RC 7268 (#31339) 08/25/78
Computer Science 47 pages

Research Report

EXEC 2

A Computer Language for Word Programming

W. E. Daniels
R. W. Ryniker II

Adapted from a document by: **C. J. Stephenson**

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

IBM Research Division
Yorktown - San Jose - Zurich

EXEC 2

A Computer Language for Word Programming

W. E. Daniels
R. W. Ryniker II

adapted from a document by

C. J. Stephenson

This document defines the EXEC 2 language, which is designed for writing command language programs. The language is system-independent in nature, and there are embodiments for VM/CMS, MVS/TSO, experimental operating systems and text editors.

This report supersedes RC6292, which defined the previous version of EXEC 2.

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

RC 7268 (#31339) 8/25/78

Copies may be requested from:

IBM Thomas J. Watson Research Center
Post Office Box 218
Yorktown Heights, New York 10598

Contents

EXEC 2

A Computer Language for Word Programming

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 EXEC 2 Language is derived in spirit from CMS EXEC; however, one of the objectives of EXEC 2 is that it should be usable in almost any host system, for programming a variety of command environments. EXEC 2 is hereinafter referred to simply as 'EXEC'.

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

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

where 'filename' is the name of the EXEC file to be executed, and 'arg1', 'arg2', ... , are arguments which are passed to it. In some command environments (such as EDITOR) the word 'EXEC' is omitted, and in others (such as CMS console command mode) it is optional.

EXEC files can have any filename. The command environments from which they are invoked may, however, impose some restrictions on the names which can be used in commands.

Each command environment from which EXEC files are to be invoked is expected to claim a filetype (or some distinguished subset of filenames) which is to be used for its EXEC files. In CMS, for example, the filetype EXEC is used for files which are invoked from CMS command mode, and the filetype EDITOR is used for edit macros.

EXEC files can have either 'F' or 'V' format, and a line length which does not exceed the implementation maximum for the host system.

Introduction to EXEC Files

EXEC files contain EXEC statements. An EXEC statement occupies one line, and may be a *comment* or an *executable statement.* A comment is a line in which the first non-blank character is an asterisk, and is ignored during execution. An executable statement consists of a sequence of *words* the first of which does not begin with an asterisk. A word is a string of contiguous non-blank characters. Words are separated from each other by one or more blanks.

An executable statement may be:

(a) a null statement (which has no effect),

(b) a command (which is issued to a command interpreter),

(c) an assignment (which manipulates EXEC variables),

or (d) a control statement (which manipulates EXEC variables, controls execution or flow through the file, or performs console input or output).

Assignments start with the name of an EXEC *variable*, and control statements start with an EXEC *control word*. EXEC variables and control words begin with an ampersand. Variables are local to the current EXEC file. Most variables are initially unset, and have an apparent null value. The variables &1, &2, ..., are special, and are initialized to the arguments 'arg1', 'arg2', ... which are passed to the EXEC file.

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

When an EXEC file is invoked, execution starts at line number 1, and proceeds sequentially, except when otherwise directed by control 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 variables, which are replaced by their values; except that if the variable is the target of an assignment, its name is retained. (A precise description is given later under *Name substitution*.) During this process, the words may grow or shrink in length.

(3) If as a result of (2) 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: if for example the value of a variable contains an embedded blank, the word containing it is still treated as one word, even though if printed it might appear as two.

(4) The statement is analysed syntactically, and executed, according to the rules which follow. Note that, except for identifying the targets of assignment, the syntax analysis is done *after* steps (1), (2) and (3) above.

EXEC Statements

(a) Null statement.

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

(b) 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 (such as CMS), or to a subcommand environment (*e.g.* an editor). When it is finished, control returns to the EXEC file, and its return code can be obtained from the predefined EXEC variable &RC.

(c) Assignments.

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

(i) null

(ii) a single word, *e.g.* ABC

(iii) an arithmetic expression, consisting of a sequence of words which represent positive or negative integers, separated by plus or minus signs, *e.g.* 3 - 4 + -11 - 00

(iv) a function invocation, *e.g.* &PIECE OF &1 2 1

(v) an arithmetic expression (as in (iii)) in which the last term is replaced by a function invocation which yields a numeric value, *e.g.* -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 arguments which 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.

(d) Control statements.

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

Predefined Variables

The following EXEC variables are initialized or maintained automatically.

&0 Initialized to the first word of the command string which is passed to the EXEC interpreter; normally has the same value as &FILENAME, but may be different if the EXEC file was invoked *via* a synonym.

&1, &2, ... These are the EXEC *arguments*. They are initialized to the arguments 'arg1', 'arg2', ..., which are passed to the EXEC file; they are reset by &ARGS or &READ ARGS; and they are temporarily reset by invocation of user-defined subroutines and functions. EXEC arguments beyond the last which is set have an apparent null value, and cannot be set explicitly. (See &N, below.)

&ARGSTRING Initialized to the command string which is passed to the EXEC file, treated as a single literal string starting with the character following the blank which terminates &0, and including any leading, embedded or trailing blanks. The initial value includes the EXEC arguments &1, &2, ..., but &ARGSTRING is not affected by changes to them.

&BLANK A word which has the value of a single blank.

&BLANKS A word which has the value of 255 blanks.

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

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

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

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

&FILENAME Initialized to the name (first qualifier) of the EXEC file; normally has the same value as &0.

&FILETYPE Initialized to the type (second qualifier) of the EXEC file (*e.g.* 'EXEC').

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

&LINE, &LINENUM Maintained as the number of the current line in the EXEC file.

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

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

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

Examples of EXEC Files

(1) The following is a sample EXEC file called GRAB EXEC, which copies a file from any CMS disk to the user's A-disk.

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

-BAD &PRINT INVALID GRAB COMMAND
&EXIT 101

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

(2) The following example is called SEND EXEC, and sends a specified CMS file to a specified user. The comments are included for tutorial purposes.

```
&TRACE OFF
*  COMMAND IS: SEND USER FILENAME FILETYPE <MODE>
*  IF THERE ARE NO ARGUMENTS GIVEN, TELL HIM HOW...

&IF &N = 0 &GOTO -TELL

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

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

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

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

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

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

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

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

*  TELL HIM INVALID SEND COMMAND, AND RETURN
*  WITH ERROR...

-BAD &PRINT INVALID SEND COMMAND
&EXIT 101

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

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

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

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

*  TELL HIM HOW...

-TELL &PRINT COMMAND IS: SEND USER FN FT <FM>
&EXIT 100
```

Control Statements

Control statements begin with a control word, which is usually followed by one or more arguments. The control words, and the rules for their use, are as follows. The symbols '<' and '>' are used to indicate optional arguments.

&ARGS <word1 <word2 ... >>

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

(See &READ ARGS; also see the predefined variable &N.)

&BEGPRINT <n <k>>
&BEGTYPE label *
1
line1
line2
...

Print at the console 'line1', 'line2', ..., truncated if necessary at column k, without removing surplus blanks or replacing any EXEC variables. Printing of the lines is terminated (a) by the end of file, (b) by the exhaustion of the count n (if given), or (c) by a line which contains the given label and nothing else. In this last case, the label must be wholly contained within the columns which would otherwise be printed (and it must be the only word within these columns): this line is not itself printed, and execution continues on the next line. The first character of a label must be a hyphen.

If the truncation column is not given, or is given as '*', the lines are not truncated.

This and '&BEGSTACK' are the only statements which can occupy more than one line, and are the only statements which permit the lines of an EXEC file to be handled literally, *i.e.* without removing surplus blanks or replacing EXEC variables.

(See &PRINT.)

&BEGSTACK <n <k <FIFO >>>
label * LIFO
1

line1
line2
...

Stack in the console input buffer 'line1', 'line2', ..., truncated if necessary at column k, without removing surplus blanks or replacing any EXEC variables. Stacking is terminated (a) by the end of file, (b) by the exhaustion of the count n (if given), or (c) by a line which contains the given label and nothing else. In this last case, the label must be wholly contained within the columns which would otherwise be stacked (and it must be the only word within these columns): this line is not itself stacked, and execution continues on the next line. The first character of a label must be a hyphen.

If the truncation column is not given, or is given as '*', the lines are not truncated. The lines are by default stacked FIFO (first in, first out), but this can be changed by giving 'LIFO' (last in, first out) as the third argument.

This and '&BEGPRINT' are the only statements which can occupy more than one line, and are the only statements which permit the lines of an EXEC file to be handled literally, *i.e.* without removing surplus blanks or replacing EXEC variables.

(See &STACK.)

&BUFFER n <comment>
*

Discard the lookaside buffer (if any) together with its contents, and create a new buffer with the capacity to hold up to n lines, or (if '*' is specified) to hold the entire EXEC file. If n is given as zero, the buffer and its contents are simply discarded.

The lookaside buffer is a contrivance which enables the EXEC interpreter to maintain a private copy of some of the more recently executed lines from the EXEC file, and to remember the whereabouts of labels to which reference has already been made. It thereby improves the performance of EXEC loops, in which the same lines and labels are used repeatedly. For maximum effect, the buffer should be large enough to contain the longest loop in its entirety, and should be set up before entering the first 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 file is subject to modification during execution; if it is, the results are unpredictable.

&CALL line-number <arg1 <arg2 ... >>
label

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

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

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

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

(See &RETURN; also see *User-defined Functions.*)

&CASE <U <comment>>
M

Translate to upper-case any lower-case alphabetic characters which are read in response to subsequent &READ statements, or do not translate them (allow 'mixed' cases), or (if no argument is given) do not change the setting. Initially the translation is set to 'U'.

(See &UPPER.)

&COMMAND <word1 <word2 ... >>

Issue to the host system the command comprising 'word1', 'word2', ..., separated from each other by a single blank. When it is finished, its

return code is obtainable from the predefined EXEC variable &RC. The statement normally has the same effect as:

```
word1 word2 ...
```

There are however the following differences.

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

(b) &COMMAND overrides any presumption of a subcommand environment (see &PRESUME), and always issues the command to the host system.

(See &SUBCOMMAND; also see the predefined variables &COMLINE and &RC.)

&DUMP ARGS
VAR<S> <var1 <var2 ... >>

Print at the console lines of the form:

```
var = value
```

either one for each EXEC argument &1, &2, ... which is set, or one for each of the variables 'var1', 'var2', ... The lines are truncated if their length exceeds the implementation limit for printed output.

&ERROR action

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

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

What happens after the action depends upon the type and consequences of the action. If it is itself a command (or subcommand) which also yields an error return code, execution stops abnormally with an error message; otherwise (unless the action causes a transfer of control) execu-

tion resumes at the line following the command which caused the action to be invoked.

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

&EXIT <return-code <comment>>
0

Stop execution of the EXEC file, and yield the given return code.

&GOTO line-number <comment>
label

Transfer control to the given line, or to the line starting with 'label'.

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

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

&IF word1 =,EQ <word2 executable-statement>
¬=,NE
<,LT
<=,¬>,LE,NG
>,GT
>=,¬<,GE,NL

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

(See &SKIP.)

&LOOP n m
label *
WHILE condition
UNTIL

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

The values of n and m (if given) must be numeric; also n must be positive, and m must not be negative.

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

The form of the condition (if given) is the same as in the &IF statement, *viz.*

word1 . =,EQ <word2>
¬=,NE
<,LT
<=,¬>,LE,NG
>,GT
>=,¬<,GE,NL

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

&PRESUME <&COMMAND>
&SUBCOMMAND environment

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

The name of the subcommand environment is not checked when the &PRESUME statement is executed. If, when a subcommand is subsequently issued, the environment does not exist, the only effect is to set a special return code (*e.g.* −3 when running in CMS).

'&PRESUME' with no arguments is equivalent to '&PRESUME &COMMAND'.

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

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

&PRINT <word1 <word2 ... >>
&TYPE

Print at the console a line containing 'word1', 'word2', ..., separated from each other by a single blank, or print a blank line if there are no words given. The line is truncated if necessary.

Unlike &BEGPRINT, the words to be printed are searched in the normal way for the names of EXEC variables, which are replaced by their values.

&READ <n>
1
*
ARGS
VAR<S> <var1 <var2 ... >>
STRING var

Read from the console, and execute or assign what is read, according to the following rules.

n, 1, * — Read n lines, or read an indefinite number of lines, and execute them individually as if they had occupied the current line in the EXEC file. Reading from the console stops when n lines have been read, or when a &BEGPRINT, &BEGSTACK, &GOTO, &LOOP or &SKIP statement is encountered. Reading from the console is suspended if a user-defined function or subroutine is invoked, and is resumed when control returns from that invocation. If another &READ statement is encountered, the number of lines to be read by it is added to the number outstanding. The value of n may be negative.

ARGS — Read a single line, assign the words in it to the EXEC arguments &1, &2, ..., and discard any other EXEC arguments which 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 &ARGS, and the predefined variable &N.)

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; or 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.

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

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

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

Lines which are read from the console are not truncated by the EXEC interpreter, *i.e.* they are unaffected by the setting of &TRUNC.

&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 arguments which was created at invocation is discarded; the previous values, and the number previously set, become accessible again. The number of lines (if any) which remain to be read from the console in response to a previous '&READ n' statement is reset to the number outstanding at the time of the invocation. Any loops which have been opened in the subroutine or function, and not closed, are aborted; any loops which 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 with an error message.

(See &CALL; also see *User-defined Functions.*)

&SKIP <n <comment>>
1

If n > 0, skip the next n lines of the EXEC file. If n < 0, transfer control to the line which 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 the number of which is zero or negative, execution stops abnormally with an error message. If control is transferred to a line below the last in the EXEC file, execution stops normally with a return code of zero.

(See &GOTO.)

&STACK <FIFO> <word1 <word2 ... >>
LIFO

Stack a line in the console input buffer containing 'word1', 'word2', ..., separated from each other by a single blank, or stack a null line if there are no words. The line is by default stacked FIFO (first in, first out), but this can be changed by giving 'LIFO' (last in, first out) as the first argument.

Unlike &BEGSTACK, the words to be stacked are searched in the normal way for the names of EXEC variables, which are replaced by their values.

&SUBCOMMAND environment <word1 <word2 ... >>

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

If the given environment does not exist, the only effect is to set a special return code (*e.g.* −3 when running in CMS).

Normally it is convenient to 'presume' the first two words of this statement, so that they do not have to appear at the beginning of every subcommand (see &PRESUME). The explicit use of the

&SUBCOMMAND statement does however permit the issue of subcommands which start with an asterisk, a hyphen, or an ampersand (*cf.* &COMMAND). Also note that the statement '&SUBCOMMAND environment' (without any additional arguments) is the only way of issuing a null subcommand.

(See the predefined variables &COMLINE and &RC.)

&TRACE <ON <comment>>
ERR
ALL
OFF

Print at the console commands (and subcommands) which are issued from the EXEC file; or print commands (and subcommands) which yield an error return (*i.e.* a return code which is not zero); or print all executable statements; or do not print any statements; or (if there is no argument) do not change the setting. The setting remains in effect until reset. Initially it is OFF.

When tracing is ON, each command is printed before it is executed, and subsequently the return code is printed if it is not zero. When ERR is in effect, commands which yield a non-zero return code are printed after execution, followed by the return code. The return code is printed on a line by itself, in the form '+++ E(nnn) +++'.

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

&TRUNC <k <comment>>
*

Set the truncation column for EXEC 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.

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

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

&UPPER ARGS
VAR<S> <var1 <var2 ... >>

Translate to upper-case any lower-case alphabetic characters in the values of the EXEC arguments &1, &2, ...; or translate to upper-case any lower-case alphabetic characters in the values of 'var1', 'var2', ...

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

(See &CASE.)

Predefined Functions

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

function-name OF <arg1 <arg2 ... >>

The names of the predefined functions, and the rules for their use, are as follows. The symbols '<' and '>' are used to indicate optional arguments.

&CONCATENATION OF <word1 <word2 ... >>
&CONCAT

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 <word>
&TYPE

Yields the value NUM if 'word' represents a valid number; otherwise yields the value CHAR.

&DIVISION OF dividend divisor
&DIV

Yields a numeric value representing the integral part of the division of the dividend by the divisor, both of which must be numeric, and the latter of which must not be zero. Example:

```
&X = &DIV OF 7 2
```

This sets &X to 3.

&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.

(*Cf.* &RIGHT OF.)

&LENGTH OF <word>

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

&LITERAL OF <string>

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

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

This results in the printed line:

```
&X = **
```

(*Cf.* &STRING OF.)

&LOCATION OF needle <haystack>

Searches 'haystack' for the first occurrence of 'needle', and yields a numeric value representing its ordinal position, or yields zero if there is no occurrence (or if the length of 'needle' exceeds that of 'haystack'). Example:

```
&X = &LOCATION OF IT GRAVITATION
```

This sets &X to 5.

(See &PIECE OF and &POSITION OF.)

&MULTIPLICATION OF i j <k ... >
&MULT

Yields a numeric value representing the result of multiplying the given arguments, all of which must be numeric, and of which there must be at least two. Example:

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

This sets &X to 120.

&PIECE OF word i <j>
&SUBSTR *

Extracts that piece of 'word' which starts at character i, with length j; or which 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 &LOCATION OF.)

&POSITION OF word <word1 <word2 ... >>

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

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

This sets &X to 3.

(*Cf.* &LOCATION OF.)

&RANGE OF stem i j

Yields a string consisting of the words which 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 which are appended to it are stripped of any plus sign or redundant leading zeros.

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

Examples:

(a) Irrespective of the values of &A, &A3, &A4 and &A5, the sequence:

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

produces the same result as:

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

(b) The sequence:

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

yields the printed line:

```
A BC DEF GHIJ KLMNO
```

(c) The sequence:

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

yields the printed line:

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

&RIGHT OF word j

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

(*Cf.* &LEFT OF.)

&STRING OF <string>

Yields the string which begins with the character following the blank which terminates 'OF' and ends with the last non-blank 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 variables, which are replaced by their values; however blanks are not removed from the string, even if they are adjacent to a word which 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
```

(*Cf.* &LITERAL 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 <arg1 <arg2 ... >>
label

The effect is to create a new generation of the EXEC arguments &1, &2, ... , initialized to 'arg1', 'arg2', ... ; and to invoke the given function, *i.e.* 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 (*q.v.*).

The new generation of arguments supersedes the arguments which were previously set, making the previous values, and the number previously set, temporarily inaccessible. On entry to the body of the function, the values of the arguments, and the number set, are as given in the function invocation. Their values, and the number 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.

Examples:

(a) The user-defined function

-OVERLAY OF layee layer

is to return the string 'layee' overlaid by 'layer'. (This is different from 'layer' as given 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...
-OVERLAY & = 1 + &LENGTH OF &2
&1 = &PIECE OF &1 &
&1 = &CONCAT OF &2 &1
&RETURN &1
```

(b) Suppose there is an external program TIME which 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
```

Name Substitution

In the *Introduction to EXEC Files*, it was stated that the words forming an executable statement are searched for the names of EXEC variables, which 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 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 which represent the EXEC control words and predefined functions, which are initialized to their own names (*e.g.* the value of '&IF' is '&IF'); and

   (b) the EXEC arguments, and the other predefined variables, which have the values specified in the section *Predefined Variables.*

(3) Inspection resumes at the next character to the left, and the procedure is repeated from (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 which 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:

(a) The sequence:

```
&X = 123
&PRINT ABC &X ABC&X 000&X
```

yields the printed line:

```
ABC 123 ABC123 000123
```

(b) The sequence:

```
&I = 2
...
&X&I = 5
&I = &I - 1
&X&I = &I + 1
...
&X = &X&I + &X&X&I
&PRINT ANSWER IS &X
```

yields the printed line:

```
ANSWER IS 7
```

Miscellaneous Notes

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

(1) Recursive execution.

An EXEC file may invoke itself recursively, or may invoke other EXEC files, by issuing the appropriate command or subcommand. EXEC files which have the filetype EXEC can, for example, be invoked by means of the statement:

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

(2) Termination of an EXEC file.

An EXEC file stops execution and returns to its caller:

(a) when an &EXIT statement is executed;

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

or (c) when an EXEC error is encountered, in which case a message is printed, and execution stops abnormally with a return code in the range 10001 to 10099 (see *Error Codes and Messages*, below).

(3) Console input buffer.

The EXEC interpreter requires the host system to support a console input buffer. This is a conceptual area in which lines can be deposited FIFO (first in, first out), or LIFO (last in, first out), and subsequently retrieved by attempts to read from the console. It provides a simple mechanism for communicating between programs. In EXEC files, lines can be deposited in the buffer with the &STACK or &BEGSTACK statements, and can be retrieved with the &READ statement.

(4) Assignment statement.

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

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

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

```
3 = 2 + 1
```

(5) Evaluation of &DATE and &TIME.

The time is taken once for each execution of a statement which 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, *e.g.* as follows:

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

(6) Size and treatment of numbers.

Words which are treated as numbers must represent integers in the range −2,147,483,648 to +2,147,483,647 inclusive. An attempt to interpret or derive a number outside this range causes numeric overflow, which results in execution stopping abnormally with an error message.

A word which represents a number is treated no differently from a word which does not represent a number except:

(a) when it appears in a context which requires a numeric value, at which point the value is obtained by conversion;

or (b) when it appears as one of the comparands in an &IF clause or a &LOOP condition, in which case its numeric value is used if both comparands represent numbers;

or (c) when it is given as the argument to &DATATYPE OF.

(7) Removing plus signs and leading zeros.

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

```
&X = 00000000000000000012
&Y = &X + 0
&PRINT &X &Y
```

This yields the printed line:

```
00000000000000000012 12
```

(8) Syntax of conditional phrases.

In the conditional phrases which 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 satisifed if and only if &1 is null or blank; and

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

is syntactically valid even if &1 is null.

A similar trick 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. (For the relative values of characters, refer to the internal codes for the EBCDIC character set, given on the IBM System/370 Reference Card.)

(9) Embedded blanks.

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

(a) &ARGSTRING is initialized to the command string exactly as passed to the EXEC file, and may therefore contain embedded blanks.

(b) The '&READ STRING var' statement assigns to the given variable the complete line exactly as read, which may contain embedded blanks.

(c) The predefined variables &BLANK and &BLANKS can be used to embed blanks in the value of a variable, *e.g.*

```
&X = &PIECE OF &BLANKS 1 80
&Y = &CONCAT OF A &BLANK B
```

(d) The predefined function &RANGE OF inserts a blank between the constituent words; and the predefined functions &LITERAL OF and &STRING OF retain embedded blanks which are given in their arguments.

(e) Embedded blanks can be transmitted from one variable to another with the assignment statement, and to the EXEC 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 arguments &1, &2, ..., contain the constituent words.

(10) &LOOP statement.

The first three words of the &LOOP statement are searched for EXEC 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). Example:

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

This results in the printed lines:

```
4
5
```

(11) Closing of loops.

A loop may be in any of three mutually exclusive states, *viz.* 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; and it becomes active again when the second loop is closed, or when a corresponding &RETURN statement is executed. A loop is closed when it is active, and when:

either (a) the requirement for termination, given in the &LOOP statement, is met;

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

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

Examples:

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

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

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

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

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

(12) Search for labels.

The search for a label which is referred to 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 heed of its context, or what follows it. It is therefore necessary to avoid using labels which would be matched by the first word of a line within the scope of a &BEGPRINT or &BEGSTACK statement.

Labels which are attached to statements are treated literally: they are not searched for EXEC variables. Labels need not be unique.

(13) Performance of label searches.

(a) &CALL, &GOTO, and user-defined functions

A &CALL or &GOTO statement, or an invocation of a user-defined function, which transfers to a label above the current statement tends to be inefficient, especially in long EXEC files.

It is preferable to use the &LOOP statement in place of an upward '&GOTO label' statement.

(b) &LOOP label ...

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

(14) EXEC words are not reserved words.

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

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

In the case of predefined variables other than the EXEC 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 arguments, and can be set explicitly (*e.g.* &2 = 1) only if they are within the range of arguments which are currently set. With this exception, EXEC 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 (*e.g.* &PIECE and &SUBSTR), an assignment to one of the names does not affect the other name or names.

It is intended that, with the exception of the arguments &1, &2, ..., EXEC words which end with a numeral will not be introduced into the

EXEC 2 Language. Therefore variables such as &A1, &A2, ..., can be relied upon to have an initial value of null. Variables the names of which do not end with a numeral should not be used in a way which relies upon their initial values, since their names may conflict with new EXEC words which are added to the language at a later time.

(15) Example of &TRACE ALL.

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

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

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

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

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

```
9. NEXT
+++ E(1) +++
--- LOOP WHILE 1 = 0
10. &EXIT 1
```

Error Codes, Messages and Abnormal Termination

If the EXEC interpreter finds an error, it prints a message of the form:

```
MISTAKE IN fn ft fm, LINE n - description of error
```

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

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
10018	NUMERIC OVERFLOW IN 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

BNF Description of the Gross EXEC 2 Syntax

The symbols < and > are used here to delimit metalinguistic variables.

<exec file>	::=	<statement>
		<exec file> <statement>
<statement>	::=	<comment>
		<label> <executable stmt>
		<executable stmt>
<comment>	::=	* <anything>
<executable stmt>	::=	<unconditional stmt>
		<if clause> <executable stmt>
<unconditional stmt>	::=	<null>
		<command>
		<assignment>
		<control stmt>
<if clause>	::=	&IF <word> <comparator> <word>
<assignment>	::=	<variable> = <rhs>
<rhs>	::=	<null>
		<word>
		<function invocation>
		<arithmetic rhs>
<arithmetic rhs>	::=	<arithmetic expr>
		<arithmetic expr> + <function invocation>
		<arithmetic expr> − <function invocation>
<arithmetic expr>	::=	<number>
		<arithmetic expr> + <number>
		<arithmetic expr> − <number>
<number>	::=	<unsigned integer>
		+<unsigned integer>
		−<unsigned integer>

History and Acknowledgements

EXEC 2, as a language, derives from CMS EXEC, which began in Cambridge, Massachusetts, in the early days of CMS. Nobody seems to want credit for the first version of CMS EXEC, which appears to have been written at the IBM Cambridge Scientific Center around 1966. It was modelled after 'RUNCOM' in CTSS (Compatible Time-Sharing System of Project MAC at M.I.T.), and provided a way of issuing a sequence of commands, each made up of eight-byte tokens, with parameter substitution *via* &1, &2, ...

This facility already exhibited two of the enduring properties of the EXEC language, *viz.* (a) the use of blank as the essential delimiter, and (b) the use of a special symbol (ampersand) to distinguish non-literals, instead of distinguishing literals as in most computer languages.

The transition to a programming language took place in 1967-68 at M.I.T.'s Lincoln Labs, during a joint study with the Scientific Center. The principal designers were Francis Belvin, Harold Feinleib, Oliver Selfridge and Joel Winnett; and the programming was done by Roger Banks during summer employment. They added user-defined variables, the assignment statement, and most of the control words (&IF, &GOTO, &LOOP, &STACK, &READ, etc.). These improvements were installed in IBM's second release of CMS by Thomas Rosato at the Scientific Center.

During this time, the work in IBM, and the joint study with Lincoln Labs, were nurtured by Norman Rasmussen, who was manager of the Scientific Center.

The EXEC language was extended and formalized in 1971 by Christopher Stephenson at IBM Research, Yorktown Heights. The general rules for replacement of EXEC variables were laid down, and predefined functions were introduced. This work was supported by Donald Rozenberg. The resulting language was described in a document written jointly with David Mainey; and the extensions were incorporated into VM/CMS by James Walsh in Cambridge.

EXEC 2 derives its overall syntax and flavour from CMS EXEC; however, it is system-independent in nature. It handles words exceeding eight bytes in length, it supports more general string handling, it extends the form of arithmetic expressions, and it enables commands to be issued to divers 'subcommand' environments. The language was proposed by Christopher Stephenson in a paper presented at the Fourth Symposium on Operating System Principles in 1973. It was defined, and first implemented, in 1976. User-defined subroutines and functions were added in 1977. Credit is due to Craig Franklin (when at Stanford University), and to Marc Auslander, Walter Daniels and Paul Kosinski (all at IBM Research), for contributions made in conversations during 1973-77.

The original EXEC 2 interpreter was written for System/370 as part of an experimental programming system at IBM Research, Yorktown Heights. It has subsequently been installed in CMS by Walter Daniels, and in TSO by Burn Lewis. It has also been incorporated in editing systems by Gordon Barnard, at the IBM Technology Data Center, Poughkeepsie; and by G. Mack Hicks and Gary Hauser at the IBM Santa Teresa Laboratory, California. A second interpreter has been written for the 801 Research Minicomputer by Albert Chang.

The EXEC 2 Language, as defined here, was revised in January, 1978, by vote of Marc Auslander, Walter Daniels, Michel Hack, Burn Lewis, Richard Ryniker and Christopher Stephenson.

This document was prepared and edited by Richard Ryniker, based on material originated by Christopher Stephenson.

APPENDIX I

EXEC 2 in CMS at Yorktown Heights

The EXEC 2 interpreter resides on the CMSSYS 19F minidisk at Yorktown Heights. It is loaded by issuing the command

```
FREELOAD EXEC EXEC2 SYS
```

after accessing that disk. This must be done before attempting to execute an EXEC file written in the EXEC 2 Language.

In order to provide the ability to execute both EXEC 2 and CMS EXEC files, some device is required to distinguish between them. If the EXEC 2 control word &TRACE starts within the first 32 characters of the first line of an EXEC file, the file is deemed to be written in the EXEC 2 Language and is executed by the EXEC 2 interpreter; otherwise, the file is executed by the CMS EXEC interpreter.

Note that the CMS EXEC interpreter tokenizes its arguments, even when called from EXEC 2. Conversely, if an EXEC 2 file is invoked from a CMS EXEC file, all arguments (&1, &2, ...) are truncated to 8 characters by the CMS EXEC interpreter before the EXEC 2 interpreter is entered. In such a case, the predefined variable &ARGSTRING is initialized to the string consisting of the arguments &1, &2, ... separated from each other by a single blank, with no leading or trailing blanks.

The maximum line-length of an EXEC file is 255. The maximum length of a line read from the console is 130, although stacked lines and lines read from the console input buffer may be as long as 255 characters. The maximum length of a word, after replacement of variables, is 255. The maximum length of a printed line is 255. The maximum length of a statement, after replacement of variables, is 511 characters (This limit is enforced only as needed for practical reasons by the interpreter, and some statements can grow to a greater length.)

Initially, if the EXEC file has a mode number of '2', there is a lookaside buffer with a capacity of 32 lines; otherwise there is no lookaside buffer (see &BUFFER).

If a nonexistent command is issued to CMS, or if a subcommand is issued to a nonexistent subcommand environment, the return code -3 in generated.

There are many programs available in the Yorktown CMS which are useful to writers of EXEC files. Here is a list of some frequently used programs, with brief descriptions of their purposes.

STACKIO	Read and write lines from and to disk files, tape, console input buffer, printer, card reader, console, ... There is a limited ability to select lines which meet designated criteria concerning their contents.
GLOBALV	A general global value mechanism which allows passing values from one program to another and can remember values from previous invocations of programs and from previous terminal sessions.
MAKBUF	Establish a new level in the console input buffer.
DESBUF	Delete level(s) from the console input buffer.
LOCALTIM	Stack the local time and date, in the same format as &TIME and &DATE.
SUID	Stack userid and system information.
OBEY	Resolve and execute a command according to the CMS command search order.
FINDSTAK	Selectively stack lines from a disk file. It offers more complete selection criteria than STACKIO.
SADT	Stack information about an accessed disk.
WAITDEV	Define, poll, wait for interrupt, or delete pseudo-devices or named interrupts; poll or wait for I/O interrupts from virtual devices.
SENTRIES	Set the return code equal to the number of stacked lines currently in the console input buffer.
SUBCOM	Establish, delete, or query the existence of a subcommand environment.
NAMEINT	Establish or clear a named interrupt.
CONVERT	Perform binary, decimal and hexadecimal conversions.
XPARSE TOKENIZE PAREN SELECT ABBREV	Aids for parsing command strings.

Comparison with Previous Version of EXEC 2

Readers who are familiar with the previous version of EXEC 2, described in RC 6292, may wish to note the following principal differences in the version described here.

1. What were previously referred to as 'special' variables are now called *predefined variables*, and what were previously referred to as 'built-in' functions are now called *predefined functions*.

2. It is now possible to write *user-defined* subroutines and functions.

3. A variable of the form &j, where j is an unsigned integer without leading zeros, cannot be set explicitly if j exceeds the number of EXEC arguments (&1, &2, ...) which are currently set.

4. The predefined variable &ARGSTRING is now set to the equivalent of '&RANGE OF & 1 &INDEX' when called from CMS EXEC.

5. CP requests are now passed to CMS for interpretation. (The previous version of the interpreter called CP directly with untokenized arguments). In order to avoid 8-byte tokenization, the CMS command 'CP' will have to be fixed to recognize the untokenized string passed by the EXEC 2 interpreter. An alternative is to call 'CPX MODULE', or, better yet, to issue 'FREELOAD CP CPX SYS' at the same time 'FREELOAD EXEC EXEC2 SYS' is issued (*e.g.* in 'PROFILE EXEC').

Note also that, due to a change in the way the System/370 clock is set at Yorktown, the predefined variables &DATE and &TIME now contain Greenwich Mean Time (GMT) instead of local time.

Comparison with CMS EXEC

Readers who are familiar with CMS EXEC may wish to note the following differences of the EXEC 2 Language compared with the EXEC of VM/CMS, Release 2.

1. Eight-byte tokenization, and special handling of parentheses, have been done away with, except as needed for interfacing with CMS. In EXEC 2, statements are composed of 'words' of up to 255 characters each.

2. Commands may be issued either to CMS or to specified 'subcommand' environments (such as an editor). The destination is controlled with the &COMMAND, &SUBCOMMAND and &PRESUME statements.

3. The &BEGPUNCH, &PUNCH, &SPACE and &TIME statements have been eliminated.

4. The &CONTROL, &TYPE and &BEGTYPE control words have been renamed &TRACE, &PRINT and &BEGPRINT.

5. The syntax of the &BEGSTACK, &BEGPRINT, &LOOP, &TRACE and &TRUNC statements has been changed.

6. On entry to an EXEC file, tracing is off, and the truncation column is set to the limit.

7. The percent sign (%) and the single quotation mark (') do not have any special effects if assigned to the EXEC arguments &1, &2, ...

8. There are new &CASE and &UPPER statements, for controlling uppercase translation.

9. There are no global variables.

10. The special variables &$, &*, &EXEC, &READFLAG and &TYPEFLAG have been eliminated.

11. There are new predefined variables &ARGSTRING, &COMLINE, &FROM, &DATE, &TIME, &BLANK and &BLANKS.

12. The word 'OF' is included in references to predefined functions.

13. The predefined function &LITERAL OF may be used only on the right-hand side of assignment statements (thus conforming to the same context rules as the other built-in functions).

14. If an assignment is made to a predefined variable, its initial value is lost through the remainder of that EXEC file, irrespective of subsequent assignments.

15. There are new predefined functions &LOCATION OF, &POSITION OF, &RANGE OF, &LEFT OF, &RIGHT OF and &STRING OF for string manipulation.

16. There are new predefined functions &DIVISION OF and &MULTIPLICATION OF for integer arithmetic.

APPENDIX II

EXEC 2 in TSO at Yorktown Heights

EXEC files are stored as members of a partitioned dataset, and may have either fixed or variable format records. The PDS, or concatenation of partitioned datasets, must be allocated with the data definition name SYSEXEC. If there is a concatenation of partitioned datasets, the block size of the second and any subsequent PDS must not exceed the block size of the first.

To activate the EXEC 2 interpreter in Yorktown TSO, issue the following commands:

```
ALLOC DD(EXEC2DD) DS('BURN.EXEC2LIB.LOAD') SHR
USERDEF TASKLIB(EXEC2DD) EXEC(EXEC2)
```

If the EXEC2 load module is placed in one of the system libraries, then the ALLOCATE command is not needed and the 'tasklib' option should be omitted from the USERDEF command.

To deactivate the EXEC 2 interpreter, issue the command

```
USERDEF NOTSKLIB EXEC(EXEC)
```

When EXEC 2 is active in TSO, the precedence of EXEC files is before CLISTs but after system commands.

The maximum data length of a record in an EXEC file is 130 bytes. The maximum length of a line which may be stacked or read from the console or console input buffer is 130. The maximum length of a word is 255. The maximum length of a statement, after replacement of variables, is 511 characters. (This limit is enforced only as needed for practical reasons by the interpreter, and some statements can grow to a greater length.)

On entry to an EXEC file, the predefined variable &FILENAME is set to the PDS member name, &FILETYPE is set to 'EXEC', and &FILEMODE is null.

An initial lookaside buffer with a capacity of 100 lines is always allocated (see &BUFFER). The meaning of the statement '&BUFFER *' is the same as for the statement '&BUFFER 100'.

The return code generated for a non-existent TSO command is 12; for an illegal TSO command, 104. The return code generated for an unknown subcommand environment is -3. (There is no mechanism currently implemented for establishing subcommand environments in TSO.)

An attempt to dynamically modify an EXEC file by replacing or deleting a member which is in the process of being interpreted has unpredictable results.

Index

Index