then the length is changed by the amount given.
w	the width of the page given as a number of 10-pitch characters. The default value is 65. If it is given as an unsigned number, then the width is set to the value given. If it is given as a signed number, then the width is changed by the amount given.
c(d),g,c(d)...	See the description of the .pdc control.
b\u	See the description of the .pdc control.

.pdc {*c*{*(d)*}} {,*g,c*{*(d)*}...} {,b\u}; page-define-column, column break (see Note)
define text columns according to the ordered set of parameters. If a value is not given for a parameter (i.e., its field is blank or null), then its default value

is used; however, if following values are given, the comma for the skipped value must still be given.

The width and depth offset values for column 1 are given first, then gutter, width, and depth offset values for the remaining columns are given. All nonzero width values given must be the same; however, gutter and depth offset values may vary from column to column. If no parameters are given or all values are 0, then the Formatter returns to page mode.

Note: This control causes a column break only if the column structure changes; otherwise, no break occurs.

The variable field parameters are:

c{d},g,c{d}... the widths *(c)* of text columns, their depth offsets *(d)*, and their separating gutters *(g)*. The widths and gutters are given in terms of 10-pitch characters and the depth offsets are given in terms of 10-pitch lines. The columns are numbered 1, 2, 3, ... with column 1 placed at the left page margin. The maximum number of columns allowed is 20.

If any width value is skipped or given as "0", it does not appear on the page and is not assigned a column number; that is, there are no breaks in the column number sequence. The default value for widths is 0.

The depth offsets are relative to the first text position on the page, that is, the first available text line position following the page header (including the header margin). If a value is given as an unsigned or positive number, the top of the column is moved downward by the given amount; if it is given as a negative number, the top of the column is moved upward (into the header margin) by the given amount. The default value for depth offsets is 0.

If any gutter value is given as "0", then the two adjoining columns have no separating space. The default value for gutters is 3.

b\u the column balancing action to be used.

The two allowed parameter values are:

- b at a page break event, balance the columns so that their bottoms are at the same page depth level. The success of this balancing is limited by the capabilities of the output device and is affected by widowing constraints. This is the default.
- u do not balance the columns, leaving their bottoms ragged.

.pdl {*n*}: page-define-length, no break
define page length only, as described for the .pd control.

.pdw {*n*}; page-define-width, no break
define page width only, as described for the .pd control.

.pfl {e\o\a} # {*n*} {*tit/e*}:page-footer-line, no break
define page footer lines according to the values given in the variable field. If no parameters are given, then all page footers are cancelled. The variable field parameters are:

e\o\a — the page position key indicating even, odd, or all pages. If this field is omitted, then all pages are assumed.

— the index value for the line.

If it is omitted or given as "0", then the current page footer block is cancelled and this line becomes line 1 of new footer block.

If it is less than or equal to the highest index number in the block, then *title* replaces that line in the block. If no *title* is given, then the line is replaced with a null line.

If it is greater than the highest index line number in the block, then *title* becomes the indexed footer line with the given number in the block and any intervening indexed lines become null lines. If no *title* is given, then the control is ignored since all lines involved would be null lines.

n — the indention value for this footer line. Note that *n* may not be given unless # is also given since they both appear as simple, unsigned numbers.

If it is omitted or given as "0", then *title* is aligned at the left page margin. If it is given as any other value, then *title* is aligned at the given position on the page.

title — the three-part title used as the footer line. Any references to symbolic variables in the title are evaluated when the line is placed on the page.

.phl {e\o\a} {# {*n*}} {*title*}; page-header-line, no break

define page header lines according to the values given in the variable field. If no parameters are given, then all page headers are cancelled. The variable field parameters are:

{e\o\a} — the page position key indicating even, odd, or all pages. If this field is omitted, then all pages are assumed.

— the index value for the line.

If it is omitted or given as "0", then the current page header block is cancelled and this line becomes line 1 of the new page header block.

If it is less than or equal to the highest index number in the block, then *title* replaces that line in the block. If no *title* is given, then the line is replaced with a null line.

If it is greater than the highest index line number in the block, then *title* becomes the indexed header line with the given number in the block and any intervening indexed lines become null lines. If no *title* is given, then the control is ignored since all lines involved would be null lines.

n — the indention value for this header line. Note that *n* may not be given unless # is also given since they both appear as simple, unsigned numbers.

If it is omitted or given as "0", then *title* is aligned at the left page margin. If it is given as any other value, then *title* is aligned at the given position on the page.

title the three-part title used as the header line. Any references to symbolic variables in the title are evaluated when the line is placed on the page.

.rd; read, no break
read one line from the user_input I/O switch and process it as an input line. Processing continues with the line following the .rd control unless the line read from user_input is a .go control that sends processing elsewhere.

.rt; return, no break
stop reading input from the current file and return to reading input from the suspended input file, if any.

If there is no suspended file (i.e., the current input file is given in the command line invoking the Formatter), then the remainder of the file is ignored and processing proceeds according to the command line parameters.

If there is a suspended file, then any arguments saved in the push-down/pop-up stack are restored to their prior values.

.spb {#}; space-block, block break (see Note)
finish the current output line, formatting any pending text as a short line, and then, if sufficient space remains in the column or page, add the given amount of space, then finish the current text block. If there is not sufficient space, then finish the current column or page and do not add white space. Any space created with this control is discarded if it appears at the top of a column or page. The default value for the parameter is 1. A blank or null line in the text has the effect of a .spb 1 control.

Note: For certain special blocks and formatting modes (e.g., page headers and keeps.) that require some other control to signal completion, this control causes only a format break, that is, it has the action of the .spf control following.

.spd *n*; space-to-depth, block break
finish the current block and then advance the page depth by adding white space as determined by the given parameter. The parameter is required; there is no default.

If the parameter is given as an unsigned number, then add a nontrimmable white space block such that the next output line appears at the page depth given. It is an error to give a value less than the current page depth.

If the parameter is given as a positive number, then add a nontrimmable white space block such that the page depth advances by the amount given. It is an error to give a negative number.

.spf {#}; space-format, format break
finish the current output line and then add the given amount of white space to the current block. The default value for the parameter is 1. If the white space is at the beginning of the text block and the text block is placed at the top of the page or column, then the space is trimmed.

.spt {#}; space-total, no break (see Note)
ensure that at least the given amount of white space appears on the page. The parameter is given as an unsigned number and its default value is 1.

This is accomplished as follows:

- calculate the space required as the difference between the given parameter and the amount of white space in the output immediately preceding occurrence of the control.
- if the space required is zero, then ignore the control.
- if the space required is greater than zero, then execute a .spb or .spf control (as appropriate for the current block) with the amount of required space.

Note: This control does not cause a break. However, if a .spb or .spf control is executed in the course of processing, then the executed control causes a break as described above.

.src *name* {*value-expr* {*by incr-expr*}}; set-reference-counter, no break
define *name* as a user counter variable. If it does not exist, then create and initialize it; otherwise, convert the existing variable to a counter variable. Both given expressions must be numeric expressions.

If *value-expr* is given, then evaluate it and assign it as the value of *name*; otherwise, do not change any existing value. If *incr-expr* is given, then evaluate it and assign it as the increment of *name*; otherwise, do not change any existing increment.

For newly created counters, the default value is 0 and the default increment is 1.

.srm *mode name*{ *name ...*}; set-reference-mode, no break
set the numeric display mode for the named variables according to the given mode keyword. *mode* is required and at least one *name* must be given. It is an error to give the name of the page counter, PageNo, in the list of names.

The valid mode keywords are:

Mode	Display
ar	Arabic numerals (0,1,2,...)
bi	binary numerals (0,1,10,11,100,...)
hx	hexadecimal numerals (0,1,2,...,D,E,F,10,11,...)
oc	octal numerals (0,1,2,...,7,10,11,...)
al	lowercase alphabetics (,a,b,...,z,aa,ab,...,zz,...)
au	uppercase alphabetics (,A,B,...,Z,AA,AB,...,ZZ,...)
rl	lowercase Roman (,i,ii,iii,iv,v,vi,...)
ru	uppercase Roman (,I,II,III,IV,V,VI,...)

.srv *name* {*value-expr*}; set-reference-value, no break
define *name* as a user variable. If it does not exist, then create and initialize it; otherwise, convert the existing variable to the type of the given *value-expr*.

If *value-expr* is given, then evaluate it and assign it as the value of *name*; otherwise, do not change any existing value. If the given expression is a numeric expression, then the variable is a numeric variable. If the given expression is a string expression, then the variable is a string variable.

.tab {*name*} {*p* {*col-spec* | [*astr*] {*a*} {*v*}:...}; table_define, no break
define a table column format according to the parameters given. If no parameters are given, then all table column formats are cancelled. No more than 20 table column formats may be defined at any one time. Up to 20 columns may be defined for a table column format. The term *col-spec* has

the form "{*w*} {*f*} {*a*} {/*str*}. See "Tables, Formatted" above for more information on additional capabilities. The variable field parameters are:

name — the name of the table column format. If this is the only parameter given, then the named table column format is cancelled.

p — the table column left margin value given in terms of 10-pitch characters as measured from the current left margin of the page or column. These values are required and must be given in steadily increasing order from left to right across the page.

w — the table column width value given in terms of 10-pitch characters. If any width value is omitted, then the width for that table column is the space from the table column left margin to the text right margin, that is, the rest of the text column. The separating comma is required in all cases.

f — the fill mode for the table column. The allowed values are:

f	filled (Default)
n	unfilled

a — the text alignment mode character for the table column. If this character is given with *col-spec*, then the fill mode character above must also be given. The allowed values are:

b	both (col-spec only and Default)
c	centered
l	left (Default with astr)
r	right
i	inside (col-spec only)
o	outside (col-spec only)

/str — the leader string to be used on the last output line (of a table entry) in the table column. It may be any character string; however, if it contains a blank (ASCII SP), it must be given as a quoted string. If a leader string is given, then both the fill mode and alignment mode characters above must be given.

astr — a string to be matched for text alignment. Each text line for the column is searched for a matching astr. If a match is found, then the first character of the matching string is aligned at the column position. If no match is found, the first character of the text line is aligned at the column position according to the text alignment character.

v — the vertical alignment mode character for the table column. The text lines for the column are aligned within the available vertical space according to this alignment character. The allowed values are:

t	top (Default)
c	centered
b	bottom

.tac {#}; table-column, format break, block break
if in table mode, finish the current output line as the last line (inserting the leader, if any) for the current table column in the current table entry and

proceed according to the value of the parameter. It is an error to give this control when not in table mode.

Note: This action implies that all text for a particular table column in a table entry should be given before switching to another table column.

The parameter is given as an unsigned integer. If it is omitted or given as "0", then finish the current table entry (with a block break) and return to the formatting parameters of the containing column without leaving table mode. If it is given as any other value, then switch to the formatting parameters of that table column.

.taf; table-off, block break
finish the current table entry as for .tac above, and leave table mode. If not in table mode, then ignore the control.

.tan *name*; table-on, block break
if not in table mode, then finish the current text block and enter table mode using the named table column format for the table column formatting parameters. If already in table mode, then finish the current table entry and switch to the named table column format without leaving table mode.

When in table mode, input text may be given in either of the following ways:

context mode
each input text line begins with a period and a single decimal digit (e.g., ".1Test line."). The digit indicates the table column for which the text is intended and it is formatted according to the parameters for that table column. By convention, the digit "0" indicates the tenth column of the format. Only the first ten table columns may be referenced with context mode.

Whenever the use of the .tac or .tan control returns to the formatting parameters of the containing column, input may be given in context mode. All control lines and any input text without the period/digit initial characters continue to be executed in, and formatted according to the parameters of the containing column. Their use may produce unexpected results and/or overprinted output. In context mode, input is limited to plain text; no special formatting features may be used.

free column mode
whenever the .tac control is used to select a table column (including the containing column), input may be given in free column mode. Almost all Formatter features may be used in free column mode; those few that may not are documented and their use is an error. For example, it is an error to give context mode input lines in free column mode; they are considered to be unknown controls.

.tcl {# {*n*}} {*title*}; title-caption-line, no break
define a text caption line according to the values given in the variable field. If no parameters are given, the caption line is a single white space line.

If a pending text caption already exists, then add the line to it; otherwise, use the line as the caption.

The variable field parameters are:

#	the amount of extra white space to be inserted *ahead* of the line. It must be given as an unsigned number. The default value is 0.
n	the indention value for this line. Note that *n* may not be given unless # is also given since they both may appear as simple, unsigned numbers. If it is omitted or given as "0", then *title* is aligned at the current text left indention. If it is given as an unsigned number, then *title* is aligned at the given position relative to the page or column margin. If it is given as a signed number, then it is used as a local adjustment to the current text left indention.
title	the three-part title used as the caption line. Any references to symbolic variables in the title are evaluated when the line is placed on the page.

.then; conditional-then, no break

begin the conditional execution group clause that is executed when the expr of the preceding elseif or .if control is true. The end of the clause is marked by a .elseif, .else, or .endif control and the eventual occurrence of one of them is required. (See "Notes" under Formatting Controls above.)

.trn {*abab...*}; translate, no break

each nonblank character *a* in the input file is replaced with its associated character *b* in the output. Any number of *ab* pairs may be given and the *ab* pairs from multiple occurrences of the control are accumulated. If a character pair is given as *aa*, then translation for *a* is cancelled without affecting any other characters being translated. If no character pairs are given, then the translation feature is disabled.

.ts *expr*; test, no break

evaluate *expr* and proceed according to the result.

If *expr* is given as a logical expression, then evaluate it as given. If it is given as a string expression, then evaluate it as though it were *expr* = "". If it is given as a numeric expression, then evaluate it as though it were *expr* = 0.

If the result of evaluation is true or *expr* is not given, then continue processing with the next input line; if it is false, then skip the next input line and continue with the second input line following.

.ttl {# {*n*}} {*title*}; text-title-line, no break

define a text title line according to the values given in the variable field. If no parameters are given, the title line is a single white space line.

If a pending text title already exists, then add the line to it; otherwise, use the line as the text title.

If there is no current text block, then use the line as the text title for the next block.

If there is a current text block and it is untitled, then prepend the line onto the current text block as a text title.

If there is a current text block and it is already titled, then insert the line between the existing text title and the first text line in the current text block.

The variable field parameters are:

#	the amount of extra white space to be inserted *after* the line. It must be given as an unsigned number. The default value is 0.
n	the indention value for this line. Note that *n* may not be given unless # is also given since they both may appear as simple, unsigned numbers. If it is omitted or given as "0", then *title* is aligned at the current text left indention. If it is given as an unsigned number, then *title* is aligned at the given position relative to the page or column margin. If it is given as a signed number, then it is used as a local adjustment to the current text left indention.
title	the three-part title used as the title line. Any references to symbolic variables in the title are evaluated when the line is placed on the page.

.ty {*expr*}; type, no break
evaluate *expr* and immediately write it back to the user on the error_output I/O switch. If *expr* is not given, a blank line is written.

.un {*n*}; undent-left, format break
refer to .unl control below for a description of this format control.

.unb {*n*}; undent-both, format break
adjust the indention for both the left and right margins *for the next output line only* according to the value of the parameter. The default value for the parameter is the current value of both the left or right indention. Positive or unsigned values of the parameter move the text outward; negative values move it inward.

.unh {*n*}; undent-hanging, format break
adjust the indention for the left margin *for the next input text line only* according to the value of the parameter and *suppress the automatic depth advance*. The line affected is formatted as unfilled and flush left at adjusted indention point. Any following text is normally formatted according to the current indention value and appears at the same page depth. The default value for the parameter is the current value of the left indention. Positive or unsigned values of the parameter move the text outward; negative values move it inward.

.unl {*n*}; undent-left, format break
adjust the indention for the left margin *for the next output line only* according to the value of the parameter. The default value for the parameter is the current value of the left indention. Positive or unsigned values of the parameter move the text outward; negative values move it inward.

.unr {*n*}; undent-right, format break
adjust the indention for the right margin *for the next output line only* according to the value of the parameter. The default value for the parameter is the current value of the right indention. Positive or unsigned values of the parameter move the text outward; negative values move it inward.

.ur *control\expr*; use-reference, no break
the given parameter is evaluated *once* as discussed in "Advanced Features" above and is then reprocessed as an input line.

If the parameter contains nested symbol delimiters, then the symbolic references at the deepest level are substituted and the nesting level is reduced by 1. For example, if the variable I contains the value "1", then the construct %%list%I%%% becomes %list1%.

.vm {*t,h,f,b*}; vertical-margin-all, no break
set vertical page margins according to the ordered set of parameters. Values are given as numbers of 10-pitch lines. If a value is not given for a parameter, (i.e., its field is blank or null), then its default value is used; however, if following values are given, the comma for the skipped value must still be given. The variable field parameters are:

t page top margin. The default value is 4.

Note: The minimum value for this parameter depends on the output device and is obtained from the device table for the device (see Appendix C). For example, Honeywell lineprinters (as configured for Multics) have a minimum top margin of 3 lines.

If the value is given as an unsigned number, the top margin is set to the value given.

If the value is given as a signed number, the top margin is changed by the amount given.

h page header margin. The default value is 2.

If the value is given as an unsigned number, the header margin is set to the value given.

If the value is given as a signed number, the header margin is changed by the amount given.

f page footer margin. The default value is 2.

If the value is given as an unsigned number, the footer margin is set to the value given.

If the value is given as a signed number, the footer margin is changed by the amount given.

b page bottom margin. The default value is 4.

Note: The minimum value for this parameter depends on the output device and is obtained from the device table for the device (see Appendix C). For example, Honeywell lineprinters (as configured for Multics) have a minimum bottom margin of 3 lines.

If the value is given as an unsigned number, the bottom margin is set to the value given.

If the value is given as a signed number, the bottom margin is changed by the amount given.

.vmb {*n*}; vertical-margin-bottom, no break
set the page bottom margin only as described for the .vm control above.

.vmf {*n*}; vertical-margin-footer, no break
set the page footer margin only as described for the .vm control above.

.vmh {*n*}; vertical-margin-header, no break
set the page header margin only as described for the .vm control above.

.vmt {*n*}; vertical-margin-top, no break
set the page top margin only as described for the .vm control above.

.wit {#}; widow-text, no break
change the widow size according to the given parameter. The parameter is given as a number of output text lines. It is an error to give a value that results in a widow size that is negative or greater than the page length. The default value for the parameter is 2.

If the parameter is given as an unsigned number, then set the widow size to the value given.

If the parameter is given as a signed number, then change the widow size by the amount given.

.wrt *path* {*string*}; write-text, no break
string (as given) with an added NL (ASCII code 012) character is written to the segment *path*. If *string* is not given, a blank line is written.

If *path* is not attached and open as an output file, it is attached and opened. If it does not exist, it is created as an empty file. If it does exist, it is truncated to an empty file by the open. All attachments are made through the vfile_ I/O module (see *Multics Subroutines*).

.wt; wait, no break
read one line from the user_input I/O switch and discard the input. Unlike the .rd control above, this control is executed during *output* rather than *input*. Processing continues with the next input line.

COMPREHENSIVE CONTROL SUMMARY

The following list presents a comprehensive grouping of all the compose formatting controls, including an identification of complexity for each. That is, one of the following categories is assigned to each control: basic, intermediate, or advanced. The format of the list is designed for ease of duplication so that a copy can be made and be available for posting near the users' terminal.

SECTION 3

WORDPRO COMMANDS

This section contains tools that may be used to:

- Translate a device description file into a binary table for use by the Formatter
- Prepare formatted documents from raw text segments for production on various documentation devices utilizing both the compose and format_document commands
- Produce a cross-reference index file from raw data
- Convert runoff input segments to compose input segments
- Display selected information from a compose device table
- Process compose output files to an online device, or to magnetic or punched paper tape
- Expand an expansion input file into an expansion output file

compdv

The compdv command is used to invoke the Device Table Compiler to translate a device description file into a binary table for use by the Formatter (see "Device Description Language" in Appendix C).

SYNTAX AS A COMMAND

compdv *path* {*-control_args*}

ARGUMENTS

path
: is the pathname of the input device description file. The entryname of the file must end with the suffix compdv, but the suffix need not be given in the command. The output segment is created (if it does not already exist) in the working directory with an entryname formed by replacing the suffix compdv with comp_dsm. Multisegment files and the star convention are not supported.

CONTROL ARGUMENTS

-check
-ck
: processes the input file, making all syntax checks and creating the ALM source intermediate file, but do not invoke the ALM assembler and do not delete the ALM source file. The default is to invoke the ALM assembler at the end of an error-free translation and to delete the ALM source file.

-list
-ls
: create an ALM assembly output listing for the translation. The default is no listing.

compose (comp)

The compose command is used to prepare formatted documents from raw text segments for production on various documentation devices including typesetters, line printers, and user terminals. Output pages are composed from various text blocks and controls provided in input files. Detailed control over page composition is provided by controls in the input file.

SYNTAX AS A COMMAND

comp *paths* {*control_args*}

ARGUMENTS

paths
are the pathnames of the input files to be formatted. The suffix compin must be the last component of the input file entryname; however, the suffix need not be given in the command line. If two or more pathnames are specified, they are treated as if compose had been invoked separately for each. Up to 200 input files may be given with one invocation of the command. Output is produced in the order in which the pathnames are given in the command line. Input files may be either single segments or multisegment files. Output files for very large documents are converted to multisegment files. The star convention is not supported.

CONTROL ARGUMENTS

all control arguments specified in the command line apply to all input file pathnames given. Control arguments may be freely intermixed with input file pathnames, except for -arguments which must be last in the command line.

-annotate {*key key key ...*}
-ann {*key key key ...*}
shows all font/pointsize changes identified by the optional key (where key may only be "font" at the present time) in an extra column to the right of the formatted text and at the output line in which they occur.

The style of a typeset document usually calls for a large number of font/pointsize changes to improve readability. However, when one is limited to a terminal and/or lineprinter for early checkout of the documents, it is very difficult to determine if the changes are all being made correctly.

-arguments *arg1 ... argn*
-ag *arg1 ... argn*
all fields following are string values to be placed in the indefinite set of program built-in variables named "CommandArg*1*" through "CommandArg*n*" where "*n*" is the count of such fields. The program built-in variable "CommandArgCount" is set to "*n*". If any argument is to contain blanks, it must be given as a quoted string.

Note: This control argument, if given, must be the last control argument in the command line.

-brief
-bf
shows only the header line of the defined error line (i.e., the count of errors), both at normal termination and in response to the program_interrupt command.

-change_bars {*x,p,l,r,d*}
-cb {*x,p,l,r,d*}
generates text change symbols in the output according to the parameters given. Change symbols are shown in the text margins as determined by controls in the text. The default for change symbol generation is OFF. All the parameters for this control argument are optional but, if any are given, they must appear in the order shown. If any parameter is skipped, its separating comma must still be given. Skipped parameters retain their default values. The parameters are:

x a change level character. If the optional change level character in any change-bar control is less than *x* (in the ASCII collating sequence sense), then no text change symbols are inserted for those controls. The *x* character may be either numeric or alphabetic. The default value for *x* is the SP (ASCII code 040) character.

p a symbol placement key character. It may have the values "l" for left margin, "r" for right margin, "i" for inside margin, or "o" for outside margin. The default for *p* is "o".

l the definition of the text change symbol to be placed to the left of text. It must be of the form *n*{*string*} where *string* is any character string and *n* is the separation from the text. The default value for *string* is a vertical bar (|) and the default value for *n* is 1. The *n* may be given without *string*, but *string* may not be given without *n*.

r the definition of the text change symbol to be placed to the right of text. It must be of the same form as *l* above.

d the definition of the text deletion symbol. It must be of the same form as *l* above except that the default for *string* is the asterisk (*).

-change_bars_art {*x,p,l,r,d*}
-cba {*x,p,l,r,d*}
as for -change_bars above except that the *strings* in the *l*, *r*, and *d* fields may be given as conventional artwork symbols (see "Creating Artwork" in Section 2). The default values for the *strings* are also artwork symbols.

-check
-ck
performs syntax checking on the input file(s) by processing all text and controls but does not produce any output. The default for this feature is OFF.

-device {*name*}
-dv {*name*}
prepares output compatible with the device specified. This control argument is used when the target device for output is not the default device for the output mode selected. If the -output_file control argument is given, the default device is "printer"; if it is not, the default device is "ascii". Any device for which *name*.comp_dsm exists is a supported device (see "Device Table Compiler" in Appendix C).

-from {*n*}
-fm {*n*}
starts printed output at page *n*. This control argument is mutually exclusive with the -pages control argument. You must give the desired *structured* page number; for example, to print the fourth page of Section 3, you must give "-page 3-4". The default value of *n* is 1.

-galley {*n1*} {,*n2*}
-gl {*n1*} {,*n2*}
produces galley format (continuous single-column text without page headers and footers) output for lines *n1* through *n2* of the input file. The default value of *n1* is 1 and the default value for *n2* is the last line in the input file. If *n2* is not given, the comma need not be given. If *n1* is not given, a comma must precede a given value for *n2*. The default for this feature is OFF.

-hyphenate {*n*}
-hyph {*n*}
-hph {*n*}
changes the default hyphenation mode from OFF to ON. The optional parameter *n* is the length of the smallest separated word part. Its default value is 2.

-indent {*n*}
-ind {*n*}
adds *n* spaces at the left page margin of the output. This space is in addition to any indention given in the text. The default value of *n* is 0 (i.e., at the left-hand mechanical stop of the output device).

-input_file *path*
-if *path*
the name of an input file even though *path* may have the appearance of a numeric parameter or a control argument.

-linespace {*n*}
-ls {*n*}
changes the default line spacing value to *n*. The linespace control uses *n* as a minimum value. The default value for *n* is 1.

-noart
-noa
disables the conversion of conventional artwork constructs and inserts space into the output at the positions that such constructs would occupy. The default for the artwork conversion feature is ON.

-nobell
-nob
suppresses the audible BELL signal when signalling the "waiting" state to the user when the -stop or -wait control argument is used.

-nofill
-nof
sets the default fill mode to OFF. The default fill mode is ON.

-number
-nb
prints input line numbers at the left margin of the output. The line numbers have the form "*i n*" where *i* is the index number of an inserted file. A list of

inserted files showing the index numbers is written on user_output after completion of all text processing. The default for this feature is OFF.

-number_brief
-nbb
prints input line numbers at the left margin of the output as for the -number control argument but the list of inserted files is not produced.

-output_file {*path*}
-of {*path*}
directs the formatted output to a file instead of to the user's terminal. The assumed output device is the Multics online printer but may be changed with the -device control argument. If *path* is not given, then the output for all given input files is written to individual output files whose names are formed by replacing the suffix compin of the input file entrynames with the suffix compout. If *path* is given, then output for all given input files is accumulated in that single bulk output file. The default for this feature is OFF; that is, formatted output is written back to the user's terminal.

-pages *page_list*
-pgs *page_list*
specifies a blank-separated list of selected pages to be printed. Each element in the *page_list* must be either a single page, *n*, or a range of pages, *n*,*n*, where *n* is a *structured* page number as for -from above. The page numbers given must steadily increase without duplication. At least one page must be specified and up to 100 list elements may be given. This control argument is mutually exclusive with the -from and -to control arguments. The default for this feature is OFF.

Note: Page number structures containing parentheses are changed by the command processor and must be given as quoted strings.

-pages_changed {x}
-pgc {x}
specifies that, of the pages selected for printing (either all pages or some subset of pages selected through use of the -pages, -from, and -to control arguments), only those pages containing text within the range of an active change-bar control or within the scope of the dot_page documentation macro are actually printed. The base pages of the dot page set (for example, page 3 of the set 3, 3.1, 3.2) are not considered part of the dot page set. This control argument is independent of the -change_bars and -change_bars_art control arguments, either of which must be given to cause the text change marks to be created. The optional parameter x chooses the change bar level and must match the x parameter given with -change_bars for proper operation. The default value for x is SP (ASCII 040) and chooses all active change levels.

-parameter {*string*}
-pm {*string*}
assigns *string* as the value of the built-in variable "Parameter". The default value for *string* is an empty string.

-passes *n*
-pass *n*
processes the input file *n* times to permit proper evaluation of expressions containing variables that are defined following their reference(s) in the text. No output is produced until the last pass. The default value for *n* is 1.

-stop
-sp
waits for a newline character (ASCII code 012) from the user before beginning each page of output and after the last page. The pause is signalled by giving two BEL/HT sequences and *returning the print head to the left margin*. If only a newline is typed, the next page is printed. If a q is typed, the command invocation is terminated gracefully. If an r is typed, the page just printed is reprinted. The default for this feature is OFF.

-to *n*
ends output after the page numbered *n* where *n* is a *structured* page number as for the -from control argument above. This control argument is mutually exclusive with the -pages control argument. The default value for *n* is the last page.

-wait
-wt
waits for a newline character (ASCII code 012) before beginning the first page of output to the terminal, but not between pages (see the -stop control argument above). The default for this feature is OFF.

compose_index (cndx)

This command processes raw index data gathered by compose and produces a cross-reference index file according to a specified format.

SYNTAX AS A COMMAND

cndx *path* {*-control_args*}

ARGUMENTS

path
: is the pathname of the compin file producing the raw index data. The compin suffix need not be given.

CONTROL ARGUMENTS

-alpha_header
-ahdr
: inserts centered uppercase alphabetic characters as group separators whenever the first character of the primary key changes.

-control_file *ctl_path*
-cf *ctl_path*
: uses *ctl_path*.cndxctl as the control file for this index. The suffix "cndxctl" is assumed if not given. The default control file is *path*.cndxctl.

-number *n*
-nb *n*
: one of the 10 (0 through 9) possible raw index data files. The default value is 0. See Notes below.

NOTES

The raw index data files are produced by compose when the .hit control (refer to Section 2) is used. The default raw data file is *path*.0.cndx. The output file is *path_entryname.n*.index in the current working directory. If the output file does not exist, it is created; if it does, it is overwritten.

The data in the raw data file is processed into an arbitrarily chosen format, the style of which is determined partially by constants built into the program and partially by statements in *path*.cndxctl. See "Index Control Files" below.

The final set of hit strings (after all raw data processing is complete) is sorted into an alphabetic collating sequence (i.e., without regard to case). The handling of certain prefix characters is provided by the use of a control directive. See "compose_index Control Directives" below.

Index Control Files

The index control file contains compose controls and text lines that partially determine the format of the index, and directives for compose_index that control

the processing of the hit strings (see "compose_index Control Directives") and complete the definition of the index format. The use of an index control file is not required. If one does not exist, defaults in the documentation macros and the program determine the format of the index.

The output file created by this program is to be treated just like any other section of the document to which it applies. Therefore, the same macro package initialization must be performed and this is done with compose controls at the beginning of the index control file. A standard macro, l0index, is provided for users who do not wish to provide their own detailed index format. The format established by this macro is the one used by all Multics and GCOS user documentation.

All hit string processing control directives are given as compose comment strings. During processing of the index control file, any line that is not recognized as a control directive is written to the output file for further processing by compose.

compose_index Control Directives

For the control directives that follow, *all* input is given in *lower-case* without regard to the case of the output.

.*blind *abcd...*

In many instances, a particular keyword will appear as such and with one or more prefix characters. A typical example is "rawo" and "^rawo" in tty_ modes. It is desirable to have such prefixed and unprefixed keys sort together in the index. To accomplished this, the sort algorithm may be made "blind" to such prefix characters by the use of this directive. The set of characters *abcd...* are treated specially as prefix characters such that keywords with them sort after their unprefixed counterparts. For example:

```
.*blind ^\$
```

Note: An older form of this directive, ".*ignore" is also supported.

The following directives apply to permuted keys only.

.*phrase *str*

Very frequently, it is necessary that a short phrase instead of a single word be a key in the index. This directive provides the ability to indicate that such phrases are to be treated as keys. Since punctutation may be wanted in the phrase, only one *str* may be given in the line. For example:

```
.*phrase access control
.*phrase pack labels
.*phrase control cards
```

.*tran *str1,str2*

Also very frequently, various grammatical forms of a root keyword or a suffixed keyword appear in an index and should be sorted together. This directive provides the ability to transform such keys *for sorting only*; the given keys will appear in the final index. For example:

```
.*tran labels,label
.*tran labeled,label
```

```
.*tran labelling,label
.*tran sys_info_$,sys_info_
```

.*excl *excl_key* {,*excl_key*}...

During permutation, many unwanted hit strings may be generated, primarily due to conjunctions, articles, prepositions, etc., in the given primary key. Further, permutation may generate unwanted hit strings that have a primary key that *is* wanted for other hits. This directive controls the exclusion of hit strings that *begin* with the partial hit string *excl_key*. Only as much of the unwanted hit string as is needed for unique identification need be given, but it must contain the entire new primary key. For example:

```
.*excl to,for,from,and,but
.*excl system~info,reporting~standard
```

convert_runoff (cv_rf)

The convert_runoff command converts a runoff input segment into a compose input segment.

SYNTAX AS A COMMAND

cv_rf *path*

where *path* is the pathname of a runoff input segment to be converted. The suffix runoff need not be given. Output is written into [wd]>entryname.compin, where entryname is extracted from path.

NOTES

All controls processed are at the beginning of lines or immediately after a series of `.ur` controls.

Warning messages are produced for conversions that are doubtful or may be ambiguous.

display_comp_dsm (ddsm)

The display_comp_dsm command displays selected information from a compose device description table, *device*.comp_dsm.

SYNTAX AS A COMMAND

ddsm *path* {*font*} {*-control_args*}

ARGUMENTS

path
is the pathname of a device description table. The entryname must end with the suffix comp_dsm but the suffix need not be given in the command line. The star convention is not supported. If this is the only argument given, summary information on all devices defined in the table is displayed. See Examples below.

font
is the external name of a font defined for the device. It may be given as family or family/member. If this argument is given, then all the graphics (Multics characters) for the named font with their widths are displayed; otherwise, information on the device is displayed.

CONTROL ARGUMENTS

-device
-dv
displays information on the named device only (including all defined fonts).

-linelength *n*
-ll *n*
sets the line length for the display to *n*. The default value is the system defined linelength for the user's terminal.

-long
-lg
displays detailed information. If *font* is given, then display all the graphics (Multics characters) for the font with the width and replacement output string for each. If *font* is not given, then display all the defined parameters for the named device.

EXAMPLES

```
!  ddsm dtc300s                          -show all devices
   Device:   dtc300s, DTC300s;
     devclass: diablo;
   Device:   vdtc, v300;
     devclass: photocomp;

!  ddsm dtc300s -dv -ll 65               -show named device with all fonts
   Device:   dtc300s, DTC300s;
     devclass: diablo;
       family: centuryschoolbook, cs, helvetica, h;
         member: /, /m, /medium, /r, /roman;
         member: /b, /bold, /boldroman, /br;
         member: /bi, /bolditalic;
         member: /i, /italic, /mediumitalic, /mi;
       family: pica10;
         member: /, /m, /medium, /r, /roman;
         member: /b, /bold, /boldroman, /br;
         member: /bi, /bolditalic;
         member: /caps;
         member: /caps_;
         member: /i, /italic, /mediumitalic, /mi;
       bachelor: ascii, 14font, 13exact, 14exact, text, footnote,
            APL, CSR, HR;
       bachelor: 10font, 13font, ASCII;
       bachelor: bold, CSBR, HBR, HBB1;
       bachelor: italic, 12font, 11exact, 12exact, CSI, HmI, ascii_;
       bachelor: 11font, ASCII_;
       bachelor: CSBI, HBI;
       family: pica12;
         member: /, /m, /medium, /r, /roman;
         member: /b, /bold, /boldroman, /br, /caps;
         member: /bi, /bolditalic, /caps_;
         member: /i, /italic, /mediumitalic, /mi;
       bachelor: pica12;
       bachelor: pica12_;
       bachelor: PICA12;
       bachelor: PICA12_;

!  ddsm dtc300s -lg -ll 65               -show long device information
   Device:   dtc300s, DTC300s;
     devclass: diablo;
   /* version: 1 (1) */                  -table version and expected version
     units: pt;
     attach: "syn_ user_output";
     comment: " Type Wheel Identification
    1 - 38101-01        PICA 10
    2 - 38510           APL 10
    3 - 38102-01        ELITE 12
   DB: dtc300s_writer_$display
   ";                                    -this is the closing quote for the
```

comment: field

```
  cleanup: "\033\033\033\006\033 0";
  defaultmargs: 48., 24., 24., 48.;
  init: pica10/m 7.2;
  interleave: on;
  letterspace: 0;
  maxpages: unlimited;
  maxfiles: unlimited;
  maxpagelength: unlimited;
  maxpagewidth: 950.4;
  minbotmarg: 0.;
  minlead: 1.5;
  minspace: 1.2;
  mintopmarg: 0.;
  stream: off;
  taperec: unlimited;
    family: centuryschoolbook, cs, helvetica, h;
      member: /, /m, /medium, /r, /roman;
      member: /b, /bold, /boldroman, /br;
      member: /bi, /bolditalic;
      member: /i, /italic, /mediumitalic, /mi;
    family: pica10;
      member: /, /m, /medium, /r, /roman;
      member: /b, /bold, /boldroman, /br;
      member: /bi, /bolditalic;
      member: /caps;
      member: /caps_;
      member: /i, /italic, /mediumitalic, /mi;
    bachelor: ascii, 14font, 13exact, 14exact, text, footnote,
          APL, CSR, HR;
    bachelor: 10font, 13font, ASCII;
    bachelor: bold, CSBR, HBR, HBB1;
    bachelor: italic, 12font, 11exact, 12exact, CSI, HmI, ascii_;
    bachelor: 11font, ASCII_;
    bachelor: CSBI, HBI;
    family: pica12;
      member: /, /m, /medium, /r, /roman;
      member: /b, /bold, /boldroman, /br, /caps;
      member: /bi, /bolditalic, /caps_;
      member: /i, /italic, /mediumitalic, /mi;
    bachelor: pica12;
    bachelor: pica12_;
    bachelor: PICA12;
    bachelor: PICA12_;

!   ddsm dtc300s CSR -11 65          -show font information
    Device:   dtc300s, DTC300s;
      devclass: diablo;
      bachelor: CSR;
      strokes: 6
      wordspace: 3,6,9
    010(-6)   040(6)    "!"(6)    042(6)    "#"(6)    "$"(6)
    "%"(6)    "&"(6)    "'"(6)    "("(6)    ")"(6)    "*"(6)
```

```
"+"(6)    ","(6)    "-"(6)    "."(6)    "/"(6)    "0"(6)
"1"(6)    "2"(6)    "3"(6)    "4"(6)    "5"(6)    "6"(6)
"7"(6)    "8"(6)    "9"(6)    ":"(6)    ";"(6)    "<"(6)
"="(6)    ">"(6)    "?"(6)    "@"(6)    "A"(6)    "B"(6)
"C"(6)    "D"(6)    "E"(6)    "F"(6)    "G"(6)    "H"(6)
"I"(6)    "J"(6)    "K"(6)    "L"(6)    "M"(6)    "N"(6)
"O"(6)    "P"(6)    "Q"(6)    "R"(6)    "S"(6)    "T"(6)
"U"(6)    "V"(6)    "W"(6)    "X"(6)    "Y"(6)    "Z"(6)
"["(6)    "\"(6)    "]"(6)    "^"(6)    "_"(6)    "`"(6)
"a"(6)    "b"(6)    "c"(6)    "d"(6)    "e"(6)    "f"(6)
"g"(6)    "h"(6)    "i"(6)    "j"(6)    "k"(6)    "l"(6)
"m"(6)    "n"(6)    "o"(6)    "p"(6)    "q"(6)    "r"(6)
"s"(6)    "t"(6)    "u"(6)    "v"(6)    "w"(6)    "x"(6)
"y"(6)    "z"(6)    "{"(6)    "|"(6)    "}"(6)    "¬"(6)
177(0)    200(6)    201(0)    202(6)    235(6)    236(6)
237(6)    240(6)    254(10)   EMd(12)   261(6)    277(0)
301(6)    (c)(12)   304(9)    o(6)      320(6)    324(12)
360(6)    375(6)    PS(6)     EM(6)     EM_(12)   EN(6)
EN_(6)    ENd(6)    THN(6)    -(6)      ``(6)     ''(6)
1hi-X(6)  424(6)    dn-arrow(6)         426(6)    dia-left(6)
delete-mark(6)      dia-right(6)        dia-top(6)          <(6)
1hi-{(6)  1hi-[(6)  left-circle(6)      437(6)    ->(6)
1hi-}(6)  1hi-](6)  right-circle(6)     444(6)    up-arrow(6)
447(6)    450(6)    451(6)    452(6)    453(0)    454(6)
455(6)    456(6)    457(6)    460(6)    461(0)    462(6)
463(6)    464(6)    465(6)    466(6)    467(0)    470(6)
471(6)    472(6)    473(6)    474(6)    475(0)    476(6)
477(6)    500(6)    501(6)    502(6)    503(0)    504(6)
505(6)    506(6)    507(6)    510(6)    511(0)    512(6)
513(6)    514(6)    515(6)    516(6)    517(0)    520(6)
521(6)    522(6)    523(6)    524(6)    525(0)    526(6)
527(6)    530(6)    531(6)    532(0)    534(6)    536(6)
537(6)
```

```
!  ddsm dtc300s CSR -lg -ll 65        -show long font information
   Device:   dtc300s, DTC300s;
     devclass: diablo;
     bachelor: CSR;
     strokes: 6
     wordspace: 3,6,9
   010(-6,"\010")  040(6," ")      "!"(6,"!")      042(6,"""")
   "#"(6,"#")      "$"(6,"$")      "%"(6,"%")      "&"(6,"&")
   "'"(6,"'")      "("(6,"(")      ")"(6,")")      "*"(6,"*")
   "+"(6,"+")      ","(6,",")      "-"(6,"-")      "."(6,".")
   "/"(6,"/")      "0"(6,"0")      "1"(6,"1")      "2"(6,"2")
   "3"(6,"3")      "4"(6,"4")      "5"(6,"5")      "6"(6,"6")
   "7"(6,"7")      "8"(6,"8")      "9"(6,"9")      ":"(6,":")
   ";"(6,";")      "<"(6,"<")      "="(6,"=")      ">"(6,">")
   "?"(6,"?")      "@"(6,"@")      "A"(6,"A")      "B"(6,"B")
   "C"(6,"C")      "D"(6,"D")      "E"(6,"E")      "F"(6,"F")
   "G"(6,"G")      "H"(6,"H")      "I"(6,"I")      "J"(6,"J")
   "K"(6,"K")      "L"(6,"L")      "M"(6,"M")      "N"(6,"N")
   "O"(6,"O")      "P"(6,"P")      "Q"(6,"Q")      "R"(6,"R")
```

```
"S"(6,"S")      "T"(6,"T")      "U"(6,"U")      "V"(6,"V")
"W"(6,"W")      "X"(6,"X")      "Y"(6,"Y")      "Z"(6,"Z")
"["(6,"[")      "\"(6,"\")      "]"(6,"]")      "^"(6,"^")
"_"(6,"_")      "`"(6,"`")      "a"(6,"a")      "b"(6,"b")
"c"(6,"c")      "d"(6,"d")      "e"(6,"e")      "f"(6,"f")
"g"(6,"g")      "h"(6,"h")      "i"(6,"i")      "j"(6,"j")
"k"(6,"k")      "l"(6,"l")      "m"(6,"m")      "n"(6,"n")
"o"(6,"o")      "p"(6,"p")      "q"(6,"q")      "r"(6,"r")
"s"(6,"s")      "t"(6,"t")      "u"(6,"u")      "v"(6,"v")
"w"(6,"w")      "x"(6,"x")      "y"(6,"y")      "z"(6,"z")
"{"(6,"{")      "|"(6,"|")      "}"(6,"}")      "¬"(6,"¬")
177(0,"\177")   200(6,"\033C|\033P0A1B4q\033\033C.")
201(0,"\033C_\033P0Y2hG1G5Q\033\033C.")
202(6,"\033C|\033P0q1B4A\033\033C.")            235(6,"d\010_")
236(6,"n\010_")                 237(6,"x\010_")
240(6,"\033C[\033P2H\033\033C]\033P0P4h\033\033C.")
254(10,"\033C\\033P0@\033\033C_\033P0WH\033\033C/\033P1W4h\033\033C.")
EMd(12,"\033P<B\033__\033P=B")
261(6,"\033C|\033P0@A\033\033C-\033P0B5s\033\033C.")
277(0,"")
301(6,"\033C|\033P0@A\033\033C-\033P0B1D4q\033\033C.")
(c)(12,"\033P4Y\033c\033P3H2AAIIHIHH3IHIIAA1IIHIHH0IHI4h")
304(9,"\033C/\033P0@\033\033C_\033P0JH\033\033C\\033P1J<X\033\033C.")
o(6,"\033P6H0@HHI2HHHH0AHHHH2IHH5{")
320(6,"\033P2@AAAAAQHHHHHHHH1QAAAAA4P")
324(12,"\033P0X2AAAAYOHHHHHH1AAHI3PAAOYIII1AAAA4X")
360(6,"\033P0H2AAAQHHHHHHH1QAAA4P")             375(6,"Z\010N")
PS(6," ")       EM(6," ")       EM_(12,"__")    EN(6," ")
EN_(6,"_")      ENd(6,"\033P<B\033_\033P=B")     THN(6," ")
+(6,"+")        ``(6,"""")      ''(6,"""")
1hi-X(6,"\033P>H0@IIIIII>X1@IIIIII<H")
424(6,"\033P=A0@AAAAAAAAH1AAAAAAAA4j")
dn-arrow(6,"\033P3B2IOHHI2HHHHIOHHHHHH5Y")
426(6,"\033P<C1@IIIOIII=C")
dia-left(6,"\033P1r2@IIIIOIIII=C")
delete-mark(6,"\033P6H0@IIII=B2@IIII7JO@HHHHHH=I")
dia-right(6,"\033P3BOIIII2IIII=[")
dia-top(6,"\033P3BOIII1III<A")
<(6,"\033P1]OAAAAAA3IAAAA2IAA3I<Y")
1hi-{(6,"\033P=R2@HHIIAAIIOIIAAIIHH=L")
1hi-[(6,"\033P=R3@HHHHOAAAAAAAAAAAAHHHH=L")
left-circle(6,"\033P=]2@HHHIHHIIIAIAAAAOIAIIIHHIHHH=C")
437(6,"\033P=R2@HHIIAAAAAAAAOIIHH=L")
->(6,"\033P7]O@AAAAAA1IAAAAOIAA1I<Y")
1hi-}(6,"\033P?RO@HHIIAAII2IIAAIIHH=1")
1hi-](6,"\033P?RO@HHHHAAAAAAAAAAAA3HHHH=1")
right-circle(6,"\033P?]O@HHHIHHIIIAIAAAA2IAIIIHHiHHH=s")
444(6,"\033P?RO@HHIIAAAAAAAA2IIHH=1")
up-arrow(6,"\033P2[OHHHHHH2IHHHHOIHH2I=[")
447(6,"\033PO@AAAAAAAAHHHH=L")  450(6,"\033PO@AAAAHHHH=J")
451(6,"\033PO@AAAAAAAA=\")      452(6,"\033P<P2@HHHHAAAAAAAA=\")
453(0,"\033P=R2@HHHHAAAA")      454(6,"\033PO@AAAAAAAA=\")
455(6,"\033P2@AAAAAAAAHHHH=1")  456(6,"\033P2@AAAAHHHH=j")
```

```
457(6,"\033PO@AAAAAAAA=\")      460(6,"\033P>PO@HHHHAAAAAAAA=\")
461(0,"\033P?RO@HHHHAAAA")      462(6,"\033PO@AAAAAAAA=\")
463(6,"\033PO@AAAAAAIIHH=L")    464(6,"\033PO@AAIIHH=J")
465(6,"\033P2@AAIIOIIAA=\")      466(6,"\033P<P2@HHIIAAAAAA=\")
467(0,"\033P=R2@HHIIAA")         470(6,"\033PO@AAAAAAAA=\")
471(6,"\033P2@AAAAAAIIHH=1")    472(6,"\033P2@AAIIHH=j")
473(6,"\033PO@AAII2IIAA=\")      474(6,"\033P>PO@HHIIAAAAAA=\")
475(0,"\033P?RO@HHIIAA")         476(6,"\033PO@AAAAAAAA=\")
477(6,"\033PO@AAAAAAIIHH=L")    500(6,"\033PO@AAIIHH=J")
501(6,"\033PO@AAAAAAAA=\")      502(6,"\033P<P2@HHIIAAAAAA=\")
503(0,"\033P=R2@HHIIAA")         504(6,"\033PO@AAAAAAAA=\")
505(6,"\033P2@AAAAAAIIHH=1")    506(6,"\033P2@AAIIHH=j")
507(6,"\033PO@AAAAAAAA=\")      510(6,"\033P>PO@HHIIAAAAAA=\")
511(0,"\033P?RO@HHIIAA")         512(6,"\033PO@AAAAAAAA=\")
513(6,"\033PO@AAAAAAAA=\")      514(6,"\033PO@AAAA=Z")
515(6,"\033PO@AAAAAAAA=\")      516(6,"\033PO@AAAAAAAA=\")
517(0,"\033P=BO@AAAA")           520(6,"\033PO@AAAAAAAA=\")
521(6,"\033P2PAAAAAAAA1XAAAAAAAA4h")
522(6,"\033P2PAAAA1XAAAA4h")
523(6,"\033P2PAAAAAAAA1XAAAAAAAA4h")
524(6,"\033P2PAAAAAAAA1XAAAAAAAA4h")
525(0,"\033P?JO@AAAA4X1@AAAA6L")
526(6,"\033P2PAAAAAAAA1XAAAAAAAA4h")
527(6,"\033P=BO@AAAAAAAAAAAA=\")
530(6,"\033P?JO@AAAAAAAAAAAA4X1@AAAAAAAAAAAA41")
531(6,"\033P7YO@IIIIIIIII=D")    532(0,"\033P=AO@AAAAAAAA=C")
534(6,"\033PO@HHHHH4H")          536(6,"\033P>K1@IIIIIIII<A")
537(6,"\033P7JO@IIIIIIII7N")
```

expand_device_writer (xdw)

The expand_device_writer command is used to invoke the Expander to expand an expansion input file into an expansion output file. The language used in the expansion input file is described above.

SYNTAX AS A COMMAND

xdw {*path*} {*-control_args*}

ARGUMENTS

path
is the pathname of the expansion input file. The entryname of this file must have the suffix xdw, but the suffix need not be given in the command line. By default, the expanded expansion output file is written to a segment in the working directory whose name is formed by stripping the suffix from the input file entryname. Multi-segment files and the star convention are not supported. If no pathname is given, input may be given to the Expander by using the -input_string control argument described below.

CONTROL ARGUMENTS

-arguments ...
-ag ...
all remaining parameters in the command line are arguments to be passed to the file or input string being expanded.

-brief
-bf
does not display the expansion usage list when the expansion is complete. (Default)

-call *command_line*
if there are no errors in processing, executes the given command line when the expansion is complete.

-input_string *string*
-instr *string*
expands the given *string* as an expansion input file. By default, the expansion is displayed and no expanded output file is created.

-long
-lg
displays the expansion usage list when the expansion is complete.

-output_file *path*
-of *path*
writes the expanded output into the segment with the given pathname. This forces no_print even if -print is also given.

-no_print
-npr
    does not display the resulting expansion. This is the default when *path* is given and is forced when -output_file is given.

-print
-pr
    dislays (or prints) the resulting expansion. This is the default when -input_string is given and is mutually exclusive with -output_file.

format_document (fdoc)

The format_document command formats text segments. Output lines are built per the embedded control lines within the input file being formatted. Although the control lines are embedded within the input text, they do not appear in the output.

SYNTAX AS A COMMAND

fdoc *path* {*-control_args*}

ARGUMENTS

path
is the pathname of an input segment or multisegment file. The suffix fdocin *must* be the last component of the entryname; however the suffix need not be supplied in the command line.

CONTROL ARGUMENTS

-indent {*n*}
-ind {*n*}
indents the output *n* spaces from the left margin. This space is in addition to any indention established by the usage of the indent control line within the text of the input file.

-output_file {*path*}
-of {*path*}
directs the output to a file instead of to the user's terminal. If *path* is not given, then the output is written to an output file whose name is formed by replacing the suffix fdocin of the input file entry name with the suffix fdocout. (The default for this feature is *off*.)

-page_numbers
-pgno
ends each page with two blank lines and a centered page number. (The default for this feature is *off*.)

Formatter Description

Basically, the format_document command (a **text formatter**) takes an input file which was created using a text editor (e.g., ted, qedx and edm), formats that file, and either displays it on the terminal or writes it to a new file with a unique name. To direct format_document to perform certain actions, the user places special lines, called **control lines**, in the input file. All control lines begin with a period and *must* be on a line by themselves. The format_document command makes certain assumptions about how the document is to be formatted (i.e., when the format_document command executes, it **defaults** to certain conditions in the absence of user-specified control lines). It assumes that the output is going to be on standard-sized paper which has 66 lines per page and that the user wants the printed lines to be 65 characters wide. These values represent an 8-1/2 by 11 inch page with one-inch margins all around. It also assumes that the user wishes to have both the left and right margins lined up evenly like the margins of this paragraph. When the user wants format_document to do

something different than the standard defaults, the user must insert the necessary control lines in the input file to accomplish the desired results.

As discussed below, line filling is the moving of words from line to line to make the line size as near to the prescribed length as possible. A control break is an action that temporarily stops this process (i.e., it processes the previous line, the line just ahead of the break) and prints this line *as is*, even if it is a short one. All of the control lines, with the exception of .pdl and .pdw, cause control breaks. A blank line or a line that starts with a space also causes a control break.

Following is a summary of the control lines recognized by the format_document command.

.alb
(align both) puts extra spaces into each line so that both the left and the right margins are even. This control line is effective only if fill (.fin) is also in effect. (Default)

.all
(align left) does not put extra spaces into the lines. The left margin is even and the right margin is ragged. This control line is effective only if fill (.fin) is also in effect.

.fif
(fill off) retains lines in the output file as they are in the input file no matter how long or short.

.fin
(fill on) restructures the input file lines to the current line length for the output file by taking a word or words from the next line in order to fill the line as close as possible to the current line length. If a line in the input file is longer than the current line length, move a word or words to the next line, etc. (See the description of the .alb and .all control lines.) (Default)

.in {*n*}
(indent) sets the indention level. It is possible to have format_document indent each line a certain number of characters. If *n* is given with a plus or minus sign, then *n* is added to or subtracted from the current indention level. If *n* is given without a sign, then *n* becomes the indention level. An error message is displayed if an indention level is less than zero or greater than the line length. (The default indention level is 0.)

.pdl {*n*}
(page length) sets the page length. If *n* is given with a plus or minus sign, then *n* is added to or subtracted from the current page length. If *n* is given without a plus or minus sign, the page length is changed to *n*. The format_document command inserts blank lines at the top and bottom of each page, so be careful not to set the page length to a value less than 13 (or less than 14 if you are having page numbers printed.) An error message is displayed if a page length of less than the required lines is given. (The default page length is 66 lines.)

.pdw {*n*}
(page width) sets the page width (line length). If *n* is given with a plus or minus sign, then *n* is added to or subtracted from the current line length. If *n* is given without a plus or minus sign, the line length is changed to *n*. An

error message is displayed if the set line length does not accommodate the input file. (The default page width is 65 characters.)

.un {*n*}
(undent) sets the indention level for the output of the next line only. If *n* has a plus sign or no sign, the line is indented *n* characters less than the current indention level. If *n* has a minus sign, the line is indented *n* characters more than the current indention level. If this seems backwards, just remember that undent goes in the opposite direction from indent. An error message is displayed if the indention that is caused by undenting is less than zero or more than the line length.

EXAMPLE

The following is an example of a business letter created using the format_document command (fdoc). Suppose you are creating a business letter that is to be printed on a standard 8-1/2 by 11 inch piece of paper and that you want the lines to be 60 characters long. You first create the input file with a text editor. In this example the input file is labeled letter.fdocin. Line numbers are shown on the example for purposes of commentary immediately following the example. All of the numbered items below are user-entered data and are not flagged with a bullet as no system responses are included.

```
1   ted
2   a
3   .pdw 60
4   .fif
5   .in 35
6   9341 Millennium Lane
7   Reston, Virginia 22061
8   November 24, 1981
9   <NL>
10  <NL>
11  <NL>
12  .in
13  Zimmerman Widget Company
14  53698 Dixie Highway
15  Drayton Plains, Michigan 48999
16  <NL>
17  <NL>
18  Dear Sir,
19  <NL>
20  .fin
21  .un -5
22  I recently purchased one of your model GX-721 widgets.
23  I feel that your engineering staff deserves high
24  praise for this new model.  It is apparent
25  that a great deal of thought has gone into its
26  design.  I am particularly pleased with the optional
27  conetop replacement mechanism.
28  <NL>
29  .un -5
30  My purpose in writing this letter, however, is to
31  obtain information.  As you are well
32  aware, the filter requires a complete overhaul after
33  each 250 hours of use.  The service brochure indicates
34  that the nearest service center to my location is in
35  Chapel Hill, North Carolina, which is a six-hour drive
36  from my residence.  If you can direct me to a service
37  center that is more convenient to my location, I would
38  be grateful.
39  <NL>
40  <NL>
41  .fif
42  .in 35
43  Sincerely yours,
44  <NL>
45  <NL>
46  <NL>
47  <NL>
48  Michael P. Marley
49  \f
50  w letter.fdocin
51  q
```

line 1
Invokes the text editor.

line 2
Places the text editor in *append* mode.

line 3
Sets the line length (page width) to 60 characters. If this control is not present, then the line length would be set to 65 characters by default.

line 4
Turns fill mode "off". The reason for turning fill off is because the text beginning on line 6 through line 8 is an address. If fill mode was not turned "off" then the address would be reformatted by fdoc, words might be moved from line to line, or extra spaces might be filled in. You do not want this to happen, so you turn fill off. The same thing is done at line 43 just prior to the closing.

line 5
Sets the indention to character position 35. Text begins at column 1 unless you change it, and since the return address is to be on the right-hand side of the letter you must set the indention to the location desired (character position 35 in this case).

line 6-8
Return address.

line 9-11
Three blank lines are inserted by pressing the newline (NL) or carriage return (CR) key three times.

line 12
Resets the indention level to 0 (the absence of a number after the control results in a default to 0).

line 13-19
Address of the recipient, two blank lines, the salutation, and another blank line.

line 20
Turns fill mode "on" (fill was turned off by the control on line 4) as you want the body of the letter filled.

line 21
The indention level is set to 0 by the control in line 12, but you want to indent the next line (*and only the next line*) by 5 characters since it begins a paragraph. To change the indention for only one line you use the undent control which works in the opposite direction of the indent control. Undent subtracts the number from the indention (i.e., if you used .un 5 it would move the indention 5 spaces to the left). You want to move 5 spaces to the right to indent the paragraph, so you use a negative number.

line 22-40
This is the body of the letter. Notice that there has been no attempt to control the entered line lengths; it is entered free-form. The fdoc command formats all of the data for you, so long as fill mode is "on". Lines can be as short or as long as you wish, even if the lines wrap around (**wrap around** is the situation where the user continues entering data until the line on the terminal has reached the right margin, at which point the system moves the cursor to character position 0 of the

next line, and data entry is continued until a newline or a carriage return is entered).

line 41
Turns fill mode "off".

line 42
Sets the indention to character position 35 so that the letter closing, signature, and sender's name appear on the right side of the page (lines 45-50).

line 49
Terminates *append* mode and returns the user to *edit* mode.

line 50
Writes the buffer contents to permanent storage. In this case the buffer is stored in a segment identified as *letter.fdocin*.

line 51
Quits the editor and returns the user to Multics command level.

Now that your input file (`letter.fdocin`) is ready, you can have it formatted and printed on the terminal for your perusal.

```
!   fdoc letter.fdocin
```

9341 Millennium Lane
Reston, Virginia 22061
November 24, 1981

Zimmerman Widget Company
53698 Dixie Highway
Drayton Plains, Michigan 48999

Dear Sir,

I recently purchased one of your model GX-721 widgets. I feel that your engineering staff deserves high praise for this new model. It is apparent that a great deal of thought has gone into its design. I am particularly pleased with the optional conetop replacement mechanism.

My purpose in writing this letter, however, is to obtain information. As you are well aware, the filter requires a complete overhaul after each 250 hours of use. The service brochure indicates that the nearest service center to my location is in Chapel Hill, North Carolina, which is a six-hour drive from my residence. If you can direct me to a service center that is more convenient to my location, I would be grateful.

Sincerely yours,

Michael P. Marley

Assume the output looks good, and you are ready to make a final copy. Since your lines are 60 characters long and you are going to print it on standard 8-1/2 by 11 inch paper, and since most terminals and printers print 85 characters in 8-1/2 inches, you will want your letter to be centered on the paper. This is where the -indent control available within the format_document command comes into play. Your lines are 25 characters shorter than the width of the paper, so if each line begins at character position 12 (roughly half of 25) your letter will be centered on the page. The command line:

```
fdoc letter -indent 12
```

accomplishes this.

Let us say, for example, that you are going to save your letter in a file so that you can print it later on another terminal or on a high-speed printer. In such a situation, you would type

```
fdoc letter -indent 12 -output_file
```

The `-output_file` control argument saves the output in a file rather than printing it on your terminal. In this example, the file is named `letter.fdocout`. You can now use the dprint or print commands (see the command descriptions in *Multics Commands*) to print the letter.

If you have a high-quality printing terminal and wish to print this letter on a piece of typing paper, you would type:

```
print letter.fdocout -stop
```

After entering this command, place the typing paper in the terminal, position it so that printing begins at the top, and then enter a carriage return (newline character). The letter is then printed, stopping at the last line. At this point, you can remove the paper and put in a new sheet (if the letter is more than one page). When the letter has been printed, you can enter another carriage return, and you are back at Multics command level.

process_compout (pco)

The process_compout command is used to process one or more compose output (compout) files to an online device, or to a magnetic or punched paper tape. All or portions of the files may be requested.

SYNTAX AS A COMMAND

pco *paths* {*-control_args*}

ARGUMENTS

paths
are the pathnames of input files to be processed. The suffix compout must be the last component of the input file entrynames (but, see -pathname control argument below); however, the suffix need not be supplied in the command line. Output is produced in the order in which the pathnames are given in the command line.

CONTROL ARGUMENTS

any control argument specified in the command line applies to *all* input file pathnames given.

-files {*n*} {,*m*}
overrides either or both of the default output file factors when writing output to magnetic or paper tape. The default output file factors are found in the header record of the input file and are set from data in the device description table. *n* is the maximum number of pages per file and *m* is the maximum number of files allowed on the tape.

-from *n*
-fm *n*
starts printed output at page *n*. This control argument is mutually exclusive with the -pages control argument. The default value of *n* is "!1". See "Page Numbers" below for a discussion of page numbers.

-mode *xxx*
selects any of the known alternative modes of output or specifies an entirely new mode.

If *xxx* is a single word, then it may be any of the built-in modes shown below or the name of an output mode given in the header record of the input file. If it is a built-in mode, then the action described below is taken; if it is a known output mode, then output is prepared according to that mode; if it is neither, then an error message is displayed giving the names of all output modes in the input file header.

If *xxx* is a quoted string containing white space, then it is used as an attach description for the output medium irrespective of any modes specified in the input file header.

If this control argument is not given, the mode used is the first known mode found in the input file header; if there are none, output is written to the user_output I/O switch.

The built-in modes are:

comment
produces a listing of the comment information in the input file header. This comment contains output mode specifications, device setup information, print wheel identification for "diablo" class terminals, and possible other miscellaneous information. It is a copy of the "Comment" information in the device description table. (See "Magnetic Tape Header Files" below for a discussion of the information shown in this display.) For example:

```
!   pco ascii.walvip -mode comment

      **  From file: >udd>m>jaf>cdv>ascii.walvip.compout
    mode:tape=tape_ibm_ -bk 800 -nlb -den 800 -fmt f -mode ascii
    setup:*****  This tape created [date_time].  *****
    setup:This tape is to be sent via FedExp to
    setup:      Honeywell, Multics Computer Operations
    setup:      5115 N 27th Ave
    setup:      Phoenix, AZ 85017
    setup:
    setup:Font setup:
    setup:       1    2    3    4    5    6
    setup:  A- 187   145  108  409 2160  534
    setup:  B-  84    85   70   69 1716 1715
    setup:  C-8398     X   71    X    X    X
    setup:
    setup:Disposition of output:
    file:
    file:Document: [compout][ioa_ "Document: ^a" [compout]]
    file:       Return[compask "  Pasteup? " no= "yes= pasted up"]
    \c[compask "original/copy? " o=original original c=copy copy].
    file:  Cost center: [compask "  Cost center: "]
    file:       Send to: [compask "       Send to: "]
    file:  Destination: [compask "  Destination: "]
    file:     Comments: [   compask "  Comments: "]
    content_file: seg, tape


!   pco -mode comment

      **  From file: >exl>cd>doc>pco.compout
     Type Wheel Identification
     1 - 38101-01        PICA 10
     2 - 38510           APL 10
     3 - 38102-01        ELITE 12
```

?
displays all the "mode:" lines in the comment information.

setup
displays all the "setup:" and "file:" lines in the comment information.

display
produces a directory of files as would be written to the tape, followed by an interpreted display of all the files (see "Display Mode Interpretations" below). This output is written to the user_output I/O switch. For example:

```
!   pco ascii.walvip -mode display
    File information:
    Directory of files as would go on tape, by file number
       #              --- pageids present ---
       0       <ASCII information file>
    Document: ascii.walvip
       1         FRONT 1
       2               2

      **  From file: >udd>m>jaf>cdv>ascii.walvip.compout

       **** FILE #1 ****

    <US>=a1=p08=f120=14500=g=.00=k1=rx=tn3
    =tl09=ta06=tm06=8=il0000=j
    7.1<EM>81-05-26<EM>1614.8<EM>ascii.walvip<EM*2>1<QC><EL>
    <1/4>=d<EN><1/4><EL>
    .
    .
    .

       **** FILE #2 ****

    <US>=a1=p08=f120=14500=g=.00=k1=rx=tn3
    =tl09=ta06=tm06=8=il0000=j
    7.1<EM>81-05-26<EM>1614.8<EM>ascii.walvip<EM*2>2<QC><EL>
    <1/4>=d<EN><1/4><EL>
    =if090=a1=p10=f120=14500
    =il0000=a1=m120.<1/8>B6 --Sups--<QL><EL>
    =il0300X=b6 &()<SS>1,-.0123456789:;<1/4><SS>4<1/8>?=a1X=b6<QL><EL>
    =a1X=b6ABCDEFGHIJKLMNOPQRSTUVWXYZ!$=a1X=b6<QL><EL>
    =a1X=b6abcdefghijklmnopqrstuvwxyz =a1X=b6<QL><EL>
    -  <QL><EL>
    .
    .
    .
    =m120,=if990=if990=if990=il0000=14500=a1=p08<1/4>=d<EN><1/4><EL>
    =if090


    <US>=s
```

display -long
as for the display mode above but also showing the evaluated comment information as it would be written to the tape. For example:

```
!   pco ascii.walvip -mode display -long
    Document: ascii.walvip
     Pasteup? () ! no
    original/copy? () ! original
      Cost center: () ! J86
          Send to: () ! JFalksen
      Destination: () ! CRF
      Comments: () ! Do not cut, send in tube
    File information:
    *****  This tape created 06/05/81  0950.3 mst Fri.  *****
    This tape is to be sent via FedExp to
         Honeywell, Multics Computer Operations
         5115 N 27th Ave
         Phoenix, AZ 85017

    Font setup:
          1    2    3    4    5    6
      A- 187  145  108  409 2160  534
      B-  84   85   70   69 1716 1715
      C-8398    X   71    X    X    X

    Disposition of output:

    Document: ascii.walvip
           Return original.
      Cost center: J86
          Send to: JFalksen
      Destination: CRF
      Comments: Do not cut, send in tube

    Directory of files as would go on tape, by file number
       #              --- pageids present ---
       0       <ASCII information file>
    Document: ascii.walvip
       1        FRONT 1
       2              2

      **  From file: >udd>Doc>jaf>cdv>ascii.walvip.compout

       **** FILE #1 ****
    .
    ... as above
    .
```

dump
produces an octal/ascii dump of the records of the input file. For example:

```
!   pco pco.compout -mode dump
```

```
Record 0 374o 252
000116 000000000002 144151141142 154157040040 040040040040 ....diablo
000122 040040040040 040040040040 040040040040 040040040040
000126 040040040040 144164143063 060060163040 040040040040     dtc300s
000132 040040040040 040040040040 040040040040 040040040040
000136 040040040040 144164143063 060060163040 040040040040     dtc300s
000142 040040040040 040040040040 040040040040 040040040040
000146 040040040040 777777777777 777777777777 777777777777     ............
000152 000000000000 000000000007 033033033006 033040060040 ............ 0
000156 000000000166 040124171160 145040127150 145145154040 ...v Type Wheel
000162 111144145156 164151146151 143141164151 157156012040 Identification.
000166 061040055040 063070061060 061055060061 011120111103 1 - 38101-01.PIC
000172 101040061060 012040062040 055040063070 065061060011 A 10. 2 - 38510.
000176 101120114040 061060012040 063040055040 063070061060 APL 10. 3 - 3810
000202 062055060061 011105114111 124105040061 062012104102 2-01.ELITE 12.DB
000206 072040144164 143063060060 163137167162 151164145162 : dtc300s_writer
000212 137044144151 163160154141 171012040040              _$display.

Record 1 5270o 2744
000000 061040040040 040040040040 040040040040 040040040040 1
000004 040040040040 040040040040 040040040040 040040040040
000010 200000000000                                        ....
   17 RAW HALT 2 preface 600000000000
000011 744000000000 000000000021 007011177177 177177177177 ................
000015 177177177177 177177177177 177000000000             ............
 2671 RAW 000000000000
000020 600000000000 000000005157 033040066012 012012137137 .......o. 6...__
000024 137137137137 137137137137 137137137137 137137137137 ________________
000030 137137033124 055137137137 137137137137 137137137137 __.T-____________
000034 137137137137 137137137137 137012012015 177160162157 __________....pro
000040 143145163163 137143157155 160157165164 054040160143 cess_compout, pc
000044 157033124055 177160162157 143145163163 137143157155 o.T-.process_com
000050 160157165164 054040160143 157012015137 137137137137 pout, pco..____
000054 137137137137 137137137137 137137137137 137137137033 ________________.
000060 124055137137 137137137137 137137137137 137137137137 T-______________
000064 137137137137 137137012012 012015033040 066120122117 _______..... 6PRO
000070 103105123123 137103117115 120117125124 054040120103 CESS_COMPOUT, PC
000074 117033040066 012012033124 004124150145 040033120065 O. 6...T.The .P5
000100 120033160162 157143145163 163137143157 155160157165 P.process_compou
000104 164040033120 065120033143 157155155141 156144040033 t .P5P.command .
000110 120065120033 151163040033 120065120033 165163145144 P5P.is .P5P.used
000114 040033120065 120033164157 040033120065 120033160162  .P5P.to .P5P.pr
000120 157143145163 163040033120 065120033157 156145040033 ocess .P5P.one .
000124 120065120033 157162040033 120065120033 155157162145 P5P.or .P5P.more
000130 012015143157 155160157163 145040033120 065130033157 ..compose .P5X.o
000134 165164160165 164040033120 065130033050 143157155160 utput .P5X.(comp
000140 157165164051 040033120065 130033146151 154145163040 out) .P5X.files
000144 033120065130 033164157040 033120065130 033141156040 .P5X.to .P5X.an
000150 033120065130 033157156055 154151156145 040033120065 .P5X.on-line .P5
000154 130033144145 166151143145 054040033120 065130033157 X.device, .P5X.o
000160 162040033120 065130033164 157040033120 065130033141 r .P5X.to .P5X.a
000164 012015155141 147156145164 151143040157 162040160165 ..magnetic or pu
000170 156143150145 144040160141 160145162040 164141160145 nched paper tape
```

-pages *n*|*n,n* ...
-pgs *n*|*n,n* ...
specifies a blank-separated list of selected pages to be printed. Each member of the list must be a single page, {*n*}, or a range of pages, {*n,n*}. The page numbers given must normally increase without duplication (however, see the discussion of page numbers in "Page Numbers" below). At least one page must be specified. This control argument is mutually exclusive with the -from and -to control arguments. The default for this feature is *off*.

-pages_changed
-pgc
specifies that only addendum pages and those pages containing text within the range of a change-bar control (from the pages specified by the "-pages" or "-from" and "-to" control arguments, if given) are to be printed.

-pathname *path*
-pn *path*
is the pathname of an input file even though it may have the appearance of a numeric parameter or a control argument, or is a compose bulk output file that does not have the suffix compout.

-stop
-sp
waits for a newline character (ASCII code 012) from the user before beginning each page of output. If only a newline is typed, the next page is printed. If "q" is typed, the command invocation is terminated gracefully. If "r" is typed, the page just printed is reprinted. The default for this feature is *off*.

-table
-tb
print a table listing information about all selected pages in the file. Only a table is produced. This control argument is mutually exclusive with all others. See "Table Option Output" below for further information.

-to *n*
ends output after page *n*. This control argument is mutually exclusive with the -pages control argument. The default value for *n* is "$". See "Page Numbers" below for a discussion of page numbers.

-volume *xx*
writes the output to the magnetic tape whose volume name is *xx*. The parameters needed for attaching the tape are provided by the device description table and are contained in the header record of the compout file. The attach descriptions in the file header may be selected or overridden by using the -mode control argument described above.

-wait
-wt
waits for a newline character (ASCII code 012) before beginning the first page of output to the terminal, but not between pages (see the -stop control argument above). The default for this feature is *off*.

Page Numbers

Pages may be referred to by several methods:

!n
means the *n*'th physical page in the file. However, *n*=0 means "go back to the beginning of the file". No page is printed for the !0.

$
means the last physical page in the file.

+n
means the *n*'th relative (to the last page referenced) page in the file.

$-n
means the *n*'th relative page from the end of the file.

page-id
is the page number constructed by compose. It may be just a simple number or a compound number such as "B-1", "3-14.2", or "i-5". The *page-id* must be an exact match for that in the document. No less than/greater than checking is possible.

A page selection could be:

```
-pages !1,!5 +19,+2 127.4,127.42 $-1,$
```

This means to process four ranges: the first five pages, then the three pages beginning with the 24th, then the 39 pages beginning with page number 127.4, then the last 2 pages of the file.

Table Option Output

The -table option prints a table of information about selected pages in a file. This information includes the physical page number and the actual page-id. Any "changed" pages are marked with the word "CHANGED". Front pages are flagged with the word "FRONT". Intentionally blank pages are flagged with the word "BLANK" and missing pages in the front/back sequence are indicated.

The format of this table is such that it can be used as a control file, either directly or after editing to remove any unwanted data. For example:

```
!   pco flow_sheet -to !7 -table

    -pathname flow_sheet -pages
    !1          /*          FRONT      1-5 */
    !2          /*                     1-6         CHANGED */
    !3          /*          FRONT      1-6.1       CHANGED */
    !4          /* BLANK               1-6.2       CHANGED */
    !5          /*          FRONT      1-7 */
                /* blank   back not supplied */
    !6          /*          FRONT      2-1 */
    !7          /*                     2-2 */
    ;
```

Magnetic Tape Header Files

The first file on a magnetic tape (noted as file 0 in the "display" output) is the ASCII-coded, evaluated comment information for all the document files on the tape. This information is obtained from the device description table and is created by use of the Comment: global statement in the device description file (see "Device Table Compiler" in Appendix C and the **"comment"** example listed under the control argument "-mode" above.) The comment may contain any information, but the following lines are important to process_compout.

Note: In the discussion below, *xxx* denotes a dynamic string that may have various values depending on its context and the use of opening and closing brackets ([]) in the string indicates that the enclosure is evaluated as an active function and is replaced with the string returned.

mode: *name=xxx*

defines an output mode, *name*, whose attachment is *xxx*. *name* is the mode name given with the -mode control argument. The first mode present is the default mode. If any others are given, they represent alternatives available. A mode defines a method of transcribing the formatted output onto some output medium.

xxx may be tape_ibm_ or tape_ansi_, each with an attach description acceptable to iox_ (see Multics Subroutines), or either of the following:

online

output is written to the user_output I/O switch. This is the default if no modes are given.

punch 6

writes 6-level TTS (or reverse TTS) code to the user_output I/O switch, assuming that the user has an appropriate paper tape punch on the terminal.

Note: An ASCII Standard 7-level paper tape punch may be used if the upper two punch channels are disabled.

leader: *xxx*

xxx is punched out in "big" letters on the paper tape leader after the file identification. Its purpose is to convey machine setup information to the typesetter operator when paper tape is the output medium. This information has no effect on magnetic tape output.

setup: *xxx*
all or part of the information necessary to process a magnetic tape on the target machine. This information may include machine setup and disposition of the tape. This keyword line may be given any number of times and the information appears only once per tape.

file: *xxx*
all or part of the information necessary for the processing of each file on a magnetic tape. This information may include the handling and disposition of the generated output from each file. This keyword line may be given any number of times and the information is repeated for each file.

content_file: {^}seg, {^}tape
controls the disposition of the "contents" file as created from the comment information discussed above. The parameters are:

seg,^seg
do/do not create the segment *vol-id*.contents in the working directory. *vol-id* is the magnetic tape volume identifier given with the -volume control argument. The segment cannot be created if no identifier is given.

tape,^tape
do/do not write the contents file as the first file on the tape.

The default is "seg,^tape" if this keyword line is not given.

pack: *spec* {,|; *spec*} ...
controls the format of the data bytes written to magnetic tape. Ordinarily, data bytes are written as normal ASCII or EBCDIC characters (depending on the tape attachment). However, because of the front-end processors on some target machines, it may be necessary to use a non-standard recording mode.

spec
may be any of the following "micro-ops":

c *n*
copy the next *n* bits of the current input byte (in the compout file) to the output data byte.

f *n*
move forward *n* bits in the current input byte (in the compout file).

b *n*
move backward *n* bits in the current input byte (in the compout file).

'*b...*'
a literal bit string of any desired length to be added to the output data byte. (The single quote is used here because the Comment: statement requires the use of the double quote.)

,
micro-op separating delimiter implying use of the current input data byte (in the compout file) for the next micro-op.

;
micro-op separating delimiter implying use of the *next* input data byte (in the compout file) for the next micro-op.

In the above, *n* is a single decimal digit and it is an error to give a value of *n* or a literal bit string that results in a bit position outside the limits of either the input or output data bytes.

Examples:

(The compout file always contains one output character per 9-bit byte stored as the low-order bits.)

TTS code is to be written to 7-track tape with the third tape channel set to binary 0.

```
f3,c2,'0',c4
```

TTS code is to be written on 9-track tape in packed mode (3 characters per 2 tape bytes).

```
f3,c6,f3,c3;f3,c3,c6
```

[compout]
when used in a file: keyword line, this active function is replaced with the name of the current compout file with the suffix removed; it is an error to use it anywhere else.

[compask *question* {*responses*}]
this active function is replaced with the response from the user to the given *question*. (See the comment and display examples listed under the control argument "-mode" above.)

question
any question or prompt string. It is written to user_output and processing waits for a response from the user. Any existing response to the question is shown in parentheses. The responses to all questions are initialized to empty strings when the processing of the comment information begins.

responses
the valid responses to the question, including any translation of responses to other strings. There may be any number of valid responses and, if none are given, any response is used as typed. Any invalid response causes the question to be repeated. Referring specifically to the "Pasteup?" question in the comment example listed under the control argument -mode above, only "yes" and "no" are valid with a "no" response being translated to an empty string and a "yes" response being translated to "pasted up". For the "original/copy?" question in the same file: keyword line, four responses are valid with the two short forms being translated to the long forms.

If only a newline (NL) is typed, the existing response is used.

If a period (.) is typed, the existing response is reset to an empty string and is used.

For example (all on one line in a compout comment string):

```
file:Return [compask "" Pasteup? "" no= ""yes= pasted up""]
  [compask ""original/copy? "" o=original original c=copy copy].
```

If two files are being processed, it causes the interaction:

```
Pasteup? () yes
```

```
original/copy () o
```

and generates the following lines:

```
Return pasted up copy.
Return original.
```

Note that the pasteup question returns a null result if the answer is no and the string "pasted up" if the answer is yes. The yes response must be quoted because it contains SPs. The other question allows the user to reply "o" instead of having to type "original", yet the result is the more meaningful whole word.

Display Mode Interpretations

The meanings of the coded symbols that may be found in the display mode output are defined below.

Each symbol may be followed by a parenthesized word (or words) that indicates the devices in whose output the symbols may be found. If no word is shown, then the symbol may appear in the display output for all devices. The words used are:

type	monospace typewriter terminals
print	line printers
diablo	Diablo-like incremental terminals
photo	phototypesetters
hyterm	Diablo 1620 and 1720 series terminals
dtc300s	DTC300 series terminals

A circumflex (^) appearing before the word is a logical "not" and means "all except".

All Honeywell-supplied display routines validate the data being displayed. Whenever a string is encountered that meets certain criteria that make it appear to be a string with known syntax, it is validated against that suspected known syntax. If any errors are discovered in the validation, one or more indented error messages are inserted into the display; the first such error message being flagged with three asterisks "***". Normal display output resumes on the next line starting with the character that triggered the validation check. When the interpretation of the entire display input string (as little as a single word or as much as an entire output page) is complete and errors have occurred, an error count message is given. For example (from a file for a Mergenthaler V-I-P typesetter):

```
<US>=a1=p10=f120=14500
=11500=m630.This is an example for use with =a3pco<EL>
=aldocumentation. It is doctored up after<EL>
composition to cause errors for the<EL>
purpose of the display.<QL><EL>
=m570.
*** 2nd char SHIFTed
    3rd char SHIFTed
=1L0000=14500=p06-=d-<EL>
=if090


<US>=s
```

```
*** Total errors: 1
```

<BSP>, <BSP*n> (^photo)
a backspace (ASCII code 010) or a string of *n* backspaces

<CR> (^photo)
a carriage return (ASCII code 015) - return to left margin without a line advance

<DT-n> (diablo)
a direct or absolute tab to position *n*

<ESC>
an "orphan" escape character - the character following is not in any escape sequence known to the display routine

<FF> (print)
a formfeed (ASCII code 014) - a page eject

<HMI *n*> (diablo)
set horizontal motion increment (space implied by space and backspace) - *n* depends on device resolution

<HMI *n*/120> (hyterm)
special form of horizontal motion increment for the hyterm device

<HT> (^photo)
a horizontal tab (ASCII code 011)

...<LF> (diablo)
a linefeed (ASCII code 012) at the end of a text line - does *not* imply a return to the left margin

<LF>, <LF*n> (diablo)
on a line by themselves, a linefeed or a string of *n* linefeeds - an indication of white lines on the page

...<NL> (type, print)
a newline (ASCII code 012) at the end of a text line - *does* imply a return to the left margin

<NL>, <NL*n> (type, print)
on a line by themselves, a newline or a string of *n* newlines - an indication of white lines on the page

<*ooo*>
octal value of any non-printing character not known to the display routine as a control character

<PLC"*c*">, <PLC"*ooo*"> (dtc300s)
set plot character - first form if character is printable, second if not

<PLT>, <SPLT>, <^PLT> (diablo)
"plot", "superplot", and "unplot", respectively - appear only in error messages (see "Plot Strings" below)

<PLT[...]> (diablo)
a plot string - everything between the brackets is an interpreted form of the plot control data (see "Plot Strings" below)

<PLT[...]CR> (hyterm)
a plot string terminated by a carriage return (see "Plot Strings" below)

<PLT{...}> (dtc300s)
a superplot string - everything between the braces is an interpreted form of the superplot control data (see "Plot Strings" below)

<VMI*n*> (diablo)
set vertical motion increment (space implied by linefeed) - *n* depends on device resolution

<...>
any control function name unique to a particular device. Among these are "QL" and "STOP" shared by almost all typesetters, "EL" (for elevate) for Mergenthaler V-I-P typesetters, and "CUT" for most phototypesetters.

PLOT STRINGS

The symbols appearing in plot strings (see "PLT" symbols above) depend on the output device and the plot mode in which it is currently operating. The Honeywell-supplied display routines support two plot modes: "plot" and "superplot" as defined below.

Plot Modes

plot mode

the horizontal and vertical increments are automatically set (by the device hardware) to the minimum (or defined) values. Plot strings are given as sequences of the four motion characters (space, backspace, linefeed, reverse-linefeed) and any selected plot character.

superplot mode

a plot character is specified, possible "pen control" is given, and the horizontal and vertical motion increments are chosen. Plot strings are given as sequences of two defined characters (usually "X" and "Y") with the plot character being printed automatically after one of them. In most devices, both the plot character and the increments may be changed without leaving and reentering plot mode.

Plot Mode Symbols

"*c...*"
the literal text string *c...*

M, *n(M)*
one or *n* occurrences of a "motion" sequence *M* that contains (in order) a horizontal motion, a vertical motion, and the plot character. Any two of the three parts may be absent. The motion directions are those of the print head relative to the paper.

The horizontal motion may be:

r	one increment to the right
l	one increment to the left
HT	ten *columns* to the right

The vertical motion may be:

d	one increment down
u	one increment up
d/2	a half-line down
u/2	a half-line up

Superplot Mode Symbols

Superplot strings contain control sequences and motion sequences. The sequences may contain two or more characters and are separated by a colon (:).

Control Sequences

A control sequence always appears first in the plot string and contains a pen control character and a quadrant selector.

The pen control character may be:

d	pen down or print
u	pen up or no print

The quadrant selector may be:

1	upper right
2	upper left
3	lower left
4	lower right
11	upper right, doubled
22	upper left, doubled
33	lower left, doubled
44	lower right, doubled

"doubled" means that the values given in the motion sequences following are doubled; that is, the superplot strings `{dl:x2}` and `{dll:xl}` are equivalent.

Motion Sequences

M, *n(M)*
: one or *n* occurrences of a "motion" sequence *M* that contains (in order) a horizontal motion and a vertical motion. Either of the parts may be absent. The motion directions are determined by the quadrant chosen in the control sequence.

The horizontal motion is:

x*n*	*n* increments

The vertical motion is:

y*n*	*n* increments

Extraneous Characters

Any characters appearing in superplot strings that are not known to the display routine as superplot characters are displayed as *c* if they are printing characters or *ooo* if they are not.

SECTION 4

WORDPRO DICTIONARIES

DICTIONARY USE

WORDPRO dictionaries are used to perform hyphenation and detect spelling errors and inconsistencies in word usage. A standard WORDPRO dictionary is provided as part of the WORDPRO system, and is easily located by any user. However, the standard dictionary can be augmented or replaced by the addition of user-supplied dictionaries to the dictionary search list used by the WORDPRO system. Search lists provide a quick way of letting Multics know where to find needed segments, (i.e., dictionaries). Throughout this section, this list of dictionaries is referred to as the *dict search list*. The search facility commands contain information which is unnecessary for users performing the standard dictionary-related functions. (The search facility commands are fully described in the *Multics Commands*.)

Standard WORDPRO Dictionary

The initial version of the standard WORDPRO dictionary contains about 30,000 words.

Note: The standard WORDPRO dictionary is at the present time a preliminary version, to be updated and revised at a future date.

Hyphenation points are selectively compiled from the most commonly used method of dividing words, since published dictionaries do vary. Syllables are specified for all words that are able to be hyphenated except homographs (i.e., words with identical spellings but different meanings and often different pronunciation as in the verb resume (to begin again) and the noun resume (a summing up)). Homographs are excluded since the differences in pronunciation affect hyphenation. In general, no technical, scientific, medical, foreign language, or other specialized words are included in this dictionary.

User-Supplied Dictionaries

Commands are provided that enable the user to create and maintain personal or private dictionaries (see "Summary of Commands" below). For each word entered into a dictionary, the user can specify the desired hyphenation points and whether the word should be trimmed (see "Unwanted Words" below for an explanation of trimming and the no-trim attribute).

In order for the commands described in this chapter to refer to a user-supplied dictionary it is necessary to include the dictionary in the dictionary search list. To do this, it is necessary to invoke either the add_search_paths or the set_search_paths command (refer to *Multics Commands*). Because setting the dictionary search list is only effective for the life of a process, it is beneficial to include the command in the user's start_up.ec.

Dictionary Files

Dictionaries are stored in dictionary files. A dictionary file can be recognized by its entryname, which always includes the suffix dict (e.g., new_words.dict). Dictionary files are not suitable for printing. The list_dict_words command lists the contents of a dictionary file.

HYPHENATION

Hyphenation is simply the division of a single word into two parts separated by a hyphen. The Multics WORDPRO compose command can perform hyphenation automatically for the user whenever a word occurring at the end of a line does not fit in the space remaining on that line.

When Hyphenation Is Needed

The process of composing text often makes hyphenation necessary in order to avoid unsightly output. If the user has requested justification of text on a line, the padding of the line with blanks sometimes generates sparsely-filled lines containing an unacceptable amount of blank space. If the user has not requested justification, then the hand margin often becomes very ragged.

Since hyphenation allows more characters to be put on a line, it can help solve both of these problems. Adding more characters to a line reduces the text padding needed when justification is performed. When justification is not performed, hyphenation results in smoother right-hand margins.

Hyphenation Problems Solved by WORDPRO

The many problems of hyphenation are solved by WORDPRO through the cooperation of the compose command and the WORDPRO dictionaries. The main hyphenation task of the compose command is to identify when hyphenation is required and then to obtain the correct hyphenation from the dictionaries. The compose command also provides the following additional hyphenation capabilities:

- the ability to explicitly turn hyphenation on or off within a document
- the ability to determine the number of blank spaces on a line after filling with complete words, and then whether extra spaces (padding) or hyphenation of a word at the end of that line is suitable

WORDPRO Hyphenation Technique

WORDPRO uses a **multiple-dictionary hyphenation technique**; i.e., it searches through more than one dictionary looking for the correct hyphenation point for a word. A user may specify an ordered list of dictionaries to be searched to ascertain the hyphenation points of words requiring hyphenation. This list of dictionaries is specified in a special list called the **dict search list**. The default dict search list specifies one dictionary, the standard WORDPRO dictionary (>unb>standard.dict).

An important point about the multiple-dictionary technique is that the dictionaries are searched in order. If the word being hyphenated is not found in the first

dictionary searched, then the next dictionary in the list is searched. When a word is found in a dictionary, it is hyphenated as specified in that dictionary. No further searching for that word is performed. If a word is not found in any of the specified dictionaries, then that word is not hyphenated. It is by this method that users can specify their preferred hyphenation and have it override any other division.

The multiple dictionary technique offers the WORDPRO user viable solutions to the problems of hyphenation. A list of the most important advantages of this technique is presented below:

- Hyphenation is automatic (i.e., if turned *on* in the compose invocation). No terminal operator interaction is required. In situations where output is directed to a file, or the composing of a document is performed by an absentee process, any requirement for online human interaction is unacceptable. It is also unlikely that any terminal operator can quickly and consistently make accurate hyphenation decisions.

- No complicated hyphenation rules are used. It is easy to understand how WORDPRO performs hyphenation. As stated above, a word is hyphenated as specified in the first dictionary in which it is found.

- Each user has complete control over hyphenation. Any Multics user may specify a private set of WORDPRO dictionaries. There are two reasons why this attribute is valuable for users. The first is that many published dictionaries disagree on preferred hyphenation. Second, different users have different preferences in hyphenation, especially how to hyphenate words that are already hyphenated (e.g., non-European -- should it be hyphenated only at its true point of hyphenation, any hyphenation point, or not at all?). By adding a private dictionary to the beginning of the dict search list, a user may specify how words contained in that dictionary are hyphenated. If a user does not like the hyphenation specified for a word in the standard WORDPRO dictionary, then that word may be added to a private dictionary and hyphenated differently or not at all. If a user wants a word hyphenated that is not contained in the standard WORDPRO dictionary, then that word may be added to a private dictionary.

- The problem of homographs can be solved. In the standard WORDPRO dictionary no hyphenation is specified for homographs. The reason is that it is better not to hyphenate a word than to hyphenate it incorrectly. A user who wants a particular instance of a homograph hyphenated can specify the hyphenation within the document, the only place that the correct hyphenation is known. This can be done by using a compose control to make external calls to the search list and dictionary commands. For example, calls can be made to add a temporary dictionary to the dict search list, to add the homograph to this dictionary, and then to delete the temporary dictionary from the dict search list after the homograph is processed by compose.

- The multiple dictionary technique fits the needs of all users. For the average user who does not demand specialized vocabulary, the standard WORDPRO dictionary alone is sufficient. No knowledge of dictionaries or search lists is required. Other users may have a second or third dictionary set up for use each time they log in and thus they also need not be concerned with how this technique works. However, other users may require absolute and dynamic control over hyphenation. For these users, absolute and dynamic control is available.

SPELLING ERRORS

WORDPRO provides tools that help detect and correct spelling errors or inconsistencies. The ability to eliminate misspelled, mistyped, and unwanted words from a document provides WORDPRO users with the means to consistently produce quality documents.

Spelling Error Detection

The WORDPRO technique used to detect spelling errors consists of three steps:

1. Make an alphabetized list of all unique words contained in a document. This list is called a wordlist.
2. Remove from the wordlist all words that can be found in a dictionary or set of dictionaries. This operation is called trimming.
3. Print the remaining words in the wordlist and check them for spelling errors.

The words printed in step 3 are those words contained in a document that could not be found in a dictionary and thus are likely to be misspelled. Normally, for a document containing perhaps thousands of words, only a small number of these words need to be checked. This makes spelling error detection fast, simple, and thorough. The search for a word in a dictionary (step 2 above) is performed in the same way as described for hyphenation (multiple dictionaries can be used). As with hyphenation, the list of dictionaries to be used is specified in the dict search list. Thus the default dictionary used for spelling error detection is the same standard WORDPRO dictionary used for hyphenation.

A separate command is provided to perform each of the three steps. However, these three commands can be combined into a single operation, if desired, by use of the Multics exec_com facility (described in *Multics Commands*).

Unwanted Words

In addition to the need to identify misspelled words in a document, there is the need to identify unwanted words. An unwanted word may be some normally-acceptable, correctly-spelled word that should not appear in a particular document (at least not without a careful check of the context in which the word is used).

An example of an unwanted word is the word "basic" within Honeywell Multics documentation. This word, when used in computer documentation, may be confused with the command used to invoke the BASIC compiler. Thus every instance of this word in a Honeywell Multics document must be identified and approved.

The problem with unwanted words is that they are very likely contained in the standard WORDPRO dictionary and thus are deleted from any list of misspelled words. It is not acceptable to solve this problem by requiring a user to maintain a private dictionary containing all of the words in the standard WORDPRO dictionary except the few words that are unwanted.

Instead, it is possible to add a word to a dictionary (presumably a private dictionary that is searched before the standard dictionary) and to specify that this word is unwanted and therefore should not be trimmed (deleted) from any list of words by the trim_wordlist command. This no-trim attribute is denoted by preceding the word with the circumflex character "^" (ASCII code 136). If a word is actually spelled with

a leading circumflex, then that circumflex must be followed by another circumflex or an equal sign.

The sequence "^^" preceding a word indicates that the word contains a leading circumflex and has the no-trim attribute. The sequence "^=" preceding a word indicates that the word contains a leading circumflex and does not have the no-trim attribute.

Wordlist Segments

Wordlists are stored in wordlist segments. A wordlist segment can be recognized by its entryname, which always includes the suffix wl (e.g., new_words.wl). A wordlist segment contains a sequence of words each separated by a newline character; thus, if a wordlist segment is printed, each line contains exactly one word. A command is provided to print a wordlist in a multiple-column format (see the print_wordlist command below).

Spelling Error Correction

When a misspelled word is detected as described above, it can be easily corrected using WORDPRO tools. The revise_words command is provided to revise all instances of specified misspellings within a document. It is not necessary for the user to locate the misspellings within the document or to know how many times each misspelling occurs.

For most errors in spelling or consistency, the intended word can be recognized from the wordlist. In some cases, however, the intended word may not be recognizable without examining the context. For this purpose, the locate_words command is provided to locate all occurrences of a given misspelling or unwanted word within a document and to print all lines in which these words appear. For each occurrence, it is possible to print not only the containing line, but surrounding lines as well.

add_dict_words (adw)

The add_dict_words command is used to add words to a WORDPRO dictionary.

SYNTAX AS A COMMAND

adw *path* {*words*} {*-control_args*}

ARGUMENTS

path
is the pathname of the dictionary to which the words are added. If *path* does not have the suffix dict, one is assumed; however, dict must be the last component of the dictionary segment name. If the dictionary does not exist, it is created.

words
are words to add to the dictionary. At least one *word* is required unless -input_file is specified. If a word is already in the dictionary with the same hyphenation and no-trim attribute, the word is ignored without comment (see "Notes" below). However, if the word is already in the dictionary with different hyphenation or no-trim attribute, then a warning is issued and the dictionary word is left unchanged.

CONTROL ARGUMENTS

-count
-ct
reports the number of words added and the total number of words in the dictionary.

-force
-fc
allows a word already in the dictionary to be replaced. This feature may be used to change the no-trim attribute or hyphenation of a word in the dictionary.

-input_file *path*
-if *path*
adds to the dictionary words contained in the segment specified by *path*. Words in this segment must be separated by newlines. This control argument may be specified more than once.

-raw
suppresses the special interpretation otherwise given to hyphen and circumflex characters (see "Notes" below).

-word *string*
adds the word *string* to the dictionary even though *string* may look like a control argument.

NOTES

The correct hyphenation of a word can be specified when it is added to the dictionary. Embedded hyphens indicate the hyphenation points. If no hyphenation points are specified, it is assumed that the word cannot be hyphenated. If a word is spelled with a hyphen, then that hyphen must be followed by another hyphen or an equal sign. The character sequence "--" indicates that the word contains a hyphen and that hyphenation may be performed at (after) the hyphen. The character sequence "-=" indicates that the word contains a hyphen, but the word may not be hyphenated at the hyphen.

If the -raw control argument is specified, no special interpretation is given to either hyphen or circumflex characters. Each such character found within a word is taken literally as a part of the word. Therefore, words added with the -raw control argument cannot have the no-trim attribute or hyphenation points.

Maximum word size is limited to 256 "literal characters". Only characters contained in the normal spelling of a word are literal characters. Thus, the special sequences "--" and "-=" both represent the single literal character "-". Literal hyphens may appear anywhere within a word. Hyphenation points, however, may not appear beyond the 33rd literal character of a word.

EXAMPLES

To add the word *test* to the dictionary good_words.dict in the working directory (it is not hyphenated), type:

```
adw good_words test
```

To add the word *example* to the dictionary Webster.dict in the user's project directory (it is hyphenated at the specified hyphenation points), type:

```
adw >udd>Project_id>Webster ex-am-ple
```

To add the word *basic* to the dictionary my_words.dict (it is hyphenated, but it is not trimmed), type:

```
adw >my_words ^bas-ic
```

To add the word *in-house* to the dictionary good_words.dict (it is hyphenated at the point of its actual hyphen), type:

```
adw good_words in--house
```

To add the word *co-star* to the dictionary good_words.dict (it is not hyphenated), type:

```
adw good_words co-star -raw
```

To add the word *right-hand* to the dictionary good_words.dict (it is not hyphenated), type:

```
adw good_words right-=hand
```

To add all words in the segment new_words to the dictionary good_words.dict, type:

```
adw good_words -input_file new_words
```

count_dict_words (cdw)

The count_dict_words command prints the number of words contained in a specified dictionary.

SYNTAX AS A COMMAND

cdw *path*

ARGUMENTS

path
is the pathname the dictionary. If *path* does not have the suffix dict, one is assumed; however, dict must be the last component of the dictionary segment name.

create_wordlist (cwl)

The create_wordlist command produces an alphabetized list of all distinct words found in the specified text segment. This list is saved in a wordlist segment that is created in the working directory. The wordlist segment is given the entryname of the text segment with a suffix of wl added. The total number of words in the text segment and the number of words put into the wordlist segment are displayed.

SYNTAX AS A COMMAND

cwl *path* {*-control_args*}

ARGUMENTS

where *path* is the pathname of the text segment.

CONTROL ARGUMENTS

-brief
-bf
suppresses the display of the total number of words in the text segment and the number of words put into the wordlist segment.

-from *n*
-fm *n*
words are processed in the text segment starting from the line number specified by *n*. If this control argument is not specified, then the text segment is processed starting from the first line.

-header
-he
displays the pathname of the text segment.

-no_control_lines
-ncl
suppresses the display of the control lines (i.e., lines that begin with a period).

-no_exclude
-ne
specifies that words containing only special characters or punctuation are not to be excluded from the wordlist (see "Notes" below).

-no_sort
-ns
specifies that the words in the wordlist segment are not to be sorted into alphabetical order. They are put into the wordlist segment in the order in which they are found in the text segment and duplications are not eliminated. (This control argument is intended for special application and should not be used for normal wordlist segment creation.)

-to *n*
words are processed in the text segment up to and including the line number specified by *n*. If this control argument is not specified, then the text segment is processed to the last line.

NOTES

Words in the text segment are separated by the following delimiter (white space) characters:

space
horizontal tab
vertical tab
newline
form feed

Punctuation characters are removed from the word. The characters "([{ are removed from the *left* side of the word. The characters ")]}.,;:?! are removed from the *right* side of the word. Also, PAD characters (octal 177) are removed from the *left* side of the word.

Additional special processing is performed on each word after all punctuation is removed. A summary of this special processing is given below:

- if the entire word is underscored, then the underscores are removed. If only part of a word is underscored, then the underscores remain.
- if the word contains no letters, i.e., consists entirely of punctuation characters or other special characters, then the word is excluded from the wordlist. The -no_exclude control argument disables the automatic exclusion of such words.

EXAMPLES

The table below shows examples of how punctuation is removed from a word and how special processing is performed.

WORD BEFORE PROCESSING	WORD IN WORDLIST
example	example
"example"	example
example.)	example
{example}	example
example	example
exam.ple	exam.ple
)example(	)example(
1000	(trimmed)
1,000	(trimmed)
7-5.2	(trimmed)
+1	(trimmed)
1/2	(trimmed)
1A	1A
1(2)	(trimmed)

delete_dict_words (ddw)

The delete_dict_words command deletes one or more words from a WORDPRO dictionary.

SYNTAX AS A COMMAND

ddw *path* {*words*} {*-control_args*}

ARGUMENTS

path
is the pathname of the dictionary. If *path* does not have the suffix dict, one is assumed; however, dict must be the last component of the dictionary segment name.

words
are words to be deleted from the dictionary. At least one *word* is required unless -input_file is specified (see below). If a word is not found in the dictionary, a warning message is issued.

CONTROL ARGUMENTS

-brief
-bf
suppresses the warning message usually given when a word is not found in the dictionary.

-count
-ct
reports the number of words deleted and the number of words in the dictionary.

-input_file *path*
-if *path*
deletes from the dictionary the words contained in the segment specified by *path*. Words in this segment should be separated by newlines. This control argument may be specified more than once.

-word *string*
deletes the word *string* from the dictionary even though *string* may look like a control argument.

NOTES

A word to be deleted from the dictionary *must* be spelled in its raw form, i.e., without indicating hyphenation points or the no-trim attribute (see add_dict_words command above).

EXAMPLES

To delete the word *example* from the dictionary Webster.dict in the Project_id directory, type:

```
ddw >udd>Project_id>Webster example
```

To delete the word *basic* from the dictionary my_words.dict in the current working directory, type:

```
ddw my_words basic
```

To delete the words *test, in-house,* and *Multics* from the dictionary good_words.dict, type:

```
ddw good_words test in-house Multics
```

find_dict_words (fdw)

The find_dict_words command finds and displays words contained in the sequence of dictionaries defined by the dict search list.

SYNTAX AS A COMMAND

fdw {*words*} {*-control_args*}

ARGUMENTS

words
are words to be found. At least one *word must* be given unless the -input_file control argument is specified.

CONTROL ARGUMENTS

-brief
-bf
suppresses the warning message usually given when a word is not found.

-dictionary
-dict
displays the pathname of the dictionary in which the word was found.

-exact_match
-exm
finds only those words that match a dictionary word exactly, i.e., no special processing is performed with respect to capitalization (see "Note" below).

-input_file *path*
-if *path*
finds words in the segment specified by *path*. Words in this segment *must* be separated by newlines. This control argument may be specified more than once.

-output_file *path*
-of *path*
writes words found into the segment specified by *path* instead of displaying words on the user's terminal. Words are separated by newlines in the output segment.

-raw
displays the words without indicating the no-trim attribute or hyphenation points (see add_dict_words command above). Otherwise, words are printed in the format accepted by add_dict_words.

-word *string*
finds the word *string* even though *string* may look like a control argument.

NOTES

When searching for a word in a dictionary, special processing of capital letters is performed unless the -exact_match control argument is specified. This special processing is identical to that performed by the trim_wordlist command below.

hyphenate_word_

The hyphenate_word_ subroutine returns the character position at which a word can be hyphenated. The word is located in a dictionary via the dict search list.

USAGE SYNTAX

```
declare hyphenate_word_ entry (char(*), fixed bin,
   fixed bin, fixed bin(35));
call hyphenate_word_ (string, space, break, code);
```

-OR-

```
declare hyphenate_word_ entry (char(*), fixed bin, fixed bin);
call hyphenate_word_ (string, space, break);
```

ARGUMENTS

string (input)
the text word that is to be split.

space (input)
the number of print positions remaining in the line.

break (output)
the number of characters from the word that should be placed on the current line; it should be at least one less than the value of *space* (to allow for the hyphen), and can be 0 to specify that the word is not to be broken. Thus if the word "calling" is to be split, and six spaces remain in the line, the procedure should return the value 4 (adjustment is performed after hyphenation).

code (output)
a standard status code. In order to retain compatibility with an older version of this subroutine, this argument is optional, depending upon how hyphenate_word_ is declared in the calling program. If this subroutine is called with only three arguments, then no code is returned.

list_dict_words (ldw)

The list_dict_words command displays a list of words in a WORDPRO dictionary.

SYNTAX AS A COMMAND

ldw *path* {*words*} {*-control_args*}

ARGUMENTS

path
is the pathname of the dictionary to be listed. If *path* does not have the suffix dict, one is assumed; however, dict must be the last component of the dictionary segment name.

words
are words to be listed. If no words are specified, and if the -input_file control argument is not specified, all words in the dictionary are listed.

CONTROL ARGUMENTS

-brief
-bf
suppresses the warning message usually given when a word is not found in the dictionary.

-input_file *path*
-if *path*
lists the words contained in the segment specified by *path*. Words in this segment should be separated by newlines. This control argument may be specified more than once.

-output_file *path*
-of *path*
writes words to be listed into the segment specified by *path* instead of printing words on the user's terminal. The words are separated by newlines in the output segment.

-raw
displays the words without indicating the no-trim attribute or hyphenation points. Otherwise, words are listed in the format accepted by the add_dict_words command above.

-word *string*
lists the word *string* even though *string* may look like a control argument.

NOTES

When listing an entire dictionary, or any large number of dictionary words, it may be convenient to use the list_dict_words command together with the print_wordlist command to obtain multiple column output. This is accomplished by using the -output_file control argument to create a wordlist (i.e., a segment whose entryname has the suffix wl). The resulting wordlist can then be printed by the print_wordlist command.

For example, the following command sequence displays the dictionary English.dict:

```
list_dict_words English -output_file dict_words.wl
print_wordlist dict_words
```

In the above example, all words in English.dict are displayed in ASCII collating order (i.e., the order in which they are stored in the dictionary). This ordering is different from the alphabetical ordering used for wordlists. However, the create_wordlist command can be used to alphabetize dictionary words. For example, the following command sequence displays the dictionary English.dict in alphabetical order. The dictionary words are in raw format.

```
list_dict_words English -raw -output_file raw_words
create_wordlist raw_words
print_wordlist raw_words
```

If raw format is not desired, the command sequence below can be used:

```
ldw English -raw -of raw_words
cwl raw_words
ldw English -if raw_words.wl -of dict_words.wl
print_wordlist dict_words
```

locate_words (lw)

The locate_words command locates all occurrences of a given word within a specified text segment. The user can specify more than one word to be located. For each occurrence of a given word within the text segment, the number of the lines containing the word is displayed.

SYNTAX AS A COMMAND

lw *path words {-control_args}*

ARGUMENTS

path
is the pathname of the text segment.

words
are words to be located in the text segment.

CONTROL ARGUMENTS

-count
-ct
displays only the number of occurrences for each word.

-from *n*
-fm *n*
the text segment is searched starting from the line number specified by *n*. If this control argument is not specified, the text segment is searched starting from the first line.

-header
-he
displays the pathname of the text segment.

-lines {*n*}
-li {*n*}
for each occurrence of a given word, the lines (and line numbers) starting *n* lines before, through *n* lines after the line containing the word are displayed. Thus, if *n* is 1, three lines are displayed. If *n* is not specified, only the line containing the word is displayed (Default).

-long
-lg
for each occurrence of a given word, the line (and line number) of that word is displayed.

-to *n*
the text segment is searched up to and including the line number specified by *n*. If this control argument is not specified, the text segment is searched to the last line.

-word *string*
locates the word *string* even though *string* may look like a control argument.

NOTES

The -count control argument is mutually exclusive with the -long and -lines control arguments.

Words are found in the text segment in the same way as described for the create_wordlist command. Words containing no letters can be found even though they are normally excluded from a wordlist.

print_wordlist (pwl)

The print_wordlist command displays (prints) the words contained in a wordlist segment in a multiple column format (see the create_wordlist command above).

SYNTAX AS A COMMAND

pwl *path* {*-control_args*}

ARGUMENTS

path
is the pathname of a wordlist segment. If *path* does not have the suffix wl, one is assumed; however, wl must be the last component of the segment name.

CONTROL ARGUMENTS

-columns *n*
-cols *n*
specifies that the output is to contain *n* columns. The default number of columns depends on the line length and the column width (see "Notes" below).

-column_width *n*
-cw *n*
specifies that the column width is *n* characters. The default column width is 20.

-output_file *path*
-of *path*
directs the output to the segment specified by *path* in a format suitable for printing on a line printer.

-page_length *n*
-pl *n*
specifies that the page length is *n* lines. The default page length is 60 if -output_file is specified; otherwise, it is 66.

-vertical_margin *n*
-vm *n*
specifies that the vertical margin size is *n* lines. The default vertical margin size is 0 if -output_file is specified; otherwise, it is 3.

NOTES

The default number of columns is the maximum number of columns that fit within the line length. If the -output_file control argument is specified, a line length of 136 is assumed. Otherwise, the line length defined for the user_output switch is used. If none is defined, a line length of 72 is assumed.

If the length of a word is greater than or equal to the column width, the word is truncated. An asterisk (*) is appended to such words to indicate truncation.

Output is divided into pages. Each page has a top and bottom vertical margin consisting of *n* blank lines where *n* is the vertical margin size. These lines are included

in the page length. The column height on a page is equal to the page length minus twice the vertical margin size. In the default case, the column height equals 60 lines whether or not the -output_file control argument is specified. On the last page of output, the column height is reduced to the minimum height needed to accommodate remaining words. If the -output_file control argument is specified, each page is terminated by an ASCII new page character (octal 014).

revise_words (rw)

The revise_words command replaces all occurrences of a given word within a specified text segment with a new word called the **revision**. The user can specify more than one word to be revised.

SYNTAX AS A COMMAND

rw *path word1 rev1 ... {wordn revn} {-control_args}*

ARGUMENTS

path
is the pathname of the text segment.

wordi
is a word in the text segment to be revised.

revi
is the revision (i.e., the replacement for *wordi*).

CONTROL ARGUMENTS

-brief
-bf
suppresses the display of the number of revisions for each *wordi*.

-from *n*
-fm *n*
revisions are made in the text segment starting from the line number specified by *n*. If this control argument is not specified, the text segment is processed starting from the first line.

-header
-he
displays the pathname of the text segment.

-lines {*n*}
-li {*n*}
for each revision made, the lines (and line numbers) starting *n* lines before, through *n* lines after the line containing the revision are displayed. Thus, if *n* is 1, three lines are displayed. If *n* is not specified, only the line containing the revision is displayed (Default).

-long
-lg
for each word revised, the line (and line number) where the revision is made is displayed.

-to *n*
revisions are made in the text segment up to and including the line number specified by *n*. If this control argument is not specified, the text segment is processed to the last line.

-word *string1 string2*
replaces the word *string1* with the revision *string2* even though *string1* may look like a control argument.

NOTES

The -brief control argument is mutually exclusive with the -long and -lines control arguments.

Words are found in the text segment in the same way as described for the create_wordlist command. Words containing no letters can be revised even though they are normally excluded from a wordlist.

EXAMPLES

To replace the word *typpoo* with the word *typo* wherever it occurs in document.compin, type:

```
revise_words document.compin typpoo typo
```

If there are two occurrences of *typpoo*, the command displays the message:

```
2 revisions for "typpoo"
```

trim_wordlist (twl)

The trim_wordlist command trims (deletes) all words in the specified wordlist segment that are found in one or more WORDPRO dictionaries. The dictionaries may be specified explicitly or else the dict search list is used. The trimmed wordlist segment replaces the original wordlist segment. The number of words trimmed and the number of words remaining in the trimmed wordlist segment are displayed.

SYNTAX AS A COMMAND

twl *path* {*dict_paths*} {*-control_args*}

ARGUMENTS

path
is the pathname of the wordlist segment to be trimmed. If *path* does not have the suffix wl, one is assumed; however, wl must be the last component of the segment name.

dict_paths
are the pathnames of dictionaries to be searched in order. If *dict_paths* does not have a suffix of dict, one is assumed; however, dict must be the last component of the dictionary segment name. If no *dict_paths* are specified, the dictionaries in the dict search list are used.

CONTROL ARGUMENTS

-brief
-bf
suppresses the display of the number of words trimmed and the number of words remaining in the trimmed wordlist segment.

-exact_match
-exm
trims only those words that match exactly a word found in a dictionary, i.e., no special processing is performed with respect to capitalization (see "Notes" below).

NOTES

For each word processed, the dictionaries are searched in the order specified or as defined in the dict search list. Normally, when a word is found in a dictionary, it is trimmed. However, if the word found has the no-trim attribute, then the word is *not* trimmed and no more dictionaries are searched for this word.

When searching for a word in a dictionary, special processing of capital letters is performed unless the -exact_match control argument is specified. Most words in a dictionary consist of all lowercase letters. These words match any representations of themselves that are either all lowercase letters, all lowercase letters with a leading capital letter, or all capital letters. Words in a dictionary that have a leading capital letter only match representations of themselves that have a leading capital letter or are all capital letters. Words in a dictionary that consist of all capital letters or mixed

lowercase and capital letters only match representations of themselves that have the identical capitalization.

The table below shows examples of different ways a word in a dictionary may be capitalized. It also shows which representations of these words match and which do not match.

WORD	MATCH	NO MATCH
example	example Example EXAMPLE	ExAmple
Multics	Multics MULTICS	multics MULTics
WORDPRO	WORDPRO Wordpro WordPro	wordpro
non-ASCII	non-ASCII	non-ascii Non-ASCII NON-ASCII

SECTION 5

SPEEDTYPE

SPEEDTYPING

The primary goal of Speedtype is to allow users to type input data more quickly. Speedtyping, quite simply, is the ability to type a document using the least possible number of key-strokes. Typing speed is therefore increased since less is typed.

Speedtype can also help improve typing accuracy. Typing accuracy is improved by defining and using symbols for words or phrases that are often mistyped. For example, the common typo `teh` (intended to be `the`) can be corrected automatically by having Speedtype expand the symbol `teh` into `the`. Even better, this typo can be eliminated entirely by typing the symbol `t` and have Speedtype expand it into `the`

Speedtype is quite similar to the Multics *abbrev* subsystem (see the description of the abbrev command in *Multics Commands*) which expands command line input. Speedtype, however, can define, maintain, and list a set of abbreviations that can be typed as input text and then expanded.

In order to avoid confusion and ambiguity in terminology between Speedtype and abbrev, the term *abbreviation* is not used when discussing Speedtype. Instead, the term *symbol* is used. All Speedtype commands are named to conform to this terminology.

SPEEDTYPE FEATURES

The primary job of Speedtype is to expand text. The following paragraphs describe the features of Speedtype that are involved in the expansion process.

Text Segments

Speedtype deals with two types of files; text segments and symbol dictionaries. A **text segment** contains the input text processed by Speedtype. This processing involves searching through the text segment and expanding all defined symbols. The expanded text is copied into an output text segment.

Speedtype processes an input text segment as just one long character string. The resulting output text segment may also be thought of as one character string. The input string is divided into pairs of tokens. Speedtype recognizes two types of tokens: **delimiter tokens** and **text tokens**. Certain ASCII characters are designated as delimiter characters (in general, white space and punctuation characters other than period). All other characters are considered text characters. Speedtype divides an input string into pairs of tokens.

```
<space>Now...<space>country
```

Not shown are the special cases that may exist at the beginning and end of an input string where one of the tokens in a pair may be missing.

Speedtype also recognizes special delimiter and text characters if they are present. If they are found in certain positions, special processing is performed. For example:

```
<space>~      - where   ~ is an escape character
|country+.    - where   | is a prefix character(s),
                        + is a suffix character, and
                        . is termination (period)
```

Speedtype performs special processing on the last character of a delimiter token and on the first and last characters of a text token. This special processing is outlined below and discussed in detail later in this section.

Escapes
Certain delimiter characters are recognized as escape characters. If the last character of a delimiter token is an escape character, then special processing is performed on the following text token.

Prefixes
Certain text characters are recognized as prefix characters. If a prefix character is found at the beginning of a text token, then special processing is performed. Recognized prefix characters are not considered part of the symbol. Prefix characters found within the text token cause no special processing and are considered part of the symbol. More than one prefix character may precede the symbol.

Capitalization
If the first character of the symbol is an uppercase letter, then the first letter of the expansion string representing this symbol is capitalized when copied into the output string.

Suffixes
Certain text characters are recognized as suffix characters. If the last character of a text token (after any trailing period is removed) is a suffix character, then special processing is performed. A recognized suffix character is not considered part of the symbol. Suffix characters found within the text token cause no special processing and are considered part of the symbol. Only one suffix character may follow the symbol.

Period
If the last character of a text token is a period ".", then it is stripped from the text token. The period is copied into the output string after the text token is processed.

Symbol Dictionaries

A **symbol dictionary** contains all of the information needed by Speedtype to expand an input string. A symbol dictionary is similar to an abbrev *profile* segment (explained in the description of the abbrev command in *Multics Commands*). A `symbol dictionary` is identified by the entryname suffix, `symbols` (e.g., `standard_words.symbols`). Speedtype allows a user to specify the symbol dictionary

used. As a default, Speedtype uses a symbol dictionary in the user's home directory. The default symbol dictionary has the pathname:

```
>udd>Project>Person_id>Person_id.symbols
```

A symbol dictionary contains three types of information. Speedtype commands allow a user to set, change, and list all of this information. The three types of information are:

Options
Several types of control information are kept in a symbol dictionary. These Speedtype *options* may be set by a user. (See the option_symbols command at the end of this section for a description of the Speedtype options.) The Speedtype options are:

Delimiters (except escapes and white space)
Escape Characters
Prefix Characters
Suffix Characters

Symbols
A symbol is a character string that represents a word or phrase. A symbol must be unique within a symbol dictionary. Since symbols are found within text tokens, they may not contain any delimiter characters. The first character of a symbol may not be a prefix character, and the last character of a symbol may not be a suffix character or a period.

Expansions
Every defined symbol has a corresponding expansion string. Expansions do not have to be unique within a symbol dictionary. An expansion may contain any character, including delimiter characters. All suffixing, capitalization, and underlining is performed on expansions, not on symbols. Associated with each expansion is information that specifies how Speedtype is to perform suffixing on that expansion.

Expansion Process

Speedtype uses the general expansion algorithm described above. However, Speedtype also performs special processing. A more detailed description of how Speedtype expands a token pair is given below:

Delimiters
Processing of the delimiter token only involves copying it into the output string.

Escape Processing
If the last character of the delimiter token is an escape character, then special processing is performed on the following text token. Escape characters contained within the delimiter token are not recognized as escapes. The most important type of escape processing involves inhibiting any processing of the following text token. Instead, the text token is just copied into the output string.

Finding the Symbol

If no escape inhibits the processing of the text token, then the next step is to find the symbol contained in the text token. This involves stripping off any prefix characters, suffix character, or trailing period. If no symbol is found within the text token (i.e., it consists of just prefix and/or suffix characters) then no further processing is performed on this text token and it is copied as is into the output string.

Decapitalization

If the text token contains a symbol, then it is placed in lowercase. This involves testing the first character of the symbol, and if it is an uppercase letter, translating it to lowercase. This translation is actually performed on a temporary copy of the symbol. The original input symbol is not modified.

Expansion

Speedtype then takes the lowercase symbol and searches for it in the current symbol dictionary. If found, the expansion for this symbol is copied into the output string, otherwise the original input symbol (and any suffix character) is copied.

Capitalization

If the input symbol was put in lowercase and replaced by expansion, then Speedtype capitalizes the expansion string copied into the output string. This involves testing the first character of the expansion string, and if it is a lowercase letter, translating it to uppercase.

Suffix Processing

If a suffix character was stripped from the symbol, and if the symbol was expanded, then Speedtype performs suffixing on the expansion string copied into the output string. This processing depends upon the suffix and how the suffix is defined for this symbol.

Prefix Processing

If any prefix characters were stripped from the symbol, then Speedtype performs prefix processing on the symbol *or* the expansion string which was copied into the output string. Prefix processing is always performed after any capitalization or suffixing.

Period Processing

If a period was stripped from the symbol, then it is added to the output string after all other processing of the text token is performed.

Escapes

The escapes recognized by Speedtype are listed below. The actual escape characters recognized are defined in a symbol dictionary and may be set by the user. Listed with each escape is its name and its default character. The special processing performed for each escape is also described.

~ (temp)

The **temp** (temporary) escape is the standard Speedtype escape. It causes Speedtype to *not* process the following text token. Thus this escape can be used to prevent a symbol from being expanded and can prohibit prefix processing

for the next text token. Instead, the text token is copied as is into the output string. The **temp** escape character itself is *not* copied into the output string.

octal 177 (pad)
The **pad** escape is useful in situations where an input text segment is also used as the output text segment and is expanded over and over. The effect of this escape is the same as that for the **temp** escape. However, unlike the **temp** escape, this escape character is copied into the output .ur string. The default character used for the **pad** escape is the **pad** character (ASCII code 177). Even though this character is copied into the output string, it is not printed. Users are cautioned that the presence of a **pad** character in the text segment may cause problems during subsequent editing.

` (perm)
The **perm** (permanent) escape is a convenient way for a user to enter a **pad** escape. The effect of this escape is the same as the **temp** escape, and like the **pad** escape, it is copied into the output string. However, the **perm** escape character is then converted to the **pad** escape character.

: (trans)
The function of the **trans** (transparent) escape is to concatenate text tokens that are processed separately. The **trans** escape character is not copied into the output string. The following text token is processed as if no escape was recognized. Any prefix processing performed on the previous text token is continued and performed on the next text token. Additional prefix processing may be specified.

; (space)
The function of the **space** escape is to generate spaces (ASCII blanks) in the output string. The processing of this escape is conditional on the first characters of the following text token. If the following text token begins with one or two numeric characters (numbers from 0 to 99), then the **space** escape character and these numeric characters are replaced in the output string with the specified number of spaces. For example, "*;5*" is replaced by five spaces in the output string. The rest of the text token is then processed normally. If the following text token does not contain a number as specified above, then the **space** escape character remains unchanged in the output string and the following text token is processed as if no escape was recognized.

Suffixes

Suffix processing is performed only on defined symbols. If a symbol is not defined, or if the specified suffix is turned off for the symbol, then no suffix processing is performed. Instead, the symbol and the suffix character are copied as is into the output string.

Appending a suffix to a symbol's expansion string is done in several different ways depending upon how the suffix is defined for the symbol. The normal way is to just addition the suffix string associated with the suffix directly to the expansion string. However, to accommodate the many anomalies of the English language, such tricks as dropping the last letter, doubling the last letter, adding letters, etc., may be performed on the expansion string in order to addition a suffix string.

A user has considerable control over how Speedtype performs suffixing. (See the add_symbols command at the end of this section for a description of how Speedtype performs suffixing.) A user may disable suffixing for a given symbol, or just disable one or more suffixes for that symbol. A user may also specify a different way to process a suffix for a symbol.

The suffixes currently recognized by Speedtype are listed below. The actual characters representing the suffixes are defined in a symbol dictionary and may be set by the user. Except for *plural*, the suffix string associated with each suffix is the suffix itself. Also listed with each suffix is the default character used to represent that suffix.

Suffix Name	Suffix String	Default Character
plural	s	+
ed	ed	-
ing	ing	*
er	er	=
ly	ly	|

Prefixes

Prefix processing is performed on the text token string copied into the output string. It is performed regardless of whether symbol expansion was performed, and is always performed after capitalization and suffixing have been performed.

The prefixes recognized by Speedtype are listed below. The actual prefix characters recognized are defined in a symbol dictionary and may be set by the user. Listed for each prefix is its name and its default character. The special processing performed for each prefix is also described.

_ **(under)**
The function of the **under** (underline) prefix is to underline the output string. The underlining is performed by taking each character of the output string and adding, in a canonical way, a backspace character and an underscore character. The resulting underlined string is in canonical form. Underlining is not performed if the output string already contains backspace characters.

| **(upper)**
The function of the **upper** (uppercase) prefix is to translate the output string into uppercase. Each lowercase letter in the output string is translated to uppercase. Characters that are not lowercase letters are not changed. If both the **upper** and **under** prefixes are recognized, then regardless of the order in which they are specified, uppercase processing is performed first.

add_symbols (asb)

The add_symbols command adds a symbol to the current symbol dictionary. All suffixes are enabled for the added symbol.

SYNTAX AS A COMMAND

asb *symbol expansion {-control_args}*

ARGUMENTS

symbol
is the symbol to be added. Its length must be 7 characters or less and it may not contain delimiter characters. Its first character may not be a defined prefix character or a capital letter, and its last character may not be a defined suffix character or a period.

expansion
is the expansion string that replaces the symbol. The length of the expansion string must not exceed 56 characters. The expansion string may contain any characters. If the expansion string contains spaces and/or tabs, then it must be enclosed in quotes.

CONTROL ARGUMENTS

-force
-fc
specifies that the replacement of an existing symbol should be done without question. If the symbol is already defined, and this argument is not specified, then the user is asked to authorize the replacement of the symbol.

-suffix *string*
enables or disables suffixing for this symbol. *string* must be either on or off. If *string* is on then suffixing is enabled and all suffixes are processed according to the default rules described in "Notes" below. If *string* is off, then all suffixes are disabled for the symbol. If this control argument is not specified, then on is assumed.

-plural *string*
defines the plural suffix for this symbol. *string* must be on or off, or a string that can be used as the plural of the expansion of this symbol. If *string* is on, then the plural suffix is enabled for this symbol and processed according to the default rules for the plural suffix. If *string* is off the plural suffix is disabled for this symbol.

-ed *string*
defines the ed suffix for this symbol. This control argument follows the same rules as the -plural control argument.

-ing *string*
defines the ing suffix for this symbol. This control argument follows the same rules as the -plural control argument.

-er *string*
defines the er suffix for this symbol. This control argument follows the same rules as the -plural control argument.

-ly *string*
defines the ly suffix for this symbol. This control argument follows the same rules as the -plural control argument.

NOTES

The *default* rule for appending a suffix string to an expansion string is a function of the suffix and the word type of the expansion string.

The word type of the expansion string is determined from its last characters. The characters `C` and `V` are used below to represent consonants and vowels. The character `X` is used to represent any character. The word types recognized and the suffix strings used are:

REF. NO.	WORD TYPES
0	other (=> none of those below)
1	XCe
2	XVe
3	XCy
4	XVy
5	Xch, Xsh, or Xex
6	CVC

REF. NO.	SUFFIX STRINGS
1	s (plural)
2	ed
3	ing
4	er
5	ly

The actions performed by Speedtype when adding a suffix string to an expansion string are:

REF. NO.	SUFFIX ACTIONS
1	add suffix string directly
2	drop last character, add suffix string
3	double last character, add suffix string
4	replace last character with `i`, add suffix string
5	replace last character with `ie`, add suffix string
6	add `e`, add suffix string

The suffix action table presented below shows the action performed by Speedtype when adding a specified suffix string to an expansion string of a given word type.

SUFFIX ACTION TABLE

Word Type Ref. No.	SUFFIX STRING REF. NO. 1	2	3	4	5
0	1	1	1	1	1
1	1	2	2	2	2
2	1	2	1	2	1
3	5	4	1	4	1
4	1	1	1	1	1
5	6	1	1	1	1
6	1	3	3	3	1

change_symbols (csb)

The change_symbols command changes the expansion or suffixing of specified symbol. Control arguments are processed one at a time. Specifying more than one control argument has the same effect as issuing the command several times with one control argument each time.

SYNTAX AS A COMMAND

csb *symbol* {*-control_args*}

ARGUMENTS

symbol
is the symbol changed. This symbol must be defined in the current symbol dictionary.

CONTROL ARGUMENTS

one or more arguments *must* be chosen from the following:

-exp *string*
where *string* represents the new expansion string for this symbol. This control argument does not change the way suffixing is performed for the symbol.

-suffix *string*
enables or disables suffixing for this symbol. *string* must be either on or off. If *string* is on, then suffixing is enabled and all suffixes are processed according to the default rules described in the "Notes" of the option_symbols command below. If *string* is off, then all suffixes are disabled for the symbol. If this control argument is not specified, then on is assumed.

-plural *string*
defines the plural suffix for this symbol. *string* must be on or off, or a string that can be used as the plural of the expansion of this symbol. If *string* is on, then the plural suffix is enabled for this symbol and processed according to the default rules for the plural suffix. If *string* is off, the plural suffix is disabled for this symbol.

-ed *string*
defines the ed suffix for this symbol. This control argument follows the same rules as the -plural control argument.

-ing *string*
defines the ing suffix for this symbol. This control argument follows the same rules as the -plural control argument.

-er *string*
defines the er suffix for this symbol. This control argument follows the same rules as the -plural control argument.

-ly *string*
defines the ly suffix for this symbol. This control argument follows the same rules as the -plural control argument.

delete_symbols (dsb)

The delete_symbols command deletes the specified symbols from the current symbol dictionary.

SYNTAX AS A COMMAND

dsb *symbols*

where *symbols* are the symbols to be deleted from the current symbol dictionary.

expand_symbols (esb)

The expand_symbols command takes an input text segment and expands it using the options and symbols defined in the current symbol dictionary.

SYNTAX AS A COMMAND

esb *input_path* {*output_path*}

ARGUMENTS

input_path
is the pathname of the input text segment.

output_path
is an optional pathname of an output text segment. If no output pathname is specified, the original contents of the input text segment are *overwritten* with the expanded material.

find_symbols (fsb)

The find_symbols command finds and lists all of the symbols associated with specified expansions contained in the current symbol dictionary. One, several, or all expansions may be listed.

SYNTAX AS A COMMAND

fsb {*expansions*} {*-control_args*}

ARGUMENTS

expansions
are optional arguments that specify expansions to find and list. If an expansion is represented by more than one symbol, all of its symbols are found and listed. If any given expansion is not found, a message is printed stating that the expansion is not defined. If no *expansions* are specified, all expansions in the current symbol dictionary are listed. The expansions are listed in order according to ASCII collating sequence.

CONTROL ARGUMENTS

-long
-lg
specifies that for each symbol listed, its expansion string with suffixing is listed for each suffix enabled for that symbol.

-option
-op
specifies that all option information for the current symbol dictionary is to be listed (see the option_symbols command for a complete description of option information). If this is the only control argument specified, only the option information is listed.

-total
-tt
specifies that the total number of symbols defined in the current symbol dictionary is to be printed. If this is the only control argument specified, only the total is printed.

list_symbols (lsb)

The list_symbols command lists one, or several, or all of the symbols defined in the current symbol dictionary.

SYNTAX AS A COMMAND

lsb {*symbols*} {*-control_args*}

ARGUMENTS

symbols
: are optional arguments that specify the symbols to list. If any given symbol is not found, then a message is printed stating that the symbol is not defined. If no *symbols* are specified, then all symbols in the current symbol dictionary are listed. The list is in ASCII collating sequence order.

CONTROL ARGUMENTS

-long
-lg
: specifies that for each symbol listed, its expansion string with suffixing is listed for each suffix enabled for that symbol.

-option
-op
: specifies that all option information for the current symbol dictionary is to be listed (see the option_symbols command for a description of option information). If this is the only control argument specified, then only the option information is listed.

-total
-tt
: specifies that the total number of symbols defined in the current symbol dictionary is to be printed. If this is the only control argument specified, then only the total is printed.

option_symbols (osb)

The option_symbols command allows a user to change certain optional control information in the current symbol dictionary. This information is summarized in "Notes" below.

SYNTAX AS A COMMAND

osb {*control_args*}

CONTROL ARGUMENTS

all except -delim set corresponding escape, prefix, or suffix characters recognized by Speedtype to the character specified by X (see "Notes" below). A complete explanation of the escape, prefix, and suffix characters is given earlier in this section.

-delim *string*
specifies a new set of delimiter characters. None of the characters in this *string* may be currently defined escape, prefix, or suffix characters.

-pad *X*
-perm *X*
-temp *X*
-trans *X*
-space *X*

-under *X*
-upper *X*

-plural *X*
-ed *X*
-ing *X*
-er *X*
-ly *X*

NOTES

A summary of all Speedtype options is given below. The default character(s) used to represent each option is also shown.

Delimiters:

Escapes (see below)
White space (space, tab, newline)
Others .,()?!<> [] {} "

Escapes:

pad	(octal 177)
perm	`
temp	~
trans	:
space	;

Prefixes:

under	_
upper	|

Suffixes:

plural	+
ed	-
ing	*
er	=
ly	|

EXAMPLES

To set the temporary escape character to &, type:

```
osb -temp &
```

print_symbols_path (psbp)

The print_symbols_path command prints the pathname of the current symbol dictionary.

SYNTAX AS A COMMAND

psbp

retain_symbols (rsb)

The retain_symbols command takes an input text segment and inserts Speedtype escape characters wherever symbols would be expanded if this text segment were being processed by the expand_symbols command. All symbols in the text segment are thus retained during future expansion.

SYNTAX AS A COMMAND

rsb *input_path* {*output_path*} {*-control_args*}

ARGUMENTS

input_path
is the pathname of the input text segment.

output_path
is the optional pathname of an output text segment. If no output pathname is specified, the original contents of the input text segment are overwritten.

CONTROL ARGUMENTS

-perm
specifies that the *perm* escape character is to be used. If no control argument is specified, -perm is assumed.

-temp
specifies that the *temp* escape character is to be used. Specifying this control argument causes the symbols in the output text segment to be retained for only one expansion.

NOTES

In addition to inserting the specified escape character wherever necessary, all existing *pad* escapes are converted to the specified escape. This allows for more convenient editing of the input text segment, since all escape characters are thus printable. (Refer to the escape description earlier in this section.)

show_symbols (ssb)

The show_symbols command shows how Speedtype expands an input string. The expansion is performed using the options and symbols in the current symbol dictionary. The expanded string is printed on the user's terminal.

SYNTAX AS A COMMAND

ssb *terms*

where *terms* are arguments that are concatenated into the input string that is expanded. These terms are separated in the input string by one space. If other spacing is desired, the input string should be enclosed in quotes.

EXAMPLES

To show the expansion for the term th which is defined in the current symbol dictionary, type:

```
! ssb th
  these
```

If the term th is not defined in the user's current dictionary then the system response would be `th`.

use_symbols (usb)

The use_symbols command sets the current symbol dictionary. All Speedtype commands then use this symbol dictionary. If this symbol dictionary does not exist, the user is asked if it should be created.

SYNTAX AS A COMMAND

usb *path*

where *path* is the pathname of the symbol dictionary that is to be the new current symbol dictionary. If *path* does not have a suffix of symbols, one is assumed; however, symbols must be the last component of the symbols dictionary segment name.

NOTES

If other Speedtype commands are issued in a user's process before the use_symbols command, then those commands use the default symbol dictionary in the user's home directory. The default symbol dictionary has the pathname:

```
>udd>Project_id>Person_id>Person_id.symbols
```

SECTION 6

LIST PROCESSING

List Processing involves maintenance, sorting, and selection of items in a list (e.g., names, words, numbers) and the production of documents that use this information. The processing steps involve creating an input file, compiling this file into a form that can be manipulated by the List Processing commands, and manipulating this new file with files that define formats for the final output.

LIST PROCESSING FUNCTIONS

The main functions of List Processing are:

- list maintenance (i.e., entry and update of information in a list)
- sorting
- selection
- report generation

An example of the use of List Processing is a dental office that maintains a list of all patients serviced by that office. The data maintained for each patient might include the patient's name, address, phone number, date of last visit, etc. When a patient first visits the office, the patient is added to the list. This involves using a text editor (see Section 2) to input information about the patient, which may be updated later. For example, on each subsequent visit, the date of last visit is updated for that patient.

This dental patient list can be used to produce various documents. For example, the dentist may want a report listing the name, address, and phone number of all patients, sorted alphabetically by patient name, or a form letter reminding the patient to visit the office for a checkup. Perhaps the dentist sends this letter to those selected patients who have not visited the office for six months or more.

On Multics, List Processing is done with a set of commands that maintain and process online lists of information. These commands can be used to produce simple reports like the ones described above; they also provide a means by which the output can be saved in a segment or directed to the terminal. Once in a segment, the output can be mailed to other users using the Multics mail facility (see *Multics Commands*), or can be further processed by the Compose Text Formatter (see Section 4 of this manual) to produce reports and form letters.

LIST PROCESSING FILES

List Processing uses three types of files (listin, lister, and listform), each type identified by its entryname suffix. A description of each of these types follows.

Listin File

A **listin file** is an ASCII file used to input and update a list. It is identified by the entryname suffix `.listin` (e.g., `monthly.listin`). Records can be added to, deleted from, or updated in this list simply by editing the file with a text editor.

The format of a listin file is simple. It consists of the following three parts:

1. Header

   The header specifies the record and field delimiter characters, the optional comment delimiter, and the field names. It is located at the beginning of the file and contains the following statements:

```
Comment_delimiter: c;          or   Cd:  c;
Record_delimiter: r;           or   Rd:  r;
Field_delimiter: f;            or   Fd:  f;
Field_names: fn1, ... fnK;     or   Fn:  fn1, ...  fnK;
Records:
```

   The Comment_delimiter statement enables comments in the listin segment and specifies the character or characters used to begin and end comments. Comments may appear anywhere that white space is allowed and are ignored. No comments may precede this statement in the listin file. If this statement is not given, then no comments are allowed in the listin file. If the string "pl1" is specified in the Comment_delimiter statement then comments begin with "/*" and end with "*/"; otherwise the comment delimiter must be exactly one character long and must be chosen from the set below, in which case that single delimiter both begins and ends comments. It should be noted that the usage of the PL/I-style comments in conjunction with the usage of the "*" as either record or field delimiters can cause problems and should be avoided if possible.

   The Record_delimiter statement specifies the character used to separate records. If this statement is not given, the default record delimiter is a dollar sign ($).

   The Field_delimiter statement specifies the character used to separate fields within a record. If this statement is not given, the default field delimiter character is an equal sign (=). Record and field delimiters must be exactly one character long, cannot be the same, and must be chosen from the following set:

   ! # $ % & * = ? @ ^ | ~

   The Records statement must be the last statement in the header, and is required. It specifies the end of the header and the beginning of the record information.

2. Fields

   The fields contain the various types of information stored in a list (e.g., first name, last name, street address, date of employment, etc.). Because data records are stored separately within a listin file, the field names must be given with each data record. Within an individual data record, a field is specified by a field delimiter character followed immediately by the field name (e.g., =lname). Any amount of white space (space, horizontal tab, vertical tab, newline, or new page) can follow the field name (e.g., =lname Smith). If the field value contains any record, field, or comment delimiters, then it must be quoted (e.g., =amount "$1.00") and in this case any embedded quotes must be doubled. A field value ends at the next record or field delimiter. Leading and trailing

white space is removed from field values, though such space can be retained by including it within quotation marks that surround the entire field value.

All of the field names used in a list must be specified in the Field_names statement of the header. Field names may be from 1 to 32 characters in length, must begin with an alphabetic character, and must contain only alphabetic, numeric, and underscore characters.

3. Data Records

   The data records (hereafter simply referred to as *records*) contain the specific information associated with the subject of each record. The beginning of each record is denoted by the record delimiter character, followed by a list of fields. A record may contain some or all of the fields defined in the header, and fields not specified for a record are considered to be null. Duplicate fields are not allowed within a record.

Lister File

A **lister file** contains the records entered and updated through a listin file and serves as the file from which the processes of merging, trimming, sorting, selecting, and document processing are performed. It is identified by the entryname suffix `.lister` (e.g., `monthly.lister`). After records have been input or updated in a listin file, the create_list command (described below) transfers them to a lister file. When a lister record is created, it is assigned a decimal identifier that is unique within the lister file and remains assigned to the record as long as the file exists. If the record is deleted, its unique identifier is not reused. If the create_list command is used to re-create an existing lister file, the unique identifiers change. The unique identifier is referred to by the reserved field name ":uid". Since lister files are formatted binary files, they cannot be displayed by using the print command, as listin files can and must be. They can only be displayed by the process_list or the display_list commands, though they can be manipulated by the append_list, copy_list, merge_list, modify_list, sort_list, and trim_list commands. Other functions are provided by the describe_list and expand_list commands (all of the lister commands are described below).

Listform File

A **listform file** defines the format of a document to be produced from a list of records. It is identified by the entryname suffix `.listform` (e.g., `monthly.listform`). Information from a list is copied into a document in the format specified by the listform being used. A single listform file may be used with a number of lister files, just as one lister file can be used with several listform files.

Three sections of a document may be defined. These three sections are the Before section, the After section, and the Record section. The Before and After sections are optional and useful for organizational purposes; i.e., the Before section may be used for headings and introductory matter, and the After section for closings and summary material. The Record section, however, is necessary for processing records. The functions of field insertion, sorting, and selection require the presence of a Record section within the listform to correspond to the lister records. Examples of these sections are in the sample listform files under "Sample List Processing Files" below.

These three listform sections are further described as follows:

Before

This section is added to the document as a preface before any records are processed. It may contain any desired text, including compose controls. The beginning of the Before section is identified by the string "<Begin before:>". The end of the Before section is identified by the string "<end;>".

Record

This section describes the document format for each lister file record processed. It contains field value strings to be copied from the lister file being processed (see "Field Insertion" below), compose controls, and any desired text. The beginning of the Record section is identified by the string "<Begin record:>". The end of the Record section is identified by the string "<end;>".

After

This section is added to the document after all records are processed. It may contain any desired text, including compose controls. The beginning of the After section is identified by the string "<Begin after:>", the end of the After section is identified by the string "<end;>".

FIELD INSERTION

In order to insert information from the lister file into the document, a field name enclosed in angle brackets (<>) is included in the record section of the listform file (e.g., <city>).

An optional field width may also be specified. For example, <city,10> specifies that the value string of the field "city" is to occupy 10 character positions. If the current value string is less than the specified field width, then it is padded on the right with blanks. If the current value string is greater than the specified field width, then it is truncated (cut off) on the right so its length is equal to the specified field width.

An optional field alignment may also be specified if a field width is specified. For example, <city,10,r> specifies that the value string of this field is to be right-aligned within the 10-character field width. The alignment indicators "l" for left and "c" for center may also be specified. If no alignment is specified, the value string is left-aligned.

To insert arguments into the document using the process_list command with the -argument control argument, an argument name (enclosed in angle brackets) may be included in the text of the before, after, or record section. The argument name is replaced by the actual argument when the section is processed. Field widths and field alignments may also be specified for argument insertions.

Argument names are of the form :arg1, arg2,... :argN. For an example of the use of the -argument control argument, see the example beneath the sample letter under "Sample List Processing Files" below. Arguments specified by the -argument control argument, but never referenced by an argument name, are diagnosed with a warning; i.e., the command function is carried out but a warning is displayed on the terminal. Arguments named in a listform segment, but never specified by the -argument control argument are also diagnosed with a warning. A null string is used in place of the missing argument. The -brief_errors control argument suppresses these warnings.

Listform files are character-oriented rather than line-oriented, so placement of field names within these files dictates the line output. Thus, when field or character strings follow immediately after a <Begin record:> control, no blank lines are inserted. When the field name or character string begins on the next line, one blank line is inserted. The following format produces no blank lines between records:

```
<Begin record:> <fname> <lname>
<end;>
```

One blank line is inserted between each record by the following listform format:

```
<Begin record:>
<fname> <lname>
<end;>
```

One blank line can also be produced by moving the <end;> control, as in:

```
<Begin record:> <fname> <lname>

<end;>
```

The unique identifier of a lister record can be inserted into a List Processing document by specifying <:uid> in the Record section of the listform file.

Finally, by use of listform files, the current date, the present time, and the number of records being processed with the current command invocation can be inserted in a List Processing document. These special fields can be inserted separately in any of the three listform sections; Before, Record, and After, using the format:

```
<:date>
<:time>
<:record_count>
```

When the process_list command is invoked with a listform file containing any of these special fields, the specified information is automatically inserted into the document being printed without any type of control argument in the command line.

ANGLE BRACKET ESCAPES

To place a single left angle bracket in the text, enter two left angle brackets (<<). A single right angle bracket is left as is in the document when it is used with two left angle brackets (e.g., <<Phoenix> becomes <Phoenix> in the output). A single right angle bracket is also left as is in the output when it is used with no left angle brackets (e.g., Phoenix> remains Phoenix>).

SORTING

The sorting process sorts records in a file according to specific criteria. These criteria are indicated in the -sort control argument which is used with both the sort_list and process_list commands. With the -sort control argument, a list of names and addresses, for instance, can be printed in alphabetical order according to first name, last name, city, street, or any field within its record.

Note: Within this section, references to alphabetical order refer to a sorting sequence identical to the ASCII collating sequence with the exception that lowercase letters immediately follow the corresponding uppercase letters.

The -sort control argument always takes a character-string argument which specifies the record fields used to control the sort. Its format is:

-sort *string*

-st *string*

Note that the character-string *string* must be surrounded by quotation marks when the string contains blanks or reserved characters and internal quotation marks must be doubled.

The character-string consists of one or more field specifications separated by spaces. The first field specification defines the primary sort field; the second field specification defines the secondary sort field; and so forth. The lister file acted upon is reordered permanently when using the sort_list command; the lister file is not altered when using the -sort control argument with the process_list command.

A field specification consists of two parts, a field name and optional order and type control arguments. The order and type control arguments can be chosen from the following:

-ascending
-asc
specifies that this field is to be sorted into ascending order. If no order control argument is specified, then ascending order is assumed (e.g., 0123456789Aa...Zz).

-descending
-dsc
specifies that this field is to be sorted into descending order.

-alphabetic
-alp
specifies that the field is to be sorted alphabetically.

-numeric
-num
specifies that this field is to be sorted numerically by temporarily converting each field value to a float decimal(29) value. Values that cannot be converted, sort as if they had the value zero.

The sort performed by the -sort control argument is stable; that is, records with equal fields stay in the same relative order, whether an ascending or descending sort is performed. For examples of the use of the -sort control argument, see "Sample List Processing Files" and the sort_list command description in this section.

SELECTION

The selection feature enables a List Processing command to select from a lister file only certain records upon which to perform its function. The command, through the use of the control argument described below, specifies requirements for desired fields. If a record meets the requirements it is processed; otherwise it is skipped.

For instance, from a complete list of names and addresses, separate lists could be printed for all entries with last name beginning with any letter, all residents of one town, all residents of one state, or all entries with the exception of those containing some specified criteria.

The -select control argument (which can be used with the copy_list, process_list, and trim_list commands) always takes a character-string argument. Its format is:

-select *string*
-sel *string*

In this argument, the character-string *string* must be surrounded by quotation marks. Each record in the specified lister file is tested to determine whether or not the record fulfills the selection criteria. Those that do are processed.

The -select control argument consists of one or more field comparisons. A field comparison involves comparing a test string to the value of the specified field in the current record. The field comparison statement always consists of three parts:

"field_name comparison_operator test_string"

where:

field_name
is the name of a field contained in the lister file. The reserved field name ":any" may be used to specify any field in the record. The reserved field name ":uid" may be used to specify the unique identifier of a record.

comparison_operator
specifies what comparison is performed. The opposite comparison is performed if the comparison operator is preceded by "not". The List Processing comparison operators are:

contain(s)
test string is contained in the field value. The comparison is made without regard to case (i.e., uppercase letters compare equal to lowercase letters).

equal(s)
test string is equal to the field value. Uppercase letters are distinct from lowercase letters with this operator.

greater
field value is alphabetically greater than the test string (e.g., 0123456789Aa...Zz).

less
field value is alphabetically less than the test string.

nequal(s)
field value string is numerically equal to the numeric value of test string.

ngreater
field value string is numerically greater than the test string.

nless
field value string is numerically less than the test string.

test_string
is the string that is compared to the field value string. The special test string ":null" is used to test whether or not the field is *null*, i.e., missing from the current record. The special test string ":numeric" is used to test whether or not the field value string is numeric, i.e., can be converted to a number. Null fields are always non-numeric.

Several field comparisons may be specified by the -select control argument. Field comparisons are combined by the logical operators "and", "or", or "not". In the absence of parentheses, the prefix "not" operator is evaluated first, then the infix "and" operator, then the infix "or" operator. Parentheses may be used to specify the exact order of evaluation. These rules are similar to the PL/I rules for Boolean expressions.

The special test strings ":null" and ":numeric" can only be used with the equal or nequal comparison operators.

The comparison operators (not) contain, (not) greater, (not) less, (not) ngreater, or (not), and nless ignore records that have null fields. Unless the special test string ":null" is used, (not) equal and (not) nequal also ignore records with null fields.

For examples of the use of the -select control argument, see "Sample List Processing Files" below, and the process_list and trim_list command described later in this section.

SAMPLE LIST PROCESSING FILES

A sample list and two specific uses (mailing list and form letter) are shown below.

Using the dental office example, the first file shown below (patients.listin) is the one containing a list of the patient's names, addresses, and other pertinent information. Its appearance is exactly as entered by the user with the text editor, except for the heading "patients.listin", which is added when the file is displayed by the print command. The listin file can be displayed by entering the command:

```
print patients.listin
```

which results in the following display:

```
                                patients.listin

Comment_delimiter: pl1;
Record_delimiter: $;
Field_delimiter: =;
Field_names: fname,lname,street,city,state,zip,phone,
  date,message;
Records:
$
 =fname John
 =lname Doe
 =street 71 Pine Street
 =city Boston
 =state Massachusetts
 =zip 02020
 =phone (617) 555-7654
 =date 770520
 =message you and your family well
$
 =fname Jane
 =lname Smith
 =street 898 Smith Avenue
 =city Needham
 =state Massachusetts
 =zip 02112
 =phone (617) 555-4567
 =date 750713
 =message you well
$
 =fname Francis
 =lname Jones       /*formerly Wilson*/
```

```
=street PO BOX 999
=city Cambridge
=state Massachusetts
=zip 02139
=phone (617) 555-7869
=date 770131
=message you well
```

The next sample file, addresses.listform, defines the format to be used while processing patients.lister (created from patients.listin) to generate the address report below (final desired output). Its appearance is exactly as entered by the user with the text editor, except for the heading "addresses.listform", which is added when the file is displayed by the print command. The listform file can be displayed by entering the command:

```
print addresses.listform
```

which results in the following display:

```
                              addresses.listform

<Begin before:>

     Dental Patient Addresses

<end;>


<Begin record:><fname> <lname>
<street>
<city>, <state> <zip>
Tel:  <phone>


<end;>


<Begin after:>     Dental Associates
<end;>
```

The create_list command makes a lister file named patients.lister from the listin file named patients.listin. The process_list command then operates on the lister file, formatting the records according to the addresses.listform file and arranging the records alphabetically by last name as specified by the -sort control argument. The following command lines are used to create the lister file and process it with the listform file:

```
create_list patients.listin
process_list patients.lister addresses.listform -sort lname
```

which results in the following display:

```
                Dental Patient Addresses

John Doe
71 Pine Street
Boston, Massachusetts 02020
Tel:  (617) 555-7654
```

```
Francis Jones
PO BOX 999
Cambridge, Massachusetts 02139
Tel:  (617) 555-7869


Jane Smith
898 Smith Avenue
Needham, Massachusetts 02112
Tel:  (617) 555-4567

          Dental Associates
```

The file, letter.listform, defines the format of the form letter, which, when manipulated with patients.lister, creates the Sample Letter (shown below) specifically for Jane Smith. Using the process_list command with the -select control argument to specify other records within patients.lister, the same letter can be composed for any or all patients on the list, using any available field as a criterion. Its appearance is exactly as entered by the user with the text editor, except for the heading "letter.listform", which is added when the file is displayed by the print command. The listform file can be displayed by entering the command:

```
print letter.listform
```

which results in the following display:

letter.listform

```
<Begin record:>
.pdl 40
.pdw 55
.inl 30
.fif
Dental Associates
1001 Jamaica Avenue
Boston, Mass. 02003
(617) 555-6000
.spb 2
.inl
<fname> <lname>
<street>
<city>, <state> <zip>
.fin
.inl
.spb
Dear <fname>:
.spb
     I hope this letter finds <message>.
It has been over six months since your last visit to our
office.  Please call and make an appointment
to have a checkup.
.spb
.inl 30
Keep smiling,
.spb 2
J. Kelly, D.M.D.
```

```
.brp
<end;>
```

SAMPLE LETTER

The process_list command selects Jane Smith from the patients.lister file, processes it with the letter.listform file, and sends it to an output file named letter.compin. This segment is then operated on by the compose command to produce the letter to Jane Smith. The following command lines are used to create the sample output from the lister file:

```
!   pls patients letter -sel fname equal Jane and lname equal Smith
    -of letter.compin
!   compose letter
```

which results in the following display:

```
                                        Dental Associates
                                        1001 Jamaica Avenue
                                        Boston, Mass. 02003
                                        (617) 555-6000

Jane Smith
898 Smith Avenue
Needham, Massachusetts 02112

Dear Jane:

     I hope this letter finds you well.  It  has  been
over  six months since your  last  visit to our office.
Please call and make an appointment to  have a checkup.

                                  Keep smiling,


                                  J. Kelly, D.M.D.
```

Now assume that Dr. Kelly takes in a partner (Dr. O'Brian). When composing the reminder in the future he wants to designate from command level whose name is to be on each letter individually. In letter.listform (see example above that displays "letter.listform"), he replaces:

```
J.  Kelly, D.M.D.
```

with an argument name of the form:

```
<:arg1>
```

The next time that a letter is needed, he types:

```
! pls patients letter -sel lname equal Doe -of letter.compin
  -ag "W. O'Brian, D.M.D."
```

and the letter is supplied with Dr. O'Brian's name in the signature block.

append_list (als)

The append_list command adds a record to a lister file.

SYNTAX AS A COMMAND

als *path -control_args*

ARGUMENTS

path
is the pathname of the lister file. The suffix lister must be the last component of the lister segment name; however, if *path* does not have a suffix of lister, one is assumed.

CONTROL ARGUMENTS

-field_name *field_name string*
-fn *field_name string*
causes the value of *string* to be assigned to the field indicated by *field_name*. If *string* contains spaces, it must be enclosed in quotes. This control argument is required and may be given more than once. Those fields for which this control argument is not given are assigned null values.

-string *string*
-str *string*
uses *string* as a character string with no special interpretation. This is useful for preventing *string* from being interpreted as a control argument. It is to be used with the -field_name control argument (e.g., "-field_name rating -string -20").

EXAMPLES

To append a record to an existing lister file, type:

```
als patients -fn fname Benjamin -fn lname Walker
```

copy_list (cpls)

The copy_list command creates a new list segment from an existing list segment. All, or selected, records of the existing list segment are copied into the new list segment. The new list segment is created in the working directory. Any existing copy of this segment is overwritten.

SYNTAX AS A COMMAND

cpls *path1 path2* {*-control_args*}

ARGUMENTS

path1
is the pathname of the existing lister file. The suffix lister must be the last component of the list segment name; however, if *path1* does not have a suffix of lister, one is assumed.

path2
is the pathname of the new list segment. The suffix lister must be the last component of the list segment name; however, if *path2* does not have a suffix of lister, one is assumed.

CONTROL ARGUMENTS

-select *string*
-sel *string*
copies records specified by *string* (the *string* argument must be enclosed in quotes). If this control argument is not specified, then all records are copied. (For a complete description of how to specify *string*, see "Selection" earlier in this section.)

-totals
-tt
displays the number of records copied.

EXAMPLES

To copy all records that have a city field equal to Boston from patients.lister into Boston_patients.lister, type:

```
cpls patients Boston_patients -sel "city equal Boston"
```

To copy all records that do not have a city field equal to Boston from patients.lister into Mass_patients.lister, type:

```
cpls patients Mass_patients -sel "city not equal Boston"
```

For more examples of the use of this control argument, see the trim_list command description.

create_list (cls)

The create_list command creates a lister file from a listin file.

SYNTAX AS A COMMAND

cls *path* {*-control_arg*}

ARGUMENTS

path
is the pathname of the listin file. The suffix listin must be the last component of the listin segment name; however, if *path* does not have a suffix of listin, one is assumed. A lister file is created in the working directory with the same entryname as *path*, and with the entryname suffix of listin changed to lister. Any existing copy of this lister file is overwritten.

CONTROL ARGUMENTS

control_arg
can be -totals or -tt to display the number of records in *path*.

NOTES

The creation of a lister file is the only List Processing operation which uses listin files as input. All other operations use lister files as input (which are unprintable files containing ASCII and binary information).

A listin file provides an ASCII representation of a list. It is used to input and update a list. The listin files can be created and updated by using any text editor.

Example

To create patients.lister from patients.listin (which contains three data records) and display the number of records in patients.listin, type:

```
! cls patients -tt
create_list: 3 records.
```

describe_list (dls)

The describe_list command displays information about a lister file.

SYNTAX AS A COMMAND

dls *path* {*-control_args*}

SYNTAX AS AN ACTIVE FUNCTION

[dls *path* {*-control_args*}]

ARGUMENTS

path
is the pathname of the lister file. The suffix lister must be the last component of the lister segment name; however, if *path* does not have a suffix of lister, one is assumed.

CONTROL ARGUMENTS

-delimiter {*record*|*field*}
-dm {*record*|*field*}
displays the value of the record or field delimiter. If the record and field keywords are omitted.then both delimiters are printed.

-field_name
-fn
displays the field_names in the lister file.

-select *string*
-sel *string*
specifies those records to be indicated by the -total control argument. If this control argument is not specified, then the total number of records in the file is used. (For a complete description of how to specify *string* see "Selection" earlier in this section.)

-total
-tt
displays the total number of records.

NOTES

If no control arguments are given, or only the -select control argument is given, then the record and field delimiters, total, and the field names are displayed.

If none or more than one of -delimiter {record|field}, -total, or -field_name are specified, the values are returned in the following order: record_delimiter, field_delimiter, total, and field_names.

Example

```
!   dls mlist
        mlist.lister        07/02/80  1606.4 mst Wed

    Total Records:          748
    Record_delimiter:       $;
    Field_delimiter:        =;
    Field_names:            name,did,addr,current,years,personid,
      alias,mproj;

!   dls mlist -sel "mproj equal SysAdmin" -total
    5
```

display_list (dils)

The display_list command displays (prints) selected portions of selected lister records.

SYNTAX AS A COMMAND

dils *path* {*-control_args*}

SYNTAX AS AN ACTIVE FUNCTION

[dils *path* {*-control_args*}]

ARGUMENTS

path
is the pathname of the lister file. The suffix lister must be the last component of the lister segment name; however, if *path* does not have a suffix of lister, one is assumed.

CONTROL ARGUMENTS

-brief_errors
-bfe
suppresses the warning when no records match the selection expression.

-field_name *field_names*
-fn *field_names*
causes the specified *field_names* to be displayed, in the order indicated. This control argument *must* be given.

-select *string*
-sel *string*
specifies those records whose fields are to be displayed. If this control argument is not specified, then all records are used. (For a complete description of how to specify *string*, type "help process_list".)

expand_list (els)

The expand_list command creates a listin segment from a lister segment. The number of records expanded is displayed. The operation performed by this command is the opposite of that performed by the create_list command.

SYNTAX AS A COMMAND

els *path* {*-control_args*}

ARGUMENTS

path
is the pathname of the lister segment. If the entryname suffix lister is not specified, then it is added. A listin segment is created in the working directory with the same entryname as *path*, and with the entryname suffix lister changed to listin. Any existing copy of this listin segment is overwritten.

CONTROL ARGUMENTS

-line_length *n*
-ll *n*
specifies that the line length of the ASCII listin segment is to be *n* characters. If this control argument is not specified, then only one field is placed on each line. A field is placed on a new line only if adding the field to the current line would exceed the specified line length. At least one field is placed on each line.

-totals
-tt
displays the number of records expanded.

NOTES

The ASCII listin segment created by this command has the following format:

- The first two lines specify the record and field delimiter characters.
- Beginning on the third line are the field names. They are separated by a comma and a space. A field name is placed at the beginning of a new line if adding it to the current line would exceed the specified line length.
- Each record begins with a line containing just the record delimiter character.
- Unless -line_length is specified, each field is placed on a separate line and indented one space.

merge_list (mls)

The merge_list command combines two lister files into a single lister file. The file resulting from the merge may be a new lister file, or it may replace an existing lister file. The fields defined in the two lister files must be identical, and the fields to be compared must be in ascending order. The comparisons are performed without regard to case (uppercase letters compare equal to lowercase letters). Sorting must be done by the sort_list command.

SYNTAX AS A COMMAND

mls *mas_path up_path* {*out_path*} {*-control_args*}

ARGUMENTS

mas_path
is the pathname of the master lister file. The suffix lister must be the last component of the lister file name; however, if *mas_path* does not have a suffix of lister, one is assumed.

up_path
is the pathname of the update lister file. The suffix lister must be the last component of the lister file name; however, if *up_path* does not have a suffix of lister, one is assumed.

out_path
is the pathname of the output lister file. The suffix lister must be the last component of the lister file name; however, if *out_path* does not have a suffix of lister, one is assumed. If this argument is not specified, the master lister file is replaced.

CONTROL ARGUMENTS

-field_name *fn1 ... fni*
-fn *fn1 ... fni*
specifies that fields *fn1* through *fni* are used as the controlling fields for the merge. (Records can only be merged if they contain the same fields, though some of those fields may be null.) The fields are compared without regard to case. If this control argument is not specified, then all fields are used to control the merge.

-totals
-tt
displays the number of records in the master, update, and output files.

Only one of the following four control arguments (-add, -and, -or, or -subtract) can be specified:

-add
copies into the output lister file all records from the master lister file *plus* all records from the update lister file. Thus records contained in both lister files are listed twice in the output file. (Default)

-and
copies into the output file those records in the master lister file that are *also* in the update lister file. That is, those records that are listed in both files are listed once in the output file; no records from the update lister file are copied.

-or
copies into the output lister file all records in *either* the master lister file *or* the update lister file. Duplicate records are copied only from the update lister file and thus appear only once in the output file.

-subtract
-sub
copies into the output lister file all records in the master lister file that are *not* also contained in the update lister file. Thus no duplicate records are copied and no records from the update lister file are copied.

NOTES

The table below shows how master and update lister files are merged for each of the four merge operations: *add*, *and*, *or*, and *sub*. The letters listed in the table body represent individual records, with duplications of letters simply representing different recordings of the same basic record. When records represented in both the master and update files are listed in the output file, the letters representing them are given the associated numeric shown in parenthesis with the identified file in order to indicate which recording of a particular record actually went into the output file.

Operation	Master File(1)	Update File(2)	Output File
add	a b c d e	d e f g h	a b c d1 d2 e1 e2 f g h
and	a b c d e	d e f g h	d1 e1
or	a b c d e	d e f g h	a b c d2 e2 f g h
sub	a b c d e	d e f g h	a b c

EXAMPLES

To copy into Boston_patients.lister all records in patients.lister that have the city field equal to Boston and print the total number of records, type:

```
copy_list patients Boston_patients -sel "city equal Boston" -tt
```

To delete from patients.lister all records that have the city field equal to Boston and print the total number of records, type:

```
! trim_list patients -sel "city equal Boston" -tt
trim_list: 1 record deleted.
```

To merge the lister files patients.lister and Boston_patients.lister using the city name as the controlling field for the merge and display the total number of records, first the

sort_list command is issued to sort the patients file into ascending alphabetical order by city:

```
sort_list patients -sort city
```

and then the merge:

```
!   merge_list patients Boston_patients -tt -fn city
    merge_list: 3 master and 1 update records merged into 4 output records.
!   merge_list patients Boston_patients out_patients -tt -fn city -and
    merge_list: 4 master and 1 update records merged into 1 output record.
```

modify_list (mdls)

The modify_list command modifies a field or fields in selected lister records.

SYNTAX AS A COMMAND

mdls *path -control_args*

ARGUMENTS

path
is the pathname of the lister file. The suffix lister must be the last component of the lister segment name; however, if *path* does not have a suffix of lister, one is assumed.

CONTROL ARGUMENTS

One or more arguments *must* be chosen from the following:

-brief_errors
-bfe
suppresses the warning when no records match the selection expression.

-field_name *field_name string*
-fn *field_name string*
causes the value of *string* to be assigned to the field indicated by *field_name*. If *string* contains spaces, it must be enclosed in quotes. This control argument is required and may be given more than once.

-select *string*
-sel *string*
specifies those records to be modified. If this control argument is not specified, then all the records are modified. (For a complete description of how to specify *string*, type "help process_list".)

-string *string*
-str *string*
uses *string* as a character string with no special interpretation. This is useful for preventing *string* from being interpreted as a control argument. It is to be used with the -field_name control argument (e.g., "-field_name rating -string -20").

-total
-tt
displays the number of records modified.

process_list (pls)

The process_list command produces a document from all or selected records in a lister file. The format of the document is defined in a listform file. Other text processors, such as compose, may be used to further format the document. By default, the document is printed on the user's terminal. Alternatively, it may be saved in a segment. For a description of the structure of a listform file and information on field insertion, angle bracket escapes, and the selection and sorting procedures (-select and -sort control arguments), see those earlier portions of this section.

SYNTAX AS A COMMAND

pls *list_path* {*form_path*} {*-control_args*}

ARGUMENTS

list_path
is the pathname of the lister file to be processed. The suffix lister must be the last component of the lister file name; however, if *list_path* does not have a suffix of lister, one is assumed.

form_path
is the pathname of the listform file that defines the format of the document. If *form_path* does not have a suffix of listform, one is assumed. If this argument is not specified, a listform file in the working directory is used that has the same entryname as *list_path*, with the entryname suffix of lister changed to listform.

CONTROL ARGUMENTS

-arguments *string*
-ag *string*
indicates that the listform segment requires arguments. If present, it must be followed by at least one argument. All arguments following this control argument on the command line are taken as arguments to the listform segment. *Thus, if present, this must be the last control argument on the command line.*

-brief_errors
-bfe
suppresses warnings about missing or extra arguments for the -ag control argument. Suppresses warning when no records are selected.

-extend
-ex
specifies that the document produced by this command is to be appended to the segment specified by path (-output_file must also be given). The default is to replace path completely.

-output_file {*path*}
-of {*path*}
specifies that the document produced by this command is saved in the segment specified by *path* (see Sample Letter in "Sample List Processing Files" earlier in

this section). If *path* is not specified, this output segment is placed in the working directory with an entry name the same as form_path and the suffix listform changed to list.

-select *string*
-sel *string*
specifies the records selected for processing. If this control argument is not specified, then all records in the list are processed (see "Selection" earlier in this section).

-sort *string*
-st *string*
sorts the records processed according to *string*, which is a string enclosed in quotes. The new ordering of the list is in effect only for the duration of the command. The lister file is not modified. If this control argument is not specified, then records are processed in the order in which they currently appear in the lister file (see "Sorting" earlier in this section).

-totals
-tt
displays the number of records processed.

EXAMPLES

Since the process_list command is an intermediate step in List Processing operations, assume that the user has already created a segment named students.listin, containing the first name, last name, city, state, and zip of three students; this segment includes three records, each consisting of the above mentioned five fields. Also assume that, using the create_list command, students.lister has been created, and a format for the list, names.listform also exists. Following is a copy of the segment students.listin:

```
Record_delimiter: $;
Field_delimiter: =;
Field_names: fname,lname,city,state,zip;
Records:
$
 =fname
 =lname    Smith
 =city     Boston
 =state    MA
 =zip      02114
$
 =fname Tim
 =lname    Jones
 =city     Cambridge
 =state    MA
 =zip
$
 =fname    Victor
 =lname    Red
 =city     Cambridge
 =state    MA
 =zip      02139
$
```

The first record in this segment has a null fname, and the second record contains a null zip field. As shown in the first field (fname) of the second record, the amount of white space between the field name and the field value is completely arbitrary (as is the space between field value and field delimiter), and makes no difference when processing.

The following listform segment does not utilize the optional *before* or *after* sections, so it creates no heading or ending lines in the final output. Following is a copy of the format-defining segment, names.listform:

```
<Begin record:>

<fname> <lname>
<city>, <state> <zip>
<end;>
```

To have the process_list command select (print) all records that have a last name (lname) field less than "m" (i.e., all persons whose last name is from A to L), type:

```
!   pls students names -sel "lname less m"
    Tim Jones
    Cambridge, MA
    r 725 0.401 3.782 61
```

To have the process_list command select (print) all records that have any field whose value is null, type:

```
!   pls students names -sel ":any equal :null"
    Smith
    Boston, MA 02114

    Tim Jones
    Cambridge, MA
    r 726 0.192 0.006 2
```

To have the process_list command select (print) all records that do not have a city field of Boston, type:

```
!   pls students names -sel "city not equal Boston"
    Tim Jones
    Cambridge, MA

    Victor Red
    Cambridge, MA 02139
    r 727 1.053 2.682 4
```

To have the process_list command select (print) all records and save the result in the segment names.list in the working directory, type:

```
pls students names -of
```

sort_list (sls)

The sort_list command sorts the records in the specified lister file. The records are sorted according to the fields specified in the -sort control argument (see "Sorting" above). Fields are sorted without regard to case; that is, they are sorted into alphabetical order and not ASCII order.

SYNTAX AS A COMMAND

sls *path -control_arg*

ARGUMENTS

path
is the pathname of the lister file to be sorted. The suffix lister must be the last component of the lister file name; however, if *path* does not have a suffix of lister, one is assumed.

CONTROL ARGUMENTS

control_arg
must be -sort *string or* -st *string* to specify how the records in the lister file are to be sorted (see "Sorting" above). If the command is invoked without specifying this control argument, the sort_list command responds with a two-line message showing proper usage.

EXAMPLES

To sort the list of patient records into ascending alphabetical order by the zip field, type:

```
sort_list patients -st zip
```

To sort the list of patient records into descending alphabetical order (most recent first) by the date field, type:

```
sort_list patients -st "date -dsc"
```

Normally, an alphabetical sort cannot be used to sort dates, but when the dates are of the form YYMMDD (year, month, day), an alphabetical sort correctly orders the dates.

To sort the list of patient records into descending alphabetical order (most recent first) by the date field, with those records having the same date sorted into ascending alphabetical order by the lname field, type:

```
sort_list patients -st "date -dsc lname -asc"
```

To sort the list of patient records into ascending alphabetical order by the lname (last name) field, type:

```
sort_list patients -st "lname fname"
```

If some records have equal lname fields, they are further sorted (ascending, alphabetically) by fname (first name).

To sort the list of patient records into ascending numerical order by the zip field (zip code address), type:

```
sort_list patients -st "zip -num"
```

trim_list (tls)

The trim_list command deletes selected records from the specified lister file. Because selection is required for trimming any lists, the select control argument must be used with this command (see "Selection" above).

SYNTAX AS A COMMAND

tls *path -control_arg {-optional_arg}*

ARGUMENTS

path
is the pathname of the lister file being trimmed. The suffix lister must be the last component of the lister file name; however, if *path* does not have a suffix of lister, one is assumed.

CONTROL ARGUMENTS

control_arg
must be -select *string* *or* -sel *string* to specify the records selected for deletion. This is a *required* argument.

optional_arg
can *only* be -totals *or* -tt to display the number of records deleted.

EXAMPLES

To select (i.e., delete) from the file patients.lister all records that have an fname (first name) field equal to John and an lname (last name) field equal to Smith, type:

```
trim_list patients -sel "fname equal John and lname equal Smith"
```

To select from patients.lister all records that have an lname equal to Doe or Jones, type:

```
trim_list patients -sel "lname equal Doe or lname equal Jones"
```

To select from patients.lister all records that have a state field equal to MA or IL, and have a zip field that is not null, type:

```
trim_list patients -sel "(state equal MA or state equal IL)
   and zip not equal :null"
```

To select from patients.lister all records that have a street field that contains the substring "PO BOX", type:

```
trim_list patients -sel "street contains ""PO BOX"""""
```

Notice the extra set of quotes required for the test string (see "Selection" earlier in this section for description of quotation marks in -select_arg).

To select from patients.lister all records that contain any field containing the substring "PO BOX", type:

```
tls patients -sel ":any contains ""PO BOX"""""
```

APPENDIX A

COMPOSE METACHARACTER TABLE

This Appendix shows the byte value assignments for the extended character set (metacharacters) used by compose while constructing the coded page image structure. The characters defined in this table may appear in the image, both singly and in various combinations. The device writer procedure decodes the charaters into printable characters and control sequences acceptable to the target device.

000 NUL	046 &	114 L	162 r
001 SOH	047 '	115 M	163 s
002 STX	050 (	116 N	164 t
003 ETX	051)	117 O	165 u
004 EOT	052 *	120 P	166 v
005 ENQ	053 +	121 Q	167 w
006 ACK	054 ,	122 R	170 x
007 BEL	055 -	123 S	171 y
010 BSP	056 .	124 T	172 z
011 HT	057 /	125 U	173 {
012 NL	060 0	126 V	174 |
013 VT	061 1	127 W	175 }
014 FF	062 2	130 X	176 ~
015 CR	063 3	131 Y	177 DEL
016 RRS	064 4	132 Z	200
017 BRS	065 5	133 [	201
020 DLE	066 6	134 \	202
021 DC1	067 7	135]	203
022 DC2	070 8	136 ^	204
023 DC3	071 9	137 _	205
024 DC4	072 :	140 `	206
025 NAK	073 ;	141 a	207
026 SYN	074 <	142 b	210
027 ETB	075 =	143 c	211
030 CAN	076 >	144 d	212
031 oct31	077 ?	145 e	213
032 SUB	100 @	146 f	214
033 ESC	101 A	147 g	215
034 FS	102 B	150 h	216
035 GS	103 C	151 i	217
036 RS	104 D	152 j	220
037 US	105 E	153 k	221
040 SP	106 F	154 l	222
041 !	107 G	155 m	233
042 "	110 H	156 n	224
043 #	111 I	157 o	225
044 $	112 J	160 p	226
045 %	113 K	161 q	227

230		316	prll	404	sup4	472	}ht
231		317		405	sup5	473	}md
232		320	PI	406	sup6	474	}bt
233		321		407	sup7	475	}hb
234		322		410	sup8	476	}fl
235		323		411	sup9	477	lptp
236		324	tmark	412	EM	500	lpht
237		325		413	EM_	501	lpmd
240		326	tfore	414	EN	502	lpbt
241		327		415	EN_	503	lphb
242		330		416	ENd	504	lpfl
243		331		417	THIN	505	rptp
244		332	approx	420	DEVIT	506	rpht
245		333		421	lquote	507	rpmd
246		334		422	rquote	510	rpbt
247		335		423	multiply	511	rphb
250		336		424	modmark	512	rpfl
251		337	infin	425	daro	513	ltp
252	mlpy	340		426	dbot	514	lht
253	pl_mi	341		427	dlvert	515	lmd
254	nabla	342		430	delmark	516	lbt
255	EMd	343		431	drvert	517	lhb
256		344		432	dtop	520	lfl
257	slash	345		433	laro	521	lltp
260		346		434	one{	522	llht
261	dagger	347		435	one[	523	llmd
262		350		436	lcirc	524	llbt
263		351		437	one(	525	llhb
264		352	theta	440	raro	526	llfl
265		353		441	one}	527	art
266		354		442	one]	530	art
267		355		443	rcirc	531	/onehi
270		356		444	one)	532	vrule
271		357		445	uparo	533	hstrt
272		360	pi	446		534	hline
273	perpen	361		447	[tp	535	hterm
274		362		450	[ht	536	lslnt
275	not_eq	363		451	[md	537	rslnt
276		364		452	[bt	540	boxtl
277	PAD	365		453	[hb	541	boxt
300		366		454	[fl	542	boxtr
301	dbldag	367		455	]tp	543	boxl
302		370		456	]ht	544	box+
303	cright	371		457	]md	545	boxr
304	delta	372		460	]bt	546	boxbl
305		373		461	]hb	547	boxb
306		374		462	]fl	550	boxbr
307		375	square	463	{tp	551	loztl
310		376	overbar	464	{ht	552	loztr
311		377	PS	465	{md	553	lozl
312		400	sup0	466	{bt	554	lozr
313		401	sup1	467	{hb	555	lozbl
314		402	sup2	470	{fl	556	lozbr
315	bullet	403	sup3	471	}tp	557	

560	624	670	734
561	625	671	735
562	626	672	736
563	627	673	737
564	630	674	740
565	631	675	741
566	632	676	742
567	633	677	743
570	634	700	744
571	635	701	745
572	636	702	746
573	637	703	747
574	640	704	750
575	641	705	751
576	642	706	752
577	643	707	753
600	644	710	754
601	645	711	755
602	646	712	756
603	647	713	757
604	650	714	760
605	651	715	761
606	652	716	762
607	653	717	763
610	654	720	764
611	655	721	765
612	656	722	766
613	657	723	767
614	660	724	770
615	661	725	771
616	662	726	772
617	663	727	773
620	664	730	774
621	665	731	775
622	666	732	776 CMODE
623	667	733	777 GMODE

APPENDIX B

REFERENCE TO COMMANDS/SUBROUTINES BY FUNCTION

This appendix contains the Multics commands and subroutines that are part of the WORDPRO system, arranged according to function.

WORDPRO COMMANDS

`compdv`
Translates a device description file into a binary table for use by the Formatter.

`compose (comp)`
Prepares formatted documents from raw text segments for various documentation devices using an extensive list of text formatting control lines and control arguments to the compose command.

`compose_index`
Produces a cross-reference index file from raw data.

`convert_runoff (cv_rf)`
Converts a runoff input segment into a compose input segment.

`display_comp_dsm (ddsm)`
Displays selected information from a compose device description table.

`expand_device_writer (xdw)`
Expands an expansion input file into an expansion output file.

`format_document (fdoc)`
Prepares formatted documents from raw text segments using a limited set of text formatting control lines.

`process_compout (pco)`
Processes one or more compose output files to an online device, or to a magnetic or punched paper tape.

DICTIONARY COMMANDS/SUBROUTINES

`add_dict_words (adw)`
Adds words to a WORDPRO dictionary.

`count_dict_words (cdw)`
Counts words in a WORDPRO dictionary.

`create_wordlist (cwl)`
Creates a wordlist segment from a text segment.

`delete_dict_words (ddw)`
Deletes words from a WORDPRO dictionary.

`find_dict_words (fdw)`
Finds words in the set of WORDPRO dictionaries defined by the search facility.

`hyphenate_word_`
Returns the character position at which a word can be hyphenated.

`list_dict_words (ldw)`
Lists words in a WORDPRO dictionary.

`locate_words (lw)`
Locates all occurrences of one or more words in a text segment.

`print_wordlist (pwl)`
Displays the words in a wordlist segment.

`revise_words (rw)`
Replaces all occurrences of one or more words in a text segment with a corresponding revision.

`trim_wordlist (twl)`
Deletes all words in a wordlist segment that can be found in a specified sequence of dictionaries.

SPEEDTYPE COMMANDS

`add_symbols (asb)`
Adds symbols to the current symbol dictionary.

`change_symbols (csb)`
Changes the expansion or suffixing of a symbol in the current symbol dictionary.

`delete_symbols (dsb)`
Deletes symbols from the current symbol dictionary.

`expand_symbols (esb)`
Expands all the symbols in a specified text segment.

`find_symbols (fsb)`
Finds and lists symbols in the current symbol dictionary that represent specified expansions.

`list_symbols (lsb)`
Lists symbols in the current symbol dictionary.

`option_symbols (osb)`
Sets options in the current symbol dictionary.

`print_symbols_path (psbp)`
Displays the pathname of the current symbol dictionary.

`retain_symbols (rsb)`
Retains all symbols in a specified text segment by placing a Speedtype escape in front of each symbol.

`show_symbols (ssb)`
Expands an input string and displays the output string.

`use_symbols (usb)`
Sets the current symbol dictionary.

LIST PROCESSING COMMANDS

`append_list (als)`
Adds a record to a lister file.

`copy_list (cpls)`
Creates a new lister file from an existing lister file.

`create_list (cls)`
Creates a lister file from a listin file.

`describe_list (dls)`
Displays information about a lister file.

`display_list (dils)`
Displays selected portions of selected lister records.

`expand_list (els)`
Creates a listin segment from a lister segment.

`merge_list (mls)`
Combines two lister files into a single lister file.

`modify_list (mdls)`
Modifies a field or fields in selected lister records.

`process_list (pls)`
Produces a document from selected records in a lister file.

`sort_list (sls)`
Sorts the records in a lister file.

`trim_list (tls)`
Deletes selected records from the specified lister file.

APPENDIX C

DEVICE SUPPORT TOOLS

DEVICE WRITER SOURCE EXPANDER

The **Device Writer Source Expander** is a special adaptation of a general text string manipulation facility that expands a device writer source expansion input file into an expanded device writer source output file. A device writer source expansion input file is a mixture of literal text and expansion constructs. The corresponding expanded device writer source output file contains the literal text, as-is, with the expansion constructs replaced by their corresponding strings (if any) that may be compiled with the PL/I compiler to obtain the device writer object module. Throughout the remainder of this section, these two files are referred to simply as the *input file* and the *output file* and the Device Writer Source Expander is referred to simply as the *Expander*.

The Expander provides these features:

- Variables and arrays with three data storage classes
- Value assignment
- Expression evaluation
- Iteration
- Conditional execution
- Internal and external expansion calling
- Active function calling

The language has intentionally been made very context-sensitive in order to allow, as much as possible, literal text to be entered as it is to be generated.

EXPANSION CONSTRUCTS

Expansion constructs are made up of expansion tokens, white space (the ASCII "motion" characters, SP, HT, LF, etc.), and literal text. Throughout the remainder of this section, **expansion constructs** are referred to simply as *constructs*.

Expansion tokens consist of an ampersand (&) followed by zero or more alphanumeric characters followed by one non-alphanumeric character. There are two types of tokens: keyword tokens and terminator tokens (See "Expansion Tokens" below for a complete list of tokens). For some tokens, the non-alphanumeric character is taken as part of the token; for others, it is considered part of the following input text; for still others (if it is white space), it is discarded. Throughout the remainder of this section, **expansion tokens** are referred to simply as *tokens*.

Every construct must begin with a keyword token and each has a very specific termination condition. There are four different classes of constructs, as determined by termination conditions.

- Self-terminating
- Matching character terminator
- General terminator token

- Specific terminator token

Constructs can be nested. When they are, the beginning and ending of each nested construct must be totally within all containing constructs. Any expansions produced by this nesting are "protected," that is, they do not change the existing syntax of the original containing construct.

In the following descriptions, constructs are defined with the skeleton:

kkk bodyttt

where

kkk
is the keyword token

body
is the body (literal text, possible white space, and nested bodies and constructs) of the construct and may be null. (This term is used in conjunction with many of the Expander features.)

The white space shown in the skeleton above preceding the body is always discarded and is *not* shown the definitions following. Any white space within the body or between the body and the terminator is either discarded or sent to the output file as the user directs. Literal text is sent to the output file without modification.

The use of the term "body" in the descriptions following implies that it is evaluated as an expression (see "Expression Evaluation" below).

ttt
is the terminator token and may be null

EXPANSION DEFINITIONS

An expansion definition begins with a definition keyword token and name, ends with a specific definition terminator token, and has a body that is a mixture of literal text, white space, and constructs consisting of any number of lines (including zero). There are two forms: the static form (shown first) and the dynamic form.

&expand<SP>*expansion-name*<NL>*expansion-body*&expend<NL>
&define<SP...>*expansion-name*<NL>*expansion-body*&dend<NL>

The *expansion-name* may be up to 26 characters long, must begin with an alphabetic, and contain only alphanumerics and "_". <SP> and <NL> are required characters. <SP...> represents one or more <SP> characters.

A **static expansion definition** may exist as all or an independent part (that is, an unnested fragment) of an expansion input file or an expansion library file. An expansion library file consists of *only* static expansion definitions with possible interspersed commentary. It produces no output other than the commentary when expanded from Multics command level since the commentary does not contain expansion calls. Any expansion input file may be accessed as though it were an expansion library file in order to use static expansions defined therein (as long as the naming requirements are met). The files may be free-standing segments or archive components and must have the name *name.xdw*.

A **dynamic expansion definition** may exist only as a nested construct within a static expansion definition or an expansion input file.

The essential difference between the two forms is that the use of a static expansion yields the *expansion-body* as given but the use of a dynamic expansion yields a static expansion definition having the expanded *name* and *expansion-body* of the dynamic expansion.

Any attempt to redefine a static expansion (by either of these two forms) in the same invocation of the Expander is an error; however, if the new definition is character-for-character identical with the existing definition, the attempted redefinition is ignored.

Examples

The static form:

```
&expand ck
          if (code ^= 0)
          then do;
             call com_err_ (code, "&name", &1);
             return;
          end;
&expend
```

Using this static expansion yields the five PL/I code lines with the two nested constructs expanded and including all the white space.

The dynamic form:

```
&expand exp_def
&define A
<Abody>;&dend
&define B
<Bbody>;&dend
&expend
```

Using the static expansion `df` yields the two new static expansions:

```
&expand A
<expanded-Abody>;&expend
&expand B
<expanded-Bbody>;&expend
```

VARIABLES AND ARRAYS

A **variable declaration construct** begins with a declaration keyword token and a name, ends with a general terminator token, and may contain array extents and/or an initial value. There are three data storage classes, each with its own keyword token:

Local, like PL/I automatic; keyword token "&loc"
Internal, like PL/I internal static; keyword token "&int"
External, like PL/I external static; keyword token "&ext"

All *variables* must be declared before they can be referenced. Local data are available only for the current invocation of the expansion in which they are declared. Internal data are available at any time after declaration for any invocation of the expansion in which they are declared. External data are available at any time after declaration in any expansion file.

Variable names may be up to 16 characters long, must begin with an alphabetic character, and contain only alphanumerics and "_". There is no conflict between

variable names and expansion names because of the syntax; however, scalar variable names and array variable names conflict. When searching for a variable name, the Expander searches the data classes in the order shown above.

Variable values are 9-bit byte character strings with a minimum size of zero and a maximum size of 1,044,480 (as determined by maximum segment size). *Numeric values* may be only decimal numbers (that is, no coded exponential forms like 2.4e4 or 10**3), have a maximum magnitude of 10s48 power and a resolution of nine decimal places. Variables declared without initial values are initialized with a null string. Throughout the remainder of this section, *assign* means that a value is given to the variable and *access* means that the value of the variable replaces the reference.

An attempt to redeclare an existing variable with different attributes is an error; however, a redeclaration with identical attributes is ignored.

All the declarations described below can define variables in any of the classes, however, the declarations and examples show only Local variables, that is, use the "&loc" keyword tokens. In any of them, "&int" or "&ext" may be substituted for the "&loc".

Scalar Variables

&loc *name*&;
&loc *name=initial-value-body*&;

Scalar variables may be declared with or without initial values. If *initial-value-body* is given, it is assigned as the initial value of *name*.

Examples

```
&loc stuff&;
```
declares a Local scalar with no initial value.

```
&loc one=1&;
```
declares a Local scalar with an initial value.

```
&loc copy_it=&it&;
```
declares a Local scalar with the current value of another variable as its initial value.

Array Variables

In this section, the terms "scalar reference", "subscripted reference", and "array reference" are used in discussing references to an array variable. The definitions of these references are given below.

Note: Throughout this section, the braces ({}) shown in reference to array variables are required as part of the construct syntax and do not mean that the enclosed expr is optional.

scalar	*&name*
subscripted	*&name{expr}* or *&name{expr1:expr2}*
array	*&name{}*

The term "*expr*", appearing here for the first time, refers to a construct that is evaluated as an arithmetic expression (see "Arithmetic Expressions" below).

The limit for the upper and lower bounds of arrays is 34,359,738,367 as determined by the maximum positive binary integer. The limit for the extent of arrays is 130,558 as determined by maximum segment size.

Any reference to any array element outside the declared extent is an error. An array access to an array with an empty extent is replaced with a null string.

FIXED ARRAYS

&loc *name*{*expr1*:*expr2*}&;
&loc *name*{*expr1*:*expr2*}=*initial-value-body*&;

Fixed arrays may be declared with or without initial values and have non-varying extents as determined by the upper and lower bounds given in their declarations. *expr1* specifies the lower bound and *expr2* specifies the upper bound. If *initial-value-body* is given, it is assigned to each element of the array being created. Fixed array elements are assigned and accessed with subscripted or array references.

Examples

```
&loc ten_nulls{1:10}&;
```
declares a Local fixed array with 10 null elements.

```
&loc fifty_5s{1:50}=5&;
```
declares a Local fixed array containing 50 elements with initial value 5.

```
&loc holders{&first:&last}&;
```
declares a Local fixed array whose extent is determined by the current values of other variables.

VARYING ARRAYS

&loc *name*{*expr1*:*expr2*}var&;

A varying array is like a fixed array except that it must be declared without initial values and the upper and lower bounds are adjusted dynamically as elements are assigned values. *expr1* specifies the minimum lower bound and *expr2* specifies the maximum upper bound. When created, the array is empty with no extent. Varying array elements are assigned and accessed with subscripted or array references, but an attempted access outside the current extent is an error.

Examples

```
&loc some_0s{1:25}var&;
```
declares a Local varying array to hold up to 25 elements.

```
&loc twoway_array{-&size:&size}var&;
```
declare a Local varying array whose maximum extent is one more than twice the value of some other variable.

LIST ARRAYS

&loc *name*{*expr*}list&;

A list array is a set of unique elements and must be declared without initial values. *expr* is the maximum number of elements the list is to hold. When created, the list is

empty with no extent. A list assignment is made with a scalar reference; the list is searched to see if the given value is there and it is added if the search fails. A list is accessed with subscripted or array references, but an attempted access outside the current extent is an error. Lists are ordered according to the order in which assignments are made.

Example

```
&loc et_dcls{20}list&;
```
declares a Local list array that can hold 20 entries.

STACK ARRAYS

&loc *name*{*expr*}fifo&;
&loc *name*{*expr*}lifo&;

A stack array may be either a **push-down/pop-up stack** (last-in-first-out or lifo) or a **linear delay queue** (first-in-first-out or fifo) and must be declared without initial values. *expr* is the maximum number of elements the stack is to hold. A stack assignment is made with a scalar reference, the given value being added as the newest element. A scalar access to a fifo stack causes the oldest element to be accessed and deleted. A scalar access to a lifo stack causes the newest element to be accessed and deleted. A stack array may also be accessed with subscripted references, but these references cause no "movement" of the stack. The subscript value 0 accesses the top-of-stack (or next-out) element, -1 accesses the next-to-top element, etc. An array access to a stack is an error.

Examples

```
&loc push_stack{25}lifo&;
```
declares a Local push/pop stack that holds up to 25 entries.

```
&loc queue{10}fifo&;
```
declares a Local queue with 10 elements.

VALUE ASSIGNMENT

&let *name=value-body*&;
&let *name*{*expr*}=*value-body*&;
&let *name*{*expr1*:*expr2*}=*value-body*&;

An **assignment construct** begins with an assignment keyword token and a name, ends with a general terminator token, and may contain array subscripts or ranges and/or a value. A value may be assigned to a scalar, an array element, or a range of array elements. If a range is specified, *value-body* is assigned to every array element in the range.

Examples

```
&let feet_per_mile=5280&;
```
assign a value to a scalar.

```
&let var{2}=&var{1}&;
```
assigns the value of the first element of the array to the second element.

```
&let array{1:5}=3&;
```
assign "3" as the value of the first five elements of array.

EXPRESSION EVALUATION

An **expression** is a collection of constructs, variable accesses, other embedded expressions, literal text, and possible white space that is replaced by the Expander with a single character string representing its value. The result of evaluating an expression is a string value or numeric value that must obey the limits mentioned earlier.

Accessing Variables

Variables are accessed by using their *name*s as though they were keyword tokens. The termination conditions for the constructs thus created depend on the form of reference and are specified in the descriptions following.

SCALAR ACCESSES

&name

The keyword token becomes a self-terminating construct, but the construct terminator (the non-alphanumeric character following the token) may not be "(" or "{". The construct is replaced by the value of *name*. This form of access may be made to scalars and stack arrays. Referencing a stack array causes the accessed value to be removed from the stack. (See "Stack Arrays" above.)

Examples

```
&let Var=foo&;
```
then:
```
&Var any literal text ...
```
becomes `foo any literal text ...`
```
if &Var>0
```
becomes `if foo>0`
```
&Var&.bar
```
becomes `foobar`
```
&Var&.(1)
```
becomes `foo(1)`

SUBSCRIPTED ACCESSES

&name{expr}
&name{expr1:expr2}
&name{expr1:expr2,string-body}

The non-alphanumeric character following *name* must be "{" and becomes part of the keyword token. The token begins a construct that is terminated by the matching "}". *expr* or *expr1* and *expr2* are evaluated (see "Arithmetic Expressions" below) to obtain the element or range of elements to be accessed.

The first form above may be used to access all array types and the construct is replaced by the value of the selected array element.

Examples

`&fixed_array{5}`
is replaced with the value of the fifth element of `fixed_array`.

`&list{&last}`
is replaced by the value of that element of `list` whose list position is given by the value of `last`.

`&stack{0}`
is replaced by the value of the next element to be recovered from the `stack`.

The second form may be used to access fixed, varying, and list arrays and the construct is replaced by the list of values of the selected range of array elements, separated by a single blank character. A subscripted access to any unassigned element in the declared extent is replaced with a null string.

Examples

`&varying_array{-2:3}`
is replaced with the six elements of `varying_array` whose subscript values lie between -2 and 3, inclusive, separated by single blanks.

`&list{1:&last}`
if `last` contains the extent of `list`, it is replaced by the entire contents of `list` separated by single blanks.

The third form may be used to access fixed, varying, and list arrays and the construct is replaced by the list of values of the selected range of array elements, separated by *string-body*. The length of *string-body* is limited to 150 characters. Literal appearances of "&" and "}" in *string-body* must be protected (see "Protected Strings" below). If the selected range is empty, the construct is replaced with a null string.

Example

`&A_list{1:&last,, }`
if `last` contains the extent of `A_list`, it is replaced by the entire contents of `A_list` separated by the string ", ".

ARRAY ACCESSES

&name{}
&name{,string-body}

Array accesses are a special case of subscripted accesses where the subscript expression is given as a null string rather than being evaluated to a null string, implying an empty range. The usage of array accesses is identical to the second and third forms of subscripted accesses above except that the range is the entire extent of the array. An array reference to a varying array, a list , or a stack addresses the current extent, not the declared extent.

Example

`&A_list{,, }`
is replaced by the entire contents of `A_list`, separated by the string ", ". (Note that result of this access is the same as that of the previous example, but does not depend on the value of some other variable.)

Accessing Arguments

An expansion may be called with a list of arguments to be used as parameters. (See "Expansion Calling" below.) The called expansion may access these arguments with argument access expressions. Arguments are accessed by using their argument list position numbers as though they were keyword tokens. The termination conditions for the constructs thus created depend on the form of reference and are specified in the descriptions following.

SINGLE ARGUMENT ACCESSES

&n
&nn

The token forms a self-terminating construct that is replaced by the value of the argument having the given position in the argument list. The construct terminator is the first non-numeric character and the number may not have more than two digits. If the reference is to an argument beyond the argument list, the construct is replaced with a null string.

Examples

`&3`
is replaced with the value of the third argument.

`&05`
is replaced with the value of the fifth argument.

`&14`
is replaced with the value of the fourteenth argument.

MULTIPLE ARGUMENT ACCESSES

&{expr}
&{expr1:expr2}
&{expr1:expr2,string-body}

At times it is necessary to reference an argument via the value of a variable, or to reference more than one argument. This is done by accessing the argument list as though it were a nameless array (see "Array Accesses" above). The keyword token is "&{" and begins a construct that is terminated by a matching "}".

Examples

`&{&arg_counter}`
is replaced by the argument whose list position is given by `arg_counter`.

`&{2:4}`
is replaced by a list of the values of the second, third, and fourth arguments, separated by a single blank.

`&{1:3, + }`
is replaced by an expression representing the sum of the first three arguments.

```
declare &{, fixed bin (17);
declare } fixed bin (17);
```
creates PL/I declarations for all the arguments.

ARGUMENT COUNT

&*

This keyword token forms a self-terminating construct that is replaced by the number of elements in the argument list with which the expansion was called.

Example

```
&{&*}
```
is replaced with the value of the last argument regardless of how many have been given.

Protected Strings

&&

This keyword token forms a self-terminating construct that is replaced by a single ampersand.

Example

```
if flag && index > 0 then do;
```
creates a PL/I logic test statement.

&"*string-body*&"

The keyword token is "&"" and it begins a construct that is terminated by the next occurrence of the same keyword token and protects any literal string. The construct is replaced by the literal, unexpanded *string-body*. *string-body* may not contain an embedded protected string.

Example

```
{&array{,&"}&" {}}
```
forms a blank separated list of array elements, each enclosed in braces.

Arithmetic Expressions

&(expr)

The keyword token is "&(" and begins a construct that is terminated by the matching ")". *expr* may contain only decimal numeric literals, embedded arithmetic expressions, and arithmetic and relational operators. *expr* is first expanded as an expansion expression and then evaluated as an arithmetic expression. The value of the arithmetic expression replaces the construct.

The arithmetic operators supported are:

+ addition
- subtraction
/ division

* multiplication

() factor grouping

Examples

```
&let array{3}=&(&array{2}+1)&;
```
assign the third element of the array a value that is one greater than the second element.

```
&array{&(2*&2+1):&(2*&3+1)}
```
access a range of array elements given by the values of the second and third arguments where the subscript expression is "2N+1".

```
&(&array{,+})&;
```
is replaced with the sum of all the elements of the array.

Note: This construct fails if the array is not fully populated, that is, if it contains any null elements, since the resulting summation construct contains a double operator that gives rise to a missing operand error (see next example.) The result of summing an empty array is a null value.

```
&(&array{,+0})&;
```
the digit "0" ensures success of the construct by representing any null elements with "+0".

The relational operators have lower precedence than the arithmetic operators, that is, within a factor group, all arithmetic is completed before any relations are tested. The result of a relational test is given a numeric value "0" representing "false" or a numeric value "1" representing "true". In the evaluation of a relational test, any term with a non-zero value is considered true.

The relational operators supported are:

= equal

^= not-equal

> greater

< less

<= less-or-equal (not-greater)

>= greater-or-equal (not-less)

Examples

```
&(&1>0)
```
is replaced with "1" if the first argument is positive; otherwise, it is replaced with "0".

```
&((&array{1}^=0)+(&array{2}^=0))
```
is true if either (value = "1") or both (value = "2") of the first two elements of array are non-zero and false (value = "0") if both are zero. (Note here that the addition operator takes on the role of the Boolean OR operator.)

```
&let flag=&((&1^=0)*(&2^=0)*(&3^=0))&;
```
assigns "0" to `flag` if any of the first three arguments is zero and "1" if all three are simultaneously non-zero. (Note here that the multiplication operator takes on the role of the Boolean AND operator.)

ITERATION

An *iteration construct* begins with an iteration keyword token, ends with a specific construct terminator token, and contains a two-part iteration body and a test clause. The test clause begins with a test keyword token, ends with a general terminator token, and contains a test body.

&do *body1* &while *test-body*&; *body2* &od

Any of *body1*, *body2*, or *test-body* may be null, however, if *test-body* is null, it is considered absolutely true and the iteration never terminates. In order to establish effective control over the iteration, either *body1* or *body2* must modify the condition tested by *test-body*.

test-body may be an arithmetic relational expression as described in "Expression Evaluation" above, or may be a string expression of either of the forms:

string-body
string-body1 RELOP string-body2

where *RELOP* is any of the relational operators discussed in "Expression Evaluation" above. *string-body* is considered false if it has any of the values "0", "F", "FALSE", or "NO" (without regard to case); any other values are considered true. For the purposes of comparison, the shorter of *string-body1* and *string-body2* is padded out to the length of the longer with ASCII blanks and the values of the characters are determined by the ASCII collating sequence. In the first form, white space is stripped from both sides of *string-body*. In the second form, white space is stripped from the left sides of the *iestring-body*ies but is retained on the right sides.

The flow of control in the iteration proceeds as follows:

1. *body1* is executed.
2. *test-body* is evaluated. If it is false, control proceeds to the construct following the iteration construct terminator. If it is true, control proceeds to step 3.
3. *body2* is executed and control goes back to step 1.

Example

```
&let vv=&*&;
&do
(&{&vv})&+
&let vv=&(&vv-1)&;
&while &(&vv>0)&;
&od
```

creates a parenthesized, comma-separated list of all the arguments with the order of the arguments inverted. Note that an array access to the argument list can be used to create a similar list with the argument in their given order.

Note: The token &+ is a white space control token and serves only to improve the readability of the expansion input file. See "Miscellaneous Features" below.

CONDITIONAL EXECUTION

A conditional execution construct begins with a conditional keyword token and a test clause, ends with a specific construct terminator token, must contain a "then" clause, may contain any number of "elseif" clauses, and may contain a single "else" clause.

```
&if test-body &then then-body
    &elseif test-body &then then-body
    &else else-body
    &fi
```

The *test-body* is the same as that for the iteration construct described above.

Note: The format shown for this construct (multiline with indents) is for clarity of presentation only and is not required for correct usage.

If the evaluation of any *test-body* results in a true value, then the corresponding *then-body* is executed and the the rest of the construct is skipped. If no *test-body* is true, then the *else-body* is executed if present.

Examples

```
&if &(&(&1)>0) &then &let sign=+&;
    &else &let sign=-&;
    &fi
```

captures the arithmetic sign of the first argument. (Note here that the nested arithmetic expression in the test clause ensures that the argument is handled correctly if it is an expression rather than a value.)

```
&if &(&*<2) &then
        &error 4,Second argument missing.&; &return
    &fi
```

reports a calling sequence error and returns to the caller.

Note: See "Miscellaneous Features" below for descriptions of the &error and &return constructs.

EXPANSION CALLING

Expansions are called (that is, execution control passed to them) by using their names as though they were keyword tokens.

&expansion-name(arg-body1,arg-body2,...)

The non-alphanumeric character following *expansion-name* must be "(" and becomes part of the keyword token. The token begins a construct that is terminated by the matching ")".

expansion-name may be either *name* or *segment$name*. The Expander keeps an internal list of all expansions it has encountered during execution. When a reference to *name* is made, the list is searched for that *name*. If the search fails, *name* is promoted to *name$name* and an external search for that name is made using the expansion search list. An explicit reference to *segment$name* causes the Expander to forego searching the internal list and make a direct external reference to the segment, again using the expansion search list if the segment is not known.

Up to 99 string arguments may be passed in the call, and each is limited to 500 characters after leading white space is discarded.

Any ",", "(", or ")" characters resulting from the expansion of an *arg-body* are literal characters; that is, they do not contribute to the syntax of the call construct.

If any *arg-body* is enclosed in parentheses, then it is considered a list argument, that is, a parenthesized list of values passed as a single argument.

Examples

```
&a_exp(abc,def)
```
calls expansion a_exp with the arguments "abc" and "def".

```
&a_exp(&
abc,def&")"
```
calls expansion a_exp with the argument "abc,def".

```
&a_exp((abc,def))
```
calls expansion a_exp with the argument " (abc,def)".

```
&let var=abc,def&;&a_exp(&var)
```
calls expansion a_exp with the argument "abc,def".

```
&a_exp(&2,&b_exp())
```
calls expansion a_exp with two arguments; the second argument of the current expansion and the expansion of "b_exp".

```
&let name=&strip_suffix(&entry,.pll)&;
```
calls expansion strip_suffix with two arguments and assigns the resulting expansion to the variable name.

ACTIVE FUNCTION CALLING

&[*active-expr*]

The keyword token is "&[" and begins a construct that is terminated by the matching "]". *active-expr* and the active function return string is limited to 500 characters. *active-expr* is first expanded as an expansion expression and then processed as an active function. The active function return string replaces the construct.

Examples

```
This file created by &[user person] on &[date] at &[time].
```
generates an audit trail time-stamp.

```
The path is &[string [dir &1]>[file &2]].
```
generates an audit trail pathname.

MISCELLANEOUS FEATURES

The miscellaneous features discussed in this section are presented alphabetically and listed below.

- Built-in functions
- Comments
- Emptying Arrays
- Error Reporting
- Expansion Debugging
- General Terminator Token
- Null Separator Tokens

- Quote Processing
- Rescanning
- Return
- White Space Control

Built-in functions

Three built-in functions are provided.

- Length
- Substr
- Usage

LENGTH FUNCTION

&length *string-body*&;

The keyword token begins a construct that ends with the general terminator token and contains a string. The construct is replaced by the number of characters in *string-body*.

The function is supported internally (rather than requiring an active function call) because of its expected high frequency of use and because the string may contain white space.

Example

```
&length &1&;
```
is replaced by the number of characters in the first argument.

SUBSTR FUNCTION

&substr *string-body*, *expr1*&;
&substr *string-body*, *expr1*, *expr2*&;
&substr *string-body*, *expr1*: *expr2*&;

The keyword token begins a construct that ends with the general terminator token and contains a string and one or two subscript expressions. The length of *string-body* is limited to 16384 characters. Both *expr*s must refer to character positions within *string-body* or the input file is in error.

This function is supported internally (rather than requiring an active function call) because of its expected high frequency of use and the extended capabilities provided.

The first form above is replaced by that part of *string-body* from character position *expr1* to the end. If *expr1* is negative, then the character position is calculated from the end of *string-body* rather than from the start.

Examples

```
&substr abcdefg,3&;
```
is replaced by `cdefg`.

```
&substr abcdefg,-3&;
```
is replaced by `efg`.

The second form is replaced by that part of *string-body* from character position *expr1* for a total resultant string length of *expr2*. If *expr1* is negative, then the

character position is calculated from the end of *string-body* rather than from the start.

If the number of characters in *string-body* following the calculated character position is less than the magnitude of *expr2*, the resultant string is padded to the required length with ASCII space (SP) characters. If *expr2* is negative, the padding is to the left of the resultant strimg; otherwise, it is to the right. If no padding is needed, then the sign of *expr2* is immaterial.

Examples

```
&substr abcdefg,2,3&;
```
is replaced by bcd.

```
&substr abcdefg,2,-3&;
```
is replaced by bcd.

```
&substr abcdefg,3,5&;
```
is replaced by cdefg.

```
&substr abcdefg,3,8&;
```
is replaced by cdefg<SP><SP><SP>.

```
&substr abcdefg,-3,8&;
```
is replaced by efg<SP><SP><SP><SP><SP>.

```
&substr abcdefg,-3,-8&;
```
is replaced by <SP><SP><SP><SP><SP>efg.

The third form is replaced by that part of *string-body* from character position *expr1* to character position *expr2*. If either *expr* is negative, then the corresponding character position is calculated from the end of *string-body* rather than from the start. Both *exprs* must refer to character positions within *string-body* or the input file is in error. Further, *epxr2* must refer to a character position to the right of that given by *expr1*.

Examples

```
&substr abcdefg,3:5&;
```
is replaced by cde.

```
&substr abcdefg,-3:-2&;
```
is replaced by ef.

USAGE FUNCTION

&usage *ioa-ctl-string*&;

This function provides a means of documenting the expansions that are used in the generation of an expansion output file. In essence, it is a means of dumping the Expander's internal expansion reference list (see "Expansion Calling" above) in a format determined by the user. It should be used only in "primary" expansion files (that is, files intended for use in the command line invoking the Expander) and not within any expansion definition. Further, it should be the last construct in the file so as to not lose any references.

ioa-ctl-string is an ioa_ control string that describes the format of the output (see *Multics Subroutines* for a description of ioa_). It is passed to ioa_$rsnnl with three string arguments each of which must have a string conversion key (^a) in the control string. The three arguments are (in the order passed):

- the pathname of the directory containing the macro input file
- the entryname of the expansion input file
- the name of the expansion

Every expansion used appears once in the display and the order is the "natural" order, that is, the order in which the reference first appeared.

Example

```
&usage /* ^a>^a -- ^a */^/&;
```

generates a list of all expansion file pathnames and expansion names as PL/I comments at the end of a generated PL/I source file.

Comments

&comment *comment*&;

The keyword token begins a construct that ends with the general terminator token and contains a comment. The comment is treated as a literal string; it is not expanded and does not contribute in any way to the processing of the expansion.

Note: In this *single* case, the general terminator token does not change the existing white space supression action (see "White Space Control" below).

Emptying Arrays

&empty *array-name*&;

The keyword token begins a construct that ends with the general terminator token and contains an array name. The array is emptied by setting its extent to zero and all its elements to null.

Error Reporting

&error *sev-expr*, *err-body*&;

The keyword token begins a construct that ends with the general terminator token and contains a severity expression and an error message body. *sev-expr* must be an arithmetic expression in the range 0–4 and is used to select one of the message forms shown below. The formatted messages are written to the error_output I/O switch.

The error message forms are:

0 for the user's information

```
NOTE:  EXPANSION <name>, line <nn>.
          <err-message>
```

1 a minor error that does not affect the validity of the output

```
WARNING EXPANSION <name>, line <nn>.
          <err-message>
```

2 a substantive error that causes the output to be invalid

```
ERROR SEVERITY 2 EXPANSION <name>, line <nn>.
          <err-message>
```

3 a major error that prevents creation of the expansion output file but allows processing to continue in order to report additional errors

```
ERROR SEVERITY 3 EXPANSION <name>, line <nn>.
        <err-message>
```

4 a fatal error that prevents further processing of the expansion input file

```
ERROR SEVERITY 4 EXPANSION <name>, line <nn>.
        <err-message>
```

Examples

```
&error 0,This code does not reference any error_table_ entries.&;
&error 1,Second argument missing, "13" assumed&;
&error 2,Source syntax error. Program will not compile.&;
&error 3,Required sections not supplied.&;
&error 4,Table name not supplied.&;
```

General Terminator Token

&;

This token is used to signal the logical end of various other constructs. It does not contribute directly to the expansion output file.

Null Separator Tokens

&.

This token acts as a terminator token and enables the copying of white space within expansion constructs into the expansion output file. It is used to resolve ambiguities that might otherwise exist and to allow expansion constructs to create white space in the expansion output file. All white space between it and the next token is copied to the expansion output file; however, white space in any comments encountered is discarded as part of the comments.

&+

This token disables the copying of white space within expansion constructs into the expansion output file. It is used to suppress the copying of white space intended solely to improve the readability of the expansion input file. All white space (and comments) between it and the next token is discarded.

Examples

```
&a_exp(A,1)
...
&2&.0
```

the third construct is replaced by 10. Were the null separator token not present, the construct would be a reference to (nonexistent) argument 20 of the expansion call.

```
&if ...
loop: do ...
...
      end loop;&fi&.

else ...
```
the white line is copied to the expansion output file to separate the "end" and "else" statements.

```
&if ...
... of the people,&+

&comment end of fragment 1&;

&fi by the people, ...
```
the expansion output file contains "... of the people, by the people, ...".

Quote Processing

The Expander is internally language-independent. However, because it can communicate with the Multics operating system and may be used to generate source code for languages supported by Multics, it must be able to manipulate quoted strings in a manner consistent with that expected by Multics. A quoted string is any string of characters enclosed within ASCII double-quote (") marks and, for this usage, limited to 16384 characters.

&quote *body*&;

The keyword token begins a construct that ends with the general terminator token. The result of the construct is a string with all quote marks doubled. Note that *body* is *not* converted from an unquoted string to a quoted string.

&unquote *body*&;

The keyword token begins a construct that ends with the general terminator token. The result of the construct is a string with all doubled quote marks reduced to single quote marks. If *body* is a quoted string, it is converted to an unquoted string.

Examples

```
Processed on : &unquote &[date_time]&;
```
strips the quote marks from the string returned by the date_time active function.

```
call my_proc ("&quote &string_arg&;");
```
ensures that any quote marks within `string_arg` are correctly passed to the procedure.

Rescanning

&scan *body*&;

The keyword token begins a construct that ends with the general terminator token. In this construct, *body* is expanded normally and then the resulting expansion is re-expanded as though it were another *body*. Normally, any constructs appearing in an

expansion are "protected"; that is, they are not subjected to further expansion. In some applications, it is necessary that any such constructs be expanded.

Examples

```
&exp_1("a,b,&[time],d")
```
exp_1 is expanded with the single argument a,b,&[time],d and contains any of the following expansion calls.

```
&exp_2(&1)
```
exp_2 is expanded with one argument, a,b,&[time],d.

```
&exp_2(&scan &1&;)
```
exp_2 is expanded with one argument, a,b,08:21,d.

```
&scan &&exp_2(&1)&;
```
exp_2 is expanded with four arguments, a, b, 08:21, and d.

Return

&return

The keyword token becomes a self-terminating construct that causes an immediate halt of processing of the current expansion.

Example

```
&if &(&*=0) &then
  &error 2,No arguments, call ignored.&; &return
&fi
```
terminates the processing of an expansion if no arguments are given.

WHITE SPACE CONTROL

White space is any of the ASCII motion characters; HT, SP, NL, VT, and FF. These characters are normally discarded when they appear as shown below; however, they may be preserved by use of the null separator tokens discussed earlier.

1. After the expansion tokens:

&+	&error	&od!&then
&;	&fi	&quote!&unquote
&do	&if	&scan!&usage
&else	&length	&substr!&while
&elseif		

2. After "(" and "," in an expansion call argument list (at level 1, that is, outside all nesting due the parenthesis usage).
3. After the ")" in the expansion of "&(expr)".
4. After "=" in &let, &loc, &int, and &ext.

EXPANSION TOKENS

As mentioned briefly above, expansion constructs fall into four different classes as determined by their termination conditions. This section lists the tokens that form

constructs in each of the four classes and then gives a sorted list of all tokens for quick reference.

Self-terminating Constructs

&&&	literal &
&*	number of arguments given
&+	begin white space skipping
&.	end white space skipping
&;	general terminator token
&*n*	argument reference (constant)
&*nn*	argument reference (constant)
&return	expansion return
&*name*	variable reference

Matching Character Terminator Constructs

&" *body* &"	protected string
&[*body*]	active function call
&*expansion-name*(*body*)	expansion call
&*name*{ *body* }	array reference
&{ *body* }	parameter reference (index or list)

General Terminator Token Constructs

&comment string &;	comment
&empty *name* &;	array emptying
&error *body* &;	error message generator
&length *body* &;	string measurement
&let *body* &;	variable value assignment
&loc *body* &;	local variable declaration
&int *body* &;	internal (static) variable declaration
&ext *body* &;	external (static) variable declaration
&quote *body* &;	quote-mark duplication
&scan *body* &;	construct rescanning
&substr *body* &;	character substrings
&unquote *body* &;	quote-mark reduction
&usage *body* &;	expansion usage reporting
&while *body* &;	do group control clause

Specific Terminator Token Constructs

&expand *body* &expend<NL>	static expansion definition
&define *body* &dend<NL>	dynamic expansion definition
&do *body* &od	limited or repetitive execution group
&if *body* &fi	conditional execution group
&else *body* &fi	if group control clause
&elseif *body* XXX	if group control clause (XXX may be &elseif, &else, or &fi)
&then *body* XXX	if group control clause (XXX may be &elseif, &else, or &fi)

Sorted Token List

The following is a sorted list of all tokens without regard to class or usage.

&"	protected string
&&	literal &
&*	number of arguments given
&+	begin white space skipping
&.	end white space skipping
&;	general terminator token
&[ ]	active function call
&{ }	parameter reference (index or list)
&comment	comment
&define	dynamic expansion definition
&dend	dynamic expansion definition terminator
&do	limited or repetitive execution group
&else	if group control clause
&elseif	if group control clause
&empty	array emptying
&error	error message generator
&expand	static expansion definition
&expansion-name( )	expansion call
&expend	static expansion definition terminator
&ext	external (static) variable declaration
&fi	if group terminator
&if	conditional execution group
&int	internal (static) variable declaration
&length	string measurement
&let	variable value assignment
&loc	local variable declaration
&n	argument reference (constant)
&nn	argument reference (constant)
&name	variable reference
&name{ }	array reference
&od	do group terminator
&quote	quote-mark duplication
&return	expansion return
&scan	construct rescanning
&substr	character substrings
&then	if group control clause
&unquote	quote-mark reduction
&usage	expansion usage reporting
&while	do group control clause

Reserved Words

It is apparent that all the keywords in the lists above are reserved words and may not be used as variable names. In addition, there are a few others that are reserved for future extensions of the Expander. The complete list is shown below.

```
arg         expand      let         substr
comment     expend      loc         then
define      ext         macro       trace
dend        fi          member      unquote
```

```
do          hbound      mend        usage
else        if          od          while
elseif      int         quote
empty       lbound      return
error       length      scan
```

Annotated Example

The following is an example of the definition, use, and result of an expansion that could aid a PL/I programmer in managing references to the Multics system error_table_. The expansion is called with an error_table_ entry name each time a reference to the entry is wanted. All the different entry names are saved in a list variable and a final call to the expansion without an entry name returns the PL/I declaration list for all the entries used.

```
    &expand et_                          Define the et_ expansion.
&int et_list{50}list&;                   Declare the error_table_ list. Note that repeated
                                         executions of this have no effect due to the
                                         identical attribute feature of variable declarations.
&if &(&*=0)                              If no argument is given, generate the error_table_
&then                                    declaration list with an array reference to et_list.
dcl  error_table_$&et_{, fixed bin(35)ext static;
dcl  error_table_$} fixed bin(35)ext static;
&else                                    However, if there is an argument, add it to et_list
&let et_list=&l&;                        if it is not already there.
error_table_$&l                          Return the error_table_ reference string for use in
                                         the PL/I program.
&fi&expend                               End if group and expansion definition.
```

Next, assume an expansion to generate PL/I source code that contains the following fragments.

```
if (code = &et_(badarg))
then code = &et_(notfound);
  ...
code = &et_(badarg);
  ...
&et_()
end;
```

Finally, when the above fragments are expanded, the following PL/I code results.

```
if (code = error_table_$badarg)
then code = error_table_$notfound;
   ...
code = error_table_$badarg;
   ...
dcl  error_table_$badarg fixed bin(35)ext static;
dcl  error_table_$notfound fixed bin(35)ext static;
end;
```

DEVICE WRITER

The *device_writer_* is the object segment that contains the procedure to convert the coded output page image created by the Formatter into the character stream needed by the output device. It operates as an external subroutine of the Formatter and, as such, is required to conform to certain conventions and restrictions. To ensure this conformance, a "skeleton" procedure is provided by a static source expansion named comp_dev_writer.xdw (see "Device Writer Source Expander" above).

The expansion provides a uniform coding style, defines all required entrypoints and their interfaces, makes certain error checks, includes the structure declarations for all the Formatter internal data bases needed, defines a number of external expansion variables (some with default initial values) that the user can change to affect the action of the Expander in generating output, and, most importantly, provides empty code fragments for all image-to-character-stream conversions that require explicit knowledge of the output device.

The code fragments are all defined as external expansion variables with null initial values. In order to activate any fragment, the user need only reassign the variable value with some PL/I code sequence. The expansion value assignment statements are written in a file named *device_writer_.pl1.xdw*. The final statement in the file must be an Expander call to comp_dev_writer.xdw.

The result of expanding *device_writer_.pl1.xdw* is a PL/I source segment named *device_writer_.pl1*. There are comments throughout that indicate where built-in pieces of code have been selected by the value of &devclass. Comments also introduce user-supplied code fragments. Users must acquaint themselves with the general structure of the writer in order to write compatible code fragments.

Note: This description is a first attempt to document a complex and sophisticated software development tool. It is difficult to determine, *a priori*, the level of detail that should be included. Therefore, the interested reader must study the released files for the Honeywell-supported devices to gain a full understanding of what is required in the creation of a device_writer_ module.

Variables and Code Fragments

The following describes the external expansion variables whose values may be set by the user. The descriptions are shown as expansion constructs with a requirement/default comment and a descriptive paragraph. The order of presentation is alphabetical; however, the fragments may appear in any order in *device_writer_.pl1.xdw*.

&ext art_proc= *PL/I-code*&; optional; default = null
any coding needed to support advanced graphic features (beyond simple plotting) in the device. This feature is not yet used by the Formatter and the variable name "art_proc" is considered as reserved for a future extension.

&ext dcls= *PL/I-code*&; optional; default = null
PL/I declarations for all variables needed by the code fragments following that are not already declared by comp_dev_writer.xdw. The PL/I compiler reports any redeclarations as errors.

Note: Understanding of the declarations and use of the expansion variables should be a primary goal in the study of the released Honeywell-supported device modules.

&ext devclass= *class*&; required; no default
the device class for the device. The value given here must be the same as that given for DevClass in *device.compdv*.

&ext device= *device*&; required; no default
the name of the device for which the procedure is being generated. All the various uses of this name (both here and in *device.compdv*) must be consistent.

&ext disp_rtn= *PL/I-code*&; optional; (see text)
the code needed to produce the interpreted display discussed under "Display Mode Interpretations" in the description of the process_compout command (see Section 3). The default is:

```
call comp_util_$display ((dev_chars));

This is the  routine used by the Formatter  to display input lines
in error messages.
```

&ext epilogue= *PL/I-code*&; optional; default = null
any coding needed to write necessary data to the output after the end of the document.

&ext file_init= *PL/I-code*&; optional; default = null
any coding needed for initialization of the writer at the beginning of an input file.

&ext foot_proc= *PL/I-code*&; optional; default = null
any coding necessary to convert footnote references into the form to be used on this device; for example, superior digits. comp_dev_writer.xdw supplies what is used for device classes typewriter and diablo.

&ext image_init= *PL/I-code*&; optional; default = null
any coding needed for initialization at the beginning of a "window image". For example, in vip7801_writer_, each output "page" is made up of windows that fit on the screen.

&ext line_finish= *PL/I-code*&; optional; default = null
any coding needed to complete the preparation of a line for the output stream.

&ext line_init= *PL/I-code*&; optional; default = null
any coding needed to begin processing of an input line image. A line image is a coded structure in the page image and may contain only part of an output line, for example, a title part or the text for one of several table columns.

&ext machines= *types*&; optional; default = terminals
used only to specialize certain descriptive comments. Other possible values are "typesetters" and "lineprinters."

&ext multi_pitch= *n*&; optional; default = 0
n may have only the values "0" and "1". "0" means that the device is a typewriter class device with a fixed pitch setting. "1" means that, even though the device is not a diablo class device, it does support more than one pitch setting.

&ext notes= *PL/I-comments*&; optional; default = null
any PL/I commentary. It is inserted into the PL/I source just ahead of the opening "procedure" statement.

&ext other_procs= *PL/I-code*&; optional; default = null
any additional internal PL/I utility procedures needed by the writer.

&ext page_finish= *PL/I-code&*; optional; default = null
any coding needed to complete (run out) a page on the device.

&ext page_init= *PL/I-code&*; optional; default = null
any coding needed for writer initialization at the beginning of an output page.

&ext plot= *PL/I-code&*; required, no default
the coding needed to do simple horizontal and/or vertical vectors and shifts on the device, regardless of what additional graphic capability the device may have.

&ext process_text= *PL/I-code&*; optional; default = null
any coding that converts the text strings in line images to output device native characters in the required code set and format.

&ext restore= *PL/I-code&*; optional; default = null
PL/I assignment statements that restore any saved user variables (see save following).

&ext save= *PL/I-code&*; optional; default = null
blank line (for example, a line containing nothing but font changes) suppression requires that certain variable values be saved at the beginning of an input line so that they may be restored if the line is truly blank. This code is PL/I assignments that add other user-defined variables to the saved data.

&ext set_font= *PL/I-code&*; optional; default = null
any coding needed to effect a font change in the device.

&ext set_media= *PL/I-code&*; optional; default = null
any coding needed to set the device to the desired font.

&ext set_ps= *PL/I-code&*; optional; default = null
any coding needed to effect a pointsize change in the device. This and set_font may be interdependent or may be completely independent, depending on the device.

&tabx= *PL/I-code&*; optional; default = null
code to support direct (or absolute) horizontal tabulation in the device.

DEVICE TABLE COMPILER

The *Device Table Compiler* is a language translator that translates the plain language description of a device intended for use by the WORDPRO Text Formatter into the required binary table form. The input to the translator consists of various statements in the language described below and contained in an unformatted stream file named *device.compdv*, where *device* is an arbitrarily chosen name for the device to be supported. This input file is referred to below as the device description file. The output is a coded binary table in a segment named *device.comp_dsm* that is accessed directly by the Formatter.

The process_compout command is referenced in the text below. The description of this command may be found in Section 3. Also, in the remainder of this section, the **WORDPRO Text Formatter** is referred to simply as the *Formatter*. Its description may be found in Section 2.

The Device Description Language

The device description file consists of nine parts that must appear in the order shown (unless otherwise noted):

Global Values (distributed)
Symbol Declarations (optional)
Media Character Table
Media Tables
View Tables
Definitions (optional)
Font Tables
Size Tables
Device Tables

GENERAL SYNTAX

Literals

A *quoted-string* means a string delimited by the double quote character ("). If a quote is needed within such a string, it must be doubled. For example:

```
"A quoted string"
"A ""quoted"" string"
```

Comments

A comment may be placed any place in the source where the syntax allows white space to appear (except within a quoted string). A comment is any string beginning with /* and ending with */. For example:

```
/* This is a comment */
/* And this is a
   multiline comment */
```

Names

A *name* means a string of not more than 32 characters beginning with an alphabetic followed by an arbitrary series of alphanumerics and/or underscores. All *names* in a device description file must be globally unique. For some usages, *name* is restricted to less than 32 characters. The restrictions are given in the discussions of the various usages. For example:

```
A
name
here_is_1
```

Fonts

Two different forms of fonts are supported: the "family" font and the "bachelor" font. A "family" is a group of fonts of different styles all of which have the same

typeface such as Century Schoolbook or Helvetica. A "bachelor" is a font that has no such close relatives such as NewsCommercialPi (NCPi) or APL.

Anywhere *font* appears, it is a *name* having the form *family/member* or *bachelor*.

Braces, Ellipses, and Vertical Lines

A term or group of terms may be enclosed in opening and closing braces ({}) and/or followed by an ellipsis (...). The braces mean that the enclosure is optional, the ellipsis means that the preceding term or term group may be repeated as desired, and the vertical line means a choice between (or among) the terms must be made. For example,

integer {, *integer*|= ...}

stands for a comma-separated list of *integers* and equal signs of any length that must start with an integer.

Input

input is a single character given by either of:

ooo
: 3 octal digits

"c"
: any single-quoted character

Range

range is an inclusive ordered set of characters given as *input*:*input*. For example:

"A":"Z"
: the uppercase alphabet

000:007
: the first eight ASCII control characters

Output

output is a blank separated list of elements selected from the following:

ooo
: 3 octal digits

"string"
: any quoted string

XXX
: any declared symbol (See "Symbol Declarations" below)

nn(*output*)
: *nn* repetitions of an *output* string

SELF
: When used in media character token definitions, means the graphic being defined. This is a reserved word; it *may not* be used as a *name*.

For example:

```
dcl: square, "Z" 010 "N";  /* a black square */
```

Media Characters

A *mediachar* is an internal token referring to some graphic symbol or control action available in the device. It may have the form of an *input*, or an eight-character *name*. Note that A and "A" are *not* the same *mediachar*.

Media Character List

A *mediacharlist* is a blank-separated list of *mediachars* given as any of the following:

ooo
: the octal value of a *mediachar* that is defined as an *Input*.

"xxx"
: a string of *mediachars*, each of which is defined as an *Input*.

yyy
: a *mediachar* that is defined as a *name*.

nn(Mediacharlist)
: *nn* repetitions of a *mediacharlist*

Media

A media table is a named aggregate of media character tokens wherein each token is assigned a character width value given in "strokes". A "stroke" is an arbitrary, dimensionless number of parts into which an EM unit is divided for the purpose of defining character widths for a device and must be at least as large as the resolution of the device. The term comes from antiquity and refers to the number of strokes required with a given size pen nib to get a line of some desired thickness.

Switch

switch is the setting of a binary switch bit. It may have two values; on or off.

Numbers

Numbers are given as one of the following forms.

integer
: a dimensionless decimal integer, for example:

```
2
253
-3
```

units
: a decimal number given in the current space measurement units (see Units under "Global Values" below), for example:

9
-1.5
97.25

SYNTAX OF THE SECTIONS

Global Values

The Global Values section is not a formally delimited section, but consists of any number of the following statements distributed randomly throughout the file. The statements define values that apply to all sections following their appearance. All have local counterparts to specify different values for a particular table.

Data dependencies affect the order in which certain statements may appear. Any such restrictions are given in the descriptions of the affected statements.

The statements all have default values that describe the default (ASCII or printer) device. Unless otherwise noted in the text, the default values are those shown in the individual examples.

Artproc: *name*{$*name*};
the entryname and optional entrypoint of the procedure that supports special artwork features for the device. This entry is normally needed only for devices having graphic features beyond the scope of plotting and simple typographic rules. The default entryname is derived from the name of the device (see Outproc below).

Note: This interface is not yet active due to lack of a specific application. The calling sequence is not yet defined. Its projected use is for the processing of half-tone raster files and generalized graphics files.

```
Artproc: ascii_writer_$artproc;
```

Attach: *quoted-string*;
the attach description to use for the output switch when formatted output is *not* being written to a file. If not given, no online output is possible for the device.

```
Attach: "syn user_output";
```

Cleanup: *mediacharlist*;
the control string that must be sent to the device to restore its normal mode of operation when interrupted in the middle of output. This string is required for plotting terminals to take them out of PLOT mode when interrupted.

```
Cleanup: "";        /* no cleanup needed */
```

Comment: *quoted-string*;
a string that is emitted to a compout file as a part of its header. It is used by the process_compout command when transcribing the file onto the output medium.

```
Comment: "";        /* null comment */
```

DefaultMargs: *units*, *units*, *units*, *units*;
the default values for the top, header, footer, and bottom page margins, respectively. This feature allows for devices (such as Braille embossers) that demand page margins other than those normally assumed for a printed document.

```
DefaultMargs; 48,24,24,48; /* 4,2,2,4 lines */
```

DevClass: *quoted-string*;
the class of the device. This string is placed in the output file header for use by the process_compout command and is used to set the DeviceClass built-in of the Formatter.

```
DevClass: "typewriter";
```

DevName: *quoted-string*;
the generic name of the machine within DevClass for which the Device Tables in this file provide support, e.g., V-I-P, Dymo, APS within "photocomp" or dtc300s, hyterm within "diablo". Note that within a generic device, such as dtc300s, there may be different specific devices (see "Device Table" selection below) for minor differences such as running in 12-pitch rather than 10-pitch. This string is also used to set the DeviceName built-in of the Formatter.

```
DevName: "ascii";
```

Endpage: *input*
the font character to select the page eject sequence for the device. A value of 000 means that there is no eject sequence.

```
Endpage: 000;       /* ascii */
Endpage: 014;       /* printer */
```

Footproc: {*name*{$*name*}} {, *font*};
the optional entryname and entrypoint of the procedure to process footnote references and the optional font for them. The default entryname is derived from the name of the device (see "Outproc" below) and the default font is the default font for the device. (See "The Device Writer" below for the description of the calling sequence for this interface.)

```
Footproc: ascii_writer_$footproc, ascii;
```

FootrefSeparator: *input* ;
the Multics character to separate multiple footnote references at the same place in the text.

```
FootrefSeparator: ""; /* parens are sufficient */
```

Interleave: *switch*;
the setting of the line sorting switch for the Formatter. If the switch is on, the output in the page image structure is sorted by the Formatter so as to appear in strictly increasing page depth order because the device does not support reverse leading to return to the top of the page for multi-column output. If the switch is off, the output lines appear in the page image by page depth within the columns, each column being a sub-array in the structure. The default value for the switch is off; it must be set on for device with DevClass values of "typewriter", "diablo", or "printer."

```
Interleave: on;     /* sort output */
```

Letterspace: *integer*;
the maximum amount of interletter space allowed, given in strokes.

```
Letterspace: 0;     /* not supported */
```

MaxFiles: *integer* | unlimited;
the maximum number of files to be written on a reel of magnetic tape. The process_compout command calls for an additional reel when this value is reached while processing "compout" files. The number of input tape reel files for some typesetters is limited by the software in their front-end computers. If this statement is omitted or is given with the keyword "unlimited", then the tape may contain any number of files.

```
MaxFiles: unlimited;
```

MaxPages: *integer* | unlimited;
the maximum number of pages to be contained in an output file for the device. The process_compout command produces output files containing no more than this number of pages. Input files for some devices are limited by such factors as size of paper tape input reel, capacity of tape cassette or film magazine, etc. If this statement is omitted or is given with the keyword "unlimited", then the file may contain any number of pages.

```
MaxPages: unlimited;
```

MaxPageLength: *units* | unlimited;
the maximum length of a page. If this statement is omitted or is given with the keyword "unlimited", then the page may be as long as the user cares to make it.

```
MaxPageLength: unlimited;
```

MaxPageWidth: *units*;
the maximum width of an output page.

```
MaxPageWidth: 979.2; /* 136 columns */
```

MinBotMarg: *units*;
the minimum page bottom margin for the device.

```
MinBotMarg: 0;      /* ascii */
MinBotMarg+ 36;     /*printer */
```

MinLead: *units*;
the minimum amount of "lead" (vertical spacing) available in the device.

```
MinLead: 12;        /* 1 line */
```

MinSpace: *units*;
the minimum value of horizontal space available in the device.

```
MinSpace: 7.2;      /* 1 column */
```

MinTopMarg: *units*;
the minimum page top margin for the device.

```
MinTopMarg: 0;      /* ascii */
MinTopMarg: 36;     /*printer */
```

Outproc: *name*{$*name*};
the entryname and optional entrypoint of the procedure that converts the coded page image structure constructed by the Formatter into a character stream acceptable to the device. This is the procedure that translates internal signal bytes into device control codes. The default entryname for the device described in *device*.compdv is *device*_writer_. (See "Device Writer" earlier in this section for the description of the calling sequence for this interface.)

```
Outproc: ascii_writer_;        /* ascii device */
```

Sizes: *name*;
the name of the default Size Table. *name* must have already been defined as the name of a Size Table section. If this statement is not given, the name of the first Size Table defined is used.

```
Sizes: onesize;
```

Stream: *switch*;
the setting of the compout file type switch for the Formatter. If the switch is set on, the compout file written when the -output_file control arg of the Formatter is given is an ASCII stream file suitable for processing with the print and dprint commands. If the switch is set off, the compout file is a sequential file containing coded binary device information that must be processed with process_compout command. Normally, this switch is set on only for the ASCII and printer devices, but it may be used for any other device that has only those features commonly found in ASCII terminals or it could be treated (by Multics) as a line printer. The default value for the switch is off; it must be set on for the ASCII device.

```
Stream: on;
```

Strokes: *integer*;
the number of strokes to be used for width values in Media Tables.

TapeRec: *integer*|unlimited;
the length in characters of records to be used when writing to a tape. If this statement is omitted or is given with the keyword "unlimited", then the tape records may contain any number of characters.

```
TapeRec: unlimited;
```

Units: *keyword*;
the physical units in which space values are given. Space values are given as normal decimal numbers, e.g., 2, 14.7, and 0.025. The valid keywords are:

pi	pica (10-pitch) monospace characters and lines
el	elite (12-pitch) monospace characters and lines
in	inches
mm	millimeters
pc	typographic picas (6 picas = 1 inch)
pt	typographic points (72 points = 1 inch)
pp	picas and points as a decimal number

```
Units: pt;          /* default is points */
```

Wordspace: *min*, *avg*, *max*, *mediachar*;
the default range of allowable interword space for devices described in the file.

min, *avg*, *max*
specify the minimum, average, and maximum values, respectively, given in strokes. They must obey the relation:

0 <= *min* <= *avg* <= *max*

and are defined as:

min
the least amount of interword space

avg
the average amount of interword space. This is the amount used for all wordspace characters in unjustified lines.

Note: This value must be the same as the width given for *mediachar*. See "Media Tables" below.

max
the maximum amount of interword space allowed before hyphenation or letterspacing is attempted. Note that justified lines may contain more than *max* space, but only in case hyphenation and letterspacing fail or are not allowed.

mediachar
the character string to be emitted for wordspace insertion.

For example:

```
Wordspace: 1,1,2,SP; /* ascii, strokes = 1 */
Wordspace: 3,6,9,SP; /* dtc300s, strokes = 6 */
```

Symbol Declarations

Symbols that represent output character strings may be defined for convenience in constructing media characters. All such symbols must be defined before their use.

dcl: *name*, *output*;

name
the name of the symbol being defined and is restricted to a maximum length of 8 characters.

output
the character string to replace a reference to the symbol.

For example:

```
dcl:    BSP,    010;
dcl:    HT,     011;
dcl:    lf,     012;
```

Media Character Table

The Media Character Table section contains the symbols and output values for all media character tokens to be used in the Media Tables following. It consists of the following media character statement.

MediaChars: *mediachar output* {, *mediachar output* ...};

dcl: BSP,:010;

mediachar
the media character token(s) being defined given as a *name*, an *input*, or a *range*.

output
the output character string to replace a reference to the token.

For example:

```
MediaChars:
    SP  " ",
    010  SELF,    014        SELF,    033   SELF,    016   SELF,
    017  SELF,    "a":"f"      SELF,    177   SELF,    USR   BSP "_",
    NIL "";
```

Media Tables

The Media Table section contains the character width values for all *mediachars* in all the physical media used by the fonts defined for the device. It consists of any number of Media: statements, each having any number of width value statements. The syntax is devised in such a way that the table may actually be formatted as a table in the input, that is, all width values in the first column are for the first *medianame*, the second column for the second *medianame*, etc.

Media: *medianame* {, *medianame* ...};
mediachar {*integer*} {, {*integer* |=} ...};
...

medianame
the name(s) of the media being defined.

mediachar
as for MediaChars: above.

integer
the width of the character given in strokes. If this value is omitted, the character is undefined in the associated media.

If "=" is given in the second or subsequent column, the value in the preceding column is repeated. It is an error to give "=" in the first column.

For example (from a Mergenthaler V-I-P description):

```
/* (A-534 is the Mergenthaler number for Universal Greek with Math.) */
/*       A-2160,     A-534,     A-187,     A-6614,     A-145,     A-108,     A-409  */
Strokes: 18;
Media:   mNCPi,     mUGM,     mCSR,     mCSRx,     mCSI,     mCSBR,     mCSBI;
A01,     09,         14,       =,        =,         =,        15,        13;
A02,     18,         15,       10,       =,         =,        =,         09;
A03,     15,         =,        10,       =,         =,        =,         09;
A04,     10,         06,       10,       =,         =,        =,         09;
A05,     06,         11,       18,       =,         17,       18,        17;
```

View Tables

A View is a switch-like "variable" (in the sense of a Multics I/O switch) through which an attachment is made to a Media Table (see "viewselect:" under Unique Local device Values below). A View may attach to only one Media Table, but a Media Table may have any number of Views attached to it. An example is the superior and inferior fonts in a Mergenthaler V-I-P typesetter that are identical in all respects except that they are on different film plaques due to their different baseline offsets. The View Table section consists of any number of the following View: statement.

View: *viewname medianame* {, *viewname medianame* ...};

viewname
the name of the View being defined.

medianame
the name of the Media Table to which an attachment will be made.

For example:

View: PICA mASC10, ELITE mASC12, APL mASC10;

Definitions

A Definition is a named aggregate of *MediaChars* that may be used in several different fonts. The Definitions section consists of any number of the following Def: statement, each followed by any number of graphic definitions.

Def: *defname*;
graphic {*viewname*} *definition*;

defname
the name of the *mediachar* aggregate being defined.

graphic
may be chosen from:

input {, *input* ...}

keyword {, *keyword* ...}
any of:

EM	EM space
EN	EN space
thin	thin space
EM-	EM dash
EN-	EN dash
hyphen	hyphen
EM_	EM-aligned dash
EN_	EN-aligned dash
PS	punctuation space
``	opening double-quote
''	closing double-quote
^0	superior digits
^1	
^2	
^3	
^4	
^5	
^6	
^7	
^8	
^9	

These keywords may be thought of as "built-in" symbols that must be assigned values if they are to be included in a font. Note that "hyphen" must be assigned a value in order that the hyphenation mechanism in the Formatter may work.

art *artname* {, art *artname* ...}

a keyword and the conventional name of the artwork construct or element selected from one of the following groups.

This group contains graphics that are complete in themselves (the so-called "one-highs").

[	opening bracket	]	closing bracket
{	opening brace	}	closing brace
(	opening parenthesis	)	closing parenthesis
|	vertical bar	‖	concatenate
•	bullet	x	multiply
m	change bar	d	delete star
/	divide	\	left slant
c	copyright	t	trademark
^	up arrowhead	v	down arrowhead
←	left arrowhead	→	right arrowhead

This group contains graphics that are parts of larger artwork constructs, e.g., rules, boxes, diamonds, and lozenges.

D^	diamond top	Dv	diamond bottom
D<	diamond left	D>	diamond right
Clf	left half-circle	Crt	right half-circle
-rul	horizontal rule	|rul	vertical rule
/rul	right slant rule	\rul	left slant rule

This group contains the parts for the multiline math symbols. The graphics for any symbol form a consistent set; if a math symbol is to be defined, all the parts must be given.

"["	"]"	"{"	"}"	"("	")"	"|"	"||"	symbol
[tp	]tp	{tp	}tp	lptp	rptp	|tp	||tp	tops
[ht	]ht	{ht	}ht	lpht	rpht	|ht	||ht	half tops
[md	]md	{md	}md	lpmd	rpmd	|md	||md	middles
[hb	]hb	{hb	}hb	lphb	rphb	|hb	||hb	half bottoms
[bt	]bt	{bt	}bt	lpbt	rpbt	|bt	||bt	bottoms
[fl	]fl	{fl	}fl	lpfl	rpfl	|fl	||fl	fillers

Note: Because the left and right parentheses are used as part of the syntax of the device description language they may not be used in forming tokens; hence the need to use the "lp" and "rp" constructs for their *artname* parts.

viewname

the name of the View: that attaches the Media Table holding the character widths to be used in calculating the width of the *definition*. The default View: is the View: attaching the Media Table for each font that refers to this Def:.

definition

replacement for *graphic* chosen from:

mediacharlist

the assigned width of the *definition* is calculated from the width values of the elements of *mediacharlist*.

mediacharlist=integer

the calculated width the *mediacharlist* (as above) is compared to *integer*. If they are the same, the value is assigned as the width of the *definition*; if not, an error message is generated. This form is useful in ensuring that plot strings are the correct width for the font in which they are to be used.

(mediacharlist)=integer

integer is assigned as the width of the *definition* without regard to the calculated width. This form is useful in forcing the width of plot strings when the calculated width is known to be wrong.

For example:

```
Def: etc;                /* miscellaneous chars */
016:017  (SELF)=0;       /* red/black ribbon shifts */
221      3(".");         /* ellipsis */
177      NIL;            /* ASCII PAD */
```

Font Table

A font table contains the width and output string for each character contained in a font. In this context, a "character" is a 9-bit byte placed in the output page image by the Formatter. This byte may be a normal ASCII graphic or a coded signal for some other output sequence.

A device description file may specify up to 100 fonts; each Font Table beginning with a Font statement and ending with the beginning of any Size Table, Device Table, or other Font Table.

A Font Table section consists of any number of Font statements, each followed by an optional local wordspace: statement and any number of ref: statements, and graphic definitions.

Font: *fontname viewname*;
{wordspace: *min, avg, max, mediachar;*}
{ref: *defname;*}
{*graphic* {*viewname*} *definition;*}

fontname
the name of the font table being defined.

viewname
the name of the default View attaching the Media Table for any graphic definitions given in this font.

wordspace: *min*, *avg*, *max*, *output*;
as for the global Wordspace described earlier but applying only to this font.

ref: *defname*;
a reference to some existing Def:.

graphic {*viewname*} *definition*;
as for "Definitions" above.

Size Table

A Size Table is a list of allowable pointsize values that may be used in conjunction with any number of Fonts in any number of Device Tables. That is to say, a Size Table may be referenced any number of times and may be used with one Font in some Device Table and a different Font in some other Device Table. A device description must contain at least one Size Table.

A Size Table section consists of exactly one Size statement of the form:

Size: *name*, *units*{, *units*}...;

name
is the internal reference name of the pointsize list being defined.

units
is a value to be entered into the list. At least one *units* value must be given.

For example:

```
Size: pitch10, 7.2;
```

Device Table

A Device Table describes a specific device and provides the data needed by the Formatter to prepare output for that device. The data in the table is gathered from default values, Global Values, Font Table references, Size Table references, and Local

Device Values. There can be any number of Device Tables in a device description file, either describing different machines that are similar enough to share many attributes, or different configurations of the same machine.

Font Tables and Size Tables may be freely shared among Device Tables. However, if a font "borrows" from some other font, then both the "loaner" and "borrower" Font Tables must be included in the Device Table. For example the Mergenthaler V-I-P font "ascii" is based on the Clarinda font but it borrows a few characters from NewsCommercialPi. Hence, if the "ascii" font is to be included in a Device Table for the V-I-P, then "NCPi" must also be included. If it is not, then the Formatter reports errors if the borrowed characters are used.

In some machines, like the Mergenthaler V-I-P, that have limited font capacity, many Device Tables are likely to be needed to describe the many different configurations. Other machines, such as the Autologic APS-5, that have large font storage capacity usually need only one Device Table.

A Device Table section begins with a Device statement and ends with the beginning of another Device Table or the end of the device description file. Global Values may also appear within a Device Table section.

Device: *name*{,*alias*} {like *device*};

name
is the name to be attached to the Device Table. It is the name by which the device is known to the Formatter and is given as a parameter with the -device control argument (of the Formatter). *name*.comp_dsm. is added to the output segment if it is not the primary entryname.

alias
is an additional name by which the device may be know, e.g., a short name. This *alias* is handled identically to the primary *name*.

device
is the name of some previously defined Device Table that is to be used as a model for this device. If this Device statement is followed immediately by another Device statement or is the last statement in the device description file, the other Device Table is referenced directly by internal pointers. If any changes to the model are made (with Global or Local Device Value statements), the other Device Table is copied as initial values for a new Device Table.

GLOBAL/LOCAL DEVICE VALUES

All the device-related items discussed in Global Values above have local counterparts. Local Device Values apply only to the Device Table in which they appear; any given are discarded when the Device Table is completed. The syntax of the Local Device Values is identical to the corresponding Global Values except that the keyword tokens are spelled with all lowercase letters. These Local Device Values are all set to their current Global or default values when a Device Table is initialized (unless *device* is used).

The Global/Local Device Values statements are listed below.

artproc: *name*{$*name*}
attach: *quoted-string*;
cleanup: *mediacharlist*;

comment: *quoted-string*;
defaultmargs: *units*, *units*, *units*, *units*;
devclass: *quoted-string*;
devname: *quoted-string*;
endpage: *input*;
footproc: {*name*{$*name*}} {, *family*|*member*|*bachelor*};
footrefseparator: *input*;
interleave: *switch*;
letterspace: *integer*;
maxfiles: *integer*|unlimited;
maxpages: *integer*|unlimited;
maxpagelength: *units*|unlimited;
maxpagewidth: *units*;
minbotmarg: *units*;
minlead: *units*;
minspace: *units*;
mintopmarg: *units*;
outproc: *name*{$*name*};
sizes: *name*;
stream: *switch*;
taperec: *integer*|unlimited;
units: *keyword*;

UNIQUE LOCAL DEVICE VALUES

The following Local Device Values have no Global Device Value counterparts.

init: *initfont initsize*;

initfont
the initial font for the device given either as *family*|*member* or *bachelor*.

initsize
the initial pointsize for the device. It must be a value in the initial Size Table for the device (see "sizes:" above).

For example:

```
init: CenturySchoolbook/medium 10;
init: ascii 7.2;
```

family: *name*{, *name*};
the external name and optional aliases of a group of fonts of different styles all of which have the same typeface.

For example:

```
family: CenturySchoolbook, CS;
```

member: */name*{, */name*, ...} *fontref*;
a member font in the preceding family.

/name
the external name and optional aliases of the member.

fontref
the name of the Font Table containing widths and replacements for characters in the font.

For example:

```
member: /medium, /m, /roman, /r CSmed;
member: /bold, /b CSbold;
```

bachelor: *name*{, *name*, ...} *fontref*;
"bachelor" fonts that have no family/member structure.

name
as for family: above.

fontref
as for member: above.

For example:

```
use: GrkMath, GM UGM;
use: APL, apl APL;
```

viewselect: *view mediacharlist*{, *view mediacharlist*, ...};
the attachment descriptions for all the fonts used in the device.

view
the name of the View through which the attachment is to be made.

mediacharlist
the character string giving the information needed by the device writer to construct the font change control that is sent to the device to cause it to select and use the desired Media. The content of this information depends on the device and the design of the device writer procedure.

For example:

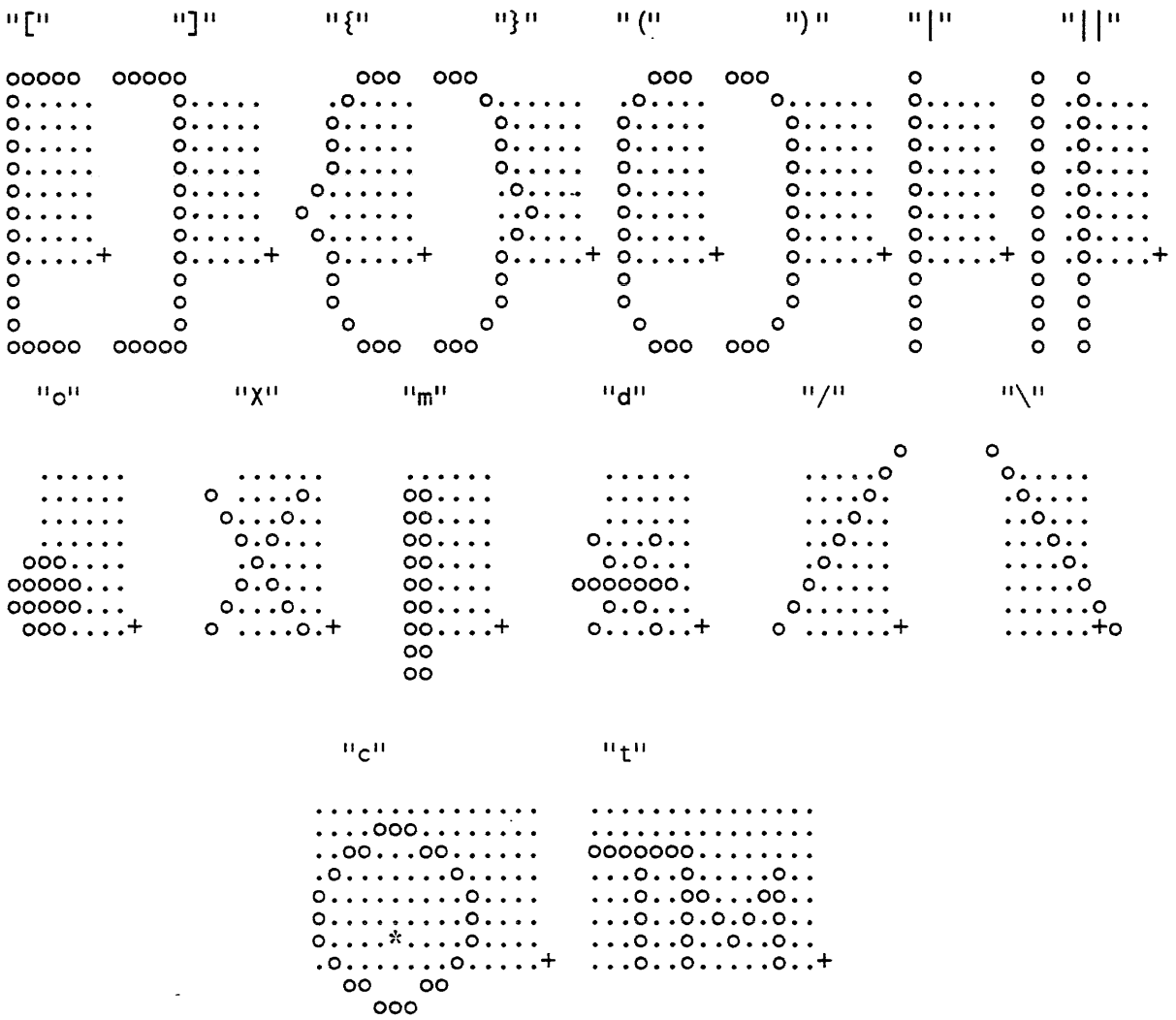
```
viewselect: vASCII Pwheel pitch10 "6";
viewselect: vAPL Awheel pitch10 "6";
```

ARTWORK PART DESCRIPTIONS

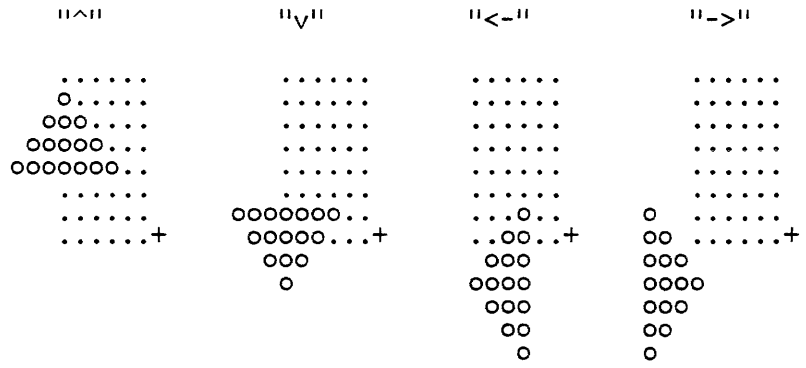
The artwork parts for incremental plotting terminals are plot strings made up of various motion characters and the dot (.) character. When strings for such a terminal are constructed, they should conform to the following specifications.

In these diagrams, the grid of dots represents the 48 possible dot positions in a print position. The starting position of the pattern is the lower left corner of the grid, that being the position at which a single "." would print in normal typing mode. If there is a "+" in a diagram, then its position is the final position of the print head; if not, the print head returns to the starting position. An "o" represents a grid position where a "." must be placed. Note that the print column for vertical lines is the left edge of the grid.

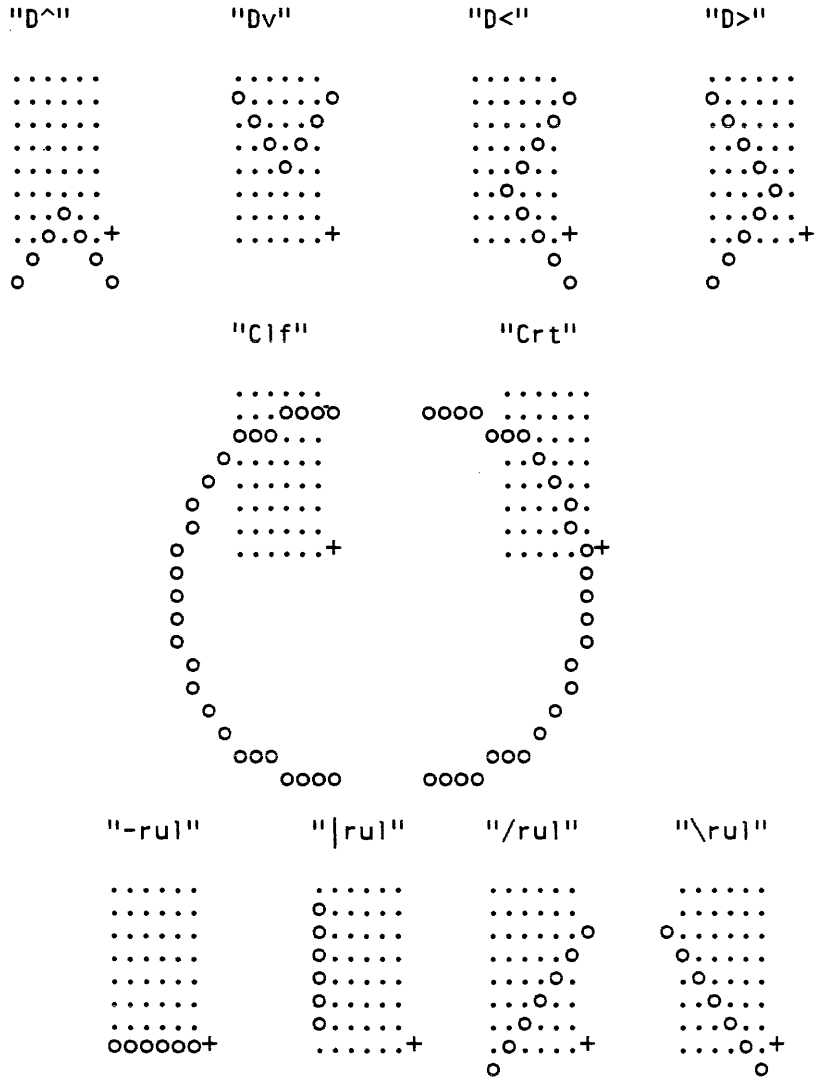
"One-High" Math Symbols



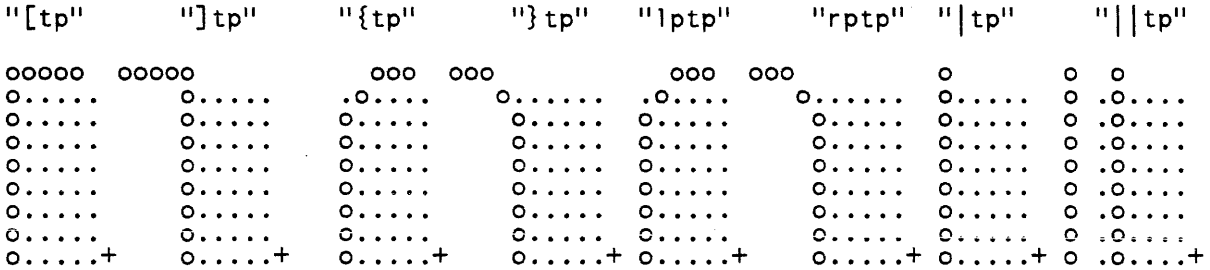
NOTE: The "*" in the "c" diagram represents the position of the "c" for "copyright". It may be changed to any other letter of the users choice.



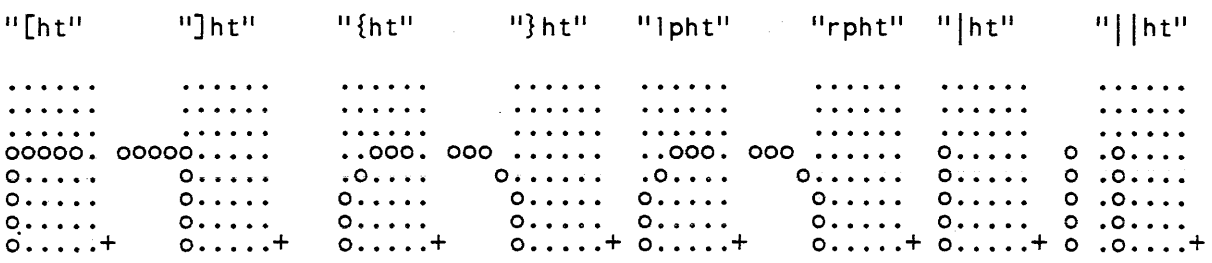
Large Artwork Elements



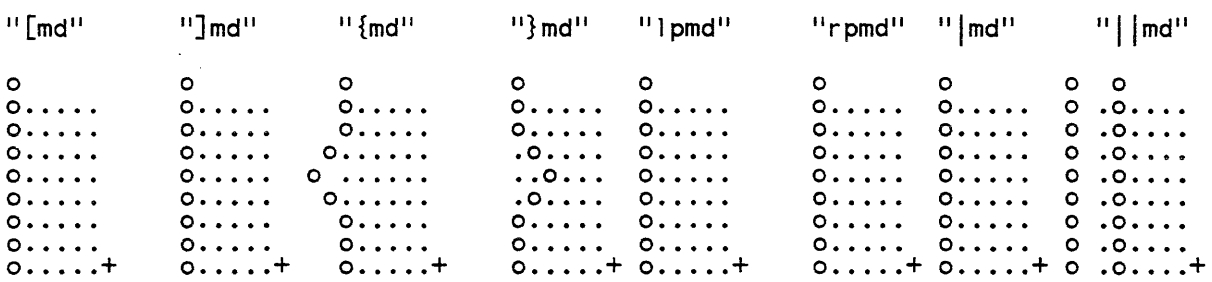
Math Symbol Top Parts



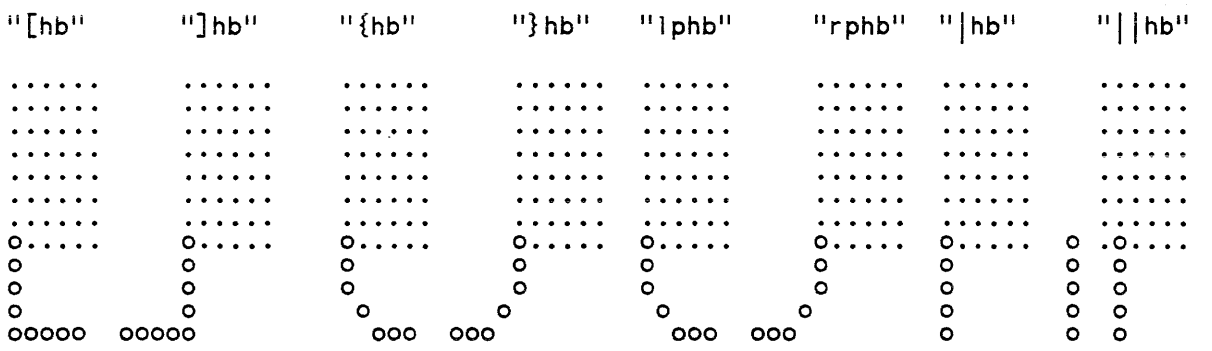
Math Symbol Half-top Parts



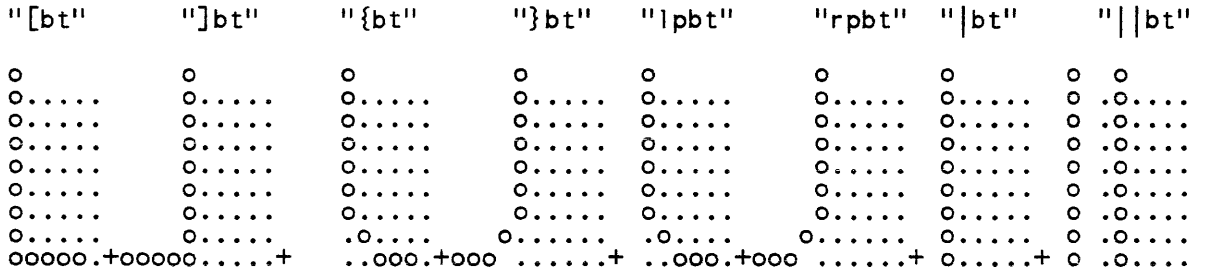
Math Symbol Middle Parts



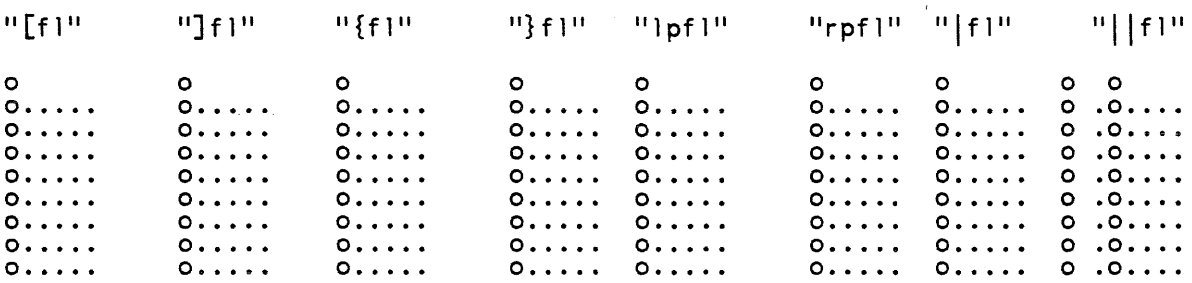
Math Symbol Half-bottom Parts



Math Symbol Bottom Parts



Math Symbol Filler Parts



APPENDIX D

GLOSSARY

The following list is WORDPRO-specific and does not duplicate common Multics glossary terms in other documentation (see *Multics Reference Manual*.)

artwork
: In compose, overstruck character patterns displayed as various symbols and line art features (e.g., diagrams, flow charts, logos).

built-in symbol
: A variable (number, on, off) which is built into one of the programs being used; not affected by user unless its value is specifically changed by a related control or control argument.

block (text block)
: In compose, the basic premise for text processing; all text material is made up of blocks of text on which compose processing takes place, using the surrounding controls as the basis for formatting.

canonicalization
: The conversion of a terminal input line into a standard (canonical) form. This is done so that lines that appear the same on the printed page, but that may have been typed differently (i.e., characters overstruck in a different order), appear the same to the system (see *Multics Reference Manual*).

carriage return
: Movement of the typing mechanism to the first column of the next line. On Multics, this action is the result of the ASCII linefeed character. The terminal type determines which key(s) the user presses to perform the equivalent action (e.g., RETURN, LF, or NL).

command
: A program designed to be called by typing its name at a terminal. Most commands are system-maintained, but any user program that takes only character-string input arguments can be used as a command (see *Multics Reference Manual*).

command level
: The process state in which lines input from a user's terminal are interpreted by the system as a command (i.e., the line is sent to the command processor). A user is at command level or when he or she logs in, or when a command completes, encounters an error, or is stopped by issuing the quit signal. Command level is normally indicated by a ready message (see *Multics Reference Manual*).

command processor
: The program that interprets the lines input at command level and calls the appropriate programs, after processing parentheses and active functions (see *Multics Reference Manual*).

compin
: A compose input file, made up of text and compose text controls (see Section 3).

compose
: A command that, given input (text and controls), formats it according to the conditions set by the user (controls and control arguments) and produces the desired output (see Section 3).

compout
: A compose output file, created (compiled) from the compin file, usually consisting of formatted text, diagrams, etc., the format having been defined by the user through the use of the controls (see Section 3).

control (control line)
: In compose or format_document, a line that defines an action to be taken while formatting the output. It always begins with a period in the first character position, followed by several alphabetic characters, and sometimes another string that further describes the action to be taken.

control argument
: An argument to a command that specifies what the user wants done, or what information the user is interested in. System control arguments begin with a hyphen, such as -all, -long, or -hold. The meaning of each control argument accepted by a specific command is given as part of the description of the command. Many control arguments have standard abbreviations such as -lg for -long. A list of the abbreviations of the most frequently used control arguments is found in Appendix A of *SDN - Standards*. (System commands are described in *Multics Commands*.)

crash
: An unplanned termination of system availability caused by problems in hardware and/or software.

delimiter
: (1) In Speedtype, a character used to delimit between text tokens (symbols including prefix and suffix characters).

  (2) In List Processing, a user-specified character used to delimit between each record in the list, between each field of each record, and between comments and data.

dictionary
: In WORDPRO, an online list of words (supplied by the system or created by the user) with which any file can be compared, to find misspelled words, unwanted words, etc.

edm
: A text editor that allows users to create input segments (text), then edit these segments making substitutions and changes.

emacs
: A text editor that allows users to create input segments (text), then edit these segments making substitutions and changes.

equation
: In compose, also called <title>, a three-part title of the form: |part1|part2|part3|.

escape character
: In Speedtype, a signal that the text token immediately following is to be processed specially, i.e., *not* expanded (see token).

expand
In Speedtype, the process by which symbols are lengthened from their shorthand-type abbreviated form to the correctly-spelled words and phrases.

fdocin
A format_document input file made up of any types of text and format_document text controls (see format_document).

fdocout
A format_document output file usually consisting of formatted text that has been defined by the user through use of controls (see control above).

file
A term that stands for segment and/or multisegment file.

format_document
A command that, given input (text and controls), formats the segment according to the conditions set by the user (controls and command control arguments) producing the desired output.

help files
See info segments.

indent
In compose and format_document, a control that indents all following text until another control indicates undent or indent zero (reset the indent to zero).

info segments
The segments whose contents are printed by invoking the help command. These segments, sometimes called help files, give information about the system. The system info segments are kept in the directory >documentation>info_segments (>doc>info). The info segments that are peculiar to an installation are kept in >doc>iml_info_segments. (see the help command in *Multics Commands*).

line art
Graphic constructs (pictures, tables, etc). that can be created online.

List Processing
A group of related commands that enable the user to create lists, define formats to be used in the manipulation of these lists, and produce personalized form letters, billing statements, reminders, etc.

lister
In List Processing, an unprintable file containing ASCII and binary information; a compiled version of the list that can be processed by the various list processing commands.

listform
In List Processing, a file that shows (defines) the format of a document to be produced; a combination of a lister file and a listform file, processed together, creates the desired end result.

listin
In List Processing, an ASCII file containing components (records), each of which is a complete entry in the list, that can be entered and updated using any text editor (see record below).

project
An arbitrary set of users grouped together for accounting and access control purposes.

Project_id
The name assigned to a project.

qedx
A text editor that allows users to create input segments (text), then edit these segments making substitutions and changes.

quit request
Several commands that read input from the keyboard and use the typed request "quit" or "q" to indicate that the user is done. This is not the same as issuing the quit signal.

quit signal
A method used to interrupt a running program. The quit condition is raised by pressing the key on a terminal, such as ATTN, BRK, INTERRUPT. This condition normally causes the printing of QUIT followed by establishment of a new command level (see *Multics Reference Manual*).

quote
A character used to delimit strings in commands and source programs. On Multics this is the double-quote (octal 042), not to be confused with the single quote or apostrophe (octal 047).

ready message
A message that is displayed each time a user is at command level. The display of this message may be inhibited, or the user may define a personal ready message. The standard system ready message tells the time of day and the number of CPU seconds, memory units, and page faults since the last ready message plus the current listener level (if greater than 1).

record
In List Processing, a division of a list containing all fields of related information grouped together. as one item in the list (e.g., name, address, city, and state of one person grouped together

recursion
The ability of a procedure to invoke itself.

Speedtype
A tool that enables the user to "type shorthand"; users can specify abbreviations (symbols) for lengthy or frequently used words and/or phrases, enabling faster input, to be later automatically expanded.

star convention
A method used by many commands to specify a group of segments and/or directories using one name (see *Multics Reference Manual*).

subsystem
A collection of programs that provide a special environment for some particular purpose, such as editing, calculation, or data management. It may perform its own command processing, file handling, and accounting. A subsystem is said to be closed if: (1) all necessary operations can be handled within the subsystem and (2) no way exists to use the normal Multics environment from within the subsystem.

suffix
The last component of an entryname (components are separated by a period (.)) that usually specifies the type of segment (e.g., **.pl1** and **.list**). A segment without a suffix is usually an object segment or data segment (see *Multics Reference Manual*).

teco
: A text editor that allows users to create input segments (text), then edit these segments making substitutions and changes.

ted
: A text editor that allows users to create input segments (text), then edit these segments making substitutions and changes.

token (text token)
: In Speedtype, the symbol used in place of a word or phrase, with a prefix, suffix, underline, or capitalize character attached.

undent
: In compose or format_document, a control used in conjunction with the indent control.

INDEX

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: MULTICS WORDPRO REFERENCE MANUAL

ORDER NO. AZ98-02

DATED JULY 1983

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

PLEASE FOLD AND TAPE–
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

CUT ALONG L
FOLD ALONG LINE
FOLD ALONG LINE

Together, we can find the answers.

Honeywell

Honeywell Information Systems
U.S.A.: 200 Smith St., MS 486, Waltham, MA 02154
Canada: 155 Gordon Baker Rd., Willowdale, ON M2H 3N7
U.K.: Great West Rd., Brentford, Middlesex TW8 9DH **Italy**: 32 Via Pirelli, 20124 Milano
Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F. **Japan**: 2-2 Kanda Jimbo-cho Chiyoda-ku, Tokyo
Australia: 124 Walker St.. North Sydney, N.S.W. 2060 **S.E. Asia**: Mandarin Plaza, Tsimshatsui East, H.K.

39364, 5C1283, Printed in U.S.A.

AZ98-02