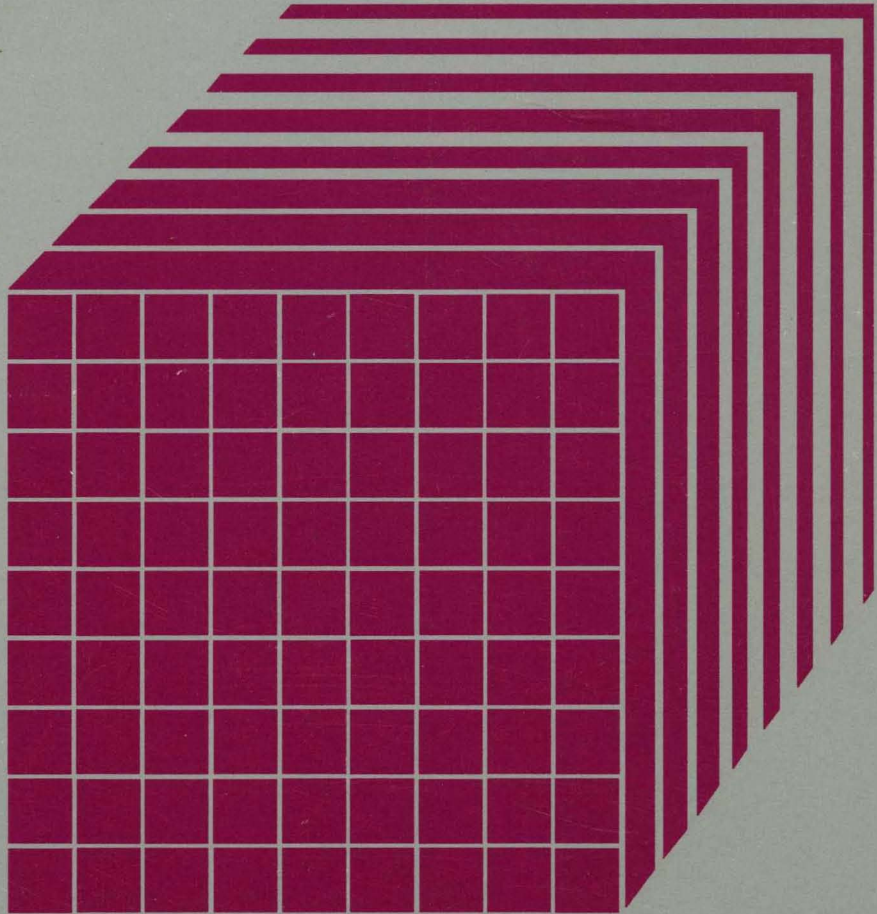
Virtual Machine/
System Product

EXEC 2 Reference

Release 5

SC24-5219-3

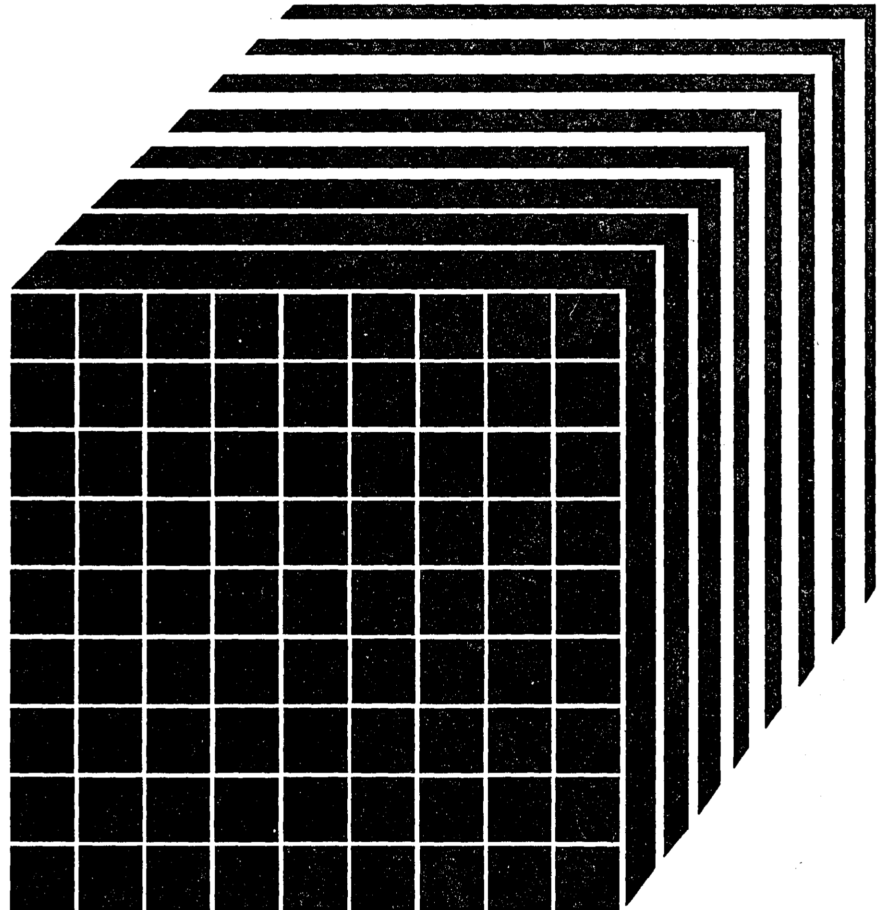


Virtual Machine/ System Product

EXEC 2 Reference

Release 5

SC24-5219-3



Fourth Edition (December 1986)

This edition, SC24-5219-3, is a major revision of SC24-5219-2, and applies to release 5 of the Virtual Machine/System Product, (VM/SP), program number 5664-167, and to all subsequent versions and releases until otherwise indicated in 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, 30xx and 4300 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

Summary of Changes

For a list of changes, see page 119.

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

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

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 readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Dept. G60, P.O. Box 6, Endicott, NY, 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 to you.

© Copyright International Business Machines Corporation 1980, 1981, 1982, 1983, 1986

Preface

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 C: 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:

- "Chapter 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.

- "Chapter 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" are also discussed.

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

- "Chapter 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.

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

- "Appendix B: 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 B also contains a discussion of variable sharing through the EXECCOMM interface.

- "Appendix C: 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 D: Writing Editor Macros" describes how to write XEDIT macros and XEDIT prefix macros. This Appendix also contains examples of XEDIT macros and XEDIT prefix macros.

- "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 C: 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 B 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.

Contents

Chapter 1: Introduction **1**
Executing EXEC 2 Programs 1
Introduction to the EXEC 2 Language 1
Types of Executable Statements 3
Rules for Interpreting Executable Statements 4

Chapter 2: EXEC 2 Statements **5**
Predefined Variables 5
Control Statements 8
Predefined Functions 40
User-Defined Functions 57

Chapter 3: Notes on EXEC 2 .... **59**
Name Substitution 59
Recursive Execution 61
Termination of an EXEC 2 File 61
Program Stack 61
Assignment Statement 62
Evaluation of &DATE and &TIME 62
Size and Treatment of Numbers 62
Removing Plus Signs and Leading Zeros 63
Syntax of Conditional Phrases 63
Embedded Blanks 64
&LOOP Statement 65
Closing of Loops 65
Search for Labels 66
Performance of Label Searches 66
EXEC 2 Words are Not Reserved Words 66
Example of &TRACE ALL 67
Truncation Column 68

Chapter 4: BNF Description of the EXEC 2 Syntax **69**

Chapter 5: EXEC 2 Errors **73**

Appendix A. Sample EXEC 2 Files **75**

Appendix B. EXEC 2 in CMS **77**
Identifying EXEC 2 Files 77
Calling EXEC 2 Programs from CMS Command Level 77
Summary of Limits for EXEC 2 Files in CMS 78
Using EXEC 2 Parameter Lists with Assembler Language Programs 79

Executing XEDIT Macros in EXEC 2 83
EXECCOMM - Sharing EXEC 2 Variables with Assembler Language Programs 83

Appendix C. EXEC 2 Primer for New Users 87
EXEC 2 Variable Names 89
Return Codes and EXEC 2 Variables 89
EXEC 2 File Arguments 90
Conditional Interpretation of Statements 91
Statement Labels 92
Assignment Statements 92
EXEC 2 Variable Evaluation 93
An Example of Generating EXEC 2 Variable Names 93
The &LOOP Control Statement 95
Making EXEC 2 Files Interact with Users 95
Using the &CASE Control Statement 99

Appendix D. Writing Editor Macros 101
What is an XEDIT Macro? 101
Creating a Macro File 102
Using XEDIT Subcommands in a Macro 102
Handling Embedded Blanks 104
Avoiding Name Conflicts 105
Walking Through An XEDIT Macro 105
Using the XEDIT EXTRACT Subcommand 110
Writing Prefix Macros 110
What Information is Passed to the Macro? 110
Creating a Sample Prefix Macro 111

Appendix E. Useful EXEC 2 Techniques 113

Summary of Changes 119

Bibliography 121
Prerequisite Publications: 121
Corequisite Publications: 121

Index 125

Figures

1. A Sample Macro: GLOBCHG 106
2. A Sample Prefix Macro: U 111

Chapter 1: Introduction

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 B, "EXEC 2 in CMS" on page 77 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.

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. This line 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 B, "EXEC 2 in CMS" on page 77 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.

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" on page 83 for possible side-effects of command execution.)

- Assignments

  An executable statement is an assignment statement 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.

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. Then its name is retained. (A precise description is given later in "Name Substitution" on page 59.) 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 "Chapter 3: Notes on EXEC 2" on page 64 on embedded blanks.

4. The statement is analyzed syntactically. Note that, except for identifying the targets of assignment, the syntax analysis is done *after* steps 1, 2, and 3 above.

Chapter 2: EXEC 2 Statements

Predefined Variables

The following EXEC 2 variables are initialized or maintained automatically:

```
&
```

& is initialized to its own name (the value "&").

```
&0
```

&0 is 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 is invoked via a synonym.

```
&1 &2 ...
```

&1 &2 ... are the EXEC 2 *arguments*. They are initialized to the arguments "arg1", "arg2", ... that 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. The arguments are reset by the control statement &ARGS and &READ ARGS. The arguments 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 they cannot be set explicitly (for example, with an assignment statement). (See the description of &N and &INDEX.)

```
&ARGSTRING
```

&ARGSTRING is 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 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
```

&BLANK is a word that has the value of a single blank.

```
&CMDSTRING
```

&CMDSTRING is 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
```

&COMLINE is maintained as the number of the line from which the last command (or subcommand) was issued from the EXEC 2 file. &COMLINE is initialized to zero.

```
&DATE
```

&DATE is 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
```

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

```
&FILEMODE
```

&FILEMODE is initialized to the filemode (third qualifier) of the EXEC 2 file.

```
&FILENAME
```

&FILENAME is initialized to the filename (first qualifier) of the EXEC 2 file.

```
&FILETYPE
```

&FILETYPE is initialized to the filetype (second qualifier) of the EXEC 2 file (for example, "EXEC").

```
&FROM
```

&FROM is maintained as the number of the line in the EXEC 2 file from which the last &GOTO statement was executed. &FROM is initialized to zero.

```
&LINE
&LINENUM
```

&LINE or &LINENUM is maintained as the number of the current line in the EXEC 2 file. If &LINE or &LINENUM is the target of an assignment statement, the value of the other variable is not affected.

```
&LINK
```

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

```
&N
&INDEX
```

&N or &INDEX is 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 if &ARGS or &READ ARGS is executed. &N or &INDEX is temporarily reset by invocation of user-defined subroutines and functions. (See the description of &1 &2 ....) If &N or &INDEX is the target of an assignment statement, the value of the other variable is not affected.

```
&RC
&RETCODE
```

&RC or &RETCODE is maintained as the return code from the last command (or subcommand) issued from the EXEC 2 file. &RC and &RETCODE are initialized to zero. If &RC or &RETCODE is the target of an assignment statement, the value of the other variable is not affected.

```
&TIME
```

&TIME is 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 following are the control words and the rules for their use.

&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 (&N or &INDEX) is the number of words given in the &ARGS statement. This number may be less than or greater than the number of arguments previously set.

For example:

```
&TRACE
&TYPE &1 &2 &3 &4
&TYPE THE NUMBER OF ARGUMENTS SET IS: &N
&ARGS FOUR FIVE SIX SEVEN
&TYPE &1 &2 &3 &4
&TYPE NOW, THE NUMBER OF ARGUMENTS SET IS: &N
&EXIT
```

Suppose the name of the above EXEC is NUMBER. If you issue the following command:

```
NUMBER RED WHITE BLUE
```

the result of the EXEC is:

```
RED WHITE BLUE
THE NUMBER OF ARGUMENTS SET IS: 3
FOUR FIVE SIX SEVEN
NOW, THE NUMBER OF ARGUMENTS SET IS: 4
```

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

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

Print at the console "line1", "line2", ... without removing surplus blanks or replacing any EXEC 2 variables. Columns 1-k are printed if "k" is specified. If the truncation column, "k", is not given, or is given as "*", the lines are not truncated by the EXEC 2 interpreter. (CMS truncates at 130 characters. See Appendix B, "EXEC 2 in CMS" on page 77.)

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 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.

These 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.

For example:

```
&TRACE
&BEGPRINT -END
ROSES ARE RED
VIOLETS ARE BLUE
-END
&EXIT
```

The result of the above EXEC is:

```
ROSES ARE RED
VIOLETS ARE BLUE
```

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

&BEGSTACK

```
&BEGSTACK   ┌      ┌    ┌    ┐┐┐
            │n     │k   │FIFO││
            │*     │*   │LIFO││
            │label └    └    ┘┘
            │1                  │
            └                   ┘

line1
line2
...
...
...
```

(In the syntax box, "1", "*" (after k) and "FIFO" are underscored as defaults.)

Place in the program stack "line1", "line2", ... without removing surplus blanks or replacing any EXEC 2 variables. Columns 1-k are stacked if "k" is specified. 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 specifying "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 that 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 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.

For example:

```
&TRACE
&BEGSTACK 2 * LIFO
THE FIRST COLOR IS RED
THE SECOND COLOR IS BLUE
&READ STRING &ONE
&READ STRING &TWO
&TYPE &ONE
&TYPE &TWO
&EXIT
```

The result of the above EXEC is:

```
THE SECOND COLOR IS BLUE
THE FIRST COLOR IS RED
```

The last line stacked is the first line read since the LIFO option was specified.

(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 B, "EXEC 2 in CMS" on page 77.)

The lookaside buffer is a device that enables the EXEC 2 interpreter to remember the location of labels that have already been referenced 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, where 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 the line starting with the given label. Control is returned to the line following the &CALL statement via the &RETURN statement.

The new generation of arguments supersedes the arguments that were previously set. The previous value of the arguments and the number of arguments previously set are temporarily inaccessible. On entry to the subroutine, the value of the arguments and the number of arguments set are determined by the arguments specified in the &CALL statement. Their values and the number of arguments set can be changed inside the subroutine in the same way as outside - 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 are 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. 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.

Suppose the following EXEC is named AVG:

```
&TRACE
&TYPE THE INITIAL VALUES OF THE ARGUMENTS ARE:
&DUMP ARGS
&CALL -AVG &2 &1
&TYPE THE AVERAGE IS: &ANSWER
&EXIT
-AVG
&TYPE THE VALUES OF THE ARGUMENTS IN THE -AVG SUBROUTINE ARE:
&DUMP ARGS
&SUM = &1 + &2
&ANSWER = &DIV OF &SUM 2
&RETURN
&EXIT
```

If you issue the command

```
AVG 76 98
```

the result is:

```
THE INITIAL VALUES OF THE ARGUMENTS ARE:
&1 = 76
&2 = 98
THE VALUES OF THE ARGUMENTS IN THE -AVG SUBROUTINE ARE:
&1 = 98
&2 = 76
THE AVERAGE IS: 87
```

Initially, &1 = 76 and &2 = 98. The &CALL statement transfers control to the label -AVG and passes the arguments 98 and 76. (Notice that the arguments in this example are passed to the -AVG subroutine in a different order than they were received.) Now, in the -AVG routine, &1 = 98 and &2 = 76.

The &RETURN statement then transfers control to the line following the &CALL statement. The initial values of &1 and &2 are accessible again - &1 = 76 and &2 = 98.

(See the description of &RETURN; also see "User-Defined Functions" on page 57.)

&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".

For example:

```
&TRACE
&TYPE ENTER YOUR NAME:
&CASE M
&READ VARS &NAME
&TYPE &NAME
&EXIT
```

The above EXEC prompts you to enter your name. If you enter your name using uppercase and lowercase characters, such as:

```
Sue
```

the result is:

```
Sue
```

However, if the "&CASE M" control statement is removed, the result is:

```
SUE
```

(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 the command 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 specifying it as an argument of &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 B, "EXEC 2 in CMS" on page 77.)
- &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 "EXECCOMM - Sharing EXEC 2 Variables with Assembler Language Programs" on page 85 for possible side-effects of command execution.)

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

Print lines at the console in 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 B, "EXEC 2 in CMS" on page 77.)

For example:

```
&TRACE
&ARGS ROSES ARE RED
&TYPE &1 &2 &3
&ONE = &1
&TWO = &2
&THREE = &3
&DUMP ARGS
&DUMP VARS &ONE &TWO &THREE
&EXIT
```

The result of the above EXEC is:

```
ROSES ARE RED
&1 = ROSES
&2 = ARE
&3 = RED
&ONE = ROSES
&TWO = ARE
&THREE = RED
```

&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. The &ERROR control statement must come before any statement that may give an error return code. If it does not, the action specified will not occur.

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 on the line following the command that caused the action to be invoked.

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

Suppose the name of the following EXEC is MEMOLIST:

```
&TRACE
&ERROR &TYPE THE RETURN CODE IS &RC
CMDCALL LISTFILE &1 MEMO A
&EXIT
```

If you want to see if the file POEM MEMO exists on your A disk, issue the command:

```
MEMOLIST POEM
```

If POEM MEMO does not exist, you will receive the message:

```
FILE NOT FOUND
THE RETURN CODE IS 28
```

However, if the file POEM MEMO does exist, you will receive the following message:

```
POEM MEMO A1
```

&EXIT	[return-code / <u>0</u> [comment]]

Stop execution of the EXEC 2 file, and yield the given return code. If the return code is specified, it 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 B, "EXEC 2 in CMS" on page 77.)

For example:

```
&TRACE
&SUM = 0
&TYPE ENTER A NUMBER:
&READ ARGS
&IF &1 < 0 &EXIT 100
&SUM = &SUM + &1
&TYPE THE SUM IS: &SUM
&EXIT
```

The above EXEC prompts you to enter a number. If a negative number is entered, the &IF statement is true. Therefore, the &EXIT control statement is executed and the result is:

```
R(00100);
```

If you enter the number 12, the result is:

```
THE SUM IS: 12
```

&GOTO	line-number / label [comment]

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 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.

For example:

```
&TRACE
&P = 1
&SUM = 0
-START &IF &P > 3 &GOTO -END
&TYPE ENTER A NUMBER:
&READ ARGS
&IF &1 < 0 &EXIT 100
&SUM = &SUM + &1
&P = &P + 1
&GOTO -START
-END
&TYPE THE SUM IS: &SUM
&EXIT
```

At the first &IF statement, the EXEC 2 interpreter compares the variable P to 3. (&P counts how many numbers have been entered.) If the comparison is true, then the &GOTO statement is executed. If the comparison is false, the &GOTO statement is ignored and execution continues with the next statement.

If you enter the numbers 10, 500, and 100 when you are prompted, the result is:

```
THE SUM IS: 610
```

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

```
&IF        word1 =|EQ                    [word2 executable-stmt]
                 ¬=|NE
                 <|LT
                 <=|¬>|LE|NG
                 >|GT
                 >=|¬<|GE|NL
```

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"). If "word2" is absent, a null string is used in its stead. The comparison is numeric if both comparatives 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.

For example:

```
&TRACE
&P = 1
&SUM = 0
-START &IF &P > 3 &GOTO -END
&TYPE ENTER A NUMBER:
&READ ARGS
&IF &1 < 0 &EXIT 100
&SUM = &SUM + &1
&P = &P + 1
&GOTO -START
-END
&TYPE THE SUM IS: &SUM
&EXIT
```

At the first &IF statement, the EXEC 2 interpreter compares the variable P to 3. (&P counts how many numbers have been entered.) If the comparison is true, then the &GOTO statement is executed. If the comparison is false, the &GOTO statement is ignored and execution continues with the next statement.

If you enter the numbers 10, 500, and 100 when you are prompted, the result is:

```
THE SUM IS: 610
```

&LOOP

&LOOP	n label	m * WHILE condition UNTIL condition

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

Loop through the following "n" lines, or loop down to (and including) the first line starting with "label". Execute the loop "m" times, 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 comparatives 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 the condition is invalid, execution stops abnormally with an error message that identifies the line containing the &LOOP statement.

For example:

```
&TRACE
&SUM = 0
&LOOP -END 3
&TYPE ENTER A NUMBER:
&READ ARGS
&IF &1 < 0 &EXIT 100
&SUM = &SUM + &1
-END
&TYPE THE SUM IS: &SUM
&EXIT
```

This &LOOP statement tells the EXEC 2 interpreter to execute the next five lines (up to and including the line beginning with the -END label) three times.

If you enter the numbers 10, 500, and 100 when you are prompted, the result is:

```
THE SUM IS: 610
```

&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 the environment does not exist when a subcommand is subsequently issued, 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 they do not have the syntax of a command.

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

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

Print at the console a line containing "word1", "word2", ..., or print a blank line if there are no words given. Each word is separated from each other by a single blank. The line is truncated if necessary. (In CMS, the line is truncated if its length exceeds 130. See Appendix B, "EXEC 2 in CMS" on page 77.)

Unlike &BEGPRINT and &BEGTYPE, surplus blanks are removed and the words to be printed are searched for EXEC 2 variables. Then these variables are replaced by their values.

For example:

```
&TRACE
&COLOR1 = RED
&COLOR2 = BLUE
&TYPE ROSES    ARE    &COLOR1
&PRINT VIOLETS    ARE    &COLOR2
&EXIT
```

The result of the above EXEC is:

```
ROSES ARE RED
VIOLETS ARE BLUE
```

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

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

Read from the console stack (program stack and terminal input buffer), 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 "*". Execute the lines individually as if they had been part of the EXEC 2 file. Reading stops (and normal execution resumes) when "n" lines are 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 is continued 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". Surplus blanks are not removed, and EXEC 2 variables are not replaced.

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 the variables on the left-hand side of an assignment statement. (See "Name Substitution" on page 59.) 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 program stack is 255. See Appendix B, "EXEC 2 in CMS" on page 77.)

Suppose you have the following EXEC named QUALIFY:

```
&TRACE
&TYPE ENTER YOUR NAME PLEASE (FIRST AND LAST):
&READ STRING &NAME
&TYPE ENTER YOUR SOCIAL SECURITY NUMBER PLEASE:
&READ VARS &NUM
&TYPE NOW, TELL US YOUR AGE:
&READ ARGS
&IF &1 < 21 &TYPE &NUM ---- &NAME IS TOO YOUNG
&IF &1 >= 21 &TYPE &NUM ---- &NAME IS OLD ENOUGH
&EXIT
```

First you are prompted to enter your name:

```
SUE SMITH
```

Then, the &READ STRING &NAME statement reads the line from the console and assigns the literal string to &NAME. Now, &NAME equals "SUE SMITH". Next you need to enter your social security number

```
111-11-1111
```

The next &READ statement assigns the word, 111-11-1111, to the variable &NUM.

If you enter 24 as your age, the EXEC 2 interpreter assigns 24 to the variable &1. Since &1 is greater than or equal to 21, the result of this EXEC, with the above data, is:

```
111-11-1111 ---- SUE SMITH IS OLD ENOUGH
```

&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 subroutine 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 console 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 left opened in the subroutine or function 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 and example of &CALL; also see "User-Defined Functions" on page 57.)

&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.

For example:

```
&TRACE
&SKIP 3
&TYPE THREE
&TYPE FOUR
&EXIT
&TYPE ONE
&TYPE TWO
&SKIP -5
&EXIT
```

The result of the above EXEC is:

```
ONE
TWO
THREE
FOUR
```

(See the description of &GOTO.)

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

Place a line in the program stack containing "word1", "word2", ..., or stack a null line if there are no words. Each word is separated from each other by a single blank. (In CMS, stacked lines are truncated at 255. See Appendix B, "EXEC 2 in CMS" on page 77.)

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. If "word1" is itself FIFO or LIFO, then it must be preceded by the FIFO or LIFO stacking choice.

Unlike &BEGSTACK, surplus blanks are removed and the words to be stacked are searched for EXEC 2 variables. Then these variables are replaced by their values.

For example:

```
&TRACE
&COLOR1 = RED
&COLOR2 = BLUE
&STACK LIFO THE FIRST COLOR IS &COLOR1
&STACK LIFO THE SECOND COLOR IS &COLOR2
&READ STRING &ONE
&READ STRING &TWO
&TYPE &ONE
&TYPE &TWO
&EXIT
```

Since the data is stacked LIFO (last-in, first-out) the result is:

```
THE SECOND COLOR IS BLUE
THE FIRST COLOR IS RED
```

(See the description of &BEGSTACK.)

&SUBCOMMAND

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

Issue to the given subcommand environment the subcommand comprising of "word1", "word2", ... Each word is separated from each other by a single blank. When the subcommand 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 "EXECCOMM - Sharing EXEC 2 Variables with Assembler Language Programs" on page 83 for possible side-effects of command execution.)

```
&TRACE     [ON    [output-action]]
           [ERR   [             ]]
           [ALL   [             ]]
           [OFF   [             ]]
           [*     [             ]]
```

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.

Suppose you have the following EXEC named TEST EXEC:

```
&TRACE
CP NMES
CP Q USERS
&TYPE HELLO
&EXIT
```

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) + + +".

The result of the above EXEC with "&TRACE ON" specified is:

```
CP NMES
+++ E(1) +++
CP Q USERS
096 USERS, 010 DIALED, 000 NET
HELLO
```

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) + + +".

The result of the above EXEC with "&TRACE ERR" specified is:

```
CP NMES
+++ E(1) +++
096 USERS, 010 DIALED, 000 NET
HELLO
```

ALL

When ALL is in effect, every executable statement is traced before it is executed, and every executable statement is preceded by its line number. Nonzero return codes are traced (as for ON and ERR). Loop conditions and lines that are read from the console stack or 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.

The result of the above EXEC with "&TRACE ALL" specified is:

```
2. CP NMES
+++ E(1) +++
3. CP Q USERS
096 USERS, 010 DIALED, 000 NET
4. &TYPE HELLO
HELLO
5. &EXIT
```

OFF

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

The result of the above EXEC with "&TRACE OFF" specified is:

```
096 USERS, 010 DIALED, 000 NET
HELLO
```

*

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. Let's change the output-action of the &TRACE statement in the above TEST EXEC so the trace information is printed somewhere other than at the console. If the &TRACE statement is changed to:

```
&TRACE ALL &COMMAND EXECIO 1 DISKW INFO SCRIPT A (STRING
```

the trace information is written to the CMS file INFO SCRIPT A. The INFO SCRIPT A file contains:

```
3. CP NMES
+++ E(1) +++
4. CP Q USERS
5. &TYPE HELLO
6. &EXIT
```

See *VM/SP CMS Command Reference* for details on the EXECIO command.

&TRUNC

&TRUNC	[k | * [comment]]

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 B, "EXEC 2 in CMS" on page 77.)

This setting affects only the reading of EXEC 2 statements from a file and the search for labels. &TRUNC does not affect lines read from the console (these lines are never truncated) or lines appearing within a &BEGPRINT, &BEGTYPE, or &BEGSTACK statement (these lines are not truncated unless the statements specify a truncation column). 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.

Suppose you had the following EXEC:

```
&TRACE
&TRUNC 19
&TYPE TYPE YOUR NAME:
&READ STRING &NAME
&TYPE YOUR NAME IS:
&TYPE &NAME
&TYPE
&BEGPRINT
THIS IS AN EXAMPLE OF THE TRUNC STATEMENT
&EXIT
```

The line "&TYPE TYPE YOUR NAME:" is truncated at column 19 and appears on the screen as:

```
TYPE YOUR NAM
```

However, if you enter the name:

```
MARGARET SMITHSONIAN
```

the result is:

```
YOUR NAME IS:
MARGARET SMITHSONIAN

THIS IS AN EXAMPLE OF THE TRUNC STATEMENT
```

The lines read from the console are not truncated. Also, the message following the &BEGPRINT statement is not truncated.

(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", ...

For example:

```
&TRACE
&TYPE ENTER YOUR FIRST NAME:
&CASE M
&READ ARGS
&TYPE &1
&UPPER ARGS
&TYPE &1
&EXIT
```

The above EXEC prompts you to enter your name. Suppose you enter your name as follows:

```
Sue
```

Because of the control statement "&CASE M", "Sue" is not translated to uppercase. However, when the &UPPER ARGS statement is interpreted, the value of &1 is translated to uppercase.

The result of the above EXEC is:

```
Sue
SUE
```

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 following are names of the predefined functions and the rules for their use.

&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

&DATATYPE OF &TYPE OF	[word]

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

Example:

```
&TRACE
&X = &DATATYPE OF -2
&Y = &TYPE OF 1
&Z = &DATATYPE OF 123HELLO
```

This sets &X to "NUM", &Y to "NUM", and &Z to "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.

If the dividend and divisor are both positive, or if they are both negative, the result is positive; if the dividend is positive and the divisor is negative, or vice versa, the result is negative; if the dividend is zero, then the result is 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

&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.

Example:

```
&TRACE
&X = &LEFT OF HELLO 3
&EXIT
```

This sets &X to "HEL".

(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.

Example:

```
&TRACE
&X = &LENGTH OF BOOKS
&EXIT
```

The value of &X is 5.

&LITERAL OF

&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

&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 TWO ONE TWO THREE
```

This sets &X to 2.

Note: "word1", "word2" ... must be listed individually. If a variable contains a string of words, the &POSITION OF predefined function will not find "word", since the variable will be treated as one word.

For example, suppose you passed an EXEC the argument string "ONE TWO THREE":

```
&X = &POSITION OF TWO &ARGSTRING
```

This will set &X to 0, since "TWO" is only part of the whole word, "ONE TWO THREE".

(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:

1. 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
   ```

2. The sequence:

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

   produces the same result as:

   ```
   &PRINT &1 &2 &3 &4 &5
   ```

   and this yields the printed line:

   ```
   A BC DEF GHIJ KLMNO
   ```

3. 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.

Example:

```
&TRACE
&X = &RIGHT OF HELLO 3
&EXIT
```

This sets &X to “LLO”.

(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

&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".

Example:

```
&TRACE
&X = &TRANS OF 85BBE 1234567890ABCDE ABCDEFGHIJKLMNOP
&PRINT &X
&EXIT
```

"word1" is "85BBE", "word2" is "1234567890ABCDE", and "word3" is "ABCDEFGHIJKLMNOP".

The first character in "word1" is "8". "word2" is scanned for the character "8". "8" is the eighth character in "word2". Now, look at the eighth character in "word3" - this character is "H". Do the same for "5" in "word1". "5" is the fifth character in "word2"; and, the fifth character in "word3" is "E". Continue this for the remaining characters in "word1".

The result is:

```
HELLO
```

&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

&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.

Example:

```
&TRACE
&X = &WORD OF ONE TWO THREE FOUR FIVE 3
&EXIT
```

This sets &X to "THREE".

(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:

line-number OF label OF	[arg1 [arg2 ...]]

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:

1. 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
   ```

2. 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
```

Chapter 3: Notes on EXEC 2

The following is a list of topics contained in this chapter:

- Name Substitution
- Recursive Execution
- Termination of an EXEC 2 File
- Program Stack
- Assignment Statement
- Evaluation of &DATE and &TIME
- Size and Treatment of Numbers
- Removing Plus Signs and Leading Zeros
- Syntax of Conditional Phrases
- Embedded Blanks
- &LOOP Statement
- Closing of Loops
- Search for Labels
- Performance of Label Searches
- EXEC 2 Words are Not Reserved Words
- Example of &TRACE ALL
- Truncation Column.

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" on page 5.

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
```

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 files written in CMS EXEC language and Restructured Extended Executor (REXX) language. EXEC 2 files that have the filetype EXEC can, for example, be invoked by means of the statement:

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

Termination of an EXEC 2 File

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

1. When an &EXIT statement is executed; or

2. 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

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

Program Stack

EXEC 2 can use the CMS program 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 program stack. It provides a simple mechanism for communicating between programs. In EXEC 2 files, lines can be deposited in the program stack with the &STACK or &BEGSTACK statements, and can be retrieved with the &READ statement.

Notes on EXEC 2

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 "EXEC 2 Words are Not Reserved Words" on page 66). 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
```

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
```

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 B, "EXEC 2 in CMS" on page 77.) 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.

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 = 0000000000000000000012
&Y = &X + 0
&PRINT &X &Y
```

This yields the printed line:

```
0000000000000000000012 12
```

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 *System/370 Reference Summary*.)

Embedded Blanks

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

1. &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.

2. The "&READ STRING var" statement assigns to the given variable the complete line exactly as read; this variable may contain embedded blanks.

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

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

4. 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.

5. 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.

&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
```

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:

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

2. 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:

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

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

2. 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.

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.

Performance of Label Searches

1. &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.

2. &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.

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.

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), EXTRACT LENGTH (which returns the length of the current line), and TOP (which moves to the first line in the file). The following sample XEDIT 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
&IF &RC ¬= 0 TOP
NEXT
&LOOP 3 WHILE &RC = 0
   EXTRACT /LENGTH
   &IF &LENGTH.1 > &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
+++ E(1) +++
3. &IF 1 ¬= 0 TOP
3. ... TOP
4. NEXT
5. &LOOP 3 WHILE &RC = 0
--- LOOP WHILE 0 = 0
6. EXTRACT /LENGTH
7. &IF 30 > 40 &EXIT
8. NEXT
--- LOOP WHILE 0 = 0
6. EXTRACT /LENGTH
7. &IF 30 > 40 &EXIT
8. NEXT
+++ E(1) +++
--- LOOP WHILE 1 = 0
9. &EXIT 1
```

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.

Chapter 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>                 ::=  *
                               *<comment_string>

<comment_string>          ::=  <character_string>
                               <comment_string> <character_string>

<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>
                               &IF <word> <comparator> <word>

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

EXEC 2 Syntax

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

<variable>          ::=  &<character_string>
                         <predefined_variable>

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

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

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

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

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

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

<predefined_variable>   ::=  &
                             &0
                             &1 &2 ...
                             &ARGSTRING
                             &BLANK
                             &CMDSTRING
                             &COMLINE
                             &DATE
                             &DEPTH
                             &FILEMODE
                             &FILENAME
                             &FILETYPE
                             &FROM
                             &INDEX
                             &LINE
                             &LINENUM
                             &LINK
                             &N
                             &RC
                             &RETCODE
                             &TIME

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

<arg_string>            ::= <word>
                            <arg_string> <word>

<letter>                ::=  a|b|c| ... |x|y|z|A|B|C ...

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

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

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

<user_function>         ::=   <unsigned_integer> OF <arg_string>
                              <label> OF <arg_string>

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

Chapter 5: EXEC 2 Errors

1. 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

2. 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

Appendix A. 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
*
*  THIS EXEC COPIES A FILE FROM ANY
*  CMS DISK TO THE USER'S A-DISK
*
*  CHECK THE NUMBER OF ARGUMENTS, AND USE FILEMODE
*  OF "*" IF IT IS NOT GIVEN ...
*
&IF &N = 0 &GOTO -TELL
&IF &N < 2 &GOTO -BAD
&IF &N > 3 &GOTO -BAD
&IF &N = 2 &ARGS &1 &2 *
*
*  COPY THE FILE SPECIFIED ONTO THE USER'S A-DISK,
*  AND EXIT WITH THE RETURN CODE FROM THE
*  COPYFILE COMMAND
*
COPYFILE   &1 &2 &3   &1 &2 A
&EXIT &RC
*
*  SEND THE USER A MESSAGE THAT THE GRAB COMMAND WAS
*  INVALID, AND EXIT WITH A RETURN CODE OF 101
*
-BAD &PRINT INVALID GRAB COMMAND
&EXIT 101
*
*  TELL THE USER HOW TO ISSUE THE GRAB COMMAND,
*  AND EXIT WITH A RETURN CODE OF 100
*
-TELL &PRINT COMMAND IS: GRAB FN FT FM
&PRINT COPY THE GIVEN FILE TO THE A-DISK,
&PRINT AND PASS 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 [FILEMODE]
*   IF THERE ARE NO ARGUMENTS GIVEN, TELL USER HOW ...
*
*   CHECK THE NUMBER OF ARGUMENTS, AND USE FILEMODE
*   OF "*" IF IT IS NOT GIVEN ...
*
&IF &N = 0 &GOTO -TELL
&IF &N < 3 &GOTO -BAD
&IF &N > 4 &GOTO -BAD
&IF &N = 3 &ARGS &1 &2 &3 *
*
*   SPOOL PUNCH TO USER'S CARD-READER, OR
*   COMPLAIN IF THE USER IS NOT KNOWN TO THE SYSTEM ...
*
CP SPOOL PUNCH TO &1
&IF &RC ¬= 0 &GOTO -BADUSER
*
*   PUNCH THE FILE, OR COMPLAIN IF FAILURE ...
*
PUNCH &2 &3 &4
&IF &RC ¬= 0 &GOTO -ERROR
*
*   TELL THE USER 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 *
&EXIT
*
*   SEND THE USER A MESSAGE THAT THE SHIP COMMAND
*   WAS INVALID, AND RETURN WITH AN ERROR ...
*
-BAD &PRINT INVALID SHIP COMMAND
&EXIT 101
*
*   SEND THE USER A MESSAGE THAT THE USERID IS NOT
*   VALID, AND RETURN WITH AN ERROR ...
*
-BADUSER &PRINT &1 IS NOT A VALID USERID
&EXIT 102
*
*   SEND THE USER A MESSAGE THAT THERE WAS AN
*   ERROR WHEN PUNCHING THE FILE; THEN UNSPOOL
*   THE PUNCH, AND RETURN WITH AN ERROR ...
*
-ERROR &PRINT ERROR &RC FROM "PUNCH" (WHILE IN SEND)
CP SPOOL PUNCH TO *
&EXIT 103
*
*   TELL THE USER HOW TO ISSUE THE SHIP COMMAND
*
-TELL &PRINT COMMAND IS: SHIP USER FN FT [FM]
&EXIT 100
```

Appendix B. EXEC 2 in CMS

Identifying EXEC 2 Files

Since all EXEC files are called in the same way, CMS examines the first statement of the EXEC 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.

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 contains the untranslated command string.

When EXEC 2 programs are invoked from another EXEC 2 program, no translation takes place, and &CMDSTRING is 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 program 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 "Control Statements" on page 8). 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 causes 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 are not typed: ERASE, LISTFILE, RENAME, STATE, and FILEDEF. (See the description of CMDCALL, in "Calling EXEC 2 Programs from CMS Command Level" on page 77 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 is reflected in the predefined variables &RETCODE and &RC.
- EXEC 2 is designed to maintain a complex program environment. For this reason, automatic clean-up is not 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.

Note: The CMS EXECOS command can be used for OS reset and VSAM clean-up.

- The length limit for values assigned via the EXECCOMM facility is 255. If the limit is exceeded, the return code from the EXECCOMM facility 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 facility 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 facility is 8 (INVALID NAME).

- If a "FETCH" reference is made to &ARGSTRING (or &CMDSTRING) via the EXECCOMM facility 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 facility, 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 "Evaluation of &DATE and &TIME" on page 62 in Chapter 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  OF
         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
         DC  A(line 1),F'len 1' Address and length of
*                                 file line 1
         DC  A(line 2),F'len 2' Address and length of
*                                 file line 2
         DC  A(line 3),F'len 3' Address and length of
*                                 file line 3
         .
         .
         .
         DC  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 XEDIT macro with a return code. The XEDIT macro then continues execution with the next statement or command. When the XEDIT macro completes, control returns to XEDIT.

See Appendix D, "Writing Editor Macros" on page 101 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   OF                  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
         DC    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 facility 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 C. 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 are needed more frequently than others. Some of the commands you used are short and easy to type, while others involved several arguments and are more difficult to issue. There may be instances when you have to look up the correct command format or issue several commands in succession to perform an operation that would be much more convenient if it were done by only one command. Command procedures, written in 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 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 you 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. The desired commands are stored in a disk file, and then they are invoked 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 console; 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 2 interpreter processes the command statements read from the disk file.

If you use 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
```

you can rename 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 2 interpreter instead of performing some operation in the CMS environment. The &TRACE ON statement tells the EXEC 2 interpreter to display on your console any commands that it issues before they are executed. A &TRACE OFF statement suppresses this display of executed commands. A &TRACE ALL statement displays EXEC 2 control statements as well as commands that are executed.

In the CMS environment, where the EXEC 2 interpreter coexists with the CMS EXEC interpreter and the System Product 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. If the first word of the file is &TRACE, the EXEC 2 interpreter processes the file. If the first record of the file begins with a /*, the System Product interpreter processes the file. If neither case occurs, the CMS EXEC interpreter processes the file.

EXEC 2 control statements make it possible to conditionally interpret statements in an EXEC 2 file, to repeat the interpretation of statements, and to control the working of the EXEC 2 interpreter in various ways. The control statements make it possible to write EXEC 2 files that perform different operations depending on the arguments entered on the EXEC 2 command line or the results of commands issued from the EXEC 2 file. This is a very important concept, for it is this ability to modify the commands issued from an EXEC 2 file (and the order in which they are issued) which underlies the most useful features of EXEC 2 files.

EXEC 2 Variable Names

EXEC 2 variables and EXEC 2 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. They 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 2 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 to manipulate an indefinite number of variables by modifying the words in a few statements instead of writing all of the variable names explicitly.

Return Codes and EXEC 2 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 2 file, the return code generated by that command can be examined and used to control the subsequent interpretation of statements in the EXEC 2 file. For example, the ERASE command displayed above in RIPPLE EXEC yields 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 2 variable &RC. EXEC 2 variables are symbols used to refer to values that may change during the interpretation of an EXEC 2 file. You can use the symbol &RC in an EXEC 2 statement to refer to the return code generated by the most recent command issued from the EXEC 2 file. One way the &RC variable might be used in the RIPPLE EXEC file is to force termination of the EXEC 2 file (before renaming any files) if the X MEMO file does not exist. To do this, use the CMS command STATE 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 2 file if the value of &RC is greater than zero. Like CMS commands and user programs, EXEC 2 files also generate return codes. If an EXEC 2 file terminates because an end-of-file is reached and there are no more statements to interpret, the return code is zero. However, various errors detected by the EXEC 2 interpreter (invalid EXEC control word, nonexistent file, and so on) causes termination with a return code greater than 10000. Or, you may write the &EXIT control statement to terminate the EXEC 2 file with a specific return code, as shown above.

The ampersand character is used at the beginning of a word to signal the EXEC 2 interpreter that this word is an EXEC 2 variable or an EXEC 2 control word. When the EXEC 2 interpreter processes a statement from an EXEC 2 file, it begins by examining each word and replacing any EXEC 2 variables with their current values. (Later, we'll see exactly how this is done.) EXEC 2 control words are like EXEC 2 variables, except their values are initialized to their names by the EXEC 2 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 is handled in a special manner by the EXEC 2 interpreter. They are called "predefined variables" because the EXEC 2 interpreter assigns values to them automatically. Some of these predefined variables are given values only once, when the EXEC 2 interpreter starts processing a file (&FILENAME is such a variable, whose value is the name of the EXEC 2 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 2 command line and is updated when an EXEC 2 control statement redefines the set of argument variables.

EXEC 2 File Arguments

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

Let's 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 Reference* for the Responses, the Error Messages, and the Return Codes issued by CMS for the STATE command.)

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

```
RIPPLE RWR
```

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 2 file, you can keep as many old files as you desire. By using some additional EXEC 2 control statements, you could rename any number of files using only one RENAME statement, interpreting it as many times as necessary, each time with different arguments.

Conditional Interpretation of Statements

Before looking at more sample EXEC 2 files, let's examine the structure of the conditional (&IF) statement more closely and introduce some other EXEC 2 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 2 interpreter.) The complete &IF statement has the format:

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

where "comparator" is =, ¬=, >, <, >=, <=, EQ, NE, GT, LT, GE, NL, LE, or NG. 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
```

terminates an EXEC 2 file only if both conditions are true.

Statement Labels

You may attach a label to an EXEC 2 statement (including the null statement, which has no words in it) so that an EXEC 2 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 2 variables. References to labels, however, may involve EXEC 2 variables. The most frequent references to statement labels are &GOTO control statements, which modify the regular, sequential processing of an EXEC 2 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 finds the label, 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. You will receive a message if the label is not found.

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 "Chapter 3: Notes on EXEC 2" on page 59.) The function of the assignment statement is to make the EXEC 2 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 2 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 2 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 to 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 2 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 2 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 2 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 2 variable, which is then replaced by the 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 2 variables to build the names of other EXEC 2 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 ==> &. 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 of the line (remember that characters are examined from right to left). Therefore, the first word keeps its initial ampersand and remains an appropriate EXEC 2 variable name. 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 2 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 2 Variable Names

We are now ready to look at an EXEC 2 file that depends on this ability to use an EXEC 2 variable to build the names of other variables. The following EXEC, named LFN, 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 2 file. Because the number of filename arguments may differ from one use to

the next, the EXEC 2 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 2 file were invoked by the command

```
LFN NEW OLD
```

The first time the LISTFILE command is issued, the EXEC 2 variable &J has 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 arguments, 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 2 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 2 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 2 is not as pretty as it could be. This is because the LISTFILE command with the LABEL option produces a title line each time it is invoked and finds at least one file meeting its argument pattern. The LABEL option includes further information you may want about the file specified, for example, the label of the disk on which the file resides.

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
```

Since the initial value of &NOHEADER is null, it 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 2 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 the LFN EXEC. The &LOOP control statement 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 2 file. The statements to be repeated (the scope of the loop) were indented to make it easier to read the EXEC 2 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 2 interpreter determines how many statements to repeat:

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

The label defining the scope of the loop must occur before the end of the EXEC 2 file or an error is reported. If there is a statement on the same line as the label, the statement is executed. In this case, the assignment statement, &J = &J + 1, is the last line of the loop. 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 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 2 Files Interact with Users

The more EXEC 2 files you write, the more difficult it is to remember the correct formats of these new user commands. You can solve this difficulty by making these EXEC 2 files self-documenting; that is, whenever they are invoked with incorrect arguments, or with a question mark as an argument, the EXEC 2 files 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 that is self-documenting:

```
&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 2 interpreter to display the words following &PRINT as a line on the user's console. The EXEC 2 interpreter substitutes the appropriate values into the EXEC 2 variables before displaying the information.

The above version of LFN EXEC generates a nonzero return code, 28, in any instance where no files were found. Since the EXEC 2 variable &NOHEADER is already being used to detect a successful invocation of LISTFILE, you can use &NOHEADER to determine whether any files were found. If the value of &NOHEADER is null after all the LIFTFILE commands have been issued, no files were found. It is not possible to simply write

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

to determine the value of &NOHEADER. If &NOHEADER is null, a syntax error in the &IF statement occurs because the &NOHEADER word would disappear and you are left with

```
&IF ¬= NOHEADER &EXIT 28
```

A solution for testing the value of an EXEC 2 variable that might be null is to use some prefix character 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 that can result after substituting for the variable &NOHEADER are:

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

or

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

For success, &NOHEADER = NOHEADER; for failure, &NOHEADER is null.

All of the previous EXEC 2 files have used only the arguments provided on the command line to determine what function they would perform.

You can also write an EXEC that interacts with you - displaying prompting messages on the console and reading instructions or values that are typed in. Before showing an example with this interaction, let's discuss the &READ control statements.

Data is read from the console using the &READ control statement. A &READ statement may read one input line and assign it to a single EXEC 2 variable:

```
&READ STRING &S
```

&S contains 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 2 variable:

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

The first word of the input line is assigned to the variable &FIRST, the second word is assigned to the variable &SECOND, etc. 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
```

This statement assigns the words in the input line to the EXEC 2 variables &1 &2 &3 ... etc. The predefined variable &N is assigned to 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 2 argument variables that may be needed later to other EXEC 2 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 2 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 2 statements using the form:

```
&READ n
```

"n" is the number of lines to read. If no explicit number of lines is given, only one line is 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 2 statements by using the file:

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

which reads statements from your console and traces their interpretation.

Here is a modified version of the LFN EXEC. It interacts with you and contains the &READ control statement.

```
&TRACE
&PRINT ENTER THE FILENAME YOU ARE INTERESTED IN
&PRINT OR PRESS ENTER TO EXIT.
&PRINT
&READ ARGS
-LOOP
  &PRINT
  LISTFILE &1 * * (LABEL
  &PRINT
  &IF &RC = 28  &PRINT THIS FILE DOES NOT EXIST.
  &PRINT ENTER ANOTHER FILENAME YOU ARE INTERESTED IN
  &PRINT OR PRESS ENTER TO EXIT.
  &PRINT
  &READ ARGS
  &IF &N ¬= 0 &GOTO -LOOP
&EXIT
```

The first two print statements tell you what information you must input. The statement, &PRINT, just leaves a blank space on the console. This is just to make the screen neater.

&READ ARGS reads the filename you entered and assigns it to the variable &1.

-LOOP is a label signalling the beginning of the loop.

The CMS LISTFILE command displays information about the file that is specified in variable &1. The LABEL option includes further information you may want to know about the file specified, for example, the label of the disk on which the file resides.

The statement, &IF &RC = 28, checks the return code from the LISTFILE command. If the return code equals 28, you receive a message that the file entered does not exist.

The next two &PRINT statements ask you if you want to inquire about any other file.

&READ ARGS again reads the filename or null character you entered and assigns it to &1.

The statement, &IF &N ¬= 0 &GOTO -LOOP, checks if you entered another filename or a null character. &N is the number of arguments set. Therefore, if &N = 0, no filename was entered and you exit the EXEC.

Using the &CASE Control Statement

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 2 interpreter, reads a console input line, it chooses whether or not to translate to uppercase. The EXEC 2 control statement

```
&CASE M
```

instructs the EXEC 2 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 2 interpreter starts processing an EXEC 2 file.

Here is an example to show you how the &CASE control statement works.

```
&TRACE
&TYPE ENTER YOUR NAME:
&CASE M
&READ VARS &NAME
&TYPE &NAME
&EXIT
```

The above EXEC prompts you to enter your name. If you enter your name using uppercase and lowercase characters, such as:

```
Sue
```

the result is:

```
Sue
```

However, if the "&CASE M" control statement is removed and you enter your name:

```
Sue
```

the result is:

```
SUE
```

Appendix D. Writing Editor Macros

The macro language is one of the most powerful facilities that the editor provides. By writing macros, you can:

- Expand the basic subcommand language
- Expand the prefix subcommand language
- Tailor the language to your own application
- Eliminate repetitive tasks.

This chapter explains how to write an XEDIT macro using the EXEC 2 language.

What is an XEDIT Macro?

An XEDIT macro is an EXEC file that is invoked from the XEDIT environment.

(A macro may also be written using the Restructured Extended Executor (REXX) language. However, all examples in this chapter use the EXEC 2 language.)

You execute a macro the same way you execute XEDIT subcommands: type the macro name on the command line (or the prefix area) and press the ENTER key. A macro may be executed by entering only its name (or synonym). The execution of the macro may also depend on arguments you enter when it is invoked.

A macro file can contain:

- XEDIT subcommands
- EXEC 2 control statements
- CMS and CP commands.

Creating a Macro File

Because an XEDIT macro is a normal CMS file, it may be created in any of the ways that CMS provides for file creation. It can even be created dynamically, by using the XEDIT multiple file editing capability (see *VM/SP System Product Editor User's Guide*). As soon as a FILE subcommand is executed for the macro file, the macro can be used.

Like any CMS file, a macro file is identified by filename, filetype, and filemode. The file identifier for a macro file must follow certain rules:

- For macros entered from the command line, the filename is a string of one to eight alphameric characters. This name is used to invoke the macro. For example, if the filename is SEND, entering "SEND" during an editing session causes the macro to be executed.

  Prefix macro filenames may be one to eight characters, but they may not contain numbers. (Because the prefix area is only five positions long, you can define a synonym for a prefix macro filename that is longer than five characters. For more information on defining synonyms for prefix macros, see *VM/SP System Product Editor Command and Macro Reference*.)

- The filetype must be XEDIT.

- The filemode can specify any of your accessed disks, for example, A1.

Using XEDIT Subcommands in a Macro

A macro can contain any XEDIT subcommand, with the following exceptions: prefix macros cannot contain READ, QUIT, FILE, and LPREFIX. However, some subcommands perform functions that are meaningful only in the context of a macro, for example, one that passes information to the EXEC 2 interpreter.

When EXEC 2 interprets a file not having a filetype of EXEC, it starts with a &SUBCOMMAND presumption of the filetype, in this case XEDIT. Therefore, you do not have 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 XEDIT 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 XEDIT macro:

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

Note that you can 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 XEDIT 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 you want to use the REPSK XEDIT file for lines starting with .SK 2, or .SK 3, or .sp 3, etc. You can use two arguments to define the lines you 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 translates input commands into uppercase before

invoking an XEDIT 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 may "RESPECT" or "IGNORE" the 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. 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, 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 XEDIT command you want, with blanks exactly where you want them, as the value of an EXEC 2 variable. Then, the FIND command is represented as a single word, and you avoid any difficulties stemming from the combination of several words to form a command. To build the FIND command, use the predefined function &CONCAT 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 you need one blank to separate the FIND XEDIT command from its operand, that blank is included by explicitly 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.

Avoiding Name Conflicts

Use the MACRO subcommand to cause the editor to execute a specified macro without first checking to see if a subcommand of the same name or a synonym exists. (This cannot be used for prefix macros.)

When a subcommand has a number as its operand, a blank is not required between the subcommand name and the operand. For example, both "NEXT8" and "N8" are interpreted by the editor as being the subcommand "NEXT 8". Therefore, if a macro name were also "N8", the macro would not be executed; the subcommand "NEXT 8" would be executed instead. To execute the macro, you could enter the following:

```
MACRO N8
```

The macro whose name is "N8" would then be executed.

The SET MACRO subcommand can be used to control the order in which the editor searches for subcommands and macros. SET MACRO ON tells the editor to look for macros before it looks for subcommands; SET MACRO OFF reverses the order.

Walking Through An XEDIT Macro

The following XEDIT macro, whose filename is GLOBCHG, is an example of a macro you may write to make life a little easier. The application is typical of a text processing file arrangement, where many SCRIPT files are imbedded in a master file, via the SCRIPT control word ".IM".

The problem with this type of setup is that if you have to make a global change throughout all the files, you have to edit each file, make the change, and then file each file.

When issued from the master file, the GLOBCHG macro edits each file, performs a global change, and files it.

The macro is invoked by entering the macro name, GLOBCHG; the arguments passed to the macro are the old data and the new data, enclosed in delimiters:

```
GLOBCHG /string1/string2/
```

For example, if a file called MASTER SCRIPT contains:

```
.IM FILE1
.IM FILE2
.
.
.
.IM FILE100
```

and the following commands are issued:

```
XEDIT MASTER SCRIPT
GLOBCHG /WAR AND PEACE/SENSE AND NONSENSE/
```

"WAR AND PEACE" is changed to "SENSE AND NONSENSE" each time it occurs in every file. (In this macro, no attempt is made to execute the change on files that may be imbedded at the next level.)

The GLOBCHG macro can also be used to delete data throughout the files, by changing a string to a null string. For example:

```
GLOBCHG /bad data//
```

The following is a listing of the macro, whose fileid is GLOBCHG XEDIT A1. After the listing, each line in the macro is explained.

```
00001 ***************************************************************
00002 **     ENTER THE OLD STRING YOU WANT CHANGED AND THE      **
00003 **     STRING YOU WANT IT CHANGED TO.  ENTER IT IN        **
00004 **     THE FORM:                                          **
00005 **         /STRING1/STRING2/                              **
00006 ***************************************************************
00007 &IF &N = 0 &GOTO -MISSING
00008 &OPERAND = &ARGSTRING
00009 PRESERVE
00010 SET MSGMODE OFF
00011 TOP
00012 FIND .IM
00013 &IF &RC ¬= 0 &GOTO -NOIMBED
00014 -LOOP
00015 STACK 1
00016 &READ ARGS
00017 &COMMAND STATE &2 SCRIPT *
00018 &IF &RC = 0 &SKIP 2
00019   &TYPE IMBEDDED FILE ' &2 SCRIPT ' DOES NOT EXIST; BYPASSED
00020   &GOTO -ENDLOOP
00021 XEDIT &2 SCRIPT (NOPROFILE
00022 EXTRACT /FNAME/FTYPE/FMODE/
00023 &TYPE PROCESSING FILE ' &FNAME.1 &FTYPE.1 &FMODE.1 '
00024 CHANGE &OPERAND * *
00025 &IF &RC ¬= 0 &TYPE NO CHANGES OCCURRED IN ' &FNAME.1 &FTYPE.1 '
00026 FILE
00027 -ENDLOOP FIND .IM
00028 &IF &RC = 0 &GOTO -LOOP
00029 RESTORE
00030 &EXIT
00031 -NOIMBED
00032 RESTORE
00033 EMSG NO IMBED FOUND.
00034 &EXIT
00035 -MISSING  EMSG EXE545E MISSING OPERAND(S)
00036 CMSG 0
00037 &EXIT
```

Figure 1. A Sample Macro: GLOBCHG

```
00001 ***************************************************************
00002 **     ENTER THE OLD STRING YOU WANT CHANGED AND THE      **
00003 **     STRING YOU WANT IT CHANGED TO.  ENTER IT IN        **
00004 **     THE FORM:                                          **
00005 **         /STRING1/STRING2/                              **
00006 ***************************************************************
```

Statements 1 through 6 are comments describing the arguments to this EXEC.

```
00007 &IF &N = 0 &GOTO -MISSING
```

If the number of arguments you passed to the macro (&N) is zero, go to the statement labeled "-MISSING" where an error message is issued. Obviously, this macro cannot work unless you tell it what to change.

```
00008 &OPERAND = &ARGSTRING
```

The user-defined variable (&OPERAND) is assigned the value of the argument string (&ARGSTRING) passed to the macro. (The argument string contains the old data string you want changed and the new data string you want it changed to.)

The next four statements in the macro are XEDIT subcommands:

```
00009 PRESERVE
```

This subcommand saves the editor settings until a subsequent RESTORE subcommand is issued (statement 32).

```
00010 SET MSGMODE OFF
```

No messages will be displayed. By turning the message mode on and off, you can select which messages you want displayed. Message mode is set OFF here to prevent messages from the FIND subcommand (statement 12) from being displayed, because the macro issues its own message (statement 33) if no imbedded files are found.

```
00011 TOP
```

Moves the line pointer to the top of the master file - the file where the macro was invoked.

```
00012 FIND .IM
```

Searches forward in the master file for the first line that contains ".IM" in column 1; that is, locate the first line that imbeds a file.

```
00013 &IF &RC ¬= 0 &GOTO -NOIMBED
```

If there is a non-zero return code from the FIND subcommand (statement 12), go to the statement labeled "-NOIMBED". This situation occurs if no ".IM" statements are found in the master file.

Statements 14 through 26 are the major loop in the macro. The global change is made on each imbedded file in this loop.

```
00014 -LOOP
```

This is the statement label that begins the loop.

```
00015 STACK 1
```

When the FIND subcommand (statement 12) locates a ".IM filename" statement in the master file, it makes that line the current line. This STACK subcommand places the current line in the program stack, so that its contents can be read by the following statement.

```
00016 &READ ARGS
```

This statement reads a line from the program stack and assigns the arguments to &1, &2, &3, ...

```
00017 &COMMAND STATE &2 SCRIPT *
```

The STATE command is a CMS command that verifies the existence of a file. This statement checks to see if the file named in the ".IM filename" statement exists. (EXEC 2 transmits the STATE command directly to CMS.)

```
00018 &IF &RC = 0 &SKIP 2
```

If the return code from the STATE command is zero, the file exists. Therefore, skip down to statement 21. If it is not zero, execute the next two statements (19-20), which comprise "file not found" processing.

```
00019   &TYPE IMBEDDED FILE ' &2 SCRIPT ' DOES NOT EXIST; BYPASSED.
00020   &GOTO -ENDLOOP
```

Statement 19 issues a message - the file, &2 SCRIPT, imbedded in the master file does not exist. Statement 20 branches to the statement label that begins the FIND loop again.

```
00021 XEDIT &2 SCRIPT (NOPROFILE
```

The XEDIT subcommand brings the imbedded file into virtual storage. The NOPROFILE option forces the editor not to execute the default PROFILE XEDIT macro.

```
00022 EXTRACT /FNAME/FTYPE/FMODE/
```

This form of the EXTRACT command places the filename, filemode, and filetype of the imbedded file into variables: &FNAME.1, &FTYPE.1, &FMODE.1, respectively. See "Using the XEDIT EXTRACT Subcommand" for further information on the EXTRACT subcommand.

```
00023 &TYPE PROCESSING FILE ' &FNAME.1 &FTYPE.1 &FMODE.1 '
```

Displays a message to let you know which file is being processed.

```
00024 CHANGE &OPERAND * *
```

The global change on one of the imbedded files. (The argument string you entered when the macro was invoked was assigned to &OPERAND in line 8. The change statement is in the form: CHANGE /string1/string2/ * *)

```
00025 &IF &RC ¬= 0 &TYPE NO CHANGES OCCURRED IN ' &FNAME.1 &FTYPE.1 '
```

If the change occurs, the return code from the CHANGE subcommand is 0. This statement checks the return code. If the return code is 0, a message is printed.

```
00026 FILE
```

The changed file is written to disk.

```
00027 -ENDLOOP FIND .IM
```

Then, the editor resumes editing the master file, searching for the next ".IM filename" statement.

```
00028 &IF &RC = 0 &GOTO -LOOP
```

If the FIND (statement 27) is successful, go through the loop again.

```
00029 RESTORE
```

If the FIND (statement 27) is not successful, restore the settings of XEDIT variables to the values they had when the PRESERVE subcommand was issued (statement 9).

```
00030 &EXIT
```

Return control to the editor; you can then issue a QUIT subcommand for the master file.

```
00031 -NOIMBED
00032 RESTORE
00033 EMSG NO IMBED FOUND.
00034 &EXIT
```

Statements 31 through 34 are executed if no ".IM" statements were found in the master file.

```
00035 -MISSING  EMSG EXE545E MISSING OPERAND(S)
```

This message is displayed in the message line if no arguments, which are required, were entered when the macro was invoked. It is a branch from statement 7.

```
00036 CMSG 0
```

In addition, the macro name (GLOBCHG) is displayed in the command line, so that you can type the arguments (string1 string2) and press the ENTER key to invoke the macro again.

```
00037 &EXIT
```

The end.

Using the XEDIT EXTRACT Subcommand

Notice in line 23 of Figure 1 on page 106 the variables &FNAME.1, &FTYPE.1, and &FMODE.1 appear. These variables are created as a result of the XEDIT EXTRACT subcommand. The EXTRACT subcommand is an extended form of the XEDIT TRANSFER subcommand. Let's compare the two subcommands.

The statement

```
TRANSFER FNAME FTYPE FMODE
```

puts the filename, filetype, and filemode of the file being edited into the program stack.

The statement

```
EXTRACT /FNAME/FTYPE/FMODE/
```

puts the filename, filetype, and filemode of the file being edited into the newly created variables &FNAME.1, &FTYPE.1, and &FMODE.1, respectively.

The EXTRACT subcommand creates similar variables for all operands available to the subcommand. See *VM/SP System Product Editor Command and Macro Reference* for more details on the EXTRACT subcommand.

Writing Prefix Macros

You can write prefix macros for a variety of purposes - from performing a function from the prefix area that is normally accomplished by entering a subcommand on the command line to creating an entirely new function.

What Information is Passed to the Macro?

An argument string is automatically passed to a prefix macro when it is invoked. It can supply a macro with information critical to its execution, like the line number of the prefix area in which the macro was entered.

The format of the argument string is as follows:

```
PREFIX  SET|SHADOW|CLEAR  pline  [op1[op2[op3]]]
```

where:

PREFIX
: indicates that this is a prefix call.

SET
: indicates that the prefix macro was entered on some line in the file displayed.

SHADOW

indicates that a prefix macro was entered on a shadow line (see SET SHADOW in the *VM/SP System Product Editor Command and Macro Reference*).

CLEAR

indicates that a new prefix subcommand or macro or new blank area replaces a previously pending prefix subcommand or macro on the same line, or the RESET subcommand was entered. In this case, this macro is invoked with "PREFIX CLEAR pline".

pline

is the line number on which the prefix macro was entered.

op1 op2 op3

are the optional operands of the macro, entered either to its left or right (for example, 5M or M5).

Creating a Sample Prefix Macro

Let's create a prefix macro, with filename U and filetype XEDIT, that translates one or more lines in a file to uppercase, which normally is accomplished by issuing the UPPERCAS subcommand in the command line. When U is entered in the prefix area of a line, that line is translated to uppercase. A number may be specified before or after the U to translate more than one line; for example, 3U=== or =U5==.

The U prefix macro may look like this:

```
00001 &PLINE  = &3
00002 &OP     = &4
00003 &IF .&OP = .  &OP = 1
00004 COMMAND :&PLINE UPPERCAS &OP
00005 &EXIT
```

Figure 2. A Sample Prefix Macro: U

```
00001 &PLINE  = &3
00002 &OP     = &4
```

Lines 1 and 2 assign the arguments to specific variables. &3 is set to the line number the prefix macro was entered on, and &4 is set to any operand that is passed.

```
00003 &IF .&OP = .  &OP = 1
```

Line 3 determines if an operand was entered. If the operand is null, a default of 1 is assumed.

```
00004 COMMAND :&PLINE UPPERCAS &OP
```

Line 4 uses :&PLINE to make the line in which the prefix macro was entered (&PLINE) the new current line, and then issues the UPPERCAS subcommand with the operand, &OP. &OP is the number of lines to be put in uppercase.

For example, if "U8" was entered in the prefix area of line 3 of a file, *&0* would be "U", *&PLINE* would be "3", and *&OP* would be "8". Then, the next eight lines, including the current line, would be put in uppercase.

Current Line Positioning

Note that in line 6, &PLINE is an absolute line number target. It is used to make the prefix line the current line because the UPPERCAS subcommand translates all lowercase characters to uppercase, starting at the current line.

When a prefix macro is finished executing, the current line is returned automatically to the line that was current when it began execution. Therefore, even though line &PLINE is made current for the UPPERCAS subcommand, the macro need not restore the current line.

Appendix E. Useful EXEC 2 Techniques

The following illustrations exhibit solutions to some EXEC programming problems. 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.

1. The statement

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

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

2. 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.

3. 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 *
   ```

4. Suppose a multi-way branch is desired, based on an argument value supplied by the caller and currently in &1. However, the value of &1 must first be tested to verify it is valid -- that is, its value is either A, B, or C. (You can expand this example to handle more than three cases.)

```
&TRACE
& = &POSITION OF &1 A B C
&IF & ¬= 0 &GOTO -&1
&TYPE INVALID CASE: &1
&EXIT
-A  &TYPE THIS IS CASE A
.
.
.
&EXIT
-B  &TYPE THIS IS CASE B
.
.
.
&EXIT
-C  &TYPE THIS IS CASE C
.
.
.
&EXIT
```

5. 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.

6. 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 therefore.

   ```
   &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.

7. 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 line constructed a string from the argument values &1 &2 ... &&N that are separated by blanks, and the second line stacks the string. 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 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 &X1, &X2, ..., &X&N. 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
```

8. 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 modification detects 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.

9. 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 program stack. The first line in the program stack 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 in the program stack 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
```

Summary of Changes

Summary of Changes
for SC24-5219-3
for VM/SP Release 5

Miscellaneous

No technical changes have been made to this book.

The contents of the front and back matter of this book has been changed to make it compatible with the rest of the VM/SP library (for example, the Table of Contents now begins on page v throughout the library).

Summary of Changes
for SC24-5219-2
for VM/SP Release 3

EXECOS

The CMS EXECOS command can be invoked within an EXEC for OS reset and VSAM clean-up.

Miscellaneous

The appendix comparing the CMS EXEC language and the EXEC 2 language has been removed.

Examples have been added to the "Control Statements" section and the "Predefined Functions" section.

A new appendix, "Appendix D: Writing Editor Macros," has been added. It describes how to write XEDIT macros and XEDIT prefix macros.

"Chapter 4: BNF Description of the EXEC 2 Syntax," has been expanded.

Minor technical and editorial changes have been made throughout this publication.

Summary of Changes
for SC24-5219-1
for VM/SP Release 2

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.

Bibliography

Prerequisite Publications:

Virtual Machine/System Product: Introduction, GC19-6200

Corequisite Publications:

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

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

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

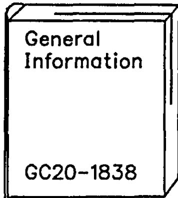
Virtual Machine/System Product: System Product Editor User's Guide, SC24-5220

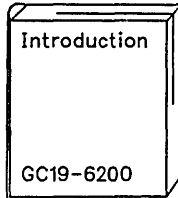
Virtual Machine/System Product: System Product Editor Command and Macro Reference, SC24-5221

System/370 Reference Summary, GX20-1850

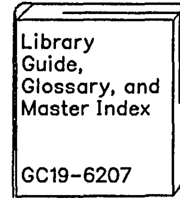
The VM/SP Library (Part 1 of 3)

Evaluation

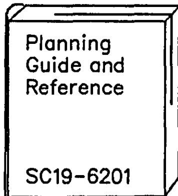


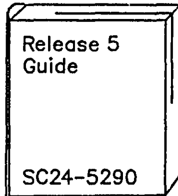


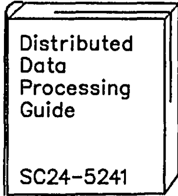
Index



Planning

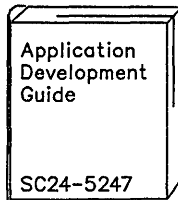


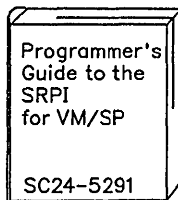




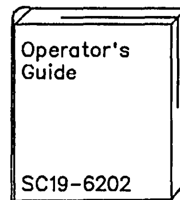
Installation

Applications



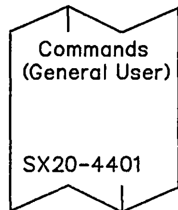


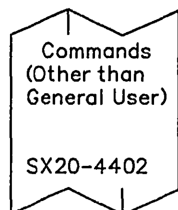
Operation

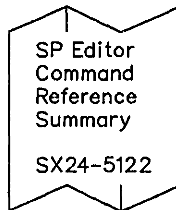


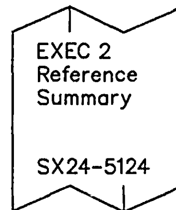
Reference Summaries

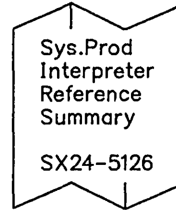
To order all of the Reference Summaries, use order number SBOF-3242

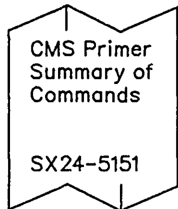


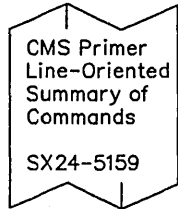


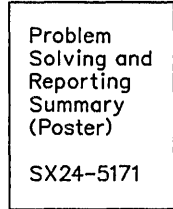


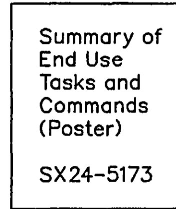






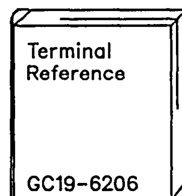


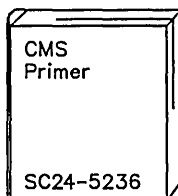


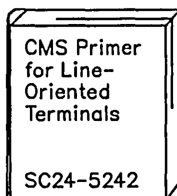


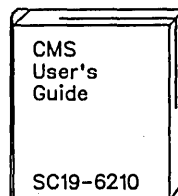
The VM/SP Library (Part 2 of 3)

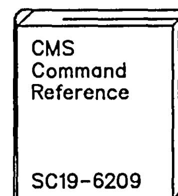
End Use

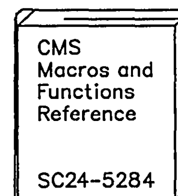


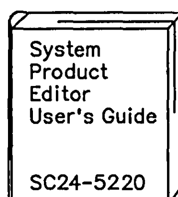


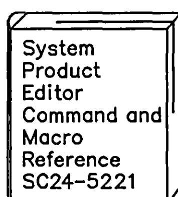


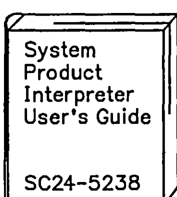


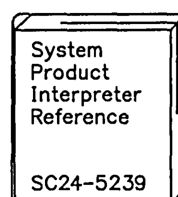


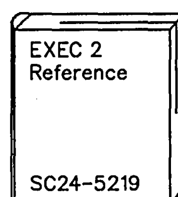


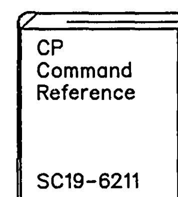


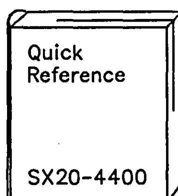




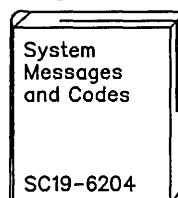


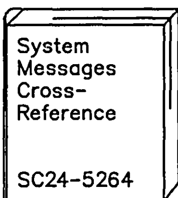


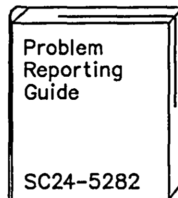


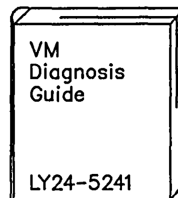


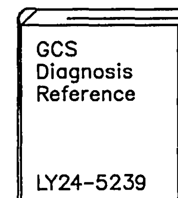
Diagnosis

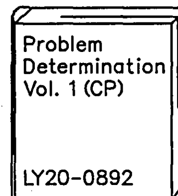


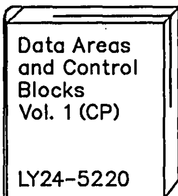


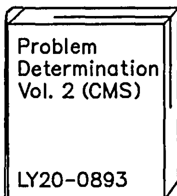


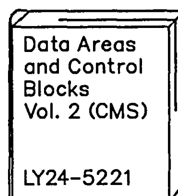


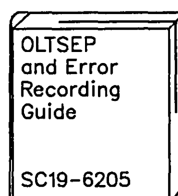


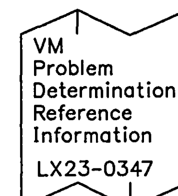


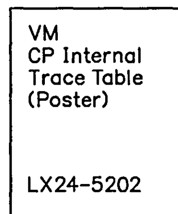






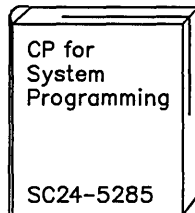


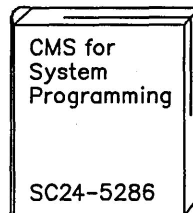


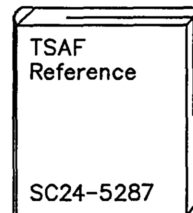


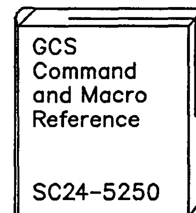
The VM/SP Library (Part 3 of 3)

Administration

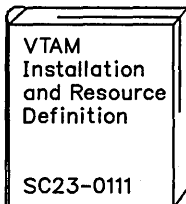


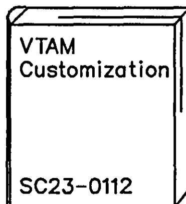


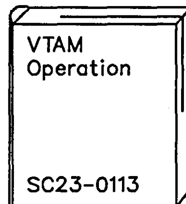


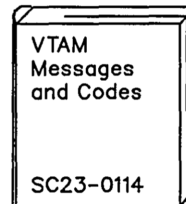


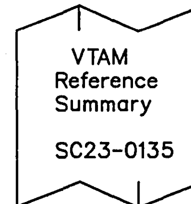
Auxiliary Communication Support

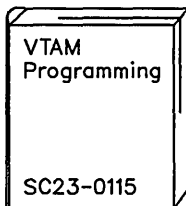


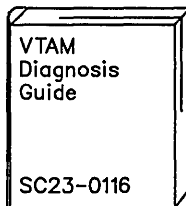


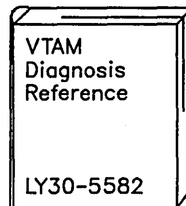


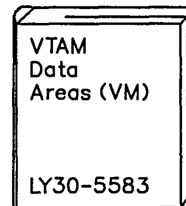


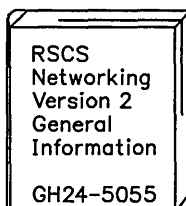


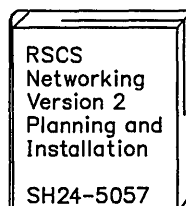


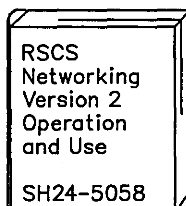


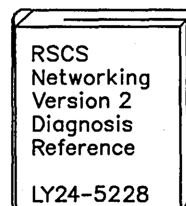


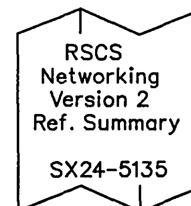


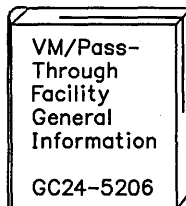


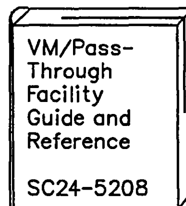


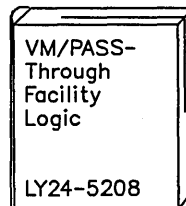












Index

Special Characters

& 5
&0 5
&ARGS EXEC2 control statements 5, 9
embedded blanks 64
&ARGSTRING EXEC2 control statement 5, 77
embedded blanks 64
&BEGPRINT EXEC2 control statement 10
truncation column 10, 68
&BEGSTACK EXEC2 control statement 12
first-in/first-out (FIFO) 12
last-in/first-out (LIFO) 12
truncation column 12, 68
&BEGTYPE EXEC2 control statement 10
truncation column 10, 68
&BLANK EXEC2 control statement 6
embedded blanks 64
&BUFFER EXEC2 control statement 14
&CALL EXEC2 control statement 15
label search 66
&CASE EXEC2 control statement 17
&CMDSTRING 6, 77
&COMLINE 6
&COMMAND EXEC2 control statement 18
&PRESUME 18, 26
&CONCAT OF 41
&CONCATENATION OF 41
&DATATYPE OF 42
&DATE 6
evaluation 62
Greenwich Mean Time (GMT) 6
&DEPTH 6
&DIV OF 43
&DIVISION OF 43
&DUMP EXEC2 control statement 19
&ERROR 20
&EXIT 21
&FILEMODE 6
&FILENAME 7
&FILETYPE 7
&FROM 7
&GOTO 22
label search 66
&IF 23
comparands 23
comparatives 23
conditional statements 91
&INDEX 7
&LEFT OF 44
embedded blanks 64
&LENGTH OF 45
&LINE 7
&LINENUM 7
&LINK 7
&LITERAL OF 46
embedded blanks 64
&LOCATION OF 47
&LOOP 24, 95
closing 65
label search 66
&MULT OF 48
&MULTIPLICATION OF 48
&N 7
&PIECE OF 49
&POSITION OF 50
&PRESUME 26
&COMMAND 18, 26
&SUBCOMMAND 26, 34
&PRINT 27
&RANGE OF 51
embedded blanks 64
&RC 8
&READ 28
&TRUNC 29, 38
ARGS 28
embedded blanks 64
examples 97
n,1,* 28
STRING 28
VARS 28
&RETCODE 8
&RETURN 31
&RIGHT OF 52
embedded blanks 64
&SKIP 32
&STACK 33
first-in/first-out (FIFO) 33
last-in/first-out (LIFO) 33
&STRING OF 53
embedded blanks 64
&SUBCOMMAND 34
&PRESUME 26, 34
&SUBSTR OF 49
&TIME 8
evaluation 62
Greenwich Mean Time (GMT) 8
&TRACE 35
* 36
ALL 36, 67
ERR 35
example 67
OFF 36
ON 35
output-action 36
&TRANS OF 54
embedded blanks 64
rules for modification 54
&TRANSLATION OF 54
embedded blanks 64

rules for modification 54
&TRIM OF 55
&TRUNC 29, 38
truncation column 38, 68
&TYPE 27
&TYPE OF 42
&UPPER 39
&WORD OF 56
&1 &2 ... 5
&ARGS 5, 9, 28
&READ ARGS 5, 28
arguments 2, 5, 9, 90
embedded blanks 64
"in memory files" 80

A

arguments 90
&1 &2 ... 2, 5, 9
assembler language programs 79-82
SVC 202 calls 79, 82
tokenized plist 80
untokenized plist 80
assigning arguments 9
assignment statements
description of 2, 62, 92
example 62, 92
assignments
See assignment statements

B

bibliography 121
BNF syntax 69

C

calling subroutines 15
case translation 17, 39
changes, summary of 119
CHAR data type 42
CMDCALL 77
CMS (Conversational Monitor System) 77-86
CMS (Conversational Monitor System) limits 78
&EXIT return codes 78
&TRACE 78
console 78
console stack 78
filename 78
line length 78
lookaside buffer 78
NUMERIC OVERFLOW 78
numeric values 78
printed line length 78
statement length 78
word length 78
column location 47
combining words 41, 53
commands 2, 3, 89
comments 1
concatenating words 41
conditional statements 91
&IF control statement 23, 63
&LOOP control statement 63
example 63
syntax 63
console stack
See program stack
control statements
&ARGS 9
&BEGPRINT 10
&BEGSTACK 12
&BEGTYPE 10
&BUFFER 14
&CALL 15
&CASE 17
&COMMAND 18
&DUMP 19
&ERROR 20
&EXIT 21
&GOTO 22
&IF 23
&LOOP 24
&PRESUME 26
&PRINT 27
&READ 28
&RETURN 31
&SKIP 32
&STACK 33
&SUBCOMMAND 34
&TRACE 35
&TRUNC 38
&TYPE 27
&UPPER 39
description of 2, 4, 8
control words
examples 2
conventions iv
Conversational Monitor System (CMS) 77-86
Conversational Monitor System (CMS) limits 78
&EXIT return codes 78
&TRACE 78
console 78
console stack 78
filename 78
line length 78
lookaside buffer 78
NUMERIC OVERFLOW 78
numeric values 78
printed line length 78
statement length 78

word length 78
current line positioning 112

D

datatype of a word 42
debugging the EXEC 2 interpreter 74
delimiters
parenthesis 5
space 5
dividing numbers 43
DMSEXE085E 73
DMSEXE175E 73
DMSEXE255T 74

E

editor macros 83, 101-104
executing 83
filetype 83
implementation 101
embedded blanks
discussion of 64
examples 64, 104
exceptions 64
handling 104
variables 64
errors
DMSEXE085E 73
DMSEXE175E 73
DMSEXE255T 74
messages 73
setting the action taken 20
evaluation of &DATE and &TIME 62
examples
&BLANK 64
assembler language programs 79-82
assignment statement 62
conditional statements 63
control words 2
generating EXEC 2 variable names 93
labels 2
leading zeros 63
name substitution 60
plus signs 63
programming techniques 113-117
SVC 202 79
tokenized plist 80
untokenized plist 80
user-defined functions 57
variable 2
exceptions
embedded blanks 64
EXEC 2 words 67
EXEC 2 errors 73
EXEC 2 files
filename, valid character for 78
filetype 1, 83
format 1
identifying 77
recursive execution 61
sample of 75
terminating 61
EXEC 2 in CMS 77-86
assembler language programs 79-82
EXECCOMM 83
identifying EXEC 2 files 77
limits in CMS 78
XEDIT macros 83
EXEC 2 interpreter
as a macro processor 82
invoked 1
EXEC 2 language 1
EXEC 2 parameter lists 79
EXEC 2 programs
assembler language programs 79-82
calling 77
EXEC 2 file 1
EXEC 2 interpreter 1
executing 1
interaction with users 95
EXEC 2 statements
comment 1
executable statement 1
EXECCOMM 83
FETCH 79
length limit for external names of shared variables 79
length limit for values assigned by 79
sharing EXEC 2 variables with assembler programs 83
STORE 79
EXECOS 79
executable statements
assignment statement 2
assignments 3
commands 2, 3
control statements 2, 4
description of 1
interpreting 4
null statement 2, 3
types 3
exit from an EXEC 2 file 21
extracting words from a string of words 56
extracting words from other words 49

F

FIFO (first-in/first-out) 33
function invocation
    predefined function 40
    user-defined functions 56
functions
    predefined 40-56
    user-defined 56

I

interpreting executable statements 4
introduction 1
issuing commands to the given subcommand environment 26, 34
issuing commands to the host system 18, 26

J

justified words
    left-justified 44
    right-justified 52

L

label
    description of 92
    example 2
    performance 66
    search 66
leading zeros
    example 63
    removing 63
left-justified 44
length limits 78
length of words, finding 45
LIFO (last-in/first-out) 33
limits for EXEC 2 files in CMS 78
literal string, evaluating 46
locating
    a word in a string of words 50
    starting column of a word in another word 47
lookaside buffer 14
looping 24, 65

M

messages
    DMSEXE085E 73
    DMSEXE175E 73
    DMSEXE255T 74
    return codes 73
mixed case data 17, 99, 103
multiplying numbers 48

N

name substitution
    examples 60
    steps 59
notational conventions iv
notes on EXEC 2 59-68
    &LOOP statement 65
    &TRACE ALL 67
    assignment statement 62
    closing loops 65
    conditional phrases 63
    embedded blanks 64
    evaluation of &DATE and &TIME 62
    label search 66
    leading zeros 63
    numbers 62
    plus signs 63
    program stack 61
    recursive execution 61
    reserved words 66
    termination 61
    truncation column 68
null statement 2, 3
NUM data type 42
numbers
    dividing 43
    multiplying 48
    range 78
    size and treatment 62

P

parameter list 79
passing arguments 9
plus signs
    example 63
    removing 63
position of a word in a string of words 50
predefined functions 40-56
    &CONCAT OF 41
    &CONCATENATION OF 41
    &DATATYPE OF 42

&DIV OF 43
&DIVISION OF 43
&LEFT OF 44
&LENGTH OF 45
&LITERAL OF 46
&LOCATION OF 47
&MULT OF 48
&MULTIPLICATION OF 48
&PIECE OF 49
&POSITION OF 50
&RANGE OF 51
&RIGHT OF 52
&STRING OF 53
&SUBSTR OF 49
&TRANS OF 54
&TRANSLATION OF 54
&TRIM OF 55
&TYPE OF 42
&WORD OF 56
format of 40
reserved words 66
predefined variables
& 5
&0 5
&ARGSTRING 5
&BLANK 6
&CMDSTRING 6
&COMLINE 6
&DATE 6
&DEPTH 6
&FILEMODE 6
&FILENAME 7
&FILETYPE 7
&FROM 7
&INDEX 7
&LINE 7
&LINENUM 7
&LINK 7
&N 7
&RC 8
&RETCODE 8
&TIME 8
&1 &2 ... 2, 5
description of 90
reserved words 66
prefix macros
current line positioning 112
description of 110
sample of 111
writing 110
Primer 87
&CASE control statement 99
&LOOP control statement 95
assignment statements 92
conditional statements 91
embedded blanks 104
file arguments 90
function of EXEC 2 language 87
implementation of editor macros 101
labels 92
looping 95
return codes 89
translating to uppercase 99
user interaction 95
variables
evaluation 93
names 89
printing lines 10, 19, 27
program stack
&BEGSTACK 12
&STACK 33
description of 61
using 61
programming techniques
examples 113-117

R

reading lines 28
recursive execution 61
removing plus signs and leading zeros 63
repeating lines 24
reserved words
predefined functions 66
predefined variables 66
return codes 73, 89-90
right-justified 52

S

sample EXEC 2 files 75
searching for a word in a string or words 50
searching for a word in another word 47
sharing EXEC 2 variables with assembler language programs 83
skipping lines 32
stacking lines 12, 33
stopping execution 21
subroutines
calling 15
returning 31
substituting variables 59
summary of changes 119
SVC 202 call
example 79
SUBCOM function 82
syntax
BNF description 69
conditional statements 63
predefined functions 40
user-defined functions 56

returning to 31

V

variables
embedded blanks 64
evaluation 93
examples 2
EXEC 2 variables 89
names 89, 93

W

WHILE keyword 24
word
definition of 1
reserved 66

X

XEDIT macros in EXEC 2
avoiding name conflicts 105
creating 102
defining 83, 101
executing 83
filetype 83
sample XEDIT macro 105
using 102
XEDIT EXTRACT subcommand 110
XEDIT prefix macros
current line positioning 112
description of 110
sample of 111
writing 110

T

terminating EXEC 2 file 61
tokenized plist
example 80
tracing 35
commands 35
commands that yield nonzero return codes 35
every executable statement 36
transferring control 15, 22, 31, 32
translating characters to other characters 54
translating to uppercase 17, 39, 77, 99
truncating
limits 78
lines 38, 68
types of executable statements
assignments 2, 3
commands 2, 3
control statements 2, 4
null statement 2, 3
typing lines 10, 19, 27

U

UNTIL keyword 24
untokenized plist
"in-memory file" 80
example 80
uppercase data 17, 77, 99
user interaction 95-99
user-defined functions
description of 57
examples 57, 58
form of 57
invocation 57
label search 57

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

File No. S370/4300-39
Printed in U.S.A.

SC24-5219-3

IBM ®

VM/SP EXEC 2 Reference
Order No. SC24-5219-3

READER'S COMMENT FORM

Is there anything you especially like or dislike about this book? Feel free to comment on specific errors or omissions, accuracy, organization, or completeness of this book.

If you use this form to comment on the online HELP facility, please copy the top line of the HELP screen.

______________ ______________ _______ **Help Information line ___ of ___**

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you, and all such information will be considered nonconfidential.

Note: Do not use this form to report system problems or to request copies of publications. Instead, contact your IBM representative or the IBM branch office serving you.

Would you like a reply? ___YES ___NO

Please print your name, company name, and address:

____

____

____

____

IBM Branch Office serving you: ____

Thank you for your cooperation. You can either mail this form directly to us or give this form to an IBM representative who will forward it to us.

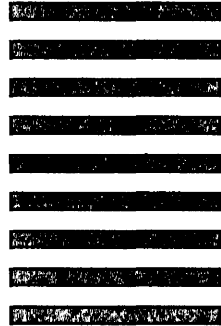
SC24-5219-3

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

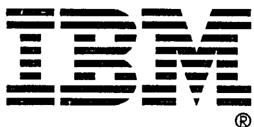


BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE:

INTERNATIONAL BUSINESS MACHINES CORPORATION
DEPARTMENT G60
PO BOX 6
ENDICOTT NY 13760-9987

Fold and tape | Please Do Not Staple | Fold and tape



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

File No. S370/4300-39
Printed in U.S.A.

SC24-5219-3

IBM ®