SC24-5219-1
File no. S370/4300-39

Program Product

IBM Virtual Machine/ System Product: EXEC 2 Reference

Program Number 5664-167

Release 2

IBM

Second Edition (April 1982)

This edition, SC24-5219-1, applies to release 2 of IBM Virtual Machine/System Product, Program Number 5664-167, and to all subsequent versions and releases until otherwise indicated in the new editions or Technical Newsletters. Changes are continually made to the information contained herein; before using this publication in connection with the operation of IBM systems, consult the IBM System/370 and 4300 Processors Bibliography, GC20-0001, for the editions that are applicable and current.

Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

Summary of Amendments
For a list of changes, see page iii.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Programming Publications, Department G60, P.O. Box 6, Endicott, New York, U.S.A. 13760. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Summary of Amendments
for SC24-5219-1

VARIABLE SHARING

Programs called from an EXEC 2 file can now directly access and manipulate all variables contained in that EXEC 2 file through an EXEC 2 facility called EXECCOMM. Variables can also be assigned values as a side-effect of command or subcommand execution.

NEW PRE-DEFINED VARIABLE

The pre-defined variable &CMDSTRING is initialized to the untranslated command string available from the command line.

The purpose of this publication is to define the EXEC 2 language. It is to be used primarily as a reference manual; it contains all of the formats, syntax rules, and descriptions of the arguments for EXEC 2 statements.

For tutorial information on using the EXEC 2 language, refer to "Appendix D: EXEC 2 Primer for New Users." The material contained therein may be used in conjunction with the reference section.

The reference section of this publication contains these parts:

- "Part 1: Introduction" summarizes what the EXEC 2 language is and what it is capable of. It introduces and defines some of the terminology used throughout this manual. EXEC 2 statements and the rules for interpreting them are also discussed.

- "Part 2: EXEC 2 Statements" discusses in detail the different types of EXEC 2 statements. This discussion is followed by illustrations of the syntax of each EXEC 2 statement and a description of the function of each statement. "User-Defined Functions" and "EXEC 2 Name Substitution" are also discussed.

- "Part 3: Notes on EXEC 2" contains detailed discussions on particular aspects of EXEC 2 that do not fit into a category by themselves.

- "Part 4: BNF Description of the EXEC 2 Syntax" contains a description of the main features of the EXEC 2 syntax in Backus-Naur Form (BNF). This section presents an alternative description of the EXEC 2 syntax for those familiar with this type of notation. This is not essential reading.

- "Part 5: EXEC 2 Errors" lists the error messages and return codes issued by the EXEC 2 interpreter.

This publication also has these appendixes:

- "Appendix A: CMS EXEC and EXEC 2 Relationship" makes a comparison between CMS EXEC and EXEC 2 statements.

- "Appendix B: Sample EXEC 2 Files" gives two examples of EXECs written in the EXEC 2 language.

- "Appendix C: EXEC 2 in CMS." This appendix discusses how CMS identifies EXEC 2 files, the limits CMS imposes on using EXEC 2, examples of using EXEC 2 with assembler language programs, and the execution of XEDIT macros in EXEC 2. Appendix C also contains a discussion of variable sharing through the EXECCOMM interface.

- "Appendix D: EXEC 2 Primer for New Users" provides a tutorial aid for users who are unfamiliar with the EXEC 2 language. This primer is intended for the person who has a modest amount of CMS experience and enough familiarity with a text editor so that the mechanics of creating a disk file present no serious difficulty. Users who have already mastered a command programming language for some other system, or who have experience with the earlier CMS EXEC facility, may prefer to read the EXEC 2 reference material instead of the primer.

- "Appendix E: Useful EXEC 2 Techniques" shows some solutions to some common EXEC 2 programming problems.

If you are unfamiliar with writing EXEC files or need tutorial information, you may find it helpful to read "Appendix D: EXEC 2 Primer for New Users" before reading the reference section of this manual.

Note: Although EXEC 2 is designed to be system independent, the implementation requirements of CMS (the host system) impose certain limits on using EXEC 2. See Appendix C for details.

NOTATIONAL CONVENTIONS USED IN THIS BOOK

The conventions used in this publication to illustrate EXEC 2 statements follow:

- Uppercase letters and punctuation marks (except as described below) represent information that must be given exactly as shown.

- Lowercase letters represent information that must be supplied by the user.

- Information contained within brackets [] represents an option that can be included or omitted.

- Vertical lists that are not enclosed in brackets represent alternatives, one of which must be given. For example:

  A
  B

- Vertical lists that are enclosed in brackets represent alternatives, one of which may be given. For example:

  [X]
  [Y]

- An ellipsis (...) indicates that a variable number of items may be included.

- Underlined elements represent an assumed (default) value in the event a parameter is omitted.

Prerequisite Publications:

IBM Virtual Machine/System Product: Introduction, GC19-6200

Corequisite Publications:

IBM Virtual Machine/System Product: System Messages, SC19-6204

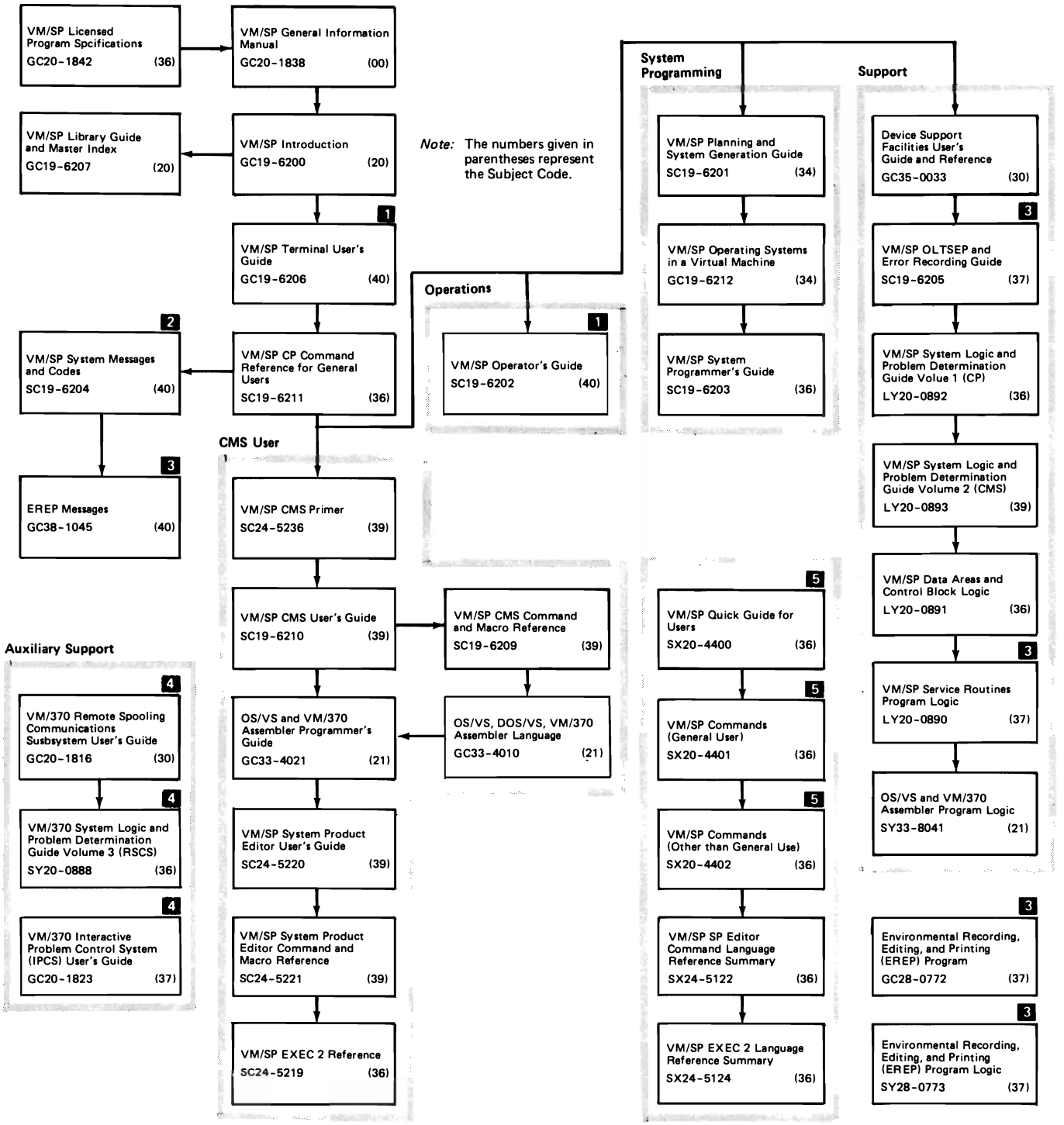
IBM Virtual Machine/System Product: CMS User's Guide, SC19-6210

IBM Virtual Machine/System Product: CMS Command and Macro Reference, SC19-6209

IBM Virtual Machine/System Product: System Product Editor User's Guide, SC24-5220

IBM Virtual Machine/System Product: System Product Editor Command and Macro Reference, SC24-5221

Publications that support VM/SP as used in conjunction with VM/370 Release 6



Legend:

1 For SNA terminal users, the prerequisite publication is: *VM/VCNA Installtion, Operations, and Terminal Use,* Order Number SC27-0502.

2 All users of virtual machine must use the *VM/SP System Messages and Codes* publication.

3 Contains information on VM/EREP support. EREP Release 3 is recommended for use with VM/SP Release 2.

4 VM/370 Release 6 components. However, the IPCS Extension Program Product (5748-SA1) and the RSCS Networking Program Product (5748-XP1) are recommended for use with VM/SP.

5 If you want all three of the Reference Summary publications, use SBOF 3820 when ordering.

EXEC 2 is intended for manipulating English-like words as they appear in computer command languages. It is also capable of performing integer arithmetic and simple string manipulation.

The notational conventions used in this publication to illustrate EXEC 2 statements are discussed in the Preface.

Executing EXEC 2 Programs

EXEC 2 programs reside in EXEC files, and are executed by the EXEC 2 interpreter. The EXEC 2 interpreter can be invoked by issuing a command such as:

```
EXEC  filename  [arg1  [arg2  ... ]]
```

where "filename" is the name of the EXEC 2 file to be executed, and "arg1", "arg2", ..., are arguments that are passed to it. In some command environments (such as XEDIT) the word "EXEC" is omitted, and in others (such as CMS console command mode) it is optional. (See Appendix C for the rules on how EXEC 2 files are distinguished from other EXEC files in CMS.)

EXEC 2 files can have any filename. EXEC 2 files have the filetype EXEC for files that are invoked from CMS command mode, and the filetype XEDIT for files used as XEDIT macros. Other filetypes may be used for EXEC 2 files that are invoked from other environments (see Appendix C).

EXEC 2 files can have either "F" (fixed) or "V" (variable) format.

Introduction to the EXEC 2 Language

EXEC 2 files contain EXEC 2 statements. An EXEC 2 statement occupies one line, and may be a comment or an executable statement. A comment is a line in which the first nonblank character is an asterisk, and is ignored during execution. An executable statement consists of a sequence of words, the first of which does not begin with an asterisk. A word is a string of contiguous nonblank characters. Words are separated from each other by one or more blanks. (Refer to Appendix C for implementation limits on EXEC 2 statements and words.)

An executable statement may be:

- a null statement (which has no effect),
- a command (which is issued to a command interpreter),
- an assignment (which manipulates EXEC 2 variables), or
- a control statement (which manipulates EXEC 2 variables, controls execution or flow through the file, or performs console input or output).

Assignments start with the name of an EXEC 2 variable, and control statements start with an EXEC 2 control word. EXEC 2 variables and control words begin with an ampersand. Variables are local to the current EXEC 2 file. Most variables are initially unset, and they have an apparent null value. The variables &1 &2 ..., are special, and are initialized to the arguments "arg1", "arg2", ..., that are passed to the EXEC 2 file. For example, if an EXEC named "TEST" was invoked as "TEST X Y Z", &0 would contain "TEST" and the arguments &1, &2, and &3 would contain X, Y, and Z, respectively.

The following are examples of variables:

```
&X
&3.1415927
&UPPER_LIMIT
&(X)
```

The following are examples of control words:

```
&TYPE
&LOOP
&EXIT
```

A label, appearing as the first word of a line, may be attached to an executable statement (including a null statement) but does not form part of the statement. A label is distinguished by its first character, which is a hyphen.

The following are examples of labels:

```
-X
-
-&A
-(TYPE)
-.-.-.-
```

When an EXEC 2 file is invoked, execution starts at line number 1 and proceeds sequentially, except when otherwise directed by control statements.

Rules for Interpreting Executable Statements

Executable statements are interpreted, one at a time, according to the following general rules. (There are a few explicit exceptions, which are noted elsewhere.)

1. The statement is scanned. This discards leading, trailing, and other surplus blanks, leaving a sequence of words separated from each other by a single blank.

2. The words forming the statement are searched for the names of any EXEC 2 variables. These variables are replaced by their values, unless the variable is the target of an assignment, its name is retained. (A precise description is given later in the section "EXEC 2 Name Substitution.") During this process, the words may grow or shrink in length.

3. If, as a result of step 1, a word is reduced to the null string, it is discarded from the statement so that the next word is deemed immediately to follow the previous one. With this exception, the words retain their identity. For example, if the value of a variable contains an embedded blank, the word containing it is still treated as one word, although when printed it might appear as two. For more details, see the section "Part 3: Notes on EXEC 2" on embedded blanks.

4. The statement is analyzed syntactically, and executed according to the rules on the following pages. Note that, except for identifying the targets of assignment, the syntax analysis is done after steps 1, 2, and 3 above.

Part 2: EXEC 2 Statements

Types of Executable Statements

- Null statement.

  A null statement is an executable statement in which the number of words is zero.

- Commands.

  An executable statement is deemed to be a command if it contains at least one word, and its first word does not start with an ampersand. It is issued immediately to the host system (CMS) or to a subcommand environment (for example, XEDIT). When it is finished, control returns to the EXEC 2 file, and its return code can be obtained from the predefined EXEC 2 variable &RC. (See the section "EXECCOMM - Sharing EXEC 2 Variables with Assembler Language Programs" for possible side-effects of command execution.)

- Assignments.

  An executable statement is an assignment if the first word starts with an ampersand and the second word is an equal sign. The first word is taken as the name of an EXEC 2 variable, and it is assigned the value of the expression that follows the equal sign. The expression may be any of the following:

  null

  a single word, for example: ABC

  an arithmetic expression, consisting of a sequence of words that represent positive or negative integers, separated by plus or minus signs, for example: 3 - 4 + -11 - 00

  a function invocation, for example:

  ```
  &PIECE  OF  &1  2  1
  ```

  an arithmetic expression (as above) in which the last term is replaced by a function invocation that yields a numeric value, for example:

  ```
  -1 + &LENGTH OF &1
  ```

  A variable of the form &j, where "j" is an unsigned integer without leading zeros, cannot be set with an assignment statement if "j" exceeds the number of EXEC 2 arguments that are currently set.

The value of the variable on the left-hand side of the assignment statement is not modified until the expression on the right-hand side has been evaluated. If an assignment statement is syntactically invalid, or if evaluation of the expression results in numeric overflow, execution stops abnormally with an error message, without further evaluation.

- Control statements.

  An executable statement is a control statement if the first word is an EXEC 2 control word and the second word either is absent or is not an equal sign. Examples of control words are &GOTO, &EXIT, &IF, and &PRINT.

Predefined Variables

The following EXEC 2 variables are initialized or maintained automatically.

&

Initialized to its own name (the value "&").

&0

Initialized to the first word of the command string that is passed to the EXEC 2 interpreter. The first word may be delimited according to the parsing rules of the host system. In CMS, &0 may be delimited by a blank or a parenthesis. Normally, this variable has the same value as &FILENAME, but it may be different if the EXEC 2 file was invoked via a synonym.

&1 &2 ...

These are the EXEC 2 arguments. They are initialized to the arguments "arg1", "arg2", ..., which are passed to the EXEC 2 file. EXEC 2 identifies individual arguments passed to it by the presence of a blank character which delimits each argument. They are reset by &ARGS or &READ ARGS, and they are temporarily reset by invocation of user-defined subroutines and functions. EXEC 2 arguments beyond the last that is set have an apparent null value, and cannot be set explicitly (for example, with an assignment statement). (See the description of &N and &INDEX.)

&ARGSTRING

Initialized to the argument string that is passed to the EXEC 2 file. It is treated as a single literal string starting with the character immediately following the blank which was used to delimit &0 (see above); or, if the delimiter is a character rather than a blank, &ARGSTRING starts with the delimiter character itself. It includes any leading, embedded, or trailing blanks. The initial value includes the EXEC 2 arguments &1 &2 ..., but &ARGSTRING is not affected by changes to them.

&BLANK

A word that has the value of a single blank.

&CMDSTRING

Initialized to the untranslated command string that is passed to the EXEC 2 file. It is treated as a single literal string starting with the first word of the command string and including any embedded or trailing blanks.

&COMLINE

Initialized to zero, and maintained as the number of the line from which the last command (or subcommand) was issued from the EXEC 2 file.

&DATE

The true date on the primary meridian (Greenwich Mean Time (GMT)) in the form YY/MM/DD. &DATE is evaluated when the statement containing it is executed. (See the description of &TIME.)

&DEPTH

Maintained as the number of user-defined functions and subroutine invocations to which return has not yet been made.

&FILEMODE

Initialized to the filemode (third qualifier) of the EXEC 2 file.

&FILENAME

Initialized to the filename (first qualifier) of the EXEC 2 file.

&FILETYPE

Initialized to the filetype (second qualifier) of the EXEC 2 file (for example, "EXEC").

&FROM

Initialized to zero, and maintained as the number of the line in the EXEC 2 file from which the last &GOTO statement was executed.

&LINE
&LINENUM

Maintained as the number of the current line in the EXEC 2 file.

&LINK

Maintained as the number of the line from which the currently executing user-defined function or subroutine was invoked, or &LINK has the value 0 if there are no user-defined functions or subroutines in execution.

&N
&INDEX

Maintained as the number of EXEC 2 arguments that are set. Initially this is the number of arguments that are passed to the EXEC 2 file. It is reset as a side effect of &ARGS and &READ ARGS. &N or &INDEX is temporarily reset by invocation of user-defined subroutines and functions. (See the description of &1 &2 ....)

&RC
&RETCODE

Initialized to zero, and maintained as the return code from the last command (or subcommand) issued from the EXEC 2 file.

&TIME

The true time-of-day on the primary meridian (Greenwich Mean Time (GMT)) in the form HH:MM:SS. &TIME is evaluated when the statement containing it is executed. (See the description of &DATE.)

Control Statements

Control statements begin with a control word, which is usually followed by one or more additional words. The control words, and the rules for their use, are as follows.

&ARGS	[word1 [word2 ...]]

Assign "word1", "word2", ..., to the arguments &1 &2 ..., and discard any other EXEC 2 arguments that were previously set. The number of arguments now set is the number of words given in the &ARGS statement, which may be less or greater than the number of arguments previously set.

(See the description of &READ ARGS; also see the predefined variables &N, &INDEX, and &1 &2 ....)

&BEGPRINT &BEGTYPE	[n [k] * [*] label 1]
line1 line2 ...	

Print at the console "line1", "line2", ..., truncated if necessary at column "k", without removing surplus blanks or replacing any EXEC 2 variables. If the truncation column is not given, or is given as "*", the lines are not truncated by the EXEC 2 interpreter. (CMS truncates at 130 characters. See Appendix C.)

The number of lines to be printed is determined by the first argument, as follows:

n,1 Print the given number of lines; or, if there are insufficient lines in the file, print all lines to the end of the file.

* Print all lines to the end of the file.

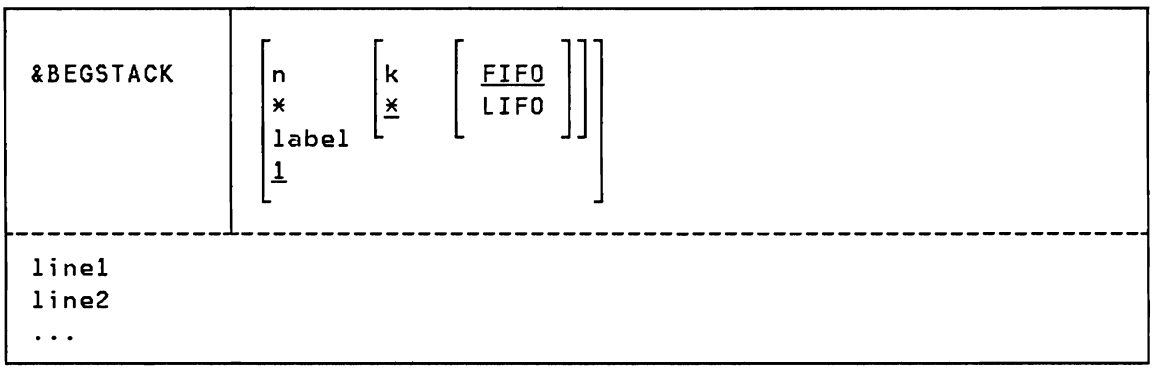
label Print down to, but not including, a line that contains the given label and nothing else; or, if such a line does not exist, print all lines to the end of the file. The label, to be recognized, must be wholly contained within the columns that would otherwise

be printed, and it must be the only word within these columns. The first character of a label must be a hyphen.

After the lines have been printed, execution continues on the line following the last one printed. If printing is terminated by a label, execution continues on the line following the label.

This and "&BEGSTACK" are the only statements that occupy more than one line. They are also the only statements that permit the lines of an EXEC 2 file to be handled literally, that is, without removing surplus blanks or replacing EXEC 2 variables.

(See the description of &PRINT and &TYPE.)



Place in the console stack "line1", "line2", ..., truncated if necessary at column "k", without removing surplus blanks or replacing any EXEC 2 variables. If the truncation column is not given, or is given as "*", the lines are not truncated. The lines are by default stacked FIFO (first in, first out), but this can be changed by giving "LIFO" (last in, first out) as the third argument.

The number of lines to be stacked is determined by the first argument, as follows:

n,1 Stack the given number of lines; or, if there are insufficient lines in the file, stack all lines to the end of the file.

* Stack all lines to the end of the file.

label Stack down to, but not including, a line which contains the given label and nothing else; or, if such a line does not exist, stack all lines to the end of the file. The label, to be recognized, must be wholly contained within the columns which would otherwise be stacked, and it must be the only word within these columns. The first character of a label must be a hyphen.

After the lines have been stacked, execution continues on the line following the last one stacked. If stacking is terminated by a label, execution continues on the line following the label.

This, &BEGPRINT, and &BEGTYPE are the only statements that occupy more than one line. They are also the only statements that permit the lines of an EXEC 2 file to be handled literally, that is, without removing surplus blanks or replacing EXEC 2 variables.

(See the description of &STACK.)

&BUFFER	n [comment] *

Discard the lookaside buffer (if any) together with its contents. Then, if "n" is given, and is positive, or if "*" is given, create a new lookaside buffer. If "n" is given, and is zero, a new lookaside buffer is not created. The value of "n" must not be negative. (In CMS, the initial buffer size is 32 lines. See Appendix C.)

The lookaside buffer is a device that enables the EXEC 2 interpreter to remember the location of labels to which reference has already been made and to keep a private copy of some of the more recently executed lines of the file. The lookaside buffer can thereby improve the performance of EXEC 2 loops, in which the same labels and lines are used repeatedly.

If "n" is given, it defines the maximum number of lines that can be kept in the buffer; if "*" is given, there is no fixed limit. For maximum effect, the buffer should be capable of keeping the longest loop in its entirety and should be set up before entering the loop. An even larger buffer may be advantageous if user-defined functions or subroutines are invoked from within a loop.

A lookaside buffer should not be used if the EXEC 2 file is subject to modification during execution. If it is used, the results are unpredictable.

&CALL	line-number [arg1 [arg2 ...]] label

Create a new generation of the EXEC 2 arguments &1 &2 ..., initialized to "arg1", "arg2", ..., and invoke the specified subroutine by transferring control to the given line, or to a line starting with the given label, in such a way as to allow control to be returned with the &RETURN statement.

The new generation of arguments supersedes the arguments that were previously set, making the previous values, and the number of arguments

previously set, temporarily inaccessible. On entry to the subroutine, the values of the arguments, and the number of arguments set, are as given in the &CALL statement. Their values, and the number of arguments set, can be changed inside the subroutine in the same way as outside, such as by assignment or with the &ARGS or &READ statement.

On return, the new generation of arguments is discarded, making the previous values, and the number of arguments previously set, again accessible. Execution resumes on the line following the &CALL statement.

The first character of a label must be a hyphen. The search for a label starts on the line following the &CALL statement; then, if a match is not found before the end of the file, the search resumes at the top. If a matching label does not exist, execution stops abnormally with an error message.

(See the description of &RETURN; also see the section "User-Defined Functions.")

&CASE	[U / M] [comment]

Translate to uppercase (U) any lowercase alphabetic characters that are read in response to subsequent &READ statements, or do not translate them (allow "mixed" (M) cases), or (if no argument is given) do not change the setting. Initially the translation is set to "U".

(See the description of &UPPER.)

&COMMAND	word1 [word2 ...]

Issue to the host system (CMS) the command comprising of "word1", "word2", ..., separated from each other by a single blank. When it is finished, its return code is obtainable from the predefined EXEC 2 variables &RC and &RETCODE. The &COMMAND statement normally has the same effect as:

```
word1  word2  ...
```

There are, however, the following differences:

- A command, the first word of which begins with an asterisk, a hyphen, or an ampersand can be issued by giving it as the argument to &COMMAND; otherwise it is interpreted as a comment, a labeled statement, an assignment, or a control statement. (Note however,

that these characters are not acceptable to CMS command mode. See Appendix C.)

- &COMMAND overrides any presumption of a subcommand environment and always issues the command to the host system (CMS).

(See the description of &SUBCOMMAND and &PRESUME; see the predefined variables &COMLINE, &RC, and &RETCODE. Refer to the section "EXECCOMM - Sharing EXEC 2 Variables with Assembler Language Programs" for possible side-effects of command execution.)

&DUMP	ARGS VAR[S] [var1 [var2 ...]]

Print lines at the console of the form:

```
var = VALUE
```

where var is &1 &2 ... or "var1", "var2", ...

ARGS Print one line for each EXEC 2 argument &1 &2 ... that is set.

VAR[S] Print one line for each of the variables "var1", "var2", ...

The lines are truncated if their length exceeds the implementation limit for printed output. (In CMS, the line is truncated if its length exceeds 130. See Appendix C.)

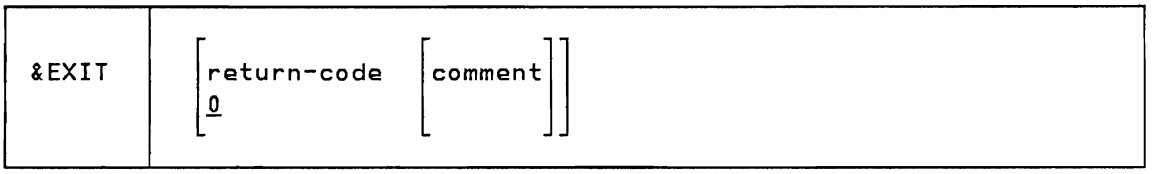
&ERROR	action

Set the action which, until further notice, is to be invoked automatically on return from any commands (and subcommands) that yield an error return code (a return code that is not zero). The action may be any executable statement, including a null statement.

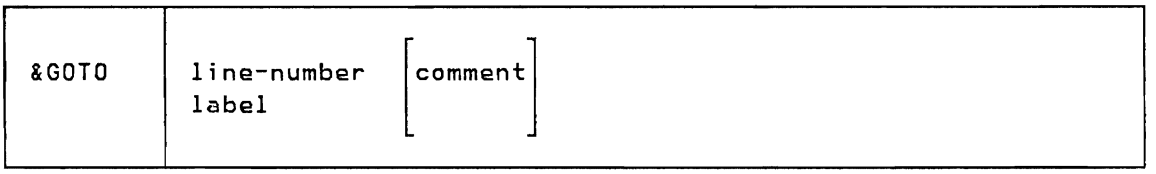
The action is not inspected at the time the &ERROR statement is executed. Instead, the search for and replacement of any EXEC 2 variables takes place each time the action is executed. The action is executed as if it occupied the same line in the EXEC 2 file as the command (or subcommand) that yielded the nonzero return code.

What happens after the action depends upon the type and consequences of the action. If it is itself a command (or subcommand) which also yields an error return code, execution stops abnormally with an error message; otherwise (unless the action causes a transfer of control), execution resumes at the line following the command that caused the action to be invoked.

Initially, the error action is set to the null statement.



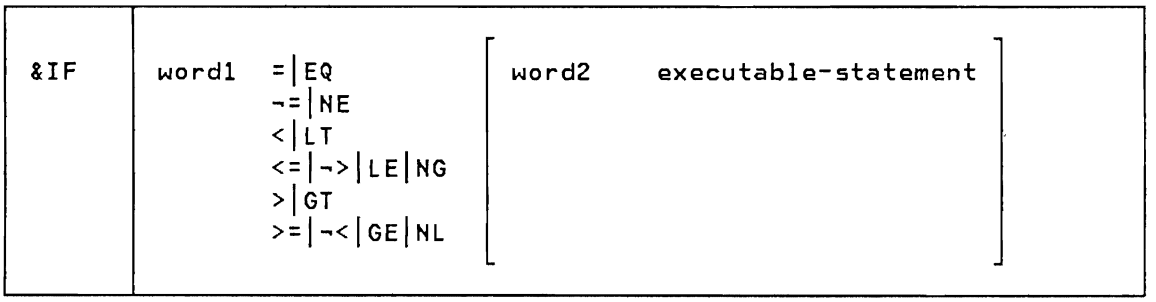
Stop execution of the EXEC 2 file, and yield the given return code. The return code must be numeric. If the given return code is not within the range of return codes acceptable to the host system, the result is defined by the implementation. (In CMS, the range is -2,147,483,648 to +2,147,483,647. See Appendix C.)



Transfer control to the given line or to the line starting with "label".

The first character of a label must be a hyphen. The search for a label starts on the line following the &GOTO statement. Then, if a match has not been found before the end of the file, the search resumes at the top. If a matching label does not exist, execution stops abnormally with an error message.

(See the description of &SKIP and &CALL; also see the predefined variable &FROM.)



If the condition is satisfied, execute the given executable statement; otherwise, proceed to the next statement. The comparative may be given in any of the forms shown (for example "=" or "EQ"). The comparison is numeric if both comparands are numeric; otherwise both comparatives are treated as character strings, and the shorter one is (for the purpose of the comparison) padded on the right with blanks. If "word2" is absent, a null string is used in its stead.

```
&LOOP   n       m
        label   *
                WHILE   condition
                UNTIL   condition
```

Loop through the following "n" lines, or down to (and including) the first line starting with "label", for "m" times, or indefinitely (*), or "WHILE" (or "UNTIL") the given condition is satisfied.

The values of "n" and "m" (if given) must be numeric; also "n" must be positive, and "m" must not be negative. If "m" is zero, the entire loop is ignored.

The first character of the label (if given) must be a hyphen. The label must be attached, as the first word of the line, to an executable statement that lies below the &LOOP statement.

The form of the condition (if given) is similar to that of the &IF statement previously described, namely:

```
word1   =|EQ                [ word2 [ comment ] ]
        ¬=|NE
        <|LT
        <=|¬>|LE|NG
        >|GT
        >=|¬<|GE|NL
```

The condition is evaluated before each iteration of the loop, including the first. If "word2" is absent, a null string is used in its stead. The comparison is numeric if both comparands are numeric; otherwise, both comparands are treated as character strings, and the shorter one is (for the purpose of the comparison) padded on the right with blanks.

If the condition is invalid, execution stops abnormally with an error message that identifies the line containing the &LOOP statement.

```
&PRESUME    [ &COMMAND
              &SUBCOMMAND  environment ]
```

Presume that any executable statements that have the syntax of a command (that is, the first word of the statement does not begin with an ampersand) are to be issued to the host system (CMS), or presume that they are to be issued to the given subcommand environment.

The name of the subcommand environment is not checked when the &PRESUME statement is executed. If, when a subcommand is subsequently issued, the environment does not exist, the only effect is to set a special return code. (In CMS, it is -3.)

The "&PRESUME" control statement with no arguments is equivalent to "&PRESUME &COMMAND".

By convention, the presumption is initially set to "&COMMAND" if the EXEC 2 file has a filetype of EXEC; otherwise, it is set to "&SUBCOMMAND filetype", where "filetype" is the filetype of the EXEC 2 file.

The presumption has no effect on &COMMAND or &SUBCOMMAND statements since these do not have the syntax of a command.

(See the description of &COMMAND and &SUBCOMMAND.)

&PRINT &TYPE	[word1 [word2 ...]]

Print at the console a line containing "word1", "word2", ..., separated from each other by a single blank, or print a blank line if there are no words given. The line is truncated if necessary. (In CMS, the line is truncated if its length exceeds 130. See Appendix C.)

Unlike &BEGPRINT and &BEGTYPE, surplus blanks are removed and the words to be printed are searched in the normal way for the names of EXEC 2 variables, that are replaced by their values.

(See the description of &BEGPRINT and &BEGTYPE.)

&READ	[n 1 * ARGS STRING var VAR[S] [var1 [var2 ...] * [* ...]]

Read from the stack (if the stack is not empty), or read from the console (otherwise). Then execute or assign what is read according to the following rules.

n, 1, * Read "n" lines, read 1 line, or read an indefinite number of lines (*), and execute them individually as if they had been part of the EXEC 2 file. Reading stops (and normal execution resumes) when "n" lines have been read, or when a &BEGPRINT, &BEGTYPE, &BEGSTACK, &EXIT, &GOTO, &LOOP, or &SKIP statement is encountered. Reading is suspended if a user-defined function or subroutine is invoked and continues when control returns from that invocation.

If a "&READ n" statement is read in response to a previous "&READ n" statement, the new value of n is added to the number of lines that remain from the previous statement. Reading stops if the number remaining becomes zero or less. The value of "n" may be negative.

If a "&READ *" statement is read in response to a previous "&READ n" or "&READ *" statement, or if a "&READ n" statement is read in response to a previous "&READ *" statement, an indefinite number of lines remain to be read.

ARGS Read a single line, assign the words in it to the EXEC 2 arguments &1 &2 ..., and discard any other EXEC 2 arguments that were previously set. The number of arguments now set is the number of words in the line, which may be less or greater than the number of arguments previously set. (See the description of &ARGS, and the predefined variables &N, &INDEX, and &1 &2 ...)

STRING Read a single line and assign it, as a literal string, to "var", without removing any surplus blanks or replacing any EXEC 2 variables.

VARS Read a single line and assign the words in it to the variables "var1", "var2", ... If the number of words in the line read exceeds the number of variables given in the statement, the surplus words are discarded. If the number of variables exceeds the number of words, the remaining variables are set to the null string. Therefore "&READ VARS" (without any variables) can be used to read a line and discard it. Asterisks (*) may be used in lieu of variable names to indicate that the corresponding words in the line read are to be discarded.

In the case of &READ ARGS and &READ VARS ..., the line that is read is scanned for words (leading, trailing, and other surplus blanks are discarded), but the words are treated as literals (there is no replacement of EXEC 2 variables).

The names of the variables in &READ VARS and &READ STRING are treated in the same way as on the left-hand side of an assignment statement. (See the section "EXEC 2 Name Substitution.") A variable of the form &j, where "j" is an unsigned integer without leading zeros, cannot be set with &READ VARS or &READ STRING if "j" exceeds the number of EXEC 2 arguments that are currently set.

Lines that are read may or may not be translated to uppercase. The case is determined by the translation mode that is set by the &CASE control statement. The &CASE control statement is issued prior to the &READ control statement. (See the description of &CASE.) However, if no case is specified, the lines read in default to uppercase.

Lines that are read are not truncated by the EXEC 2 interpreter; they are unaffected by the setting of &TRUNC. (See the description of &TRUNC.)

(In CMS, the maximum length of a line read from the console is 130, and the maximum length of a line read from the console stack is 255. See Appendix C.)

&RETURN	[word] [comment]

Return control to the most recent subroutine invocation (&CALL statement) to which return has not yet been made; or return "word" (or the null string) to the most recent user-defined function invocation to which a value has not yet been returned.

The generation of EXEC 2 arguments that was created at invocation is discarded. The previous values and the number of arguments previously set become accessible again. The number of lines (if any) that remain to be read from the stack or console in response to a previous "&READ n" statement is reset to the number outstanding at the time of the invocation. Any loops that have been opened in the subroutine or function, and not closed, are aborted; and any loops that were open at the time of invocation are reinstated.

If there is both a subroutine invocation and a function invocation to which return has not yet been made, return is to the more recent point of invocation. If there is neither, execution stops abnormally with an error message.

(See the description of &CALL; also see the section "User-Defined Functions.")

&SKIP	[n [comment]]
	[1 [ ]]

If n > 0, skip the next "n" lines of the EXEC 2 file. If n < 0, transfer control to the line that is "-n" lines above the current line. If n = 0, transfer control to the next line.

If an attempt is made to transfer control to a line number that is zero or negative, execution stops abnormally with an error message. If control is transferred to a line below the last in the EXEC 2 file, execution stops normally with a return code of zero.

(See the description of &GOTO.)

&STACK	[FIFO | LIFO] [word1 [word2 ...]]

Place a line in the console stack containing "word1", "word2", ..., separated from each other by a single blank, or stack a null line if there are no words. (In CMS, stacked lines are truncated at 255. See Appendix C.) The line is by default stacked FIFO (first in, first out), but this can be changed by giving "LIFO" (last in, first out) as the first argument.

Unlike &BEGSTACK, surplus blanks are removed and the words to be stacked are searched in the normal way for the names of EXEC 2 variables, that are replaced by their values.

(See the description of &BEGSTACK.)

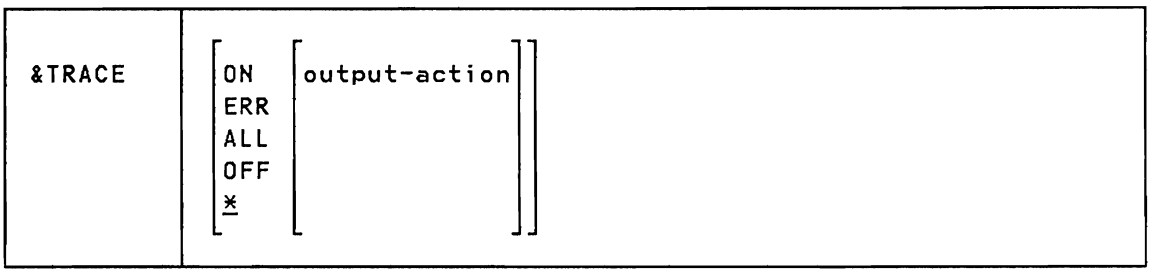
&SUBCOMMAND	environment [word1 [word2 ...]]

Issue to the given subcommand environment the subcommand comprising of "word1", "word2", ..., separated from each other by a single blank. When it is finished, its return code is obtainable from the predefined EXEC 2 variable &RC.

If the given environment does not exist, the only effect is to set a special return code. (In CMS, it is -3.)

Normally, it is convenient to "presume" the environment so that this control statement does not have to be issued for every subcommand (see the description of &PRESUME, above). The explicit use of the &SUBCOMMAND statement does, however, allow subcommands that start with an asterisk, a hyphen, or an ampersand to be issued. (Compare with the description of &COMMAND.) Also note that the statement "&SUBCOMMAND environment" (without any additional arguments) is the only way of issuing a null subcommand.

(See the description of &COMMAND; also see the predefined variables &COMLINE, &RC, and &RETCODE. Refer to the section "EXECCOMM - Sharing EXEC 2 Variables with Assembler Language Programs" for possible side-effects of command execution.)



where "output-action", if given, is:

```
     &PRINT    [word1  [word2 ...]]
or:
     &COMMAND        word1 [word2 ...]
or:
     &SUBCOMMAND     environment [word1 [word2 ...]]
```

Trace commands (and subcommands) that are issued from the EXEC 2 file; or trace commands (and subcommands) that yield an error return code (a return code that is not zero); or trace all executable statements; or do not trace any statements; or (if "*" is given) do not change the setting. The setting remains in effect until reset. The initial setting is OFF.

Trace information can be printed at the console, or passed to a command (or subcommand) for processing. The trace destination is determined by the output action, as described below.

ON
: When tracing is ON, each command is traced before it is executed. Subsequently, the return code is traced if it is not zero. The return code is traced on a line by itself in the form "+++ E(nnn) +++".

ERR
: When ERR is in effect, commands that yield a nonzero return code are traced after execution, followed by the return code. The return code is traced on a line by itself in the form "+++ E(nnn) +++".

ALL
: When ALL is in effect, every executable statement, preceded by its line number, is traced before it is executed. Nonzero return codes are traced (as for ON and ERR). Loop conditions and lines that are read from the console are also traced. The statement following an &IF clause, the action given in an &ERROR statement, and the conditional phrase in a &LOOP statement are traced as literal words (that is, without replacement of any variables). These statements and phrases are traced again, with the normal replacement of variables, at the time of their execution. A statement that is executed as a consequence of a satisfied &IF clause is preceded in the trace by an ellipsis. Words that exceed 24 characters in length are truncated in the trace at 21 characters and followed by an ellipsis. Statements that exceed 80 characters in length (with the line number and

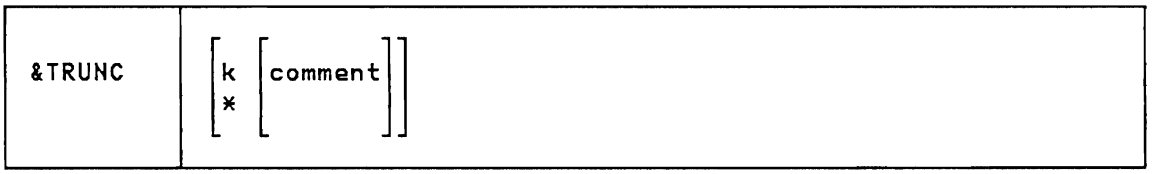
preceding ellipsis, if present) are truncated in the trace at an integral number of words and followed by an ellipsis.

OFF Do not trace any statements. This is the initial setting.

* Do not change the setting. "&TRACE" without arguments is equivalent to "&TRACE *".

output-action
The output action gives the destination of the tracing information. The words in it are searched in the normal way for the names of EXEC 2 variables. These variables are replaced by their values, and the resulting sequence of words is set aside. When a trace line is produced, it is prefixed with the sequence of words, and the resulting EXEC 2 statement is executed without tracing. (See the description of &PRINT, &TYPE, &COMMAND, and &SUBCOMMAND). If the return code from the command or subcommand is nonzero, execution stops abnormally with an error message.

Initially the output action is set to "&PRINT", which causes the trace to be printed at the console. If the output action is not given, the previous action remains in effect.



Set the truncation column for EXEC 2 statements to "k", or set it to the maximum value (*), or (if no argument is given) do not change it. Initially, it is set to the maximum value. (In CMS, the maximum value is 255. See Appendix C.)

This setting affects only the reading of EXEC 2 statements from a file and the search for labels; it does not affect lines read from the console (that are not truncated) or lines appearing within a &BEGPRINT, &BEGTYPE, or &BEGSTACK statement (that are separately controlled). This setting does not affect the length to which a statement can grow during or after replacement of EXEC 2 variables.

Changing the truncation column has the side-effect of purging the lookaside buffer (if there is one), and may consequently degrade performance if done within a loop.

(See the description of &BUFFER.)

&UPPER	ARGS VAR[S] [var1 [var2 ...]]

Translate to uppercase any lowercase alphabetic characters in the values of the EXEC 2 arguments &1 &2 ..., or translate to uppercase any lowercase alphabetic characters in the values of "var1", "var2", ...

A variable of the form &j, where "j" is an unsigned integer without leading zeros, cannot be translated with &UPPER VARS if "j" exceeds the number of EXEC 2 arguments that are currently set.

(See the description of &CASE.)

Predefined Functions

A predefined function can be invoked only in the last term on the right-hand side of an assignment statement. The invocation takes the form:

```
function-name  OF  [arg1  [arg2  ... ]]
```

The names of the predefined functions, and the rules for their use are as follows.

&CONCATENATION OF &CONCAT OF	[word1 [word2 ...]]

Concatenates "word1", "word2", ..., into a single word, without intervening blanks; or yields the null string if there are no words. Example:

```
&A = **
...
&B = &CONCAT OF XX &A 45
&PRINT &B
```

This results in the printed line:

```
XX**45
```

&DATATYPE OF &TYPE OF	[word]

Yields the value NUM if "word" represents a valid (signed or unsigned) number; otherwise, yields the value CHAR.

&DIVISION OF &DIV OF	dividend divisor

Yields a numeric value that results from dividing the dividend by the divisor. Both the dividend and the divisor must be numeric and the divisor must not be zero.

In precise terms, the value is the integral part of the division of the absolute value of the dividend by the absolute value of the divisor, or minus this value if the dividend is not zero and the sign of the dividend differs from that of the divisor.

Examples:

```
&W = &DIV OF 7 2
&X = &DIV OF -7 -2
&Y = &DIV OF -7 2
&Z = &DIV OF 0 -2
```

This sets &W to 3, &X to 3, &Y to -3, and &Z to 0.

&LEFT OF	word j

Yields a string of length "j" in which "word" is left-justified and either padded with blanks or truncated on the right.

(See the description of &RIGHT OF.)

&LENGTH OF	[word]

Yields a numeric value representing the length of the word (that is, the number of characters in it); or yields zero if the word is absent.

&LITERAL OF	[string]

Yields the literal string that begins with the character following the blank that terminates "OF" and ends with the last nonblank character before or at the truncation column. Any leading or embedded blanks are retained, and the search for and replacement of any EXEC 2 variables that may appear in the string is suppressed. Example:

```
& = &LITERAL OF &X =
&X = **
&PRINT & &X
```

This results in the printed line:

```
&X = **
```

(See the description of &STRING OF.)

&LOCATION OF	needle [haystack]

Searches "haystack" for the first occurrence of "needle", and yields a number indicating its starting position, or yields zero if there is no occurrence (or if the length of "needle" exceeds that of "haystack"). Example:

```
&X = &LOCATION OF ANN LIZANNE
```

This sets &X to 4.

(See the description of &PIECE OF, &SUBSTR OF, and &POSITION OF.)

&MULTIPLICATION OF &MULT OF	i j [k ...]

Yields a numeric value representing the result of multiplying the given words. There must be at least two words given (i and j), and each word must be numeric (signed or unsigned). Example:

```
&X = &MULT OF 4 5 6
```

This sets &X to 120.

&PIECE OF &SUBSTR OF	word i [j | *]

Extracts that piece of "word" that starts at character "i", with length "j"; or that starts at character "i" and runs to the end of the word (*).

The value of "i" (and "j" if given) must be numeric; also "i" must be positive, and "j" must not be negative.

If the value of "i" exceeds the length of the word, the value of the function is the null string. If "j" is given, but exceeds the remaining length of the word, the remaining length is used instead.

Example:

```
&A = &PIECE OF ABCDE 2 3
&B = &PIECE OF ABCDE 2 999
```

```
&C = &PIECE OF ABCDE 33 2
&PRINT &A &B &C ***
```

This results in the printed line:

```
BCD BCDE ***
```

(See the description of &LOCATION OF.)

&POSITION OF	word [word1 [word2 ...]]

Compares "word" with "word1", "word2", ..., looking for a match, and yields a numeric value representing the position of the first matching word, or yields zero if "word" does not match any of the other words (or if there are no other words given). Example:

```
&X = &POSITION OF THE NOW IS THE TIME
```

This sets &X to 3.

(See the description of &LOCATION OF and &WORD OF.)

&RANGE OF	stem i j

Yields a string consisting of the words that are composed by appending to the given stem the numbers i, i+1, ..., j, the words being separated from each other by a single blank; or yields the null string if i > j.

The stem is treated as a literal until after the composition is performed. The numbers that are appended to it are stripped of any plus sign or redundant leading zeros.

The composed names are searched for any EXEC 2 variables, which are replaced by their values in the usual way. If, as a result of this, a word is reduced to the null string, it is discarded from the result, and the next word is deemed immediately to follow the previous one.

Examples:

A. Irrespective of the values of &A, &A3, &A4, and &A5, the sequence:

```
&X = &RANGE OF &A 3 5
&PRINT &X
```

produces the same result as:

```
&PRINT &A3 &A4 &A5
```

B. The sequence:

```
&ARGS A BC DEF GHIJ KLMNO
...
&X = &RANGE OF & 1 &N
&PRINT &X
```

yields the printed line:

```
A BC DEF GHIJ KLMNO
```

C. The sequence:

```
&X = &RANGE OF AB -2 +2
&PRINT &X
```

yields the printed line:

```
AB-2 AB-1 AB0 AB1 AB2
```

&RIGHT OF	word j

Yields a string of length "j" in which "word" is right-justified and either extended with blanks or shortened on the left.

(See the description of &LEFT OF.)

&STRING OF	[string]

Yields the string that begins with the character following the blank that terminates "OF" and ends with the last nonblank character before, or at, the truncation column, suppressing the removal of any leading or embedded blanks in the string.

Each word in the string is searched in the usual way for the names of EXEC 2 variables. These variables are replaced by their values. However, blanks are not removed from the string, even if they are adjacent to a word that is reduced to the null string.

Example:

```
&A = STRING
&B = ENDS
&X = &STRING OF A PIECE OF &A    HAS TWO &B
&PRINT &X
```

This yields the printed line:

A PIECE OF STRING HAS TWO ENDS

(See the description of &LITERAL OF.)

&TRANSLATION OF &TRANS OF	word1 [word2 [word3]]

Makes a copy of "word1", modifies the characters in it as directed by "word2" and "word3", and yields the resulting string.

The rules for modification are as follows. Each character of the copy is considered in turn, and:

1. if "word2" does not contain a matching character, the character in the copy is left unchanged; or

2. if "word2" contains a matching character, in position "i" (or if it contains several matching characters, the first of which occupies position "i"), the character in the copy is replaced by the ith character of "word3", or by a blank if "word3" is not given or contains fewer than "i" characters.

The result has the same length as "word1".

Examples:

1. The sequence:

```
&X = ABC123,XYZ
&X = &TRANS OF &X ABCDEF, abcdef
&PRINT &X
```

   yields the printed line:

```
abc123 XYZ
```

2. The sequence:

```
&YY/MM/DD = 80/10/29
&MM/DD/YY = &TRANS OF 45678312 12345678 &YY/MM/DD
&PRINT &MM/DD/YY  &YY/MM/DD
```

   yields the printed line:

```
10/29/80 80/10/29
```

&TRIM OF	[word]

Yields a string consisting of "word" with any trailing blanks removed, or yields the null string if "word" is not given.

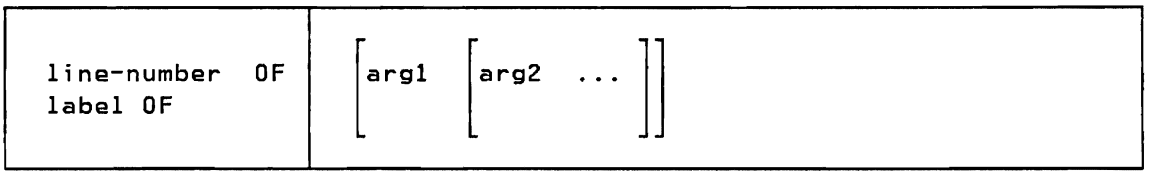
&WORD OF	[word1 [word2...]] i

Yields the ith word from the given list of words, or yields the null string if "i" is zero or exceeds the number of words that are given. The value of "i" must be numeric, and "i" must not be negative.

(See the description of &POSITION OF.)

User-Defined Functions

A user-defined function can be invoked only in the last term on the right-hand side of an assignment statement. The invocation takes the form:



The effect is to create a new generation of the EXEC 2 arguments &1 &2 ..., initialized to "arg1", "arg2", ..., and to invoke the given function; that is, to transfer control to the given line, or to a line starting with the given label, in such a way as to allow a value to be returned with the &RETURN statement.

The new generation of arguments supersedes the arguments that were previously set, making the previous values and the number of arguments previously set temporarily inaccessible. On entry to the body of the function, the values of the arguments, and the number of arguments set, are as given in the function invocation. Their values, and the number of arguments set, can be changed in the body of the function in the same way as outside, such as by assignment or with the &ARGS or &READ statement. On return, the new generation of arguments is discarded, and the previous values, and the number of arguments previously set, become accessible again.

The first character of a label must be a hyphen. The search for a label starts on the line following the function invocation. Then, if a match is not found before the end of the file, the search resumes at the top. If a matching label does not exist, execution stops abnormally with an error message.

(See the description of the &CALL and &RETURN control statements.)

Examples:

A. The user-defined function

```
-OVERLAY  OF  layee  layer
```

is to return the string "layee" overlaid by "layer". (The result will be different from "layer" only if "layee" is longer than "layer".) Here is the body of the function, preceded by an example of its invocation:

```
&S = -OVERLAY OF &S *
...
* THIS FUNCTION USES "&" AS A TEMPORARY VARIABLE
-OVERLAY & = 1 + &LENGTH OF &2
&1 = &PIECE OF &1 &
&1 = &CONCAT OF &2 &1
&RETURN &1
```

B. Suppose there is an external program TIME that stacks the CPU time consumed in (say) microseconds. The user-defined function -TIME OF is to return this number as its value, relieving its caller of the need to issue the external command, check the return code, and read the answer. Here is the body of the function, preceded by an example of its use:

```
&T = -TIME OF
...   (sequence to be timed)
&T = 0 - &T + -TIME OF
&PRINT TIME CONSUMED WAS &T
...
-TIME &COMMAND TIME
&IF &RC ¬= 0 &GOTO -UNEXPECTED
&READ ARGS
&RETURN &1
-UNEXPECTED &PRINT UNEXPECTED ERROR FROM TIME
&EXIT &RC
```

EXEC 2 Name Substitution

The words that form an executable statement are searched for the names of EXEC 2 variables. These variables are replaced by their values. This is done according to the following steps:

1. Each word is inspected for ampersands, starting with the rightmost character of the word and proceeding to the left.

2. If an ampersand is found, then it, with the rest of the word to the right, is taken as the name of an EXEC 2 variable and replaced (in the word) by its value. This may increase or decrease the length of the word. Initially, all variables have a null value, except:

   a. the variables that represent the EXEC 2 control words and predefined functions; they are initialized to their own names (for example, the value of "&IF" is "&IF"); and

   b. the EXEC 2 arguments, and the other predefined variables, that have the values specified in the section "Predefined Variables."

3. Inspection resumes at the next character to the left, and the procedure is repeated from step 2 above, until the word is exhausted.

There is an exception if the word is the target of an assignment. In this case, inspection for ampersands stops on the second character of the word.

Note that any characters that are substituted are not themselves inspected for ampersands. They are, however, included in the name of the next variable if another ampersand is found to the left.

These rules make it possible to construct arrays of subscripted variables.

Examples:

1. The sequence:

```
(Original file)              (After Substitution)

&X = 123                     2. &X = 123
&PRINT ABC &X ABC&X 000&X    3. &PRINT ABC 123 ABC123 000123
```

   yields the printed line:

```
       ABC 123 ABC123 000123
```

2. The sequence:

```
(Original file)              (After Substitution)

&I = 2                       2. &I = 2
&X&I = 5                     3. &X2 = 5
&I = &I - 1                  4. &I = 2 - 1
&X&I = &I + 1                5. &X1 = 1 + 1
&X = &X&I + &X&X&I           6. &X = 2 + 5
&PRINT ANSWER IS &X          7. &PRINT ANSWER IS 7
```

yields the printed line:

```
ANSWER IS 7
```

3. The sequence:

```
(Original file)              (After Substitution)

&X = &CONCAT OF X &BLANK X   2. &X = &CONCAT OF X   X
&&X = 7                      3. &X X = 7
&DUMP VARS &X &&X            4. &DUMP VARS &X &X X
```

yields the printed line:

```
&X = X X
&X X = 7
```

1. Recursive execution
2. Termination of an EXEC 2 file
3. Console input buffer
4. Assignment statement
5. Evaluation of &DATE and &TIME
6. Size and treatment of numbers
7. Removing plus signs and leading zeros
8. Syntax of conditional phrases
9. Embedded blanks
10. &LOOP statement
11. Closing of loops
12. Search for labels
13. Performance of label searches
14. EXEC 2 words are not reserved words
15. Example of &TRACE ALL
16. Truncation column

1. Recursive execution.

   An EXEC 2 file may invoke itself recursively, or may invoke other EXEC 2 files, by issuing the appropriate command or subcommand. (EXEC 2 files may also invoke CMS EXEC files. See Appendix C.) EXEC 2 files that have the filetype EXEC can, for example, be invoked by means of the statement:

   ```
   &COMMAND  EXEC  filename  <arg1  <arg2  ... >>
   ```

2. Termination of an EXEC 2 file.

   An EXEC 2 file stops execution and returns to its caller:

   a. when an &EXIT statement is executed; or

   b. when an attempt is made to pass control to a line beyond the last (for example by "falling off" the end of the file), in which case a return code of zero is used; or

   c. when an EXEC 2 error is encountered, in which case a message is printed and execution stops abnormally.

3. Console input buffer.

   EXEC 2 can use the CMS console input buffer (sometimes referred to as the console stack). This is a conceptual area in which lines can be deposited FIFO (first in, first out), or LIFO (last in, first out), and subsequently retrieved by attempts to read from the console. It provides a simple mechanism for communicating between

programs. In EXEC 2 files, lines can be deposited in the buffer with the &STACK or &BEGSTACK statements, and can be retrieved with the &READ statement.

4. Assignment statement.

   The word immediately following the target of an assignment must be a literal equal sign. It cannot be an EXEC 2 variable that has the value of an equal sign nor an EXEC 2 variable that is discarded from the statement due to having a null value. Conversely, if an equal sign is to be the first word following a control word, either it must be given as an EXEC 2 variable that has the value of an equal sign, or there must be an intervening word that reduces to the null string; otherwise, the statement is interpreted as an assignment, and (if it is valid as such) the control word is assigned a new value (see below, under "EXEC 2 words are not reserved words"). With this exception, a word that is discarded due to having a null value has no effect on whether a statement is interpreted as an assignment, even if it occurs at the beginning of the statement. For example, in the sequence:

```
&X =
&LOOP 2 2
   &X &Y = 2 + 1
   &X = &PRINT
```

   the first statement in the loop is executed as an assignment to &Y, and then (the second time) as a &PRINT statement, resulting in the line:

```
3 = 2 + 1
```

5. Evaluation of &DATE and &TIME.

   The time is taken once for each execution of a statement that refers to the predefined variable &DATE or &TIME. Therefore, multiple references to these variables within a statement yield the same values. If consistency (rather than currentness) is required over a range exceeding one statement, then the values of &DATE and &TIME must be assigned to ordinary variables. For example,

```
&STACK LIFO &DATE &TIME
&READ VARS &D &T
```

6. Size and treatment of numbers.

   Words that are treated as numbers must represent integers. No limit is imposed on the size of a number that appears in a comparison, or as an argument to the predefined function &DATATYPE OF. In contexts that require numeric values, numbers must lie within a range that is defined by the implementation. (In CMS, the range is -2,147,483,648 to +2,147,483,647. See Appendix C.) An attempt to interpret a number outside the allowable range, or to derive such a number by

arithmetic, causes numeric overflow. This overflow causes execution to stop abnormally with an error message.

7. Removing plus signs and leading zeros.

   A plus sign, and any redundant leading zeros, can be stripped from a numeric quantity by performing an arithmetic operation on it.

   Example:

```
&X = 000000000000000000012
&Y = &X + 0
&PRINT &X &Y
```

   This yields the printed line:

```
000000000000000000012 12
```

8. Syntax of conditional phrases.

   In the conditional phrases that occur in the &IF and conditional &LOOP statements, a missing second comparand is regarded as a null string. The first comparand and the comparator must always be present; otherwise execution stops abnormally with an error message. If there is a risk of the first comparand having a null value, syntactic validity can be ensured by prefixing both comparands with the same character. For example, the clause

```
&IF /&1 = /
```

   is satisfied if, and only if, &1 is null or blank; and

```
&IF /&1 = /PRINT
```

   is syntactically valid even if &1 is null.

   A similar technique can be used to force character-string comparisons even if both of the comparands are numeric. (In this case, the prefix must not be numeric.) For example, if it is known that &1 has a numeric value, the clause

```
&IF /&1 < /0
```

   is satisfied if and only if &1 begins with a plus or minus sign. If &1 is equal to "1", the clause is false. However, if &1 is equal to "+1", the clause is true, since "+" is less than "0" in a character-string comparison. (For the relative values of characters, refer to the internal codes for the EBCDIC character set, given in IBM System/370 Reference Summary, GX20-1850.)

9. Embedded blanks.

   With a few exceptions, EXEC 2 does not embed blanks in the values of variables. The exceptions are as follows:

   a. &ARGSTRING is initialized to the string containing the EXEC 2 arguments, and &CMDSTRING is initialized to the command string exactly as passed to the EXEC 2 file. Therefore, these variables may contain embedded blanks.

   b. The "&READ STRING var" statement assigns to the given variable the complete line exactly as read, that may contain embedded blanks.

   c. The predefined variable &BLANK can be used to embed blanks in the value of a variable, for example:

```
&Y = &CONCAT OF A &BLANK B
```

   d. The predefined function &RANGE OF inserts a blank between each word; the predefined functions &LITERAL OF and &STRING OF retain embedded blanks that are given in their arguments; and the predefined functions &LEFT OF, &RIGHT OF, and &TRANSLATION OF can yield leading, embedded, or trailing blanks.

   e. Embedded blanks can be transmitted from one variable to another with the assignment statement, and to the EXEC 2 arguments &1 &2 ... with the &ARGS statement or by invocation of user-defined subroutines and functions.

   Embedded blanks are always significant. For example, "&IF " is not recognized as "&IF"; and "10 " and " 10" cannot be used as numbers.

   Embedded blanks can be removed from the value of a variable by stacking it and rereading it as a sequence of words. Suppose, for example, that a line to be read from the console is required both in its literal form (with embedded blanks, if any) and as a series of normal words (without embedded blanks). The following sequence achieves this:

```
&READ STRING &S
&STACK LIFO &S
&READ ARGS
```

   Now &S contains the literal string, and the EXEC 2 arguments &1 &2 ..., contain the constituent words.

10. &LOOP statement.

    The first three words of the &LOOP statement are searched for EXEC 2 variables (in the normal way) when the &LOOP statement is executed. However, the remainder of the statement (which is present only if "WHILE" or "UNTIL" is given) is saved without inspection. This

saved phrase is then interpreted as a condition each time around the loop (including the first time). For example:

```
&J = 3
&LOOP 2 UNTIL &J = 5
   &J = &J + 1
   &PRINT &J
```

This results in the printed lines:

```
4
5
```

11. Closing of loops.

A loop may be in any of three mutually exclusive states: active, suspended, or closed. A loop becomes active when execution of its defining &LOOP statement begins. It is suspended if another loop becomes active before the first is closed or if a user-defined subroutine or function is invoked. It becomes active again when the second loop is closed or when a corresponding &RETURN statement is executed. A loop is closed when it is active, and when either:

a. the requirement for termination, given in the &LOOP statement, is met; or

b. control is transferred to a line outside the scope of the loop by any means other than invocation of a user-defined function or subroutine.

In addition, the &EXIT statement closes all loops, and the &RETURN statement closes any loops that have been opened during execution of a user-defined subroutine or function.

Examples:

a. In the following sequence, the &SKIP statement closes the loop after ten iterations, since it transfers control to a line below the last in the loop.

```
&J = 0
&LOOP 2 *
   &J = &J + 1
   &IF &J > 9 &SKIP 0
```

b. In the following sequence, the second loop closes the first loop since it causes control to be transferred to a line outside the scope of the first loop.

```
&LOOP 1 *
   &LOOP 1 1
      & =
```

The first loop would similarly be closed, for the same reason, if the second loop statement were replaced by a &BEGPRINT, &BEGTYPE, or &BEGSTACK statement which occupied more than one line.

12. Search for labels.

    The search for a label to which reference is made in a &CALL, &GOTO, or &LOOP statement, or in the invocation of a user-defined function, involves examination of the first word on each line, without regard to its context, or what follows it. It is, therefore, necessary to avoid using labels that would be matched by the first word of a line within a &BEGPRINT, &BEGTYPE, or &BEGSTACK statement.

    Labels that are attached to statements are treated literally; they are not searched for EXEC 2 variables. Labels need not be unique.

13. Performance of label searches.

    a. &CALL, &GOTO, and user-defined functions

       A &CALL statement, a &GOTO statement, or an invocation of a user-defined function that transfers to a label above the current statement tends to be inefficient, especially in long EXEC 2 files. It is preferable to use the &LOOP statement in place of an upward "&GOTO label" statement.

    b. &LOOP label ...

       A "&LOOP label ..." statement is converted, at the time of its execution, into the equivalent "&LOOP n ..." statement. Therefore, the overhead for finding the label is incurred only once, when the loop is entered, irrespective of the number of iterations.

14. EXEC 2 words are not reserved words.

    EXEC 2 control words, predefined functions, and predefined variables are known as EXEC 2 words. EXEC 2 words begin with an ampersand; but, unlike ordinary variables, they have an initial value that is not null.

    The initial value of EXEC 2 control words and predefined functions is the word itself (for example, the value of "&IF" is "&IF"). If one of these words is assigned a different value (for example, &IF = ABC), then the feature that it represents in the language is lost to the EXEC 2 file unless it, or another variable, is reset to the old value (for example &IFX = &LITERAL OF &IF) and used appropriately.

    In the case of predefined variables other than the EXEC 2 arguments, the special properties of a variable disappear if an explicit assignment is made to it. For example, the statement:

```
&TIME = &TIME
```

inhibits further automatic updating of the variable &TIME.

Words of the form &j, where "j" is an unsigned integer without leading zeros, are reserved for the EXEC 2 arguments. They can be set explicitly (for example, &2 = 1) only if they are within the range of arguments that are currently set. With this exception, EXEC 2 words are not reserved words, and can, if desired, be used like ordinary variables.

&READ VARS, &READ STRING, and &UPPER VARS are treated as explicit assignments to the variables given; &ARGS, &READ ARGS, and &UPPER ARGS are not treated as explicit assignments to &N or &INDEX.

If a feature, function, or value is accessible through more than one name (for example, &PIECE OF and &SUBSTR OF), an assignment to one of the names does not affect the other name or names.

With the exception of the arguments &1 &2 ..., there are no EXEC 2 words that end with a numeral, and it is intended that no such words will ever be introduced. Therefore, variables such as &A1, &A2, ..., can be relied upon to have an initial value of null. However, the names of variables that do not end with a numeral should not be used in a way that relies upon their initial value being null.

15. Example of &TRACE ALL

Assume that an editor accepts the requests NEXT (which moves down the file, and yields a return code of zero unless the end of file is reached), LENGTH (which stacks the length of the current line), and TOP (which moves to the first line in the file). The following sample edit macro (called LONGER) searches for the next line that is longer than the given length (passed to the EXEC file as an argument).

```
&TRACE ALL
NEXT 0
&IF &RC ¬= 0 TOP
NEXT
&LOOP 4 WHILE &RC = 0
   LENGTH
   &READ VAR &L
   &IF &L > &1 &EXIT
   NEXT
&EXIT &RC
```

If the macro is invoked at the end of the file, the search starts from the top.

Suppose that the macro is invoked with the parameter 40 at the end of a file containing two lines, both of length 30. This is the trace:

```
2. NEXT 0
+++ E(1) +++
3. &IF 1 ¬= 0 TOP
3. ... TOP
4. NEXT
5. &LOOP 4 WHILE &RC = 0
--- LOOP WHILE 0 = 0
6. LENGTH
7. &READ VAR &L
30
8. &IF 30 > 40 &EXIT
9. NEXT
--- LOOP WHILE 0 = 0
6. LENGTH
7. &READ VAR &L
30
8. &IF 30 > 40 &EXIT
9. NEXT
+++ E(1) +++
--- LOOP WHILE 1 = 0
10. &EXIT 1
```

16. Truncation Column

A truncation column may be specified with the &BEGSTACK, &BEGTYPE, &BEGPRINT, and &TRUNC statements.

In all cases the truncation column is the last column in which characters are significant. Characters in columns that are beyond the truncation column are ignored.

Example:

```
----:----1----:----2
&TRUNC 10
&X = ABCDEFGHIJK
```

This sets &X to ABCDE.

Part 4: BNF Description of the EXEC 2 Syntax

What follows is a description of the EXEC 2 syntax in Backus-Naur Form (BNF). This is an alternative to the other descriptions in this manual and is not essential reading.

The items enclosed in the angular brackets "<" and ">" are variables (nonterminal symbols). These items are replaced by the items to the right of "::=". ("::=" means "is to be replaced by".) The items to the right of "::=" may give exact replacements, other variables to be replaced, or the final step of the syntax breakdown. Items in capital letters are exact replacements. Items in lowercase, not surrounded by the angular brackets, are the final step (terminals) of the syntax breakdown.

```
<exec_file>             ::=  <statement>
                             <exec_file> <statement>

<statement>             ::=  <comment>
                             <label> <executable_stmt>
                             <executable_stmt>

<comment>               ::=  * anything

<label>                 ::=  -<word>

<executable_stmt>       ::=  <unconditional_stmt>
                             <if_clause> <executable_stmt>

<word>                  ::=  <number>
                             <character_string>
                             <variable>

<unconditional_stmt>    ::=  <assignment>
                             <control_stmt>
                             <command>
                             null

<if_clause>             ::=  &IF <word> <comparator> <word>

<number>                ::=  <unsigned_integer>
                             +<unsigned_integer>
                             -<unsigned_integer>

<character_string>      ::=  <character>
                             <character_string><character>
```

```
<variable>              ::=   &<character_string><letter>
                              &<character_string><variable>
                              &<character_string>symbol
                              &symbol

<assignment>            ::=   <variable> = <rhs>

<control_stmt>          ::=   &ARGS
                              &BEGPRINT
                              &BEGTYPE
                              &BEGSTACK
                              &BUFFER
                              &CALL
                              &CASE
                              &COMMAND
                              &DUMP
                              &ERROR
                              &EXIT
                              &GOTO
                              &IF
                              &LOOP
                              &PRESUME
                              &PRINT
                              &READ
                              &RETURN
                              &SKIP
                              &STACK
                              &SUBCOMMAND
                              &TRACE
                              &TRUNC
                              &TYPE
                              &UPPER

<command>               ::=   CP command
                              CMS command
                              XEDIT command (if working with
                               an XEDIT macro)

<comparator>            ::=   =|EQ
                              ¬=|NE
                              <|LT
                              <=|¬>|LE|NG
                              >|GT
                              >=|¬<|GE|NL

<unsigned_integer>      ::=   <digit>
                              <unsigned_integer><digit>

<character>             ::=   <letter>
                              <unsigned_integer>
                              symbol

<letter>                ::=   a|b|c|d| ... |x|y|z
```

```
<rhs>                    ::=   <word>
                               <function_invocation>
                               <arithmetic_rhs>
                               null

<digit>                  ::=   0|1|2|3|4|5|6|7|8|9

<function_invocation>    ::=   &CONCAT OF
                               &CONCATENATION OF
                               &DATATYPE OF
                               &DIV OF
                               &DIVISION OF
                               &LEFT OF
                               &LENGTH OF
                               &LITERAL OF
                               &LOCATION OF
                               &MULT OF
                               &MULTIPLICATION OF
                               &PIECE OF
                               &POSITION OF
                               &RANGE OF
                               &RIGHT OF
                               &STRING OF
                               &SUBSTR OF
                               &TRANS OF
                               &TRANSLATION OF
                               &TRIM OF
                               &TYPE OF
                               &WORD OF
                               user-defined function

<arithmetic_rhs>         ::=   <arithmetic_expr>
                               <arithmetic_expr> + <function_invocation>
                               <arithmetic_expr> - <function_invocation>

<arithmetic_expr>        ::=   <number>
                               <arithmetic_expr> + <number>
                               <arithmetic_expr> - <number>
```

Part 5: EXEC 2 Errors

If the EXEC 2 interpreter finds an error, it issues the following message:

ERROR IN EXEC FILE fn ft fm, LINE nnn - description of error

(In CMS, this is message DMSEXE085E.)

Execution of the EXEC 2 file then stops abnormally with one of the following return codes.

Return Code	Description of Error
10001	FILE NOT FOUND
10002	WRONG FILE FORMAT
10003	WORD TOO LONG
10004	STATEMENT TOO LONG
10005	INVALID CONTROL WORD
10006	LABEL NOT FOUND
10007	INVALID VARIABLE NAME
10008	INVALID FORM OF CONDITION
10009	INVALID ASSIGNMENT
10010	MISSING ARGUMENT
10011	INVALID ARGUMENT
10012	CONVERSION ERROR
10013	NUMERIC OVERFLOW
10014	INVALID FUNCTION NAME
10015	END OF FILE FOUND IN LOOP
10016	DIVISION BY ZERO
10017	INVALID LOOP CONDITION
10019	ERROR RETURN DURING &ERROR ACTION
10020	ASSIGNMENT TO UNSET ARGUMENT
10021	STATEMENT OUT OF CONTEXT
10097	INSUFFICIENT STORAGE AVAILABLE
10098	FILE READ ERROR nnn
10099	TRACE ERROR nnn

The EXEC 2 interpreter also issues the following messages:

INVALID EXEC COMMAND

(In CMS, this is message DMSEXE175E.)

Return Code: 10000

INSUFFICIENT STORAGE FOR EXEC INTERPRETER

(In CMS, this is message DMSEXE255T.)

Return Code: 10096

The &CRASH statement is useful in debugging the EXEC 2 interpreter (module DMSEXE). It is intended for use only by IBM or customer system support personnel. Note that the &CRASH command is not used for debugging programs or EXEC 2 files written in the EXEC 2 language. For information on debugging EXEC files written in the EXEC 2 language, refer to the &TRACE statement in this book.

A complete description of the &CRASH command can be found in VM/SP System Programmer's Guide.

Appendix A: CMS EXEC and EXEC 2 Relationship

Converting CMS EXEC Files to EXEC 2 Files

CMS EXEC files continue to be supported without modification. However, to take advantage of the new function and performance available under EXEC 2, you must convert your EXEC files to conform to EXEC 2 language. The first step in converting CMS EXEC files to EXEC 2 files is to change the &CONTROL statement to &TRACE. This statement determines which EXEC interpreter will handle the EXEC file. &CONTROL indicates the CMS EXEC interpreter, and &TRACE indicates the EXEC 2 interpreter. This must be the first record in the EXEC file.

Next, the CMS EXEC statements must be converted to their corresponding EXEC 2 counterparts. A comparison between the language definitions of CMS EXEC and of EXEC 2 follows. A section listing unique EXEC 2 functions follows the comparison sections.

EXEC Statement Comparison

Control Statements

CMS EXEC	EXEC 2
&ARGS	Supported; does not have a limit of 30 arguments.
&BEGEMSG	Not supported
&BEGPUNCH	Not supported
&BEGSTACK ALL parameter specifies stacking of the entire line up to 130 characters. The absence of ALL results in truncation at 72 characters.	ALL parameter is not supported. If a truncation value is not specified, the lines are not truncated. &END is not supported as a data list delimiter. The number of lines to be stacked can be specified with the "n" parameter. An "*" will stack lines to the end of the file. The "label" parameter allows lines to be stacked down to a specified label. Parameters

	FIFO and LIFO are supported.
&BEGTYPE	(See comment for &BEGSTACK; also supported as &BEGPRINT.)
&CONTINUE	Treated as a null statement.
&CONTROL	&TRACE. &TRACE does not support the following parameters: MSG, TIME, PACK, NOMSG, NOTIME, NOPACK. Uses parameter ON in place of CMS. Parameters OFF, ERROR(ERR), ALL are supported. "*" has been added.
&EMSG	Not supported
&END	Not supported
&ERROR	Supported; &CONTINUE is treated as a null statement.
&EXIT	Supported
&GOTO The parameter TOP directs the EXEC processor to the first line of the program.	TOP is not explicitly supported. The line number value 1 provides equivalent function. The line number and label parameters are supported.
&HEX	Not supported
&IF	Supported; &$ and &* are not supported.
&LOOP	The conditional expression must be preceded with "WHILE" or "UNTIL". "*" has been added.
&PUNCH	Not supported
&READ	Supported; "*" and STRING have been added.
&SKIP	Supported
&SPACE	Not supported; see &PRINT.
&STACK	Supported. HT and RT are not supported. SET CMSTYPE HT and SET CMSTYPE RT are their equivalents.

&TIME Used to request timing information during execution.	Not supported as a control statement. Used as a predefined variable to display the true time-of-day on the primary meridian (Greenwich Mean Time (GMT)). See &TIME under the section "PREDEFINED VARIABLES" in this publication.
&TYPE	Supported; also supported as &PRINT.

Predefined Functions

CMS EXEC	EXEC 2
&CONCAT	Supports &CONCAT OF and &CONCATENATION OF
&DATATYPE	Supports &DATATYPE OF and &TYPE OF
&LENGTH	Supports &LENGTH OF
&LITERAL	Supports &LITERAL OF
&SUBSTR	Supports &SUBSTR OF and &PIECE OF

Predefined Variables

CMS EXEC	EXEC 2
&0 Represents the filename.	Normally, this parameter represents the filename, but this may be different if the EXEC was invoked by a synonym.
&1 &2 ...	Supported; does not have a limit of 30 arguments.
&*	Not supported, but see &POSITION OF and &LOCATION OF.

&$	Not supported, but see &POSITION OF and &LOCATION OF.
&DISKX	Not supported
&DISK*	Not supported
&DISK?	Not supported
&DOS	Not supported
&EXEC	Supported as &FILENAME
&GLOBAL	Not supported
&GLOBALn	Not supported
&INDEX	Supported; also supported as &N, does not have a limit of 30 arguments.
&LINENUM	Supported; also supported as &LINE
&READFLAG	Not supported
&RETCODE	Supported; also supported as &RC
&TYPEFLAG	Not supported

Functions Unique to EXEC 2

Control Statements

&BUFFER

&CALL

&CASE

&COMMAND

&DUMP

&RETURN

&SUBCOMMAND

&TRACE (similar to &CONTROL in CMS EXEC)

&TRUNC

&UPPER

Predefined Functions

&DIVISION OF and &DIV OF

&LEFT OF

&LOCATION OF

&MULTIPLICATION OF and &MULT OF

&POSITION OF

&RANGE OF

&RIGHT OF

&STRING OF

&TRANSLATION OF and &TRANS OF

&TRIM OF

&WORD OF

Predefined Variables

&

&ARGSTRING

&BLANK

&CMDSTRING

&COMLINE

&DATE

&DEPTH

&FILEMODE

&FILENAME

&FILETYPE

&FROM

&LINK

&TIME

Other

User-Defined Functions

Blanks in values of variables

Arbitrary characters allowed in variable names

Appendix B: Sample EXEC 2 Files

1. This sample EXEC 2 file, called GRAB EXEC, copies a file from any CMS disk to the user's A-disk.

```
&TRACE
&IF &N = 0 &GOTO -TELL
&IF &N < 2 &GOTO -BAD
&IF &N > 3 &GOTO -BAD
&IF &N = 2 &ARGS &1 &2 *
COPYFILE   &1 &2 &3   &1 &2 A
&EXIT &RC

-BAD &PRINT INVALID GRAB COMMAND
&EXIT 101

-TELL &PRINT COMMAND IS: GRAB FN FT [MODE]
&PRINT COPIES THE GIVEN FILE TO THE A-DISK,
&PRINT AND PASSES BACK THE RETURN CODE FROM
&PRINT 'COPYFILE'.
&EXIT 100
```

2. This sample EXEC 2 file, called SHIP EXEC, sends a specified CMS file to a specified user. The comments are included for tutorial purposes.

```
&TRACE

*  COMMAND IS: SHIP USER FILENAME FILETYPE [MODE]
*  IF THERE ARE NO ARGUMENTS GIVEN, TELL USER HOW...

&IF &N = 0 &GOTO -TELL

*  CHECK THE NUMBER OF ARGUMENTS, AND USE FILEMODE
*  OF "*" IF IT IS NOT GIVEN...

&IF &N < 3 &GOTO -BAD
&IF &N > 4 &GOTO -BAD
&IF &N = 3 &ARGS &1 &2 &3 *

*  SPOOL PUNCH TO RECIPIENT'S CARD-READER, OR
*  COMPLAIN IF RECIPIENT IS NOT KNOWN TO SYSTEM...

CP SPOOL PUNCH TO &1 CLASS A
&IF &RC ¬= 0 &GOTO -BADUSER

*  PUNCH THE FILE, OR COMPLAIN IF FAILURE...
```

```
PUNCH &2 &3 &4
&IF &RC ¬= 0 &GOTO -ERROR

*  TELL RECIPIENT WHAT HAS BEEN DONE; THEN UNSPOOL
*  THE PUNCH, AND RETURN WITH SUCCESS...

CP MSG &1 I HAVE PUNCHED YOU MY FILE &2 &3 &4
CP SPOOL PUNCH TO * CLASS A
&EXIT

*  TELL RECIPIENT INVALID SHIP COMMAND, AND RETURN
*  WITH ERROR...

-BAD &PRINT INVALID SHIP COMMAND
&EXIT 101

*  TELL RECIPIENT GIVEN USERID IS NOT VALID, AND
*  RETURN WITH ERROR...

-BADUSER &PRINT &1 IS NOT A VALID USERID
&EXIT 102

*  TELL RECIPIENT ERROR WHEN PUNCHING FILE; THEN
*  UNSPOOL PUNCH AND RETURN WITH ERROR...

-ERROR &PRINT ERROR &RC FROM "PUNCH" (WHILE IN SEND)
CP SPOOL PUNCH TO * CLASS A
&EXIT 103

*  TELL USER HOW...

-TELL &PRINT COMMAND IS: SHIP USER FN FT [FM]
&EXIT 100
```

Appendix C: EXEC 2 in CMS

Identifying EXEC 2 Files

Since both CMS EXEC and EXEC 2 files are called in the same way, CMS examines the first statement of the EXEC 2 file to determine which EXEC interpreter must handle it. If the first statement of the EXEC file is &TRACE, CMS calls the EXEC 2 interpreter to handle it. If the first statement is not &TRACE, CMS calls the CMS EXEC interpreter to handle it.

Calling EXEC 2 Programs from CMS Command Level

When EXEC 2 programs are called from command level, the command verb (which becomes &0) and the arguments (which individually become &1 &2 ... and collectively become &ARGSTRING) are translated to uppercase. &CMDSTRING will contain the untranslated command string.

When EXEC 2 programs are invoked from another EXEC 2 program, no translation takes place, and &CMDSTRING will be the same as the &STRING OF &0 &ARGSTRING (if &0 was delimited by a blank) or &CONCAT OF &0 &ARGSTRING (if &0 was delimited by a parenthesis).

It is possible to 'pretend' a command-level call by using the CMS command, CMDCALL. CMDCALL converts EXEC 2 extended plist function calls to CMS extended plist command calls. The use of CMDCALL in an EXEC 2 exec allows the message 'FILE NOT FOUND' to be displayed for the ERASE, LISTFILE, RENAME, and STATE commands. Also, an EXEC 2 program invoking another EXEC 2 program will have the same results as an EXEC 2 program being called from command level. &0, &1 &2 ..., and &ARGSTRING will be translated as stated above.

In either case, calling an EXEC 2 program from command level or invoking an EXEC 2 program from another EXEC 2 program, the CMS convention that parentheses are token delimiters is applied to separate &0 from &ARGSTRING, but it is not applied to delimit &1, &2, ... from each other.

Summary of Limits for EXEC 2 Files in CMS

Some CMS limits that apply to EXEC 2 files are:

- EXEC 2 files used as CMS command files must have the word &TRACE as the first word in the first record of the file. In subcommand environments, such as XEDIT for XEDIT macros, the word &TRACE is optional.
- The maximum length of an EXEC 2 line is 255.
- The maximum length of a statement, after replacement of variables, is 511. (This limit is enforced only as needed by the interpreter; some statements can grow to a greater length.)
- The maximum length of a word, after replacement of variables, is 255.
- The maximum length of a line read from the console is 130, and from the console stack is 255.
- The maximum length of a printed line is 130.
- An EXEC 2 filename can be from one to eight characters long. The valid characters are A-Z, 0-9, $, #, @, +, : (colon), - (hyphen), and _ (underscore). The filetype must be EXEC for files that are invoked from CMS command mode and XEDIT for files used as XEDIT macros.
- All EXEC 2 files have an initial lookaside buffer of 32 lines (see the &BUFFER description in the "Control Statements" section). The &BUFFER 0 statement must be issued to delete the lookaside buffer if the file is to be modified while being executed.
- In a context that requires numeric values, numbers must be in the range -2,147,483,648 to +2,147,483,647.
- In CMS, return codes for the &EXIT control statement are limited to the range -2,147,483,648 to +2,147,483,647. Attempts to exceed these limits will cause the EXEC 2 file to stop abnormally with an error message (NUMERIC OVERFLOW).
- CMS commands issued from EXEC 2 files are invoked in such a way that most information and error messages issued by the following CMS commands will not be typed: ERASE, LISTFILE, RENAME, STATE, and FILEDEF. (See the description of CMDCALL, in section "Calling EXEC 2 Programs from CMS Command Level" above, for an exception to this statement.) This is also true for any other system or user command that makes a distinction in its operation based on flags passed in register 1. However, note that a nonzero return code from any of these commands will be reflected in the predefined variables &RETCODE and &RC.

- EXEC 2 is designed to maintain a complex program environment. For this reason, automatic clean-up will not be invoked at the completion of each command within the EXEC. It is the programmer's responsibility to ensure that any necessary clean-up functions (i.e. STRINIT, OS RESET, VSAM CLEAN-UP, etc.) are invoked when needed.

- CP and CMS commands issued from an EXEC 2 file must be in uppercase.

- The length limit for values assigned via the EXECCOMM interface is 255. If the limit is exceeded, the return code from the EXECCOMM interface is 16 (INVALID VALUE).

- The length limit for the external name of a shared variable is 254. If the limit is exceeded, the return code from the EXECCOMM interface is 8 (INVALID NAME).

- If a "STORE" reference is made to an unset EXEC 2 argument (i.e. a variable of the form &i where "i" is an unsigned number without leading zeros that exceeds the number of EXEC 2 arguments that are currently stored), no assignment is performed, and the return code from the EXECCOMM interface is 8 (INVALID NAME).

- If a "FETCH" reference is made to &ARGSTRING (or &CMDSTRING) via the EXECCOMM interface and the length of &ARGSTRING (or &CMDSTRING) exceeds 255, a length of 256 is recorded. If the length of the caller's area exceeds 255, the value is truncated without any error indication.

- If a "FETCH" reference is made to &TIME or &DATE via the EXECCOMM interface, the time-of-day returned is the same for all references from a given program invocation, since (as far as the EXEC 2 interpreter is concerned) the same statement is still in execution (see note 5, "Evaluation of &DATE and &TIME," in section 3).

Using EXEC 2 Parameter Lists with Assembler Language Programs

The calls illustrated below are made via CMS SVC 202 calls.

1. EXEC 2 interpreter calling another program:

```
For &COMMAND word0 word1 ... wordn
     R0 = A(NPLIST)
     R1 = A(tokenized CMS plist)
     High-order byte of R1 is X'01'.

For &SUBCOMMAND word0 word1 ... wordn
     R0 = A(NPLIST)
     R1 = A(=CL8'word0')
     High-order byte of R1 is X'02'.
```

where:

```
 NPLIST  DS  0F
         DC  A(COMVERB)
         DC  A(BEGARGS)
         DC  A(ENDARGS)
         DC  A(0)
         .
         .
         .

 COMVERB EQU *          the command verb
         DC  C'word0'
         DC  C' '       optional blanks
 BEGARGS EQU *          the argument string
         DC  C'word1'
         DC  C' '
         DC  C'word2'
         DC  C' '
         .
         .
         .
         DC  C'wordn'
 ENDARGS EQU *
```

2. Calling the EXEC 2 interpreter with a tokenized plist only:

```
R0 = irrelevant
R1 = A(CMS tokenized plist)
High-order byte of R1 as from LA, BAL, or BALR.
```

The value of &ARGSTRING in this case is set as if by the EXEC 2 statement:

```
   &ARGSTRING = &RANGE OF & 1 &INDEX
```

3. The EXEC 2 interpreter can be passed an extended plist, that specifies an untokenized argument string. In addition, the parameter list may precisely identify the EXEC file to be executed (and thereby specify a filetype other than EXEC, or an explicit filemode); or it may identify an "in-memory file." An "in-memory file" is similar in concept to a file on disk, but it is resident in memory.

```
R0 = A(NPLIST)
R1 = A(CMSPLIST)
High-order byte of R1 is X'01'.

 NPLIST  DS  0F
         DC  A(0)          (ignored by EXEC 2)
         DC  A(BEGARGS)
         DC  A(ENDARGS)
         DC  A(0)  or  A(FBLOCK)
         .
         .
         .
CMSPLIST DS 0F
         DC CL8'EXEC'
         DC CL8'filename' (Ignored if file block is given)
         .                (Always ignored by EXEC 2 interface)
         .
         .
*    If no FBLOCK is given for the above instruction in the NPLIST
*    (i. e. A(FBLOCK) is zero), the filename of the EXEC file is
*    taken from the second 8-byte token of the area addressed
*    by register 1.  This will be the value after synonym resolution
*    so it may be different from &0.

         .
         .
         .
 BEGARGS EQU *             the argument string
         DC  C'amp0'       no embedded blanks, becomes &0
         DC  C' '          single blank separates &0 from &ARGSTRING
         DC  C'argstring'  becomes &ARGSTRING
 ENDARGS EQU *
         .
         .
         .
 FBLOCK  DS  0F             ** File Descriptor **
         DC  CL8'filename'  if blank, &0 will be used - see &0
         DC  CL8'filetype'  may be blanks for &PRESUME &COMMAND
         DC  CL2'filemode'  should be given as '*', or blanks for
*                           in-memory files

*   IMPORTANT NOTE: The default &PRESUME setting is as follows:
*
*   No file block given:                    &COMMAND
*   File block given, filetype blank:       &COMMAND
*   File block given, filetype non-blank:   &COMMAND filetype
```

```
*
*   Thus, if a filetype of EXEC is explicitly specified in the file
*   block, the default presumption will be &SUBCOMMAND EXEC, and not
*   &COMMAND, even though an EXEC file of filetype EXEC will be
*   executed.
*
*   The following is an FBLOCK extension block. The first
*   halfword specifies how many words are in the extension
*   block. CMS requires a value of either zero or two.

         DC  XL2'0002'            Number of full words that follow
         DC  AL4(PGMFILE)         Address of the in-memory
                                  EXEC 2 descriptor
         DC  AL4(PGMEND-PGMFILE)  Number of bytes in the
                                  descriptor

*    If no "in-memory file" is provided, the values in
*    the extension must either both be zero, or be
*    omitted by changing the XL2'0002' to XL2'0000'.

 PGMFILE DS  0F                   in-memory EXEC 2 Program
         DS  A(line 1),F'len 1'   Address and length of
                                  file line 1
         DS  A(line 2),F'len 2'   Address and length of
                                  file line 2
         DS  A(line 3),F'len 3'   Address and length of
                                  file line 3
         .
         .
         .
         DS  A(line n),F'len n'   Address and length of
*                                 file line n
 PGMEND  DS  0H

*   The above fields are not checked by the interpreter, but they
*   are used in error messages and in the predefined variables
*   &FILENAME, &FILETYPE, and &FILEMODE.  If they contain embedded
*   blanks, the results are unpredictable.
```

4. Using the EXEC 2 interpreter as a macro processor.

The use of EXEC 2 programs as macros or command files for user specified command processors requires functions provided by the CMS SUBCOM function.

The following paragraphs describe how to use SUBCOM and the EXEC 2 interpreter to implement a macro facility.

Issue SUBCOM SVC 202 to set up an entry point in the command processor. (For information on how to do this, refer to VM/SP System Programmer's Guide under SVC 202 and SUBCOM/DYNAMIC LINKAGE.)

Call EXEC 2 as in example 3 above. The filetype from the file descriptor block becomes the default &PRESUME &SUBCOMMAND environment except when it is blank, in which case the default filetype is EXEC, and the default presumption is &PRESUME &COMMAND.

When subcommands are encountered in the macro, the EXEC 2 interpreter will call the entry point specified in the SUBCOM call. This entry point may then take whatever action is necessary with the command.

Upon return, the EXEC 2 interpreter continues with the next statement or command.

When the EXEC 2 file terminates, control is returned to the initiating program at the calling point.

Executing XEDIT Macros in EXEC 2

The basic subcommand language of the XEDIT editor can be extended by writing macros that are executed by the EXEC 2 interpreter.

These XEDIT macros are CMS files with the filetype of XEDIT.

When the EXEC 2 interpreter encounters an XEDIT subcommand, it sends the command to XEDIT for execution.

XEDIT processes the command and returns to the EXEC 2 file with a return code. The EXEC 2 file then continues execution with the next statement or command. When the EXEC 2 file completes, control returns to XEDIT.

See VM/SP System Product Editor Command and Macro Reference for further information on XEDIT macros.

EXECCOMM - Sharing EXEC 2 Variables with Assembler Language Programs

EXEC 2 permits programs called from an EXEC 2 file to access all EXEC variables used within that EXEC file. Variables accessed in this manner are called "shared variables." The EXECCOMM facility of EXEC 2 provides this variable sharing environment. Using the "FETCH" and "STORE" functions of EXECCOMM, programs can directly access and manipulate EXEC 2 variables. Also, the execution of commands or subcommands can result in assignments to some of these variables as a side-effect of their execution. It is also possible to create new variables in the called program.

When variables are stored by a program, their names are checked for validity, but no substitution is carried out by EXEC 2. In other words, names passed through EXECCOMM are taken exactly as is, and embedded ampersands (&) do not cause multiple substitution.

Variables are identified by an "external name," which is the same as their "internal name," but without the leading ampersand. For example, to "fetch" a value contained in the internal variable "&VALUE," a program should use the external name "VALUE."

The facility works as follows:

When EXEC 2 starts to interpret a new EXEC or XEDIT macro, it first sets up a subcommand entry point called EXECCOMM. When a program (command or subcommand) is called by EXEC 2, it may in turn use the current EXECCOMM entry point to Store or Fetch variable values.

To access variables, the EXECCOMM entry point is invoked using both the normal and the extended Plist (see below, also see the VM/SP System Programmer's Guide). SVC 202 should be issued with register 1 pointing to the normal Plist and the top flag byte of register 1 set to X'02'.

On return from the SVC, register 15 contains a summary return code for the entire Plist. The possible return codes are:

Return Code	Meaning
0 or positive	Entire Plist was processed. Register 15 is the composite OR-ing of the SHVRET flags (see below).
-1	Invalid entry conditions.
-2	Insufficient storage was available for the requested operation. Processing was terminated.
-3 from SUBCOM	No EXECCOMM entry point found (i.e. not called from inside a EXEC 2 Exec).

The register 1 Plist: Register 1 should point to a Plist which consists of the eight character string "EXECCOMM".

The register 0 Plist: Register 0 should point to the SUBCOM Plist. The first word of the SUBCOM plist should also point to the word "EXECCOMM." No argument string should be given, so the second and third words should be the same (e.g. point to the same address or both 0). The fourth word of the Plist should point to the first of a chain of one or more request blocks.

The call is made via CMS supervisor call SVC 202, with the Plist registers set up as follows:

```
         R0 = A(NPLIST)           (see below)
         R1 = A(CL8'EXECCOMM')    high-order byte = X'02'
```

where:

```
    NPLIST DS   0F                subcommand Plist
           DC   A(CL8'EXECCOMM')  same as register 1, but with 0 in the
*                                  high-order byte
           DC   A(ARGS)           null argument string
           DC   A(ARGS)           end address of null argument string
           DC   A(SHRLIST)        pointer to first variable access
*                                  request block
```

The request block: Each request block in the chain must be laid out as follows:

```
*********************************************************************
* SHVBLOCK:  Layout of shared-variable Plist element.               *
*********************************************************************
*
SHRLIST  DS    0F          Variable Access Request Block
SHVNEXT  DS    A           Chain pointer (0 if last block)
SHVUSER  DS    F           Not used, available for private use
SHVCODE  DS    CL1         Individual function code
SHVRET   DS    XL1         Individual return code flag
         DS    H'0'        Not used, should be zero
SHVBUFL  DS    F           Length of 'FETCH' value buffer
SHVNAMA  DS    A           Address of external variable name
SHVNAML  DS    F           Length of external variable name
SHVVALA  DS    A           Address of value buffer (0 = 'none')
SHVVALL  DS    F           Length of value (set by 'FETCH')
   .     .     .             .    .    .     .  .     .
   .     .     .             .    .    .     .  .     .
   .     .     .             .    .    .     .  .     .
*
*         Function Codes (SHVCODE)
*
SHVFETCH EQU   C'F'   FETCH - Copy value to caller's area
SHVSTORE EQU   C'S'   STORE - Store from value supplied by caller
*
*         Return Code Flags (SHVRET)
*
SHVCLEAN EQU   X'00'  (Decimal 0)   Execution was OK
SHVTRUNC EQU   X'04'  (Decimal 4)   Truncation occurred during 'FETCH'
SHVBADN  EQU   X'08'  (Decimal 8)   Invalid variable name (e.g. too long)
SHVBADV  EQU   X'10'  (Decimal 16)  Value too long  - "STORE" not
*                                    performed
SHVBADF  EQU   X'80'  (Decimal 128) Invalid function code (SHVCODE)
*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

A typical calling sequence for the EXECCOMM interface might be:

```
         .     .
         .     .
         .     .
         LA    R0,NPLIST              Subcom Plist as shown
         LA    R1,=CL8'EXECCOMM'      Name of Subcom entry point
         ICM   R1,B'1000',=X'02'      Insert 'subcommand call' flag
         SVC   202                    Issue SVC
         DC    AL4(1)                 Sequential return
         LTR   R15,R15                Check for a negative return code
         BM    DISASTER               If yes, quit
*        Execution was okay
         .     .
         .     .
         .     .
```

The specific actions for each function code are as follows:

S Store variable. SHVNAMA contains the address of the external variable name, and SHVNAML contains the length of this name. SHVVALA contains the address of the buffer where the "value" of SHVNAMA is stored, and SHVVALL contains the length of the "value." The external name (SHVNAMA) is checked (e.g. length limitations), and the corresponding internal variable (same name as the external name, only with a leading ampersand (&)) is set to the value of the external variable. If a "STORE" reference is made to an unset EXEC 2 argument (i.e. a variable of the form &i where "i" is an unsigned number without leading zeros that exceeds the number of EXEC 2 arguments that are currently stored), no assignment is performed. The SHVBADN bit is set to X'08' (INVALID NAME).

F Fetch variable. SHVNAMA contains the address of the external variable name, which is the same as the internal variable name that you want to fetch, but without the leading ampersand (&). SHVNAML contains the length of this external name. SHVVALA contains the address of a buffer where the fetched variable value will be copied, and SHVBUFL contains the length of the buffer. The external variable name (SHVNAMA) is checked (e.g. length limitations), and the internal variable is located and copied into the buffer. The total length of the fetched variable is placed in SHVVALL, and if the fetched value was truncated because the buffer was not big enough, the SHVTRUNC bit is set to X'04'. If the referenced variable is shorter than the length of the buffer, no padding is done.

If there is insufficient storage (return code -2), some of the SHRLIST elements may not have been processed. These elements (including the SHVRET field) are left unchanged.

Note: The value returned by a FETCH operation is a snapshot of the internal variable at the time the operation is done. The returned value is therefore unaffected by subsequent STORE operations to the same internal variable (even within the same list).

Appendix D: EXEC 2 Primer for New Users

The function of a command programming language such as EXEC 2 is to improve the effectiveness of a programming system by matching the available commands to the particular needs and applications of each user. As a CMS user, you probably have observed that some commands were needed more frequently than others. Some of the commands you used were short and easy to type while others involved several arguments and were more difficult to issue. There may have been instances when you had to look up the correct command format or issue several commands in succession to perform an operation which would be more convenient if it were done by only one command. Command procedures, written using the EXEC 2 language, can adapt existing commands to user needs by storing commands that are issued frequently, and in the sequence that you wish them executed, in a disk file. Within this file, the validation of arguments can be checked and default values can be supplied. (A default value is a specific value assumed when an argument has not been explicitly specified. Usually, default values are chosen to be the most frequently used argument values, so that the convenience of not having to write that particular value is realized as many times as possible.) The name of the file containing these commands becomes a new command name, and hence, a new CMS command. The format of this new command can be tailored to the individuals needs.

To illustrate this, assume you have the files listed in the first column of the following table, and wish to rename them as indicated in the second column:

Current Name	Desired Name
X MEMO	NEW MEMO
NEW MEMO	OLD MEMO
OLD MEMO	(erased)

The commands used to perform this operation are straightforward, though they are a bit lengthy because two of the three fileids must be repeated and filemodes are required for the RENAME commands:

```
ERASE OLD MEMO
RENAME NEW MEMO A OLD MEMO A
RENAME X MEMO A NEW MEMO A
```

EXEC 2 makes it easy for the user to issue a sequence of commands by typing only a single command line. This is achieved by storing the desired commands in a disk file, and invoking the stored commands by typing the file's name as the command name. Such files of stored commands must have a filetype of EXEC. Note that other filetypes are possible, but they cannot be called directly by a command that you type at your terminal; they can be invoked from a program, such as a text editor. When CMS reads a command typed by the user, it searches for a

disk file having the same filename as the typed command name and a filetype of EXEC. If such a file is found, the EXEC interpreter interprets the command statements read from the disk file.

If we used a text editor to create the following file named RIPPLE EXEC:

```
&TRACE ON
ERASE OLD MEMO
RENAME NEW MEMO A OLD MEMO A
RENAME X MEMO A NEW MEMO A
```

we could command the renaming of the files described above by typing the line:

```
RIPPLE
```

The first line of the RIPPLE EXEC file is an EXEC 2 control statement. Such statements affect the operation of the EXEC interpreter instead of performing some operation in the CMS environment. The &TRACE ON statement tells the EXEC interpreter to display on your console any commands that it issues before they are executed. A &TRACE OFF statement would suppress this display of executed commands. A &TRACE ALL statement would display EXEC control statements as well as commands that are executed.

In the CMS environment, where the EXEC 2 interpreter coexists with the CMS EXEC interpreter, a second purpose is served by the &TRACE statement. Whenever an EXEC file is to be interpreted, the first record of the file is read and scanned to see if the first word is &TRACE. If it is, the file is deemed to be an EXEC 2 file; otherwise, the CMS EXEC interpreter is used to interpret that file.

EXEC control statements make it possible to conditionally interpret statements in an EXEC file, to repeat the interpretation of statements, and to control the working of the EXEC interpreter in various ways. They make it possible to write EXEC files which perform different operations depending on the arguments entered on the EXEC command line or the results of commands issued from the EXEC file. This is a very important concept, for it is this ability to modify the commands issued from an EXEC file (and the order in which they are issued) which underlies the most useful features of EXEC files.

Commands, Return Codes, and EXEC Variables

Every command executed in CMS issues a return code indicating the success or failure of the operation requested. This return code is a numeric value that is passed back to the caller of the command. If a command is issued from an EXEC file, the return code generated by that command can be examined and used to control the subsequent interpretation of statements in the EXEC file. For example, the ERASE

command displayed above in RIPPLE EXEC will yield a return code of: 0 (zero) if it succeeds in erasing a file, 28 if the file to be erased does not exist, 36 if the file exists but is on a read-only disk, and other values for less common conditions.

A command's return code is saved by the EXEC 2 interpreter as the value of the EXEC variable &RC. EXEC variables are symbols that are used to refer to values that may change during the interpretation of an EXEC file. You can use the symbol &RC in an EXEC statement to refer to the return code generated by the most recent command issued from the EXEC file. One way the &RC variable might be used in the RIPPLE EXEC file is to force termination of the EXEC file (before renaming any files) if the X MEMO file does not exist. To do this, the CMS command STATE is used to determine whether X MEMO exists on the A-disk. STATE generates a return code of 0 if the designated file exists, or a return code greater than 0 if it does not.

```
&TRACE OFF
STATE X MEMO A
&IF &RC > 0 &EXIT 1
ERASE OLD MEMO
RENAME NEW MEMO A OLD MEMO A
RENAME X MEMO A NEW MEMO A
```

The third statement in this file (&IF ...) tests the return code from STATE, and uses the &EXIT control statement to force immediate termination of the EXEC file if the value of &RC is greater than zero. Like CMS commands and user programs, EXEC files also generate return codes. If an EXEC file terminates because an end-of-file is reached and there are no more statements to interpret, the return code will be zero. However, various errors detected by the EXEC 2 interpreter (invalid EXEC control word, nonexistent file, and so on) will cause termination with a return code greater than 10000. Or, you may write the &EXIT control statement to terminate the EXEC file with a specific return code, as shown above.

The ampersand character is used at the beginning of a word to signal the EXEC interpreter that this word is an EXEC variable or an EXEC control word. When the EXEC 2 interpreter processes a statement from an EXEC file, it begins by examining each word and replacing any EXEC variables with their current values. (Later, we'll see exactly how this is done.) EXEC control words are like EXEC variables, except their values are initialized to their names by the EXEC interpreter (that is, the value of &TRACE is &TRACE, the value of &IF is &IF, etc.).

&RC is one of a group of variables that are handled in a special manner by the EXEC interpreter. They are called "predefined variables" because the EXEC interpreter assigns values to them automatically. Some of these predefined variables are given values only once, when the EXEC interpreter starts processing a file (&FILENAME is such a variable, whose value is the name of the EXEC file being processed). Other predefined variables are assigned values whenever some specific action occurs. Examples are &RC, which is set to the return code value whenever a command is issued, and &N, which is initially set to the

number of arguments present on the EXEC command line and is updated when an EXEC control statement redefines the set of argument variables.

EXEC File Arguments

The EXEC variables &1 &2 &3 ... are used to refer to the arguments in the EXEC command invoking the file. The value of &1 is the first word following the name of the EXEC file in the command line, &2 is the second word, etc. If you refer to an argument that was not present in the command line (such as &1, if no operands were written), its value will be null, and that word will disappear from any statement in which it is used. The same is true for a reference to any other EXEC variable that has not been assigned a value, or has been explicitly assigned the null value.

We will now modify the RIPPLE EXEC again so that it accepts the name of any MEMO file as an argument instead of always using the file X MEMO:

```
&TRACE OFF
STATE &1 MEMO A
&IF &RC > 0 &EXIT &RC
ERASE OLD MEMO
RENAME NEW MEMO A OLD MEMO A
RENAME &1 MEMO A NEW MEMO A
```

Here the return code from STATE is used as the return code from the RIPPLE EXEC file. A nonzero value indicates failure of the RIPPLE command, and provides a little more information than simply returning a value of 1. (Refer to VM/SP CMS Command and Macro Reference for the Responses, and the Error Messages and Return Codes issued by CMS for the STATE command.)

With this RIPPLE EXEC file, we could have any number of current, or working, MEMO files, each with a different filename. Whenever we wish to rename one of them (RWR MEMO, for example) we could use the command:

```
RIPPLE RWR
```

to rename it, making the original filename available to be used again. There will always be copies of the last two files renamed, in case a need arose to use one of them again. Files more than two iterations old are automatically erased.

There is no limit (other than disk capacity) to the number of files that can be kept. By adding more RENAME commands to the EXEC file, we can keep as many old files as we desire. By using some additional EXEC control statements, we could rename any number of files using only one RENAME statement, interpreting it as many times as necessary, each time with different arguments.

EXEC Variable Names

EXEC variables and EXEC control words always start with an ampersand. The ampersand may be followed by any other characters, up to a maximum length of 256 characters (including the initial ampersand). This is the maximum length allowed for any word; it is also the maximum length allowed for any line in an EXEC 2 file.

The characters ampersand and blank have special meanings, and cannot be made part of a variable name by simply writing them as part of a word. A blank denotes the end of a word, so it can not be included as part of the word. An ampersand denotes the beginning of an EXEC variable name. That name (including the ampersand) is replaced with the value of the variable when the word containing it is evaluated during statement interpretation. Value substitution for variable names makes it possible to put blanks or ampersands (or any other characters) into names, but it's principal benefit is the ability to manipulate an indefinite number of variables by modifying the words in a few statements instead of writing all of the variable names explicitly.

Conditional Interpretation of Statements

Before looking at more sample EXEC files, we will examine the structure of the conditional (&IF) statement more closely and introduce some other EXEC control statements. The &IF statement is actually a compound statement. The first part defines a condition; the second part may be any executable statement, which is interpreted only when the condition is true. (An executable statement is any statement except a comment. Comment statements have an asterisk as their first nonblank character, and are ignored by the EXEC interpreter.) The complete &IF statement has the format:

```
&IF  word1  comparator  word2  statement
```

where "comparator" is =, ¬=, >, <, >=, or <=. The comparison is performed numerically if both word1 and word2 are numeric data items; it is performed on a character basis if either is not numeric. Thus, "&IF 2 = +2" is true and "&IF 000 = 0" is true, but "&IF 1. = 1" and "&IF +A = 10" are false. A numeric data item consists of decimal digits, optionally preceded by a plus or minus sign. EXEC 2 does not support fractional numbers as numeric data.

The "statement" part of an &IF statement may be another &IF statement. Therefore, several conditions may be written in one conditional statement, with the last "statement" interpreted only when all of the conditions are true. Thus,

```
&IF &1 = A &IF &2 = B &EXIT
```

will terminate an EXEC file only if both conditions are true.

Statement Labels

You may attach a label to an EXEC statement (including the null statement, which has no words in it) so that an EXEC control statement can reference the labeled statement. The label must be the first word of the statement, and it must start with a hyphen. EXEC 2 does not consider a label to be part of a statement, so it is not inspected for EXEC variables. References to labels, however, may involve EXEC variables. The most frequent references to statement labels are &GOTO control statements, which modify the regular, sequential processing of an EXEC file. A typical &GOTO statement is:

```
&GOTO -END
```

which means continue interpretation of statements with the next statement having the label -END.

When a &GOTO statement is interpreted, EXEC 2 searches for the specified label by reading successive statements from the disk file and examining the first word of each statement to determine if it is the desired label. If it is, sequential interpretation of statements resumes with that statement. If the end of the disk file is encountered without finding the specified label, EXEC 2 continues to read statements, starting at the beginning of the file, until either the desired label is found, or all statements before the one being interpreted have been examined.

Assignment Statements

The EXEC 2 assignment statement is a special case, in that it is recognized when the second word of the statement (not counting a label) is an equal sign and the first word starts with an ampersand. (This is a simplification of the actual rule, which is discussed in Note 4 of "Part 3. Notes on EXEC 2.") The function of the assignment statement is to make the EXEC variable specified by the first word have the value specified by the expression following the equal sign. Thus,

```
&OPTION = GESUNDHEIT
```

assigns the value GESUNDHEIT to the EXEC variable &OPTION.

```
&ITEM = &ITEM + 2
```

increments the value of &ITEM by 2, assuming the value of &ITEM was numeric to start with (if it was not numeric, EXEC 2 considers it an error and terminates interpretation of the EXEC file). The following statement:

```
&L = &LENGTH OF &OPTION
```

uses the predefined function &LENGTH OF to compute the number of characters in the value of the variable &OPTION; that number is then assigned as the value of the variable &L. If &OPTION has the value GESUNDHEIT, then &L would be assigned the value 10. The right side of an expression in an assignment statement is the only place to use a predefined (or user-defined) function in EXEC 2. There are several predefined functions used in the EXEC files discussed later.

It is possible to set a variable to the null value by using an assignment statement:

```
&NOTHING =
```

and it is possible, of course, to have labels on assignment statements:

```
-SETONE &ONE = 1
```

EXEC Variable Evaluation

It is time to explain in detail how EXEC 2 examines a word for variable names and replaces them with values. Inspection for EXEC variables is performed by examining the characters in a word from right to left. Whenever an ampersand is detected, the ampersand and all characters to the right of it are taken as the name of an EXEC variable, which is then replaced by that variable's current value. After a value has replaced a variable name in a word, the inspection process resumes with the next character to the left, so it is possible to use EXEC variables to build the names of other EXEC variables.

To illustrate, if &X = 1 and &1 = FIRST, the word &&X means &1, which is replaced by the value FIRST. Suppose the value of &1 is an ampersand instead of FIRST; then, &&X ==> &1 ==> &, but no further substitution occurs, since there are no more characters of the original word to be inspected.

In the case of an assignment statement, the inspection of the first word for ampersands is stopped just before the first character has been tested (remember that characters are examined from right to left). Therefore, that word retains its initial ampersand and remains an appropriate name for an EXEC variable. Retention of the initial ampersand of a word also occurs in other contexts where a variable name is required (the &READ VARS and &UPPER VARS statements, for example).

Recall that there are no undefined EXEC variables. If an EXEC variable has no default or explicitly assigned value, its value is taken to be null (the character string that has no characters in it, and whose length is zero).

An Example of Generating EXEC Variable Names

We are now ready to look at an EXEC file that depends on this ability to use an EXEC variable to build the names of other variables. LFN EXEC uses the CMS command LISTFILE to display information about all of the files on all accessed disks that have the filenames (arguments) specified on the command line invoking the EXEC file. Because the number of filename arguments may differ from one use to the next, the EXEC variable &J is used to select the next argument to use in the LISTFILE command.

```
&TRACE
&J = 1
-LOOP LISTFILE &&J * * (LABEL
&J = &J + 1
&IF &J <= &N &GOTO -LOOP
```

Suppose this EXEC file were invoked by the command

```
LFN NEW OLD
```

The first time the LISTFILE command is issued, the EXEC variable &J will have the value 1, so &&J ==> &1 ==> NEW and the command passed to CMS is

```
LISTFILE NEW * * (LABEL
```

After the first LISTFILE command, the value of &J is incremented from 1 to 2, and the &IF statement is interpreted. Since there are two argument words, NEW and OLD, the value of &N is 2, the condition part of the &IF control statement is true, and the &GOTO statement is executed. Interpretation of EXEC statements continues with the LISTFILE statement again, but this time &&J ==> &2 ==> OLD and the command issued is

```
LISTFILE OLD * * (LABEL
```

After &J is incremented to 3, the &IF condition is false, so the &GOTO statement is not interpreted and the EXEC file terminates with a return code of zero.

If more than one of the specified filenames is found on a disk, the output generated by this EXEC is not as pretty as it could be. This is because the LISTFILE command produces a title line each time it is invoked and finds at least one file meeting its argument pattern. The following elaboration of LFN EXEC uses the return code generated by the LISTFILE command to detect when the title line is first displayed and uses the NOHEADER option in subsequent LISTFILE commands to prevent duplicate title lines from being displayed.

```
&TRACE
&J = 1
-LOOP LISTFILE &&J * * (LABEL &NOHEADER
&IF &RC = 0 &NOHEADER = NOHEADER
&J = &J + 1
&IF &J <= &N &GOTO -LOOP
```

Here, we take advantage of the fact that the initial value of &NOHEADER is null, so that word disappears the first time the LISTFILE command is interpreted. When the command is successful (that is, it produces a return code of zero), the EXEC variable &NOHEADER is given the value NOHEADER, and all subsequent LISTFILE commands have the NOHEADER option following the LABEL option.

The &LOOP Control Statement

There is another way of writing this EXEC file. That is by using the &LOOP control statement, which is more efficient because it eliminates the need for repetitively interpreting the &IF statement and searching the file for the label -LOOP:

```
&TRACE
&J = 1
&LOOP 3 &N
  LISTFILE &&J * * (LABEL &NOHEADER
  &IF &RC = 0 &NOHEADER = NOHEADER
  &J = &J + 1
```

The &LOOP statement can take several forms. Here, it specifies that the three lines following the &LOOP statement are to be repeated &N times; that is, for as many times as there are arguments to the EXEC file. The statements to be repeated (the scope of the loop) were indented to make it easier to read the EXEC file. It is often more convenient to use a label reference in a &LOOP statement instead of an absolute count of the number of statements to be repeated. In this case, the label is written in place of the count and the EXEC interpreter determines how many statements to repeat:

```
&TRACE
&J = 1
&LOOP -X &N
  LISTFILE &&J * * (LABEL &NOHEADER
  &IF &RC = 0 &NOHEADER = NOHEADER
  -X &J = &J + 1
```

The label defining the scope of the loop must occur before the end of the EXEC file or an error is reported. It is valid to have a loop count of zero, in which case no statements within the loop are interpreted.

This would happen in the above EXEC if it were invoked with no arguments.

A loop statement that defines its scope through the use of a label reference is more resistant to errors introduced because of a change than a loop statement that specifies an absolute number of lines. The label reference avoids a common error: forgetting to update the line count in a &LOOP statement when a change is made that alters the number of statements within the scope of the loop.

Making EXEC Files Interact with Users

One of the problems accompanying the writing of EXEC files to extend the user's command set is that it becomes more difficult to remember the correct formats for invoking this larger set of commands. A very useful technique is to make the EXEC files self-documenting; that is, whenever they are invoked with incorrect arguments, or with a question mark as an argument, they display a description of the correct command format and whatever additional description the writer deems appropriate. Such additional information might be a description of what the file does and how to use it, or perhaps a reference to a MEMO file or a publication containing more information. Here is a version of LFN EXEC with such a feature:

```
&TRACE
&IF &N = 0 &GOTO -TELL
&IF &N = 1 &IF &1 = ? &GOTO -TELL
&J = 1
&LOOP -X &N
  LISTFILE &&J * * (LABEL &NOHEADER
  &IF &RC = 0 &NOHEADER = NOHEADER
  -X &J = &J + 1
&IF /&NOHEADER = / &EXIT 28
&EXIT
-TELL &PRINT Format is:  &FILENAME fn1 fn2 ...
&PRINT Uses LISTFILE to display information about
&PRINT all files with filenames fn1, fn2, etc.
&EXIT 100
```

The &PRINT control statement directs the EXEC interpreter to perform its usual replacement of EXEC variables with values, then to display the words following &PRINT as a line on the user's console.

The above version of LFN EXEC generates a nonzero return code in any instance where no files were found. Since the EXEC variable &NOHEADER was already being used to detect a successful invocation of LISTFILE, an appropriate test after all the LISTFILE commands have been issued is to return a nonzero code whenever the value of &NOHEADER is null. It is not possible to simply write

```
&IF &NOHEADER ¬= NOHEADER &EXIT 28
```

In the case where &NOHEADER is null, this would cause a syntax error in the &IF statement because the &NOHEADER word would disappear and we would be left with

```
&IF ¬= NOHEADER &EXIT 28
```

A solution for testing the value of an EXEC variable that might be null is to use some prefix on both the variable and the value compared with it. In the case of LFN EXEC, the slash is that prefix, and the two statements which can result after substituting for the variable &NOHEADER are:

```
&IF /NOHEADER = / &EXIT 28

&IF / = / &EXIT 28
```

for success (&NOHEADER = NOHEADER) or failure (&NOHEADER is null), in that order.

All of the previous EXEC files have used only the arguments provided on the command line to determine what function they would perform. You can also write an EXEC file that interacts with the user, displaying prompting messages on the console and reading instructions or values which are typed.

When CMS or CP reads a command line, it translates the command line into uppercase before interpreting it. When a program, such as the EXEC interpreter, reads a console input line, it chooses whether or not to translate to uppercase. The EXEC control statement

```
&CASE M
```

instructs the EXEC interpreter to read subsequent input lines in mixed case (uppercase and lowercase combined) while

```
&CASE U
```

requests translation into upper case. &CASE U is the initial setting when the EXEC interpreter starts processing an EXEC file.

Data is read from the console using the &READ control statement. A &READ statement may read one input line and assign it as the value of a single EXEC variable:

```
&READ STRING &S
```

&S will contain the entire text of the input line, including all blanks. Alternatively, the input line can be separated into words and each word assigned to an EXEC variable.

```
&READ VARS &FIRST &SECOND &THIRD &FOURTH
```

If there are more variables than words in the input line, those variables remaining after all words have been used are assigned the null value. If there are more words than variables, the extra words are ignored. If you don't know how many words will be on an input line, it is often convenient to use the statement:

```
&READ ARGS
```

which redefines the EXEC argument variables &1 &2 &3 ... etc., and assigns to &N the number of words (arguments) in the input line. All of the prior values for &1 &2 ... etc. are lost when this is done. So, remember to assign any EXEC argument variables that will be needed later to other EXEC variables before interpreting a &READ ARGS statement. The predefined variable &ARGSTRING is not affected by a &READ ARGS statement. Its value continues to be the original argument string passed to the EXEC file, or whatever value the user last gave it in an assignment statement.

It is possible to read lines from the console and interpret them as EXEC statements using the form:

```
&READ n
```

where "n" is the number of lines to read. If no explicit number of lines is given, only one line will be read. An asterisk (*) may be used in place of a number to denote that statements are to be read from the console until a statement which modifies sequential processing of lines is interpreted (&EXIT, &GOTO, &SKIP, etc.). It is easy to test the effect of various EXEC statements by using the file:

```
&TRACE ALL
&READ *
```

which reads statements from your console and traces their interpretation.

The next example is CPM EXEC, which enhances the CP MSG command in two respects. First, it transmits multiline messages to one or to a group of VM users (the same message is sent to all of the specified users). Second, it transmits messages in uppercase and lowercase.

```
&TRACE
&CASE M
&IF &N = 0 &GOTO -TELL
&IF &1 = ? &GOTO -TELL
&READ STRING &
&LOOP -E UNTIL /& = /
  &X = 1
  &LOOP -EE &N
    CP MSG &&X &
    -EE &X = &X + 1
  -E &READ STRING &
&EXIT
-TELL &BEGPRINT -X
Format is:  CPM user1 <user2 ...>
CP 'MSG' console function is used to send lines to the
specified users.  Enter blank or null line to end.
-X
&EXIT 101
```

This EXEC uses two loops, one nested inside the other. The scope of both loops is defined by labels. The inner loop containing the statements

```
CP MSG &&X &
-EE &X = &X + 1
```

is similar to the loop in LFN EXEC; that is, it is interpreted once for each argument. The outer loop uses a condition like an &IF statement to determine when repetition of the loop will end. In this case, we wish to repeat the loop until the user enters a null or blank line from his console. The EXEC variable & (that is the shortest possible name for an EXEC variable) contains the string read from the console, and the word "UNTIL" identifies the nature of the condition being tested; that is, repeat the loop until the condition is true. Because the value of & may be null, we use the prefix technique discussed before to avoid a null value from destroying the syntax of our &LOOP statement. We could also have written the first &LOOP statement like this:

```
&LOOP -E WHILE /& ¬= /
```

which repeats while the condition is true (that is, until the condition is false). The choice between these two forms is purely a personal matter of what the EXEC writer sees as easier to write or understand.

At label -TELL in this EXEC file, we see an example of a statement which, unlike all those seen before, requires more than one line in the EXEC file. This, and the &BEGSTACK statement that has a similar syntax, are the only statements that can use more than one line of an EXEC file. The lines following &BEGPRINT are not examined for EXEC variables. The EXEC interpreter prints each line exactly as it is read from the disk file, until either the end-of-file occurs, or a line is reached that contains only the label specified in the &BEGPRINT statement.

EXEC 2 Implementation of Editor Macros

CMS commands are not the only commands that may be executed from an EXEC 2 file. An important application of EXEC 2 is the creation of editor macros; that is, procedures that issue commands to an editor instead of, or in addition to, the regular CMS command interpreter. The benefits of such procedures are the same as for EXEC 2 files containing normal CMS commands; the avoidance of repetitive keying of commands by the user, and the ability to build new commands that are specially adapted to particular applications, making the user-to-application interface more efficient.

XEDIT, the System Product Editor, causes CMS to establish an interface through which EXEC 2, or another program, can issue editing commands. Commands issued through this interface are called "XEDIT subcommands" because they are not interpreted by CMS in its regular command environment, but instead are delivered to XEDIT, which interprets them in its own environment.

When XEDIT receives a command that it does not recognize as one of its basic editing commands, it determines (using the CMS command STATE) whether there is an EXEC file (an edit macro) that implements this command. If there is, the editor invokes the EXEC 2 interpreter to interpret that file.

The EXEC file author has several means to designate whether a command in his file is for a particular subcommand environment or for CMS. The most explicit of these is to use the &SUBCOMMAND control statement. Thus, an edit macro that has to issue a NEXT 8 command to XEDIT could use the statement:

```
&SUBCOMMAND XEDIT NEXT 8
```

Since most edit macros contain many subcommands, it can be convenient to tell the EXEC 2 interpreter to send all commands to the XEDIT environment by executing the statement:

```
&PRESUME &SUBCOMMAND XEDIT
```

All command statements interpreted after the above statement are presumed to be subcommands for the XEDIT environment, and are treated as though they were prefaced by &SUBCOMMAND XEDIT. This presumption does not affect commands explicitly directed to another environment by the &SUBCOMMAND statement, so

```
&SUBCOMMAND ZOO ELEPHANT 3
```

would continue to send the command ELEPHANT 3 to the ZOO environment, irrespective of a &SUBCOMMAND XEDIT presumption.

How, then, may a normal CMS command be executed if all commands are presumed to be subcommands? By using the &COMMAND control statement,

which always treats the accompanying command as a regular CMS command. For example:

```
&COMMAND STATE &1 MEMO *
```

will always be interpreted as a CMS command. A &SUBCOMMAND presumption may be reset, and regular CMS command processing resumed by executing the statement:

```
&PRESUME &COMMAND
```

At the beginning of this primer, it was noted that a program could call the EXEC 2 interpreter to interpret a file that had a filetype other than EXEC. The System Product Editor (XEDIT) does this when it encounters an edit macro. This macro must have the filetype XEDIT instead of EXEC. When EXEC 2 interprets a file not having a filetype of EXEC, it starts with a &SUBCOMMAND presumption of the filetype. Thus, there is no need to preface XEDIT subcommands in an XEDIT macro with &SUBCOMMAND XEDIT, unless the default &SUBCOMMAND presumption has been explicitly changed. It is necessary, however, to preface regular CMS commands with &COMMAND if they are not to be passed to XEDIT. XEDIT macros do not require an initial &TRACE statement to indicate that they should be interpreted by the EXEC 2 interpreter because that is indicated by the way in which XEDIT invokes the EXEC 2 program.

To illustrate just how simple an edit macro can be, consider the case where it is desired to replace lines that currently contain:

```
.SK 3
```

with the three lines:

```
.SK
.CE ----------
.SK
```

This can be done using the XEDIT commands:

```
FIND .SK 3
REPLACE .SK
INPUT .CE ----------
INPUT .SK
```

If those commands are put into a file named REPSK XEDIT, they may be executed by simply entering the command

```
REPSK
```

in the XEDIT environment. Of course, this only affects the next occurrence of the ".SK 3" line. All occurrences could be changed by writing a loop in the edit macro:

```
FIND .SK 3
&LOOP 4 UNTIL &RC ¬= 0
  REPLACE .SK
  INPUT .CE ----------
  INPUT .SK
  FIND .SK 3
```

Note that here take advantage of the fact that XEDIT subcommands generate return codes indicating their success or failure much like regular CMS commands. In this example, the FIND command generates a return code of zero if it succeeds in finding the specified text, and a return code of one if it fails.

The above example contains all uppercase data, but it may be necessary to process mixed case data in edit macros. EXEC 2 statements may be written in whatever case you desire, but control words such as &LOOP and predefined variables such as &RC must be in uppercase. Variables to which you assign values, such as &X or &ZILCH, may be written in uppercase or lowercase, but remember that &ZILCH and &zilch are two distinct variables. Likewise, &LOOP is an EXEC 2 control word, but &loop is a variable. You can use variables such as &JuGGerNauT if you like pressing the shift key.

Suppose we want to use the REPSK XEDIT file for lines starting with .SK 2, or .SK 3, or .sp 3, etc. We can use two arguments to define the lines we are interested in finding, as follows:

```
FIND &1 &2
&LOOP 4 UNTIL &RC ¬= 0
  REPLACE .SK
  INPUT .CE ----------
  INPUT .SK
  FIND &1 &2
```

This works fine, but the question of case rises again. If the editor is operating in CASE U, it will translate input commands into uppercase before invoking an edit macro. Therefore, if a REPSK .sp 3 command is to work properly (meaning it is to look for ".sp 3", not ".SP 3"), it must be entered while XEDIT is in mixed case mode. (XEDIT allows a second argument on a CASE subcommand, indicating whether locate and find operations "RESPECT" or may "IGNORE" case when comparing characters. Using the "IGNORE" value produces a different effect than the above macro, because REPSK .sp 3 would find lines starting with any of these: ".sp 3", ".sP 3", ".Sp 3", ".SP 3".

Handling Embedded Blanks

If you wanted to find a line starting with the words ".SK" and "3" separated by two blanks, the above macro would fail. This happens because when EXEC 2 prepares a command, it builds a parameter list by concatenating all the words of the command (after variable substitution)

with a single blank between words. If a word is null (that is, it has zero characters in it), the word and its delimiting blank disappear from the command.

To handle a case having two blanks between words, we can rewrite REPSK XEDIT using the predefined variable &ARGSTRING. This variable has an initial value of the entire string of arguments passed to the EXEC file. This string does not include the command name used to invoke the EXEC file, nor the blank separating it from the argument string. It does include all blanks separating the argument words, plus any additional blanks preceding or following those words.

```
&C = &CONCAT OF FIND &BLANK &ARGSTRING
&C
&LOOP 4 UNTIL &RC ¬= 0
  REPLACE .SK
  INPUT .CE ----------
  INPUT .SK
  &C
```

The idea here is to build the edit command we want, with blanks exactly where we want them, as the value of an EXEC variable. Then, the FIND command is represented as a single word, and we avoid any difficulties stemming from the combination of several words to form a command. To build the FIND command, we use the predefined function &CONCATENATION OF, whose value is the string obtained by placing all of its argument values (after variable substitution) side by side without any intervening blanks. Since we need one blank to separate the FIND edit command from its operand, that blank is included explicitly by using the predefined variable &BLANK, whose value is a single blank character.

Actually, it really wasn't necessary to build the FIND command quite so carefully. It would work equally well using FIND &ARGSTRING, but the method displayed above is more general, and can be used to build any possible command.

An Example of EDIT and CMS Commands in One File

The next example is an XEDIT macro that assumes the user is editing a file such as CMS EXEC, produced when the CMS LISTFILE command is used with the EXEC option. Each line of this file identifies a disk file. The format of the lines are:

```
&1 &2 filename filetype filemode ...
```

and the function of this macro is either (1) to edit the file identified by the current line, or (2) if the value of the first argument is ERASE, to execute a CMS ERASE command to remove that file from disk, then to delete the identifying line from the current editor display. This function is useful for cleaning a minidisk of files that are no longer wanted, because often a file must be examined before the decision to

erase it can be made. The ability to examine and erase files without having to type their fileids can be a boon to the user who must cope with a dozen, or maybe a hundred, files.

```
&F = &1
STACK 1
&READ ARGS
&IF &N = 0 &EXIT
& = &LITERAL OF &1
&IF & = &1 &ARGS &3 &4 &5
&IF &N < 2 &EXIT
&IF /&F = /ERASE &GOTO -ERASE
XEDIT &1 &2 &3
&EXIT
-ERASE &COMMAND ERASE &1 &2 &3
DELETE
```

The first statement remembers the optional argument as the value of &F, so that later a branch can be made to -ERASE in case the argument value is ERASE. Next, the command STACK puts the current line from the file being edited into the console stack (explained in the next paragraph). This is to prepare for the following &READ ARGS statement which reads the line and separates it into words that are used to redefine the argument variables &1 &2 ... etc. The predefined variable &N, which records the current number of arguments, is also updated by the &READ ARGS statement.

The console stack may be thought of as a buffer or staging area for console input. Whenever CMS is asked to read the next input line from a terminal, it first checks to see if any lines are already available in the console stack. If there is a line in the console stack, it is used as the next input line; if there is no line in the console stack, then CMS waits until a line has been entered from the console. Programs can create their own "input" lines, and put them into the console stack. This is what the STACK edit command does. The console stack is an important facility for passing data between programs and EXEC files.

If there is nothing on the line stacked, &EXIT terminates execution of the EXEC file immediately. Otherwise, a test is made to see whether the first word was literally &1, as would be the case for a line from a CMS EXEC file produced by LISTFILE. The predefined function &LITERAL OF suppresses variable substitution in its argument string, thereby providing a means of assigning values containing ampersands and blanks to EXEC variables.

If an initial, literal &1 is present, an &ARGS statement redefines the argument values to eliminate the &1 (and the &2 assumed to follow it); therefore, what remains is the filename, filetype, and filemode. The &ARGS statement works by evaluating the words following &ARGS, clearing the current argument variables (for example, assigning the null string as their value), and assigning the value of the first non-null word to &1, the second to &2, etc. The variable &N is then assigned the new number of arguments. Note that even if there used to be more arguments

than were specified in the &ARGS statement, all the old values are cleared. The value of &ARGSTRING is not affected by an &ARGS statement.

Next in the macro, is a test to ensure that there are at least two arguments left to be used as a filename and filetype. Then we test &F and either branch to -ERASE, or issue the needed XEDIT command. Note that, if the file is to be erased, the CMS ERASE command is explicitly invoked by the &COMMAND statement. If the &COMMAND control word was not used, EXEC 2 would presume the command was for the editor and mistakenly pass it to XEDIT. After the ERASE command, a DELETE command tells XEDIT to remove the current line, which identified the file just erased, from the file being edited.

PARA -- A Complex XEDIT Macro

The next example illustrates the complex text manipulation that is possible with edit macros implemented in the EXEC 2 language. The objective is to take a specified number of lines from the current file and reformat them into a paragraph. Arguments from the command line specify the number of lines to be reformatted and the left and right hand margins between which the new paragraph is to be formed. This edit macro is particularly useful for editing MEMO files, or any files that contain text that might be presented on a display screen. Modifications to the PARA macro can adapt it for dealing with comment lines in programs written in assembly language, FORTRAN, PL/I, and even in EXEC files.

This file is rather long, so it will be analyzed in two parts. First, we will look at the part that actually does the formatting -- the kernel, so to speak. Then we will surround that kernel with statements that expand upon the basic command syntax, supplying default values for arguments that were omitted and checking to see that argument values that were written are not flawed (by typing errors, for example) in ways that could lead to a failure in the interpretation of the EXEC file. You will find that, after the kernel has been developed, about twice as many statements are used in support of that kernel than are in the kernel itself. This is not unusual for this type of file.

The format of the PARA macro is:

```
PARA nlines leftmargin rightmargin
```

where "nlines" is the number of lines in the file to be reformatted, starting with the current line. "Leftmargin" and "rightmargin" specify the columns within which the reformatted text is to be placed; that is, there will be leftmargin-1 blanks at the beginning of each new line, and no line will exceed the position defined by rightmargin, unless a single word exceeds the paragraph width.

The basic plan of the kernel is to implement two nested loops. The outer loop, interpreted as many times as there are input lines, uses the

STACK subcommand to put a line from the file being edited into the console stack. Then the outer loop uses a &READ ARGS statement to define the words in the stacked line as EXEC arguments. The inner loop is interpreted once for every word, and builds an output line by concatenating individual words and blanks onto the end of the string &S until the next word causes the length of &S to exceed the right margin. At that point, the existing &S string is put into the edit file and the next line is started with the word that did not fit into the previous string. Whenever there are no more argument words to add to &S, another iteration of the outer loop occurs until all input lines have been processed. A more detailed analysis of the file follows this listing.

```
&CASE M
&STACK LIFO &1       &2      &3
&READ VARS  &NLINES  &MLEFT  &MRIGHT
&SINIT = &LEFT OF &BLANK &MLEFT
&SINIT = &PIECE OF &SINIT 2
&S = &SINIT
&LOOP -Z &NLINES
STACK
&IF &RC ¬= 0 &GOTO -END
&READ ARGS
DELETE
&X = 1
&LOOP -Z UNTIL &X > &N
&T = &CONCAT OF &S &&X &BLANK
& = &LENGTH OF &T
&IF & < &MRIGHT &GOTO -OK
UP
INPUT &S
NEXT
&T = &CONCAT OF &SINIT &&X &BLANK
-OK &S = &T
-Z &X = &X + 1
*
-END UP
&IF /&S ¬= / INPUT &S
```

The first six lines perform initialization functions. &CASE M is set, so that subsequent lines read from the console stack will not be translated into uppercase. The values of the first three arguments are assigned to the variables &NLINES, &MLEFT, and &MRIGHT. This is so that these values will be available later, after the argument variables &1 &2 ... have been redefined to refer to words from a line in the edit file. It is possible to use three assignment statements to achieve this, namely:

```
&NLINES = &1
&MLEFT = &2
&MRIGHT = &3
```

but the same effect is realized by stacking the three values in one line in the console stack, then using a &READ VARS statement to read the stacked line and assign the words in it to the desired variables.

When a program puts a line into the console stack, it specifies whether the line is to be put "at the end of the stack," where it will be read after all of the lines already in the stack, or whether it is to be put "at the beginning of the stack," where it will be read first, when the next input line is requested. These two choices are named "First-In First-Out" and "Last-In First-Out," respectively, and are frequently designated by the initials FIFO and LIFO. In this case, where we wish to immediately use the line we stack, we explicitly designate LIFO in the &STACK statement. This avoids any possible complication from lines that may already be in the console stack.

After assigning values to the variables &NLINES, &MLEFT, and &MRIGHT, we prepare the initial value for the string &S. This value contains &MLEFT-1 blanks. The predefined function &LEFT OF (and the similar function &RIGHT OF) always generates a result containing the number of characters specified by the second argument. These characters are obtained from the word given as the first argument, adding blanks or truncating on the right according to whether that word is shorter or longer than the specified length (on the left, in the case of &RIGHT OF).

The predefined function &PIECE OF acts just as its name implies: its value is the specified piece of the word supplied as its first argument. The second argument denotes where the result piece starts (1 means start with the first character, etc.). An optional third argument may be used to define how long the result piece is to be. If it is omitted, the result contains all characters from the starting character specified by argument 2 to the end of the word. If the specified length is larger than the number of characters available, the result is the same as if the length argument were omitted.

This use of &PIECE OF serves to shorten the value of &SINIT by one character, yielding a string containing exactly the number of blanks desired. The value of the variable &S, which will eventually become a reformatted output line, is initialized to the value of &SINIT.

The statements that obtain input data to be reformatted are:

```
&LOOP -Z &NLINES
STACK
&IF &RC ¬= 0 &GOTO -END
&READ ARGS
DELETE
&X = 1
```

Since we are to start with the current line in the edit file, STACK puts it into the console stack. In case there is no current line (XEDIT is at the end of file, for example) we test the return code from STACK and branch if it is not zero, meaning no line was stacked. If the STACK command succeeds, &READ ARGS reads the stacked line and assigns the

words of the line to the variables &1 &2 ..., while also assigning the number of words (the new number of arguments) to &N. Since we now have the words from the line, a DELETE command removes the original line from the file. &X will be used as an index denoting which word (from the set of new argument words &1 &2 ...) is to be processed next, so it is assigned the value 1 each time a new line is acquired.

The statements to format a new line consist of:

```
&LOOP -Z UNTIL &X > &N
&T = &CONCAT OF &S &&X &BLANK
& = &LENGTH OF &T
&IF & < &MRIGHT &SKIP 4
  UP
  INPUT &S
  NEXT
  &T = &CONCAT OF &SINIT &&X &BLANK
&S = &T
-Z &X = &X + 1
```

Note that this inner loop also terminates with the statement labeled -Z. There is no problem here; the interpretation of the two loops can be understood by imagining there are unique labels for each loop, the first one for the inner loop, the last one for the outer loop.

This inner loop is interpreted once for each argument (zero times in the case of a blank line). The variable &X is initially 1, and is incremented by one for each subsequent iteration, so the word &&X denotes the first, second, ..., nth argument value during the first, second, ..., nth iteration of the loop. The variable &T is assigned the string containing the current contents of the output line, &S, followed by the current word, &&X, and a blank. The length of the string &T is then compared with the right margin value, &MRIGHT. If the string length is less than the margin, four lines are skipped to where &S is assigned the value of &T because we now know the current word will fit into the current output line. Finally, the statement labeled -Z increments the value of &X by one, so that the next time around, this inner loop uses the next input word.

If the length of the temporary variable &T exceeds &MRIGHT, the current word will not fit into the output line &S, so the skip does not occur. Instead, &S is inserted into the edit file above the current line, because we do not want to mistake an output line for the next input line. The NEXT command restores the current line pointer so that it points to our next input line. We assign to the variable &T the initial value of &S (the string containing &MLEFT-1 blanks), followed by the word &&X (which did not fit into the line just put into the edit file), and a single blank to separate this word from the next word. With this value assigned to &T, we can continue as we did for the case where the word fit, assigning the value of &T to the variable &S, incrementing the value of &X, and looping for the next word.

Finally, we reach the -END label. We can arrive here either because the attempt to STACK a line failed, or simply because after processing all

the lines specified, we fell through the outer loop. In either case, the only thing left to do is to back up one line, and if there are any words in the string &S, insert that final output line into the edit file.

The PARA XEDIT file discussed above performs the indicated function, but it suffers from two deficiencies which we shall now correct. The first problem manifests itself if the user incorrectly specifies an argument. For example, if the user typed U instead of 7 for the first argument, the &LOOP -Z &NLINES statement would fail because of incorrect syntax and EXEC 2 would immediately terminate interpretation of the file. The second problem is simply one of convenience. It should not be necessary to enter the paragraph margins each time the macro is used. Instead, some default value should be assumed whenever an explicit value is not used.

While we are fixing up those two problems, we will take advantage of the opportunity to incorporate two new features. If asterisk (*) is used as the first argument (the number of lines to reformat), the macro will process all lines from the current line until the next blank line or to the end of the file if no blank line is found. Since this will also be used as the default "number of lines to process," it will be possible to issue the command PARA with no arguments. This default value will be especially useful because the second feature we shall add is a test to see whether the line following the last reformatted line is blank, and, if it is, advance the current line pointer to the line following such a blank line. Then, we will be able to enter one PARA command specifying or implying "*" as the number of lines to process, reformat an entire paragraph, then use the XEDIT command "=" (which means "repeat the last command typed") to reformat the next paragraph. The "=" command may then be repeated to process as many consecutive paragraphs as desired.

Note that it isn't possible to assume a default value for one argument if an explicit value is given for a later argument. For example, if PARA EXEC is invoked to reformat the lines before the next blank line, but the left margin is to be 15, the default value for the first argument must be explicitly written so that the second argument is really the second argument and not mistaken for the first.

The second new feature supplies an option, INDENT, which allows us to request that the first line of a paragraph be indented by a desired amount.

Since the function of the PARA command has now become a bit elaborate, with default values, optional arguments, where the current line pointer ends up, etc., it would also seem like a good idea to include a TELL function in case the user can't remember the details of the command format or operation.

When all of these features have been included, our little 25-line edit macro has grown to almost 100 lines. However, the kernel described above is basically unchanged. The other features are implemented by relatively short series of statements that do not interact with one another, so they should be comprehensible.

Here is the complete, final form of the PARA XEDIT file. Following it is a detailed discussion of the its parts.

```
&IF &N = 1 &IF &1 = ? &GOTO -TELL
* Establish default values, initialize variables.
&STACK LIFO *       1      65      0        NLINES MLEFT MRIGHT 1
&READ VARS  &NLINES &MLEFT &MRIGHT &INDENT  &X1    &X2   &X3    &X
* Test arguments for valid values.
&IF &N = 0 &GOTO -DOIT
&IF &1 = * &X = 2
&LOOP -NEXTARG UNTIL &X > 3
  &IF &X > &N &GOTO -DOIT
  & = &DATATYPE OF 0&&X
  &IF & ¬= NUM &GOTO -TESTOPT
  &&X&X = &&X
  -NEXTARG &X = &X + 1
&IF &X > &N &GOTO -DOIT
*
* Test for valid option.
-TESTOPT & = &PIECE OF &&X 1 1
&IF & ¬= ( &GOTO -TELL
& = &LENGTH OF &&X
&IF & = 1 &X = &X + 1
&IF & > 1 &&X = &PIECE OF &&X 2
&IF &X > &N &GOTO -DOIT
& = &LENGTH OF &&X
& = &PIECE OF INDENT&BLANK 1 &
&IF &&X ¬= & &GOTO -TELL
&X = &X + 1
&INDENT = 5
&IF &X > &N &GOTO -DOIT
& = &DATATYPE OF &&X
&IF & ¬= NUM &GOTO -TELL
&INDENT = &&X
&IF &N > &X &GOTO -TELL
*
-DOIT &IF &NLINES ¬= * &GOTO -SKIPSEARCH
* Convert * into number of lines
TRANSFER    LENGTH  LINE
&READ VARS  &L      &CURLINE
&LOOP 4 UNTIL &L = 0
  NEXT
  &IF &RC ¬= 0 &GOTO -EOF
  TRANSFER    LENGTH
  &READ VARS  &L
-EOF TRANSFER   LINE
&READ VARS    &ELINE
&NLINES = &ELINE - &CURLINE
:&CURLINE
*
-SKIPSEARCH &CASE M
&SINIT = &LEFT OF &BLANK &MLEFT
&SINIT = &PIECE OF &SINIT 2
& = &INDENT + &MLEFT - 1
```

```
&IF & < 0 & = 0
&S = &LEFT OF &BLANK &
&ARGS
&LOOP -Z &NLINES
STACK
&IF &RC ¬= 0 &GOTO -END
&READ ARGS
DELETE
&X = 1
&LOOP -Z UNTIL &X > &N
&T = &CONCAT OF &S &&X &BLANK
& = &LENGTH OF &T
&IF & < &MRIGHT &SKIP 4
  UP
  INPUT &S
  NEXT
  &T = &CONCAT OF &SINIT &&X &BLANK
&S = &T
-Z &X = &X + 1
*
-END TRANSFER   LENGTH
&READ VARS      &L
UP
&IF /&S ¬= / INPUT &S
&IF &L = 0 NEXT 2
&EXIT
*
-TELL XEDIT PARA TELL
&BEGSTACK -X

Format is:  PARA <n <left <right>>> <(Indent <i>>
 Defaults:        *  1     65                  5

The PARA XEDIT Macro reformats the current line, and the next n-1
lines using the specified "left" and "right" margins.  Optionally,
the first line reformatted may be indented "i" spaces.  If "*" is
specified for the number of lines, lines are reformatted until a
blank line is encountered.

The reformatted text replaces the original text, and the current
line pointer is set to the last line of reformatted text, or, if
a blank line follows the last reformatted line,
the current line pointer is set to the line
following the blank line that terminated reformatting.

-X
&STACK
SET CASE M
INPUT
:8
```

----------END OF EXAMPLE----------

The XEDIT file starts with the traditional test for an information query. If there is only one argument, and if it is a question mark, go to label -TELL where a complete description of the PARA command is put into the console stack. XEDIT is commanded to edit the file PARA TELL, and the stacked lines are read into that file. Since the editor will display that file, the user will see the entire description on his display screen and may use the QUIT XEDIT command to return to the file he was editing.

If this is not simply a request for information, a set of EXEC variables are assigned initial values by the &STACK LIFO and the following &READ VARS statements. These work just as in the kernel described before, but this time there are more variables involved. The extra spaces in the commands are only to improve readability; they do not affect interpretation of the commands. Note that there are some comment statements in this file, which should make it easier for a user to locate and modify various features such as default values.

```
* Test arguments for valid values.
&IF &N = 0 &GOTO -DOIT
&IF &1 = * &X = 2
&LOOP -NEXTARG UNTIL &X > 3
  &IF &X > &N &GOTO -DOIT
  & = &DATATYPE OF 0&&X
  &IF & ¬= NUM &GOTO -TESTOPT
  &&X&X = &&X
  -NEXTARG &X = &X + 1
```

Now that we have assigned default values for the variables &NLINES, &MLEFT, &MRIGHT, and &INDENT, the above statements examine the arguments and change any of the default values to valid values given with the command. Instead of repeating the statements to test that an argument is an unsigned integer, a loop is used. The variable &X is an index to the argument value being examined and, at the same time, an index to the name of the related EXEC variable, stored as the values of &X1, &X2, and &X3. As soon as an argument value is found not to be an unsigned integer, a branch is made to -TESTOPT to check that it is a valid INDENT option.

The &DATATYPE OF predefined function returns the value NUM if its argument is an integer, or CHAR if its argument is not an integer. Since this edit macro will not accept a signed integer for any of the first three arguments (number of lines, left margin, and right margin), &DATATYPE OF 0&1 will be NUM only if &1 is an unsigned integer, etc.

Note how the assignment statement

```
&&X&X = &&X
```

works: if &X is 1, &&X denotes the value of &1, while &&X&X ==> &&X1 ==> &NLINES, the name of the EXEC variable initialized to the default value for "nlines".

Now, statements to check the validity of any MARGIN arguments:

```
* Test for valid option.
-TESTOPT & = &PIECE OF &&X 1 1
&IF & ¬= ( &GOTO -TELL
& = &LENGTH OF &&X
&IF & = 1 &X = &X + 1
&IF & > 1 &&X = &PIECE OF &&X 2
&IF &X > &N &GOTO -DOIT
& = &LENGTH OF &&X
& = &PIECE OF INDENT&BLANK 1 &
&IF &&X ¬= & &GOTO -TELL
&X = &X + 1
&INDENT = 5
&IF &X > &N &GOTO -DOIT
& = &DATATYPE OF &&X
&IF & ¬= NUM &GOTO -TELL
&INDENT = &&X
&IF &N > &X &GOTO -TELL
```

When the above section of the macro is entered, &&X is the first argument value that could not be a specification for number of lines, left margin, or right margin. Our PARA command syntax, then, requires that the first character of this word be a left parenthesis. This is the first test. If it is not a left parenthesis, hence not a proper INDENT option, branch to -TELL and display the correct command format for the user. Once we know the left parenthesis is present, we must then find out whether the word starting with the left parenthesis contains any more characters, that is, the word INDENT, or an abbreviation for it. Possibly the user separated the word INDENT from the left parenthesis by a blank, leaving a left parenthesis alone in this argument word. &LENGTH OF tells us this, so if the left parenthesis was the sole character in the current argument word, we simply advance the index variable &X by one and look for INDENT in the next argument word. If the left parenthesis was followed by other characters, we use the &PIECE OF function to "cut away" the parenthesis, leaving the remainder of the word to be tested for INDENT.

As always, before using the next argument word, we must make a test to see that we haven't run out of argument values. As long as &X <= &N, there is at least one argument word, &&X, to be examined. We use &LENGTH OF again to determine the length of the argument, then we use this length to form a piece of the word INDENT of the same length as the argument. (Actually, we append a blank to INDENT, so that in case someone used INDENTURE or INDENTZ, etc., as an option, it will not be accepted.) Now that we have both the argument and the acceptable value of the same length, if they are not equal, go to -TELL to explain things.

If the word INDENT was properly typed, it may be followed by an optional indention amount. Therefore the variable &X is incremented again. This is so that it may be compared with &N to see if another argument is available, and access it if there is. Before making this test, however, the default value of &INDENT is changed from 0 (appropriate when INDENT

is not used) to 5, the default indention amount when INDENT is specified. If another argument word remains, and if it is an integer (this time, a signed number is permitted), that value is assigned to &INDENT. Finally, we check to see if any additional arguments were given in the command. If they were, branch to -TELL since we have no idea what they could mean.

Only one thing remains to be done before entering the reformatting kernel. All of the command arguments have been verified and default values have been supplied where appropriate, but it may be necessary to convert the number of lines (the value of &NLINES) from asterisk into an actual number of lines. This is done by these statements:

```
* Convert * into number of lines
TRANSFER    LENGTH  LINE
&READ VARS  &L      &CURLINE
&LOOP 4 UNTIL &L = 0
  NEXT
  &IF &RC ¬= 0 &GOTO -EOF
  TRANSFER    LENGTH
  &READ VARS  &L
-EOF TRANSFER   LINE
&READ VARS    &ELINE
&NLINES = &ELINE - &CURLINE
:&CURLINE
```

The TRANSFER command is the mechanism whereby an XEDIT macro may query the value of internal XEDIT values, such as the xname and type of the edit file, length of the current line, line number of the current line, etc. The TRANSFER command arguments denote a set of variables whose values XEDIT stacks LIFO in one line. Sometimes, more than one value is given for a single variable name, such as CURSOR. In this case, however, the command:

```
TRANSFER    LENGTH  LINE
```

causes a line containing two words to be stacked. The first word is the length of the current line, excluding trailing blanks. Therefore, if the value of &L is zero, we know the current line is a blank line. The second word is the current line number. This is remembered as the value of &CURLINE, so that we can return to this line after finding the next blank line.

The loop advances the current line pointer to the next line and checks for a blank line. If the return code from NEXT is nonzero, it means we have encountered the end-of-file or end-of-range, which we treat the same as finding a blank line. If NEXT succeeds, the value of &L is updated and the loop continues until &L = 0.

Once a blank line is found, another TRANSFER command retrieves the line number of this line and the necessary number of lines to process is simply the difference between this line number and the line number at which we started. After computing the difference, the command :&CURLINE

restores the current line pointer to the value it had when the macro was entered, and we are ready to enter the reformatting kernel.

The only difference between this kernel and the one discussed above concerns the setting of the initial value of &S before entering the outer loop. Instead of setting it to &SINIT, the following statements are used in order to accommodate the possibly of a negative value for the value of &INDENT:

```
& = &INDENT + &MLEFT - 1
&IF & < 0 & = 0
&S = &LEFT OF &BLANK &
```

Some final words about the PARA XEDIT file. Like other edit macros, this one is not really final, for many users will think of features they would like to add, or changes they could make to better fit it to their needs. Many such changes can be made rather simply, and the description or documentation for them carried in the text of the macro itself and presented to the user at his command (PARA ?).

For example, if you would like two blanks following a word ending with a period, add the statements:

```
& = &RIGHT OF &&X 1
&IF & = . &S = &CONCAT OF &S &BLANK
```

just before the statement labeled -Z. Or, if you wanted to process comments in EXEC files, automatically inserting "*" before each line, you could replace the inner loop statement in the reformatting kernel:

```
&LOOP -Z UNTIL &X > &N
```

with the following:

```
&IF &N = 0 &GOTO -Z
& = &PIECE OF &1 1 1
&IF & = * &1 = &PIECE OF &1 2
&LOOP -Z UNTIL &X > &N
&IF /&&X = / &GOTO -Z
```

and change both of the INPUT commands to "INPUT * &S".

Do not believe that all of this can be done instantly. This PARA XEDIT file was written by the author in about two hours (including time to correct a few errors), but he has had a lot of experience. The first few XEDIT macros that you write will probably take a long time because you are learning about EXEC 2 and you are also learning a new way of using XEDIT. However, the reward for writing them is more than simply a better understanding of XEDIT and EXEC 2, or some justified feeling of accomplishment. The time will come when your work can be done easier and more efficiently if you write or modify an EXEC file, then you will gain the greatest benefit from the effort invested in learning about EXEC 2.

Some Final Words

This primer has illustrated barely half of the total facilities in EXEC 2, though it has discussed many ideas needed to understand and start writing EXEC 2 files. This introduction, and perhaps a little practice, should make it possible for the novice to better use the information in the EXEC 2 Reference section. Like any programming language, facility with EXEC 2 is gained through experience in using it. Often it is easier (and more educational) to try something out, making changes as errors are detected, than to ponder each statement. The trace facility helps in this. It lets the user watch while his program is being executed and identify most errors as they occur.

Appendix E: Useful EXEC 2 Techniques

The following illustrations exhibit solutions to some EXEC programming problems. These solutions frequently involve nonobvious uses of predefined functions to achieve the desired result in a minimum of statements. There has been no attempt to present a comprehensive catalog of solutions. The objective is to give the reader some insight into the possibilities inherent in the EXEC 2 functions.

The statement

```
& = &DATATYPE OF +&1
```

sets & to 'NUM' if, and only if, &1 contains an unsigned integer.

If &J is an unsigned integer not exceeding 99999999, the statement

```
&J = &RIGHT OF 0000000&J 8
```

extends it with leading zeros to a total length of 8.

A string of any number of blanks, 23 for example, can be created by:

```
&B23 = &LEFT OF &BLANK 23
```

A string of some character other than blanks, asterisks for example, is easily obtained from the string of blanks by using the &TRANSLATION OF predefined function:

```
&*23 = &TRANSLATION OF &B23 &BLANK *
```

A multi-way branch is desired, based on an argument value supplied by the caller and currently in &F. However, the value of &F must first be

tested to verify it is valid -- that is, its value is either CASE1, CASE2, etc.

```
& = &POSITION OF &F CASE1 CASE2 CASE3 ...
&IF & ¬= 0 &GOTO -&F
&TYPE INVALID CASE: &F
  ...
-CASE1
  ...
-CASE2
  ...
```

The statement

```
& = &LOCATION OF /&1 //PRINT
```

sets & to 2 if, and only if, &1 contains the word "PRINT" or an abbreviation for it. Note that & would have the value 1 if &1 is null.

Suppose &1 is as given on entry, and is, therefore, known not to contain any blanks. Then the following sequence transfers control to the label -BLUE if &1 contains the word "BLUE" or an abbreviation for it, to the label -GREEN if &1 contains the word "GREEN" or an abbreviation for it, ..., or to the label -ERR if &1 is null or does not contain a color or an abbreviation therefor.

```
&X = &LITERAL OF ERR /ERR /BLUE /GREEN /RED /YELLOW
& = 1 + &LOCATION OF /&1 &X
& = &PIECE OF &X &
&STACK LIFO &GOTO -&
&READ
```

The first statement assigns to &X the string containing all of the expected labels prefaced with / and separated by blanks. In addition, the first word (ERR) is included in case the value of &1 does not appear in &X, and the second word (/ERR) is included in case the value of &1 is null. The third statement assigns to & that part of &X starting with the desired label. A &GOTO statement is then stacked. This statement is read and interpreted by the last, &READ statement. When the stacked line is read, it is broken into words and examined in the ordinary way, so the desired label becomes the &GOTO operand, and any surplus data from the original value of &X is treated as a comment.

The argument values are to be assigned to the variables &Xi, for i = 1, 2, ..., &N. The object of this is to make it possible to reuse the numeric variables without losing access to the current arguments. Calling a user-defined function which needs the argument values that existed before the function was invoked illustrates such a need.

```
&S = &RANGE OF & 1 &N
&STACK LIFO &S
&S = &RANGE OF *X 1 &N
&S = &TRANS OF &S * &
&STACK LIFO &READ VARS &S
&READ
```

The first two lines construct a string from the argument values &1 &2 ... &&N separated by blanks, and stack it. A corresponding string of variable names is then created in two steps. First, a string of words *X1 *X2 ... *X&N is built, then all of the asterisks in that string are translated to ampersands. The string of variable names is then used when stacking a &READ VARS statement. The final statement causes the just stacked &READ VARS statement to be read and interpreted by EXEC 2. When executing this statement, the previously stacked argument values are read and assigned to the desired variables. Note that use of & as a temporary variable is avoided so that its predefined value (ampersand) will be available as an argument to &TRANS OF.

If only a (contiguous) subset of the current arguments are to be transferred to the variables &Xi, the arguments to &RANGE OF may be adjusted as required. If the values of the original arguments, instead of the current argument values, were desired, the first two lines could be replaced with:

```
&STACK LIFO &ARGSTRING
```

To verify that a value is a valid hexadecimal number (contains no characters other than the digits 0-9 and the letters A-F):

```
& = &TRANS OF &HEXNUM 0123456789ABCDEF
&IF /& ¬= / &GOTO -BADHEX
```

The first statement uses &TRANSLATION OF to translate all valid characters in &HEXNUM into blanks. Then, the &IF condition succeeds only if the translation contained something other than blanks (since the shorter word is extended with blanks for purposes of comparing the two strings). This corresponds to the presence of one or more untranslated (that is, invalid) characters in &HEXNUM.

This scheme works only if it is known that there are no blanks embedded in &HEXNUM, or if blanks are acceptable characters. The following elaboration will detect embedded blanks as invalid characters:

```
&Z = &CONCAT OF &BLANK 0123456789ABCDEF
& = &TRANS OF &HEXNUM &Z *
&IF /& ¬= / &GOTO -BADHEX
```

Here, a blank in &HEXNUM is explicitly translated into an asterisk so that it forces the subsequent comparison to fail.

The following EXEC file is useful when it is necessary to extract information delimited by parentheses within a string. Blanks and nested parentheses are retained, so PAREN EXEC may be invoked multiple times when there are nested parentheses. The result is two lines put into the console stack. The first one, stacked LIFO, contains all characters of the original argument string except the first left parenthesis, the characters following it to the matching right parenthesis, and that right parenthesis. The second line contains the data excised from the first line without the delimiting parentheses, but includes any nested parentheses.

```
&TRACE
&A = &ARGSTRING
& = -1 + &LOCATION OF ( &ARGSTRING
&IF & < 0 &GOTO -END
&A = &PIECE OF &ARGSTRING 1 &
& = & + 2
&B = &PIECE OF &ARGSTRING &
&IF .&B EQ     &GOTO -END
& = 1 + -NESTED OF 1
&Z = &PIECE OF &B &
&A = &CONCAT OF &A &Z
& = & - 2
&B = &PIECE OF &B 1 &
-END &STACK LIFO &A
&STACK LIFO &B
&EXIT
* Recursive subroutine to balance parentheses.
* &1 = index into string &B where search is to start.
* Returns index into &B of matching ).
-NESTED &ARGS &1 0 0 0
&LOOP -X *
  &2 = &PIECE OF &B &1
  &3 = &LOCATION OF ) &2
  &4 = &LOCATION OF ( &2
  &IF &4 ¬= 0 &IF &4 < &3 &SKIP 3
    &IF &3 = 0 &3 = 1 + &LENGTH OF &2
    &3 = &1 + &3 - 1
    &RETURN &3
  &2 = &1 + &4
  -X &1 = 1 + -NESTED OF &2
```

This implementation of PAREN illustrates the use of a recursive user-defined function. Notice the &ARGS statement at the beginning of -NESTED which creates three local variables (&2, &3 and &4) each time the function is entered. This automatically associates a unique group of EXEC variables with every invocation of the function (in addition to the function's explicit arguments). Because these variables are unique to an individual invocation of the user-defined function, they are guaranteed not to conflict with any other EXEC variable name. Actually, in this instance the technique is not necessary. The &ARGS statement could be eliminated, and the variables &2, &3, and &4 renamed &S, &L, and &R, without introducing an error. An error would occur only if a subsequent modification of the EXEC file introduced one of those variable names outside of the -NESTED function.

The following version of PAREN EXEC illustrates an alternative implementation which doesn't use a user-defined function:

```
&TRACE
&A = &ARGSTRING
& = -1 + &LOCATION OF ( &ARGSTRING
&IF & < 0 &GOTO -END
&A = &PIECE OF &ARGSTRING 1 &
& = & + 2
&B = &PIECE OF &ARGSTRING &
&IF .&B EQ .   &GOTO -END
&LP = 1
& = 1
&LOOP -X UNTIL &LP = 0
  &S = &PIECE OF &B &
  &R = &LOCATION OF ) &S
  &IF &R = 0 &GOTO -END
  &L = &LOCATION OF ( &S
  &IF &L ¬= 0 &IF &L < &R &SKIP 3
    & = & + &R
    &LP = &LP - 1
    &GOTO -X
  & = & + &L
  &LP = &LP + 1
  -X
&Z = &PIECE OF &B &
&A = &CONCAT OF &A &Z
& = & - 2
&B = &PIECE OF &B 1 &
-END &STACK LIFO &A
&STACK LIFO &B
&EXIT
```

&

"

A

B

C

D

E

IBM VM/SP: EXEC 2 Reference (File No. S370/4300-39) Printed in U. S. A. SC24-5219-1

SC24-5219-1

IBM VM/SP
EXEC 2 Reference
SC24-5219-1

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate. Comments may be written in your own language; English is not required.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

	Yes	*No*
• Does the publication meet your needs?	☐	☐
• Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

• What is your occupation? _____

• How do you use this publication:

As an introduction to the subject?	☐	As an instructor in class?	☐
For advanced knowledge of the subject?	☐	As a student in class?	☐
To learn about operating procedures?	☐	As a reference manual?	☐

Your comments:

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

If you would like a reply, please supply your name and address on the reverse side of this form.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Cut or Fold Along Line

Fold and Tape | Please Do Not Staple | Fold and Tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department G60
P. O. Box 6
Endicott, New York 13760

Fold | Fold

If you would like a reply, *please print:*

Your Name _____

Company Name _____ *Department* _____

Street Address _____

City _____

State _____ *Zip Code* _____

IBM Branch Office serving you _____

IBM®

IBM VM/SP: EXEC 2 Reference (File No. S370/4300-39) Printed in U. S. A. SC24-5219-1

IBM VM/SP
EXEC 2 Reference
SC24-5219-1

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate. Comments may be written in your own language; English is not required.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

	Yes	*No*
• Does the publication meet your needs?	☐	☐
• Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

• What is your occupation? _____

• How do you use this publication:

As an introduction to the subject?	☐	As an instructor in class?	☐
For advanced knowledge of the subject?	☐	As a student in class?	☐
To learn about operating procedures?	☐	As a reference manual?	☐

Your comments:

If you would like a reply, please supply your name and address on the reverse side of this form.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Cut or Fold Along Line

IBM VM/SP: EXEC 2 Reference (File No. S370/4300-39) Printed in U. S. A. SC24-5219-1

Fold and Tape | Please Do Not Staple | Fold and Tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department G60
P. O. Box 6
Endicott, New York 13760

Fold | Fold

If you would like a reply, *please print:*

Your Name _____

Company Name _____ *Department* _____

Street Address _____

City _____

State _____ *Zip Code* _____

IBM Branch Office serving you _____

IBM®