File No. S370-36
Order No. GC20-1812-1

Systems

IBM Virtual Machine Facility/370: EXEC User's Guide

Release 2 PLC 11

This publication is for those VM/370 users who want to use the Conversational Monitor System (CMS) EXEC facilities. The CMS EXEC facilities enable a user to define new CMS commands that are combinations of existing CP and CMS commands. The new commands, called EXEC procedures, are usually created using the CMS Editor.

This publication tells the user how to:

- Use the CMS EXEC facilities.
- Code EXEC control statements.
- Build EXEC procedures.
- Enter EXEC procedures into the system.
- Invoke EXEC procedures.

A prerequisite publication is the *IBM Virtual Machine Facility/370: EDIT Guide,* Order No. GC20-1805.

IBM

Second Edition (April 1975)

This edition, GC20-1812-1, is a reprint of GC20-1812-0 incorporating changes released in the following Technical Newsletters:

GN20-2635 (dated December 17, 1973)
GN20-2637 (dated March 29, 1974)
GN20-2651 (dated January 30, 1975)

This edition applies to _Release 2 PLC 11_ (Program Level Change) of IBM Virtual Machine Facility/370 and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters This edition does not make GC20-1812-0, or the above listed TNLs obsolete.

Changes are periodically made to the specifications herein; before using this publication in connection with the operation of IBM systems, consult the latest _IBM System/360 and System/370 Bibliography_, Order No. GA22-6822, and its _Virtual Storage Supplement_, Order No. GC20-0001, for the editions that are applicable and current.

Technical changes and additions to text and illustrations are indicated by a vertical bar to the left of the change.

Requests for copies of 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, VM/370 Publications, 24 New England Executive Park, Burlington, Massachusetts 01803. Comments become the property of IBM.

This publication describes the Conversational Monitor System (CMS) EXEC facilities available to VM/370 users. It contains both introductory and reference information about the EXEC facilities for any VM/370 users who want to use them. In addition, it contains a section that illustrates many useful techniques of EXEC procedure design and implementation.

To use this book effectively, you should have a general understanding of basic programming techniques such as branching and looping, as well as an understanding of CMS procedures, CMS commands, and the CMS Editor.

This book contains four major sections and an Appendix:

- The "Introduction" discusses the EXEC facilities and their relationship to the VM/370 system. It also tells how to invoke and create an EXEC procedure.
- "Using the CMS EXEC Facilities" describes the three parts of the CMS EXEC facilities:
  - The EXEC command.
  - The EXEC files.
  - The EXEC interpreter.
- "EXEC Control Statements" defines three types of EXEC statements (execution control, built-in functions, and special variables), and describes each control statement in detail. This section contains the main portion of the reference material in the book.
- "Building EXEC Procedures" illustrates several techniques of EXEC design. This section shows you how to create EXEC procedures and control their execution. It contains most of the examples to be found in this book.
- "Appendix A: EXEC Control Statement Summary" provides a quick-reference summary of the EXEC control statements, which you may find helpful after you have learned to use them.

Examples of EXEC statements and procedures are found throughout this book, but they are concentrated in the "Building EXEC Procedures" section. You can refer to this section for examples as you read the control statement descriptions, or you can read the entire book once from front to back, then use the "EXEC Control Statements" section and "Appendix A: EXEC Control Statement Summary" for reference purposes.

Users of the IBM 3277 Display Station (also called the "3270") should note the 3270 equivalents of terms in this book that refer to IBM 2741 Communication Terminals. These are as follows:

- The equivalent of the RETURN key on a 2741 is the ENTER key on a 3270.
- Output that is "typed" at a 2741 is "displayed" at a 3270.

Because there is no TAB key on a 3270, one of the 3270 program function keys should be set to define tab characters.

The differences between 3270s and 2741s are fully described in the *IBM Virtual Machine Facility/370: EDIT Guide*, Order No. GC20-1805.

PREREQUISITE PUBLICATION

The *IBM Virtual Machine Facility/370: EDIT Guide*, Order No. GC20-1805, is a prerequisite for understanding the use of the CMS Editor and EDIT subcommands, which you will need in order to create EXEC procedures.

Summary of Amendments
for GC20-1812-0
as updated by GN20-2651
VM/370 Release 2 PLC 11

NEW OPERANDS FOR &CONTROL

New: Program Feature

Two new operands, MSG and NOMSG, have been added to the &CONTROL control statement.

DOCUMENTATION CHANGES

The information about PROFILE EXEC procedures, return codes, stacking, EXEC file record lengths, and EDIT macros has been clarified.

Summary of Amendments
for GC20-1812-0
as updated by GN20-2637
VM/370 Release 2 PLC 1

&EXEC SPECIAL VARIABLE

Maintenance: Documentation Only

We have included information about the &EXEC special variable.

EDIT MACROS

Maintenance: Documentation Only

We have corrected the example of EDIT macro usage.

Summary of Amendments
for GC20-1812-0
As Updated by GN20-2635
VM/370 Release 1 PLC 13

ASSIGNMENT STATEMENT

Maintenance: Documentation Only

This change clarifies the paragraph under "Executable Statements" on page 11. This change also affects "Assignment Statements" on page 13 and the sample EXEC procedure on page 59.

USE OF EXEC PROCEDURES FOR THE CMS BATCH FACILITY

Maintenance: Documentation Only

The EXEC procedure example on page 59 of the section titled 'Building EXEC Procedures' is changed.

The CMS EXEC facilities enable you to define new CMS commands that are combinations of existing CP and CMS commands. The new commands, called EXEC procedures, are usually created using the CMS Editor.

In some respects, an EXEC procedure provides facilities similar to those of an OS cataloged procedure. When an EXEC procedure is invoked, it represents a sequence of commands that are executed according to the logic control statements defined in the EXEC procedure.

You can create simple EXEC procedures that execute several frequently used commands, or you can devise complex EXEC procedures that test several logical conditions before deciding whether or not to execute a command. The logical capabilities in the EXEC processor are controlled with statements similar to the IF/THEN, GOTO, DO, and LOOP statements familiar to high-level language users.

An EXEC procedure is created by placing a selected sequence of commands in an EXEC file. An EXEC file can have any valid CMS filename and filemode, but it must have a filetype of EXEC. EXEC files are made up of fixed length records up to 130 characters long. Each record consists of one CMS command or EXEC control statement. (A CP command can be entered in an EXEC file by using the CMS command CP.)

Although EXEC files are usually created by using the CMS Editor, they can also be created by an option of the LISTFILE command, by reading a card file from the user's virtual reader, or by a user program.

INVOKING AN EXEC PROCEDURE

To invoke an EXEC procedure, you enter the CMS command EXEC, followed by the filename of the EXEC file wanted and, optionally, a list of arguments. When you are in CMS command mode, you may omit the initial word EXEC, thus invoking the EXEC procedure simply by entering the filename.

You can invoke an EXEC procedure by filename because when a CMS command is entered at a terminal, CMS first searches for an EXEC procedure by that name, then for a regular CMS command module by that name. Therefore, if an EXEC procedure has the same name as a CMS command module, the EXEC procedure is always executed instead of the command. The CMS command of the same name can then be invoked within the EXEC prodedure.

Note: When a CMS command is issued in a user program, the request is handled by an SVC (Supervisor Call), and the search of EXEC procedures is not made. In this case, an EXEC procedure can be invoked explicitly.

When an EXEC file is invoked, the EXEC interpreter controls the execution of the procedure, substituting values for EXEC variables where required, and passing control to CMS for execution of CMS commands.

The EXEC interpreter can manipulate argument lists, thus allowing the user to pass arguments to the EXEC procedure when it is invoked. Before a command in the EXEC file is executed, each variable in it is temporarily replaced by the corresponding argument from the argument list that was specified when the EXEC procedure was invoked. Use of these variable arguments thus permits great flexibility in command execution within the EXEC procedure.

The concepts introduced in the preceding paragraphs are discussed in greater detail later in this book. At this point, however, you can see that the EXEC facilities provide you with a powerful tool that you can use to develop your own command language or set of operating procedures.

WRITING AN EXEC PROCEDURE

Once you have designed an EXEC procedure, you can enter it into the VM/370 system in several ways:

1. By punching cards, which are then read via the real system card reader and the user's virtual reader.

2. By using the CMS Editor to enter input lines into a CMS file.

EXEC files can also be created by a user program or by the CMS LISTFILE command. Regardless of which method is used, the format of the entered statements is basically the same. Only one CMS command or EXEC control statement may be entered per card or card image. CMS commands must be in the same format as they would be if you entered them from a terminal.

To use the CMS Editor to create an EXEC file, enter the command:

```
EDIT filename EXEC
```

where the filename is any valid CMS filename. The filetype of EXEC is required. The CMS Editor automatically places a limit of 80 characters on the input line length, and translates all entered lowercase characters to uppercase.

If the EXEC file specified in the EDIT command is a new file, the message:

```
NEW FILE:
EDIT:
```

is displayed at the terminal. You can then type in the INPUT subcommand and start entering input lines as soon as the Editor replies, as follows:

```
input
INPUT:
(Begin entering input lines.)
```

End each input line by pressing the RETURN key (or the ENTER key on a 3270 display station). When you have finished entering input, return to EDIT mode by pressing the RETURN key again. If the file needs no corrections, you can save it by typing in the FILE subcommand. The data is stored and control returns to the CMS environment.

You can execute an EXEC file created in this way by typing in "EXEC filename", or simply by typing in the filename. You can examine its contents either by displaying it at a terminal using the CMS TYPE command, or by printing it on the system printer using the CMS PRINT command.

The preceding description of the CMS Editor identifies only a few of the Editor functions that may be useful in creating and editing EXEC files. For a more complete discussion of the CMS Editor, refer to the _VM/370: EDIT Guide_.

USING THE CMS EXEC FACILITIES

This section describes the three major parts of the CMS EXEC facility:

1. The EXEC command, which initiates CMS execution of an EXEC file.

2. The EXEC files, which contain sequences of CMS commands and EXEC control statements.

3. The EXEC interpreter, which analyzes each statement in an EXEC file before CMS executes the procedure.

Each of these items is described in greater detail under a separate heading.

THE EXEC COMMAND

The EXEC command executes one or more CMS commands or EXEC control statements contained in a specified EXEC file, allowing a sequence of commands to be executed by issuing a single command. If this command is entered from the CMS command mode, but not nested within another EXEC procedure, the initial word EXEC may be omitted. This technique is known as "implied EXEC." The format of the EXEC command is:

```
| EXec | fname [args...]                                          |
```

fname specifies the filename of a file containing one or more CMS commands to be executed. The filetype of the file must be EXEC and the file must be fixed format, with a logical record length of up to 130 characters.

args specify the arguments to replace the numeric variables in the specified EXEC file. Within an EXEC file, up to thirty symbolic variables may be used, each one indicated by an ampersand (&) followed by an integer ranging from 1 to 30, to indicate values which are to be replaced when the EXEC file is executed.

The EXEC interpreter assigns the arguments to symbolic variables in the order in which they appear in the argument list. For example, each time an &1 appears in a line within the EXEC, the first argument specified with the EXEC command temporarily replaces the &1, the second argument specified with the EXEC command replaces &2, and so on, to the _n_th argument of the EXEC command.

If the percent sign (%) is used in place of an argument, the EXEC interpreter ignores the corresponding variable in all the commands that refer to that variable. If the EXEC file contains more variables than arguments given with the EXEC command, the EXEC interpreter assumes that the higher-numbered variables are missing, and CMS ignores the higher-numbered variables when it executes the command.

EXEC FILES

An EXEC file is a CMS data file that can contain CMS commands and EXEC control statements.

EXEC files can be created with the CMS Editor, by punching cards, by a user program, or by the CMS LISTFILE command. An EXEC file can contain up to 4096 lines. When you create an EXEC file using the CMS Editor, the record length defaults to 80 characters, unless you use the LRECL option of the EDIT command to specify up to 130 characters. However, the EXEC interpreter only processes the first 72 characters of the records (no matter what the record length actually is) unless you do one of the following:

1. Specify &BEGPUNCH ALL, to give you access to the first 80 characters of each record that follows, up to the next &END control statement.
2. Specify &BEGTYPE ALL or &BEGSTACK ALL, to give you access to all 130 characters of each record that follows, up to the next &END control statement.

If you do not specify the ALL option, EXEC will use only 72 characters of each record.

If you have a command line longer than 72 characters, you can stack that command following an &BEGSTACK ALL statement. This will enable ycu to issue a command line up to 130 characters long.

EXEC files consist of two types of statements: executable and nonexecutable. Each type is discussed below.

NONEXECUTABLE STATEMENTS

A nonexecutable statement in an EXEC file is one that begins with an asterisk (*) and may or may not contain text. These statements are for use as comment statements and are ignored during EXEC interpretation and processing.

EXECUTABLE STATEMENTS

An executable statement in an EXEC file is any statement that does not begin with an asterisk. These statements consist of data items which are strings of contiguous nonblank characters separated by blanks. Four classes of executable statements are recognized by the EXEC interpreter:

1. Null statements.
2. CMS commands.
3. Assignment statements.
4. Control statements.

Each of these statement classes is discussed under a separate heading. In addition, labels, user-defined variables, and special EXEC variables are discussed in relation to assignment statements and control statements.

Null Statements

A null statement is an executable statement in which the number of data items is zero. A blank line is a null statement.

Labels

A label in an EXEC procedure begins with a hyphen (dash), and contains up to seven additional alphameric characters. A label can be placed in front of (on the same line as) a CMS command or EXEC control statement. A label is often the object of a branching control statement, such as &GOTO or &LOOP.

When searching for a label, the EXEC interpreter examines only the first word on a line. Therefore, avoid any label names that appear as the first word of a line within the scope of an &BEGPUNCH, &BEGSTACK, or &BEGTYPE control statement.

CMS Commands

The EXEC interpreter considers an executable statement as a CMS command if the first data item does not start with an ampersand or asterisk. (A label can precede a CMS command.) CMS executes the command immediately. When CMS finishes execution, it returns control to the EXEC file, and places the completion code from the CMS command in the special EXEC variable &RETCODE.

Any valid CMS command may be included in an EXEC file. CP commands may be included by prefixing the desired command with the CMS "CP" command. Another EXEC procedure may be invoked by prefixing its filename with the CMS command "EXEC".

Note: When in CMS command mode, the commands CP and EXEC are not required; they can be implied. These prefixes are required only when these commands are invoked from an EXEC procedure or a user program, or when you have used the CMS command SET to set implied CP commands (IMPCP) or implied EXEC commands (IMPEX) off.

Variables

In an EXEC procedure you can use two types of variables: user-defined variables and special EXEC variables.

User-defined variable names begin with an ampersand (&) and contain up to seven additional characters. These variables can contain numeric or alphameric data, and must be initialized by the user. They can be used for almost any purpose, much as user-defined variables in a high-level language program are used.

Special EXEC variables follow the same naming conventions as user-defined variables, and also contain the same types of data (numeric or character values). The special EXEC variables, however, are initialized and set by the EXEC interpreter. The user can examine their contents, but in general, he cannot change them.

Variables are used extensively in assignment statements, which are discussed next.

Assignment Statements

An assignment statement is a statement in which a variable symbol is assigned a value. The statement takes the form:

```
&variable = ae
```

&variable is a variable symbol which must be preceded by the ampersand and followed by a blank.

ae is an arithmetic expression, the value of which is assigned to &variable each time the statement is executed. ae must be preceded by a blank and may be any of the following:

1. A single data item, such as ABC or 194.
2. An arithmetic expression, consisting of a sequence of data items that possess positive or negative integral values and are separated by plus or minus signs, such as 3 - 4 + -11 - 00.
3. A built-in function followed by its arguments, for example, &SUBSTR &1 2 1. (Built-in functions are discussed under a separate heading.)

As in other programming languages, the result of the expression to the right of the equal sign is placed in (assigned to) the variable named on the left of the equal sign (the target).

Leading zeros can be removed from a numeric quantity by performing some arithmetic operation on it and assigning the result to some variable. For example, the statements:

```
&X = 00012
&TYPE &X
&X = &X + 0
&TYPE &X
```

result in the printed lines:

```
00012
12
```

when the statements are executed in an EXEC procedure.

Note: The data item immediately following the target of an assignment statement must be a literal equal sign (=), and not an EXEC variable that has the value of an equal sign. Conversely, if an equal sign is to be the first data item following an EXEC control word (see "Control Statements" below), then it must be specified as an EXEC variable that has the value of an equal sign, and not as a literal equal sign. Otherwise, the statement is interpreted as an assignment statement, and (if it is valid as such) the control word is thereafter treated as a variable.

Control Statements

An executable statement is a control statement if the first data item is an EXEC control word and the second data item is not an equal sign, except that a control statement may be preceded by a label. Examples of control words are:

```
&GOTO
&EXIT
&IF
```

Control statements begin with a control word, which is usually followed by a list of data items and, in some cases, by additional lines of data. Control statements provide the means by which the user can control the execution of his EXEC procedure. The &IF control word, for example, can establish a conditional test, and a branch (&GOTO) can be taken if the test condition is met. Techniques for controlling EXEC logic are described in the section entitled "Building EXEC Procedures." The control words, and the rules for their use, are described individually in a later section entitled "EXEC Control Statements."

THE EXEC INTERPRETER

The EXEC interpreter examines each statement in an EXEC file when the file is invoked for execution. Except where specifically stated otherwise, data lines read from an EXEC file are truncated at column 72, and lines read from a terminal are truncated at column 130. All nonexecutable statements (comments) are ignored. Executable statements are interpreted, one at a time, in the following sequence.

1. Except for those commands that accept a line of data (an arbitrary, unsubstituted, collection of words) as an argument, the words forming a statement are "tokenized." That is, each word is treated as an eight character quantity and is padded with blanks or truncated, as necessary.

2. The tokens are then searched for the names of any special EXEC variables, which are replaced by their values. However, if a token is the target of an assignment statement, the name of the variable is retained. Tokens in a statement are then searched for the names of user-defined variables, which are replaced by values as follows:

   - Each token is scanned for ampersands (&), starting with the rightmost character of the token.

   - If an ampersand is found, then it, with the rest of the token to the right, is taken as a name. Then:

     IF it is the name of an active variable,
     THEN it is replaced (in the token) by the value assigned to the variable;

     IF it is the name of an EXEC keyword,
     THEN it is left unaltered;

     ELSE it is replaced (in the token) by blanks.

     An EXEC keyword is a control word, a built-in function, or either of the special tokens &$ and &*. Any token formed is padded with blanks or truncated (as necessary) to maintain a length of eight characters.

- Scanning resumes at the next character to the left, and the procedure is repeated until the token is exhausted. However, if the token is the target of an assignment statement, scanning for ampersands effectively stops before the leftmost character of the token is reached, because a variable name must be retained as the assignment target.

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

This processing makes it possible to simulate the effects of subscripted variables. For example, the sequence:

```
&X = 123
&TYPE ABC &X ABC&X 000000&X
```

yields the printed line:

```
ABC 123 ABC123 00000012
```

The sequence:

```
Entered                 After Substitution
&I = 2
...
&X&I = 5                &X2 = 5
&I = &I - 1             &I = 1
&X&I = &I + 1           &X1 = 2
...
&X = &X&I + &X&X&I      &X = &X1 + &X2 or &X = 2 + 5
&TYPE ANSWER IS &X
```

yields the printed line:

```
ANSWER IS 7
```

3. If at this point a token is entirely blank, it is discarded from the statement. The next token is deemed to immediately follow the previous one.

   A final token of blanks is added to any EXEC statement that is syntactically invalid if doing so makes the statement syntactically valid. For example:

   ```
   &BLANK =
   &TYPE
   &LOOP 3 &X NE
   ```

4. The statement is analyzed syntactically, and either passed to CMS or executed. When control returns from CMS, or the EXEC control statement has been executed, the EXEC interpreter examines the next statement in the file.

Note: The substitution process is performed each time a statement is interpreted. Thus, a variable could contain a different value each time the statement containing the variable is interpreted. In this way, the same line can be executed as an entirely different statement on different occasions. For example, in the following EXEC procedure:

```
&ARGS ASSEMBLE MYFILE
&SKP = 0
-EX &1 &2 &3 &4
&SKIP &SKP
&ARGS PRINT MYFILE LISTING
&SKP = 3
&GOTO -EX
```

the statement labeled -EX is executed the first time as:

```
     ASSEMBLE MYFILE
```

and the second time as:

```
     PRINT MYFILE LISTING
```

at which point the &SKIP statement branches to the end of the EXEC file.

EXEC CONTROL STATEMENTS

The EXEC control statements are grouped into three main categories:

- Execution control statements, which control the logic flow of the EXEC procedure.
- Built-in functions, which provide special services to the EXEC user.
- Special variables, which contain particular values or perform specific functions during EXEC processing.

Each of these categories is discussed in this section. The individual control statements are discussed under the category headings.

EXECUTION CONTROL STATEMENTS

The execution control statements determine what is to be done within an EXEC procedure. They are used to control logic flow; to communicate with a terminal, a user program, or the VM/370 system; or to create output files via the user's virtual punch.

Each of the execution control statements is fully described under a separate heading, including syntactic specifications. They are presented in alphabetical order. Usage examples and implementation techniques and suggestions are found in the section entitled "Building EXEC Procedures."

&ARGS CONTROL STATEMENT

The &ARGS control statement allows the user to redefine the value of one or more arguments during EXEC processing. The format of the &ARGS control statement is:

&ARGS	[arg1 [arg2 ... [argn]]]

&ARGS is used to redefine the variables &1, &2, ..., &n with the values specified by arg1, arg2, ..., argn, and reset the special variable &INDEX to the number of variables thus redefined. Up to 18 numeric variables can be redefined by an &ARGS control statement. The remaining variables (through &30) are set to blanks. (The &READ control word can be used to read a list of arguments from the terminal.)

&BEGPUNCH CONTROL STATEMENT

The &BEGPUNCH control statement heads a list of one or more lines to be spooled to the user's virtual punch. The list of lines to be punched is followed by the control statement &END. The format of the &BEGPUNCH control statement is:

```
┌──────────────────────────────────────────────────────────────────┐
|  &BEGPUNCH  |  [ALL]                                              |
├──────────────────────────────────────────────────────────────────┤
|   line1                                                           |
<   line2                                                           >
<     :                                                             >
<                                                                   >
|   linen                                                           |
├──────────────────────────────────────────────────────────────────┤
|    &END     |                                                     |
└──────────────────────────────────────────────────────────────────┘
```

&BEGPUNCH punches line1, line2, ..., linen to the card punch, without tokenizing them. No substitution is performed on any untokenized data. The lines are normally truncated at column 72 and padded with blanks to fill an 80-column card; truncation can be avoided by specifying the option ALL, in which case data can occupy columns 73 to 80. The list of lines to be punched is terminated by a line in which the control statement &END starts in column 1. (The &PUNCH control word can be used to spool a single line of tokens to the punch.)

&BEGSTACK CONTROL STATEMENT

The &BEGSTACK control statement heads a list of one or more lines to be placed in the user's console input stack. The list of lines to be stacked is followed by the control statement &END. The format of the &BEGSTACK control statement is:

```
┌──────────────────────────────────────────────────────────────────┐
|             |  ┌    ┐  ┌   ┐                                      |
|  &BEGSTACK  |  |FIFO|  |ALL|                                      |
|             |  |LIFO|  └   ┘                                      |
|             |  └    ┘                                             |
├──────────────────────────────────────────────────────────────────┤
|     line1                                                         |
<     line2                                                         >
<       :                                                           >
<                                                                   >
|     linen                                                         |
├──────────────────────────────────────────────────────────────────┤
|     &END    |                                                     |
└──────────────────────────────────────────────────────────────────┘
```

&BEGSTACK stacks line1, line2, ..., linen in the terminal input buffer, without tokenizing them. No substitution is performed on any untokenized data. The lines are normally stacked FIFO (first in, first out), but this can be changed by specifying the option LIFO (last in, first out). The lines are normally truncated at column 72, but this can be avoided by specifying the option ALL, in which case data can occupy columns 73 to 130. The list of lines to be stacked is terminated by a line in which the control statement &END starts in column 1. (The &STACK

control word can be used to place a single line of tokens in the console input buffer.)

&BEGTYPE CONTROL STATEMENT

The &BEGTYPE control statement heads a list of one or more lines to be typed on the user's terminal. The list of lines to be typed is followed by the control statement &END. The format of the &BEGTYPE control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
│ &BEGTYPE │ [ALL]                                                     │
├──────────────────────────────────────────────────────────────────────┤
│     line1                                                            │
<     line2                                                            >
<       :                                                              >
<                                                                      >
│     linen                                                            │
├──────────────────────────────────────────────────────────────────────┤
│     &END    │                                                        │
└──────────────────────────────────────────────────────────────────────┘
```

&BEGTYPE types line1, line2, ..., linen at the user's terminal, without tokenizing them. No substitution is performed on any untokenized data. The lines are normally truncated at column 72, but this can be avoided by specifying the option ALL, in which case data can occupy columns 73 to 130. The list of lines to be typed is terminated by a line in which the control statement &END starts in column 1. (The &TYPE control word can be used to type a single line of tokens at the terminal.)

&CONTINUE CONTROL STATEMENT

The &CONTINUE control statement instructs the EXEC interpreter to process the next statement in the EXEC file. The format of the &CONTINUE control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
│ &CONTINUE  │                                                         │
└──────────────────────────────────────────────────────────────────────┘
```

&CONTINUE is generally used in conjunction with an EXEC label (for example, -LAB &CONTINUE) to provide a branch address for &ERROR, &GOTO, and other branching statements. &CONTINUE is the default action taken when an error is detected in processing a CMS command.

&CONTROL CONTROL STATEMENT

The &CONTROL control statement instructs the EXEC interpreter how to handle the typing of various information messages at the user's terminal. The format of the &CONTROL control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
│                                                                      │
│  &CONTROL  │ ┌      ┐ ┌        ┐ ┌       ┐ ┌      ┐                  │
│            │ │OFF   │ │TIME    │ │PACK   │ │MSG   │                  │
│            │ │ERROR │ │NOTIME  │ │NOPACK │ │NOMSG │                  │
│            │ │CMS   │ └        ┘ └       ┘ └      ┘                  │
│            │ │ALL   │                                                │
│            │ └      ┘                                                │
└──────────────────────────────────────────────────────────────────────┘
```

&CONTROL sets the characteristics of terminal typecut of executicn messages until further notice.

OFF	Do not type any CMS commands as they are executed in this EXEC procedure, nor any return codes that may result.
ERROR	Type only those CMS commands that result in a nonzero return code, and type the return code.
CMS	Type each CMS command as it is executed, but type the return code only if it is nonzero.
ALL	Type every executable statement as it is executed, and type any nonzero return codes from CMS commands.
TIME	Include the time-of-day value with each CMS command printed in the execution summary. This operand is effective only if CMS cr ALL is specified.
NOTIME	Do not include the time-of-day value with CMS commands printed in the execution summary.
PACK	Pack the lines of the execution summary so that surplus blanks are removed from the typed lines.
NOPACK	Do not pack the lines of the execution summary.
MSG	Do not suppress the "FILE NOT FOUND" message if it is issued by the following commands when they are invoked from an EXEC procedure: ERASE, LISTFILE, RENAME, or STATE.
NOMSG	Suppress the "FILE NOT FOUNC" message if it is issued when the ERASE, LISTFILE, RENAME, or STATE commands are invoked from an EXEC procedure.

On entry to an EXEC file, the default settings for &CONTROL are:

```
CMS NOTIME PACK MSG
```

Each operand remains set until explicity reset by another &CONTROL statement.

&END CONTROL STATEMENT

The &END control statement marks the end of a list of one or more lines that began with an &BEGPUNCH, &BEGSTACK, or &BEGTYPE control statement. The format of the &END control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
│  &END  │                                                             │
└──────────────────────────────────────────────────────────────────────┘
```

The lines between the &BEGPUNCH, &BEGSTACK, or &BEGTYPE control statement and the terminating &END control statement are handled as untokenized lines of data. The &END control word must begin in column 1 to be recognized as a termination instruction.

&ERROR CONTROL STATEMENT

The &ERROR control statement specifies an action to be taken when a CMS command returns with an error return code. The format of the &ERROR control statement is:

```
┌─────────────────────────────────────────────────────────────────────┐
| &ERROR  | action                                                    |
└─────────────────────────────────────────────────────────────────────┘
```

&ERROR tells the EXEC interpreter to perform the specified action following any CMS command that yields an error return code (that is, a return code that is not zero). The action can be any executable statement.

What happens next depends upon the type and consequences of the action. If the action specified is a CMS command that also yields an error return code, then the EXEC interpreter types an error message and exits from the file; otherwise (unless the action causes a transfer of control), execution resumes at the line following the CMS command that caused the action to be executed. On entry to an EXEC file, the action is set to continue processing (that is, &CONTINUE).

The error message typed by the EXEC interpreter is:

```
(811) ERROR IN &ERROR ACTION
```

Additional information about this and other error messages is found under the heading "Recognizing EXEC Processing Errors" in the section entitled "Building EXEC Procedures."

Note: The words following the &ERROR control word are saved in an unscanned format, and substitution for any variables among them is performed dynamically (if the occasion arises) after obtaining a nonzero return code from a subsequent CMS command.

&EXIT CONTROL STATEMENT

The &EXIT control statement causes an exit from the current EXEC file. The format of the &EXIT control statement is:

```
┌─────────────────────────────────────────────────────────────────────┐
|         | ┌           ┐                                             |
|  &EXIT  | |return-code|                                             |
|         | |     0     |                                             |
|         | └           ┘                                             |
└─────────────────────────────────────────────────────────────────────┘
```

&EXIT tells the EXEC interpreter to exit to the next higher level of control with the specified return code. If the exit is taken from a nested EXEC procedure, control passes to the calling EXEC procedure. If the exit is taken from a first-level EXEC procedure, control passes to CMS.

If you do not specify a return code, a normal exit with a return code of zero is taken. You can specify the special variable &RETCODE to return the completion code from the most recently executed CMS command.

Alternatively, in the &EXIT control statement, you can specify your own variable, which you set to a numeric value earlier in the EXEC procedure. Then when you exit from the procedure, this value will appear as the return code. This can be useful if your EXEC procedure sets the variable to one value if it executes one branch and another value if it executes another branch; the EXIT return code will show which branch was executed.

&GOTO CONTROL STATEMENT

The &GOTO control statement transfers control to a specific location in the EXEC procedure. Execution then continues at the location that is branched to. The format of the &GOTO control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
│  &GOTO   │ ( TOP          )                                          │
│          │ { line-number  }                                          │
│          │ ( label        )                                          │
└──────────────────────────────────────────────────────────────────────┘
```

&GOTO transfers control to the top of the EXEC file, to a given line, or to the line starting with the specified label.

The first character of a label must be a hyphen (minus sign). You can attach a label to any executable statement as the first token on the line. Scanning for a label starts on the line following the &GOTO statement, goes to the end of the file, then to the top of the file, and (if unsuccessful) ends on the line above the &GOTO statement. If more than one statement in the file has the same label, the first one encountered by these rules satisfies the search.

&GOTO is commonly used to branch based on some conditional test, such as an &IF control statement.

&IF CONTROL STATEMENT

The &IF control statement allows you to conditionally execute statements in your EXEC procedure. The format of the &IF statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
│  &IF   │ ( token1 ) ( EQ ) ( token2 )  executable-statement          │
│        │ { &$     } ( NE ) { &$     }                                │
│        │ ( &*     ) { LT } ( &*     )                                │
│        │            { LE }                                           │
│        │            ( GT )                                           │
│        │            ( GE )                                           │
└──────────────────────────────────────────────────────────────────────┘
```

&IF tells the EXEC interpreter to determine whether the condition expressed is true or false. If the condition is true, the executable statement is processed. If the condition is false, processing continues with the next line in the EXEC procedure.

The analysis of the conditional expression proceeds as follows:

1. Token1 and token2 are syntactically examined, and any substitutions to be made are performed.
2. The tokens are compared according to the comparison operation specified.
3. If the comparison is true, the executable statement is processed
4. If the comparison is false, control passes to the next statement in the EXEC procedure.

The executable statement can be any valid EXEC control statement or CMS command. (Another &IF control statement can be specified as the executable statement for up to three levels of nesting.)

The special token &$ is interpreted as "any of the variables &1, &2, ...&n," and the special token &* is interpreted as "all of the variables &1, &2, ..., &n."

The comparison is arithmetic only when both comparands are numeric; in all other cases, a logical comparison is made. The comparison operators must be specified as shown. They are interpreted as follows:

EQ	equals
NE	not equal
LT	less than
LE	less than or equal to (not greater than)
GT	greater than
GE	greater than or equal to (not less than)

&LOOP CONTROL STATEMENT

The &LOOP control statement describes a loop through the FXEC procedure, including the conditions for exit from the loop. The format of the &LOOP control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
|  &LOOP   | { n     }{ m         }                                     |
|          | { label }{ condition }                                     |
└──────────────────────────────────────────────────────────────────────┘
```

&LOOP executes the following n lines, or down to (and including) the line starting with label, for m times, or until the specified condition is satisfied. A value must be specified for each parameter in the &LOOP statement.

The values of n and m (if given) must be positive integers from 0 to 4095.

The first character of the label name (if given) must be a hyphen, and the label must be specified, as the first token on the line, in an executable statement that lies below the &LOOP statement.

The syntax of the exit condition (if given) is the same as that in the &IF statement. The condition is always tested before executing the loop. Thus, if the condition is met, the loop is not executed.

When loop execution is complete, control passes to the next statement following the end of the loop. Loops may be nested up to four levels deep. All nested loops may end at the same label, if desired.

&PUNCH CONTROL STATEMENT

The &PUNCH control statement causes a string of eight-character tokens to be directed to the user's virtual punch. The format of the &PUNCH control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
| &PUNCH  |  tok1 [tok2 ... [tokn]]                                    |
└──────────────────────────────────────────────────────────────────────┘
```

&PUNCH punches a card containing the tokens tok1, tok2, ..., tokn. The card is padded with blanks or truncated, as necessary, to fill an 80-column card. The tokens, as punched, are separated from each other by a single blank. Any tokens longer than eight characters are left justified and truncated on the right.

To punch one or more lines of untokenized data, use the &BEGPUNCH and &END control statements.

&READ CONTROL STATEMENT

The &READ control statement reads one or more lines of data from the user's terminal. The format of the &READ control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
|         |  ┌                            ┐                            |
|  &READ  |  |n                           |                            |
|         |  |1                           |                            |
|         |  |ARGS                        |                            |
|         |  |VARS  var1 [var2 ... [varn]]|                            |
|         |  └                            ┘                            |
└──────────────────────────────────────────────────────────────────────┘
```

&READ reads the next n lines from the terminal and treats them as if they had been in the EXEC file; or it reads a single line, assigns the tokens in it to the arguments &1, &2, ..., &n, and resets the special variable &INDEX to the number of arguments thus set; or it reads a single line and assigns the tokens in it to the variables var1, var2, ..., varn (this does not reset &INDEX).

If n is specified, reading from the terminal stops when n lines have been read, or when an &LOOP statement or a statement that transfers control is encountered. If an &READ statement is encountered, the number of lines to be read by it is added to the number outstanding.

The variables var1, var2, ..., varn, if specified, are scanned in the same way as if they appeared on the left-hand side of an assignment statement. If no variable names are specified, no data is read from the terminal. Any data entered is lost.

If no operands are specified, &READ 1 is assumed by default.

&SKIP CONTROL STATEMENT

The &SKIP control statement causes a specified number of lines in the EXEC file to be skipped. The format of the &SKIP control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
│          │ ┌   ┐                                                      │
│  &SKIP   │ │ n │                                                      │
│          │ │ 1 │                                                      │
│          │ └   ┘                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

&SKIP passes over and ignores the next n lines of the EXEC file if n is greater than zero. If n is less then zero, &SKIP transfers control to the line that is n lines above the current line. If n equals zero, &SKIP transfers control to the next line. If end-of-file is reached during the skip operation, the EXEC file returns control to CMS.

If n specifies a position before the beginning of the EXEC file, an error results. See the discussion of error messages under the heading "Recognizing EXEC Processing Errors" in the "Building EXEC Procedures" section.

If no number is specified, &SKIP 1 is assumed by default.

&SPACE CONTROL STATEMENT

The &SPACE control statement types a specified number of blank lines at the user's terminal. The format of the &SPACE control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
│           │ ┌   ┐                                                     │
│  &SPACE   │ │ n │                                                     │
│           │ │ 1 │                                                     │
│           │ └   ┘                                                     │
└──────────────────────────────────────────────────────────────────────┘
```

If no number is specified, &SPACE 1 is assumed by default.

&STACK CONTROL STATEMENT

The &STACK control statement causes a single data line to be stacked in the terminal input buffer. The format of the &STACK control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
│          │ ┌    ┐                                                     │
│ &STACK   │ │FIFO│  [tok1 [tok2 ... [tokn ]]]                          │
│          │ │LIFO│                                                     │
│          │ └    ┘                                                     │
└──────────────────────────────────────────────────────────────────────┘
```

&STACK places a line in the terminal input buffer containing the tokens tok1, tok2, ..., tokn, or places a null line in the buffer if no tokens are specified. The line is normally stacked FIFO (first in, first out), but this can be changed by specifying the LIFO option (last in, first out). The tokens, as stacked, are separated from each other by a single blank.

You can stack CMS Immediate commands in the terminal input buffer, just as you can stack any other commands. In fact, you can stack any data at all in the terminal input buffer. The data need not be a valid command.

Note: The Logical Line End character is a hexadecimal '15'. CMS routines do not process the symbolic pound sign (#) as a logical line end character (that is, do not translate it to X'15') unless the user does the following:

1. Changes or turns off the Logical Line End character via the CP TERMINAL command.
2. Uses the EDIT subcommand ALTER to cause the CMS Editor to automatically convert the # character (or some other character) to X'15'.

To enter one or more untokenized lines into the terminal input buffer, use the &BEGSTACK and &END control statements.

&TIME CONTROL STATEMENT

The &TIME control statement determines what timing information is typed at the user's terminal after each CMS command is executed. The format of the &TIME control statement is:

```
┌──────────────────────────────────────────────────────────────────────┐
│          │ ┌     ┐                                                    │
│ &TIME    │ │ON   │                                                    │
│          │ │OFF  │                                                    │
│          │ │RESET│                                                    │
│          │ │TYPE │                                                    │
│          │ └     ┘                                                    │
└──────────────────────────────────────────────────────────────────────┘
```

The &TIME control statement can be used to type timing information in the form:

```
T=x.xx/y.yy hh:mm:ss
```

where:

x.xx	is the virtual CPU time used since it was last reset in the current EXEC file.
y.yy	is the total CPU time used since it was last reset in the current EXEC file.
hh:mm:ss	is the actual time of day in hours:minutes:seconds.

The CPU times are set to zero before the execution of the first statement in the EXEC file, and are set to zero (reset) whenever timing information is printed.

ON	Resets the CPU times before every CMS command, and prints the timing information on return. If the &CONTROL control statement is set to CMS or ALL, the printing of the timing information is followed by a blank line.
OFF	Does not automatically reset the CPU times before every CMS command, nor does it print the timing information on return. OFF is the initial setting.
RESET	Performs an immediate reset of the CPU times.
TYPE	Types the current timing information (and resets the CPU times).

&TYPE CONTROL STATEMENT

The &TYPE control statement prints a line of tokens at the user's terminal. The format of the &TYPE control statement is:

&TYPE	tok1 [tok2 ... [tokn]]

&TYPE prints at the terminal a line containing the tokens tok1, tok2, ..., tokn. The tokens, when typed, are separated from each other by a single blank. To print one or more untokenized lines at the terminal, use the &BEGTYPE and &END control statements.

BUILT-IN FUNCTIONS

An EXEC built-in function consists of the name of the function and, usually, a list of arguments. Built-in function names are EXEC keywords, and start with an ampersand. With the exception of &LITERAL, a built-in function is recognized only if it appears as a token immediately following the equal sign of an assignment statement. Each of the built-in functions is described under a separate heading.

&CONCAT BUILT-IN FUNCTION

The &CONCAT function creates a concatenated string of tokens to be assigned to a user-defined variable. The format of the &CONCAT function is:

&CONCAT	tok1 [tok2 ... [tokn]]

&CONCAT concatenates the tokens tok1, tok2, ..., tokn into a single token, with a maximum length of eight characters. This function is recognized only on the right-hand side of an assignment statement. For example:

```
&A = **
    •
    •
    •
&B = &CONCAT XX &A 45
&TYPE &B
```

Results in the printed line:

```
XX**45
```

If the concatenated token is longer than eight characters, the data is left justified and truncated on the right.

&DATATYPE BUILT-IN FUNCTION

The &DATATYPE function determines whether the contents of the specified token is alphabetic or numeric data. The format of the &DATATYPE function is:

&DATATYPE	token

The result of the &DATATYPE function has the value NUM or CHAR, depending on the data type of the specified token. This function is recognized only on the right-hand side of an assignment statement.

&LENGTH BUILT-IN FUNCTION

The &LENGTH function determines the number of nonblank characters in the specified token. The format of the &LENGTH function is:

&LENGTH	token

The result of the &LENGTH function is the number of nonblank characters in the specified token. This function is recognized only on the right-hand side of an assignment statement.

&LITERAL BUILT-IN FUNCTION

The &LITERAL function instructs the EXEC interpreter to handle the specified token literally, without substituting for any variables in it. The format of the &LITERAL statement is:

&LITERAL	token

The &LITERAL function causes the EXEC interpreter to use the literal value of the specified token without substitution. This function is recognized anywhere in an executable statement. For example:

```
     &X = **
     &TYPE &LITERAL &X EQUALS &X
```

Results in the printed line:

```
     &X EQUALS **
```

&SUBSTR BUILT-IN FUNCTION

The &SUBSTR function creates a substring of specified characters from a specified token. The format of the &SUBSTR function is:

&SUBSTR	token i [j]

The result of the &SUBSTR function extracts part of the specified token that starts at character i, with length of j; or that starts at character i and runs to the end of the token if j is not specified. The values of i and j (if given) must be positive integers. This function is recognized only on the right-hand side of an assignment statement. For example:

```
     &A = &SUBSTR ABCDE 2 3
     &TYPE &A
```

Results in the printed line:

```
     BCD
```

SPECIAL VARIABLES

EXEC special variables are of two types: numeric and keyword. The numeric variables are those from &0 through &30. Keyword variables are those that have special meaning to the EXEC interpreter. They include the special variables &* and &$, which are used to define conditions in &IF and &LOOP control statements. The special variables are discussed under separate headings in alphabetical order.

&EXEC SPECIAL VARIABLE

The &EXEC special variable is the name of the EXEC file. This variable cannot be set explicitly by the user, but it can be examined and tested.

&N SPECIAL VARIABLE

The &n special variable represents the numeric variables &0 through &30. When an EXEC file is invoked, &0 is set to its filename by the EXEC interpreter. The other numeric variables from &1 to &30 are initialized to arguments that are passed to the EXEC file (if any). Each numeric variable can contain up to eight alphameric characters.

The numeric variable &n is ignored when n is negative or greater than 30, or when n is greater than the number of arguments supplied when the EXEC command is issued. The numeric variables can be reset by either an &ARGS or &READ ARGS control statement.

An argument can be set to blanks by assigning it a percent sign (%) when invoking the EXEC procedure, in an &ARGS control statement, or in an &READ ARGS control statement.

&GLOBAL SPECIAL VARIABLE

The &GLOBAL special variable contains the recursion level of the EXEC interpreter. Since the EXEC interpreter can handle up to 19 levels of recursion, the value of &GLOBAL can range from 0 through 19. This variable cannot be set explicitly by the user, but it can be examined and tested.

&GLOBALN SPECIAL VARIABLE

The &GLOBALn special variable represents the variables &GLOBAL0 through &GLOBAL9. These variables hold integral numeric values that can be set explicitly by the user. They are all initially set to 1. Unlike other EXEC variables, these can be used to communicate between different recursion levels of the EXEC interpreter.

Note: The EXEC interpreter can handle up to 19 levels of recursion.

&INDEX SPECIAL VARIABLE

The &INDEX special variable contains the number of arguments passed to the EXEC procedure. Since up to 30 arguments can be passed to an EXEC procedure, the value of &INDEX can range from 0 through 30.

Although the user does not set this variable explicitly, it is reset by an &ARGS or &READ ARGS control statement. &INDEX can be examined to determine the number of active arguments in the EXEC procedure.

&LINENUM SPECIAL VARIABLE

The &LINENUM special variable contains the current line number in the EXEC file. Since an EXEC file can contain up to 4096 lines, the value of &LINENUM can range from 0 to 4095. This variable cannot be set explicitly by the user, but it can be examined and tested.

&READFLAG SPECIAL VARIABLE

The &READFLAG special variable contains the value CONSOLE or STACK, depending on whether an attempt to read from the user's terminal would obtain a physical line from the terminal or a logical line from the terminal input buffer (console stack). This variable cannot be set explicitly by the user, but it can be examined and tested.

&RETCODE SPECIAL VARIABLE

The &RETCODE special variable contains the return code from the most recently executed CMS command. &RETCODE can contain only integral numeric values (positive or negative), and is set after each CMS command is executed. This variable can be examined, tested, and changed by the user, but changing it is not recommended. To specify a return code on exit from an EXEC procedure, use the &EXIT control statement.

&TYPEFLAG SPECIAL VARIABLE

The &TYPEFLAG special variable contains the value RT (resume typing) or HT (halt typing), depending on the value of the console output flag. This variable cannot be set explicitly by the user, but it can be examined and tested.

USER-DEFINED VARIABLES

An EXEC user can define and manipulate his own variables in an EXEC procedure. The name of a user-defined variable must begin with an ampersand (&) that is followed by a string of up to seven alphameric characters, at least one of which is not a number.

When you choose a name for a variable, be careful not to choose a name that is an EXEC special variable (&EXEC, &RETCODE, etc.).

The EXEC subcommands that begin with an ampersand can also be used as user variables, but then they will not be executed as subcommands. For example, if you specify:

```
&TYPE = QUERY
&TYPE USERS
```

the EXEC interpreter will execute the command:

```
QUERY USERS
```

User-defined variables can be used for any purpose in an EXEC procedure, and can contain any type of data. The user is, therefore, responsible for validating any data in a variable before using it. For example, a character variable should not be added to a numeric variable under normal circumstances.

A user variable is most often defined by entering it into the EXEC file as the receiving field of an assignment statement. It is usually initialized by the expression that makes up the other half of the same assignment statement. For example, to define a counter and initialize it to 99, the user could enter:

```
&COUNTER = 99
```

This counter could then be manipulated by such control statements as:

```
&LOOP -LOOPEND &COUNTER LE 0
&COUNTER = &COUNTER - 1
-LOOPEND &CONTINUE
```

This loop decreases the value of the counter by one each time through the loop, then exits when the counter value equals zero.

The user can define similar variables throughout his EXEC file. Further examples of user variables are found throughout the next section, "Building EXEC Procedures."

BUILDING EXEC PROCEDURES

Previous sections have described the functions and syntax of the EXEC control statements, built-in functions, and special variables. This section shows you how to use these EXEC facilities, along with CMS commands, to simplify operation of your CMS virtual machine. Using EXEC procedures, you can:

- Catalog frequently used sequences of commands, such as those used to assemble or compile a program.
- Create interactive procedures for such purposes as form letters, teaching a new user about the system, or generating reports.
- Control the execution of jobs by the CMS Batch Facility.
- Define special EXEC files, called EDIT macros, that can be used by the CMS Editor.
- Specify the operating characteristics of your virtual machine in a special EXEC file called the PROFILE EXEC.

The following discussions illustrate some of the techniques you can use to create EXEC files and control their execution.

PASSING ARGUMENTS TO AN EXEC PROCEDURE

Arguments can be passed to an EXEC procedure in two ways:

1. By specifying them when the EXEC file is invoked.
2. By entering them from the terminal in response to an &READ ARGS control statement.

In addition, arguments can be set explicitly within the EXEC file by issuing an &ARGS control statement. The arguments are assigned to the numeric variables &1, &2, ..., &n, and the special variable &INDEX is set to the number of arguments that are assigned to numeric variables.

While up to 30 arguments can be passed to an EXEC file, only 19 tokens can appear in any single EXEC statement. Thus, to pass arguments to all 30 numeric variables, you may need to use the &READ ARGS control statement. For example, to assign values to the 30 numeric EXEC variables, you could use the following EXEC statement:

```
&READ ARGS
```

This statement reads an input line from the terminal and assigns the arguments specified to their corresponding numeric variables. Thus, if 30 arguments are entered in response, then all 30 numeric variables are set.

An argument can be set to blanks by assigning it a percent sign (%) when invoking the EXEC procedure, when issuing an &ARGS control statement, or when responding to an &READ ARGS control statement. For example, the statements:

```
&ARGS A % B
&TYPE &1 &2 ** &3
```

result in the printed line:

```
A ** B
```

After the arguments have been assigned to their corresponding numeric variables, they can be examined, tested, and manipulated at will. Two useful tests are discussed under the next two headings.

CHECKING FOR THE PROPER NUMBER OF ARGUMENTS

Although you can pass up to 30 arguments to an EXEC procedure, it is more common for an EXEC procedure to be designed to expect a few specific arguments that determine how it is executed.

One way to find out if all the expected arguments are present is to check the &INDEX special variable. For example, if you create an EXEC procedure that manipulates a CMS file, you may need to know the filename and filetype of the file. In this case, your EXEC procedure can check to make sure that at least two arguments are entered (which could be a filename and filetype), as follows:

```
&IF &INDEX LT 2 &EXIT 16
```

The result of this statement is to execute the next instruction in the EXEC file if the value of &INDEX is equal to or greater than 2, or to exit from the current EXEC procedure with a return code of 16 if the value of &INDEX is less than 2.

After you have determined that the number of arguments passed is correct, then you can examine the individual arguments for correctness.

CHECKING FOR THE LENGTH OF AN ARGUMENT

An argument passed to an EXEC procedure can be up to eight characters long. Arguments longer than eight characters are left justified and truncated on the right when they are assigned to their corresponding numeric variables.

In many cases, you may know that an argument should be a specific number of characters. When the proper length of an argument is known, you can use the &LENGTH built-in function to see if the entered data is the correct length. For example, suppose that the first argument can be a number up to five digits long. In this case, you can make sure that this limit is observed by coding:

```
&LIMIT = &LENGTH &1
&IF &LIMIT GT 5 &EXIT &LIMIT
```

The result of these statements is to assign the length of the variable &1 to the user-defined variable &LIMIT, then if the value in &LIMIT is equal to or less than 5, execute the next instruction in the EXEC file. If the value of &LIMIT is greater than 5, an exit is taken from the EXEC procedure with a return code that specifies the erroneous value in &LIMIT.

If your EXEC procedure expects an argument that is exactly five characters long, you can perform a similar test by changing the GT (greater than) to NE (not equal) in the &IF statement.

After the preliminary checking of arguments has been done, you can perform any other tests that seem necessary. In many cases, you may want to check for specific values. Some techniques for doing this checking are discussed under the next heading.

CHECKING FOR A SPECIFIC ARGUMENT

When your EXEC procedure expects a specific value to be passed in the argument list, you can check for the presence of the argument in two ways, depending on its positional importance.

When the argument is expected to be in a specific location in the list, you can check for it in that position. For example, if you create an EXEC procedure to handle files with a specific filetype, you can make sure that the filetype is entered as the second argument. If the filetype you expect is ASSEMBLE, the checking statement could be:

```
     &IF &2 NE ASSEMBLE &EXIT 4
```

In this case, if the second argument is ASSEMBLE, the next statement in the EXEC procedure is executed. Otherwise, an exit is taken with a return code of 4.

When it does not matter where in the argument list the expected value appears, you could code:

```
     &IF &* NE ASSEMBLE &EXIT 4
```

In this case, the exit is taken only when none of the arguments passed has a value of ASSEMBLE. If any one of them is the character string ASSEMBLE, then the next sequential statement in the EXEC procedure is executed.

COMMUNICATING WITH A TERMINAL

One of the facilities available to the EXEC user is the ability to communicate with an interactive terminal. EXEC procedures can be designed to display informational messages, prompt the user for specific data, produce form letters, and perform other similar functions. This section shows you how to create EXEC control statements to do some of these things.

READING DATA FROM A TERMINAL

When an EXEC procedure is invoked, arguments can be passed to it in the invoking command line. After the EXEC procedure begins execution, however, the only way you can pass new data to it from your terminal is in response to an &READ control statement. For example, suppose you want to enter some CMS command under a particular set of circumstances, but not at any other time. You can avoid testing for the particular set of circumstances in the EXEC procedure by simply issuing the &READ control statement, as follows:

```
&READ
```

The person at the terminal now has to decide what to enter. If the situation calls for the defined CMS command, it can be entered; if the required conditions are not met, he can simply press the Return key to enter a null line. The entered line is treated as though it had been in the EXEC file all along.

If you want to read in some new arguments, simply code:

```
&READ ARGS
```

The arguments entered at the terminal are tokenized and assigned to their corresponding numeric variables (the first argument to &1, the second to &2, and so on). Up to 30 arguments can be entered in response to an &READ ARGS control statement.

When you want to assign arguments to specific variables in the EXEC procedure, you can do this by coding:

```
&READ VARS &varname1 &varname2 [...]
```

where &varname1 is the name of the variable to which the first argument is assigned, &varname2 is the name of the variable to which the next argument is assigned, and so on. Since a line in an EXEC procedure can contain up to 19 tokens, up to 17 named variables can be set by the response to a single &READ VARS control statement.

Note: Be aware of the difference between &READ ARGS and &READ VARS. &READ ARGS always assigns the response tokens to the numeric variables &1, &2, etc. &READ VARS assigns the response tokens to the variables specified in the variable list. Thus, if you want to pass more than 30 data items to the EXEC, you can assign the first 30 to the numeric variables with a response to &READ ARGS, then assign as many more as desired to named variables with responses to &READ VARS statements.

Just as you can instruct an EXEC procedure to read data from your terminal, so can you have the EXEC procedure type data at your terminal. The next section shows you some of the techniques for typing data at a terminal.

TYPING DATA AT A TERMINAL

An EXEC procedure can be coded to type three different kinds of data at a terminal:

1. Lines of tokenized data, which are typed one at a time by the &TYPE control statement.
2. Lines of untokenized data, which are typed by specifying the &BEGTYPE control statement, then the lines of data, then the &END control statement.
3. Records from a CMS file, which are typed by specifying the CMS TYPE command with appropriate parameters.

You can also check a special EXEC variable (&TYPEFLAG) to determine whether or not to send output to a terminal. For example, if the terminal has suppressed typing, it is a waste of system resources to send data to it.

In addition to these methods of explicitly controlling typing of data at a terminal, the user can occasionally receive messages from the EXEC interpreter and CMS commands. You can control the typeout of some of these messages by specifying certain options of the &CONTROL statement, as described in the section on "Checking for Execution Errors."

Typing a Single Line of Tokens

The &TYPE control statement can be used to type a line of tokens at a terminal. The tokens can be user-defined variables, special EXEC variables, self-defining terms, or any of the 30 numeric variables. If a token in an &TYPE statement is longer than eight characters, it is left-justified and truncated on the right.

To type a line of data to a terminal, for example, you could enter:

```
&TYPE THIS IS THE MESSAGE
```

These tokens are all self-defining terms less than eight characters long. The resulting typed line appears at the terminal as:

```
THIS IS THE MESSAGE
```

To type the value of an EXEC variable, simply enter the variable as one of the tokens in the &TYPE statement, as follows:

```
&TYPE &INDEX &RETCODE
```

The terminal output will display the current values of &INDEX and &RETCODE. For example,

```
3    0
```

would type at the terminal if the current value of &INDEX is 3 and the current value of &RETCODE is 0.

Up to 18 tokens can be typed by a single &TYPE control statement. If you want to type a line longer than 18 words, or if you want to type one or more lines of untokenized data, you can use the &BEGTYPE and &END control statements, which are described next.

Note: If you want to type variable data, such as the value of &INDEX, you must use &TYPE instead of &BEGTYPE.

Typing More Than One Line of Data

When you want to type a word longer than eight characters at a terminal, you need to use the &BEGTYPE control statement, since the &TYPE statement edits data items into 8-character tokens. One or more data lines can be specified between the &BEGTYPE statement and a following &END control statement, which terminates the list of data lines.

Under normal circumstances, data lines are truncated at column 72. If you want to type data lines longer than this (up to 130 characters), you can specify the ALL option on the &BEGTYPE control statement, as follows:

```
&BEGTYPE ALL
data line of 85 characters
data line of 108 characters
data line of 63 characters
&END
```

This specification turns off the truncation at column 72, and permits the first two lines (of 85 and 108 characters) to type as specified.

Because the data lines controlled by &BEGTYPE are not tokenized, no substitution for EXEC variables is made before typing begins. Thus, if you want to type the contents of an EXEC variable, you must use the &TYPE control word (you cannot use &BEGTYPE). For example, the entered lines:

```
&BEGTYPE
&INDEX ARGUMENTS WERE PASSED
&END
```

would type at the terminal, when the EXEC procedure is executed, as follows:

```
&INDEX ARGUMENTS WERE PASSED
```

The &INDEX specification is not recognized as an EXEC variable because it appears in an untokenized data line. Note that if the &TYPE control statement is used to type this data line, the word ARGUMENTS is truncated to eight characters, as follows:

```
&TYPE &INDEX ARGUMENTS WERE PASSED
```

yields, when the value of &INDEX is 3:

```
3 ARGUMENT WERE PASSED
```

where the last letter of ARGUMENTS has been truncated.

In addition to using the &TYPE and &BEGTYPE control statements, you can use the CMS TYPE command to display part or all of a selected CMS file at a terminal. Use of the CMS TYPE command is discussed next.

Typing a CMS File

You can use the CMS TYPE command to type part or all of a CMS file at a terminal. A complete description of the TYPE command is found in the *VM/370: Command Language Guide for General Users*, Order No. GC20-1804. For purposes of this discussion, however, you need to know that the the filename and filetype of the file to be typed must be entered, and that you can specify the lines where typing is to begin and end, as follows:

```
TYPE filename filetype [begin [end]]
```

The filename, filetype, begin, and end values can be specified when the EXEC file is created, or they can be coded as EXEC variables and assigned values when the EXEC procedure is executed.

Note: If you do not specify a beginning line for typing the first line of the file is assumed. If you do not specify an ending line for typing, the last line of the file is assumed.

For example, to type the first 23 lines of a CMS file named PGM3 ASSEMBLE at a terminal, you could create an EXEC procedure that explicitly defines these values, as follows:

```
TYPE PGM3 ASSEMBLE 1 23
```

If you want to make your EXEC procedure more general, however, you can let the TYPE parameters be numeric variables that are assigned values when the command line is analyzed by the EXEC interpreter. For example, the line:

```
TYPE &1 &2 &3 &4
```

can be set up to type any part of any CMS file by invoking the EXEC that contains it with different arguments. If the name of the EXEC file containing this line is TYPEOUT, a specification of:

```
TYPEOUT MYFILE EXEC
```

causes all the lines in the CMS file named MYFILE EXEC to be typed at the requestor's terminal, while a specification of:

```
TYPEOUT PGM9 COBOL 1 130
```

causes lines 1 through 130 in the CMS file named PGM9 COBOL to be typed out at the requestor's terminal.

Although the examples shown here do little more than the TYPE command itself, you can see how inclusion of a generalized CMS TYPE command in an EXEC file can extend the communication facilities between EXEC procedure and terminal. The EXEC procedure can find out what values to assign to the TYPE parameters by issuing an &READ ARGS control statement, or by using arguments that were passed when the EXEC procedure was invoked.

Before typing any data at a terminal, your EXEC procedure may want to check the output flag to make sure the terminal is receiving typed data. This procedure is discussed next.

Examining the Output Flag

The special EXEC variable &TYPEFLAG always contains one of two character string values, RT or HT. The RT characters mean that the terminal is accepting typeout, while the HT characters mean that the terminal has suppressed typeout. Since the typeout to a terminal involves overhead in both CMS and CP, you can make more efficient use of system resources if you do not use CMS and CP functions when they are not required.

When a terminal has suppressed typing, you can set up an EXEC control statement that avoids any attempt to type data at the terminal, as follows:

```
&IF &TYPEFLAG EQ HT &GOTO -NOTYPE
&TYPE ANY NUMBER OF DATA LINES
-NOTYPE &CONTINUE
```

The execution of this EXEC segment is determined by the setting of &TYPEFLAG. If it contains the value RT, the branch to -NOTYPE is not taken, and the next instruction is executed. If it contains the value HT, the branch to -NOTYPE is taken, and no data lines are typed at the terminal.

Note: While you cannot explicitly set the &TYPEFLAG variable in your EXEC procedure, you can examine its contents at any time, and move its value to some other variable if desired.

LOGIC CONTROL IN AN EXEC PROCEDURE

Another major facility available to the EXEC user is the ability to control the execution of an EXEC procedure by testing conditions and branching to different statements based on the results of testing. This section shows you how to set up conditional execution paths in an EXEC procedure, including such techniques as IF/THEN, GOTO, and fixed loops.

LABELS IN AN EXEC PROCEDURE

In many instances, an execution control statement in an EXEC procedure causes a branch to a particular statement that is identified by a label. The rules and conventions for creating syntactically corrent EXEC labels are relatively simple:

1. A label must begin with a hyphen (dash), and must have at least one additional character following the hyphen.
2. Up to seven additional alphameric characters may follow the hyphen (with no intervening blanks).
3. A label name may be the object of an &GOTO or &LOOP control statement.
4. A label that is branched to must be the first token on a line. It may stand by itself, with no other tokens on the line, or it may precede an executable CMS command or EXEC control statement.

The following are examples of correct use of labels:

```
&GOTO -LAB1
-LAB1
-LAB2 &CONTINUE
-CHECK &IF &INDEX EQ 0 &GOTO -EXIT
&IF &INDEX LT 5 &SKIP
-EXIT &EXIT 4
&TYPE &LITERAL &INDEX VALUE IS &INDEX
```

The following examples of label usage are incorrect for the reas indicated:

-LABELING	More than 8 characters
&SKIP -4LINES	Not object of &GOTO or &LOOP
-	No alphameric characters included
&EXIT -EXIT	Label not first token on line
- BLANKS	Blanks between hyphen and other characters

You will find that EXEC labels are useful in controlling the logic flow through your EXEC procedures. Further examples of label usage are found throughout the rest of this book.

CONDITIONAL EXECUTION WITH THE &IF STATEMENT

The main tool available to you for controlling conditional execution in an EXEC procedure is the &IF control statement. The &IF control statement provides the decision-making ability that you need to set up conditional branches in your EXEC procedure.

One approach to decision-making with &IF is to compare the equality (or inequality) of two tokens, and perform some action based on the result of the comparison. When the comparison specified is true, the executable statement is executed. When the comparison is false, control passes to the next sequential statement in the EXEC procedure. An example of a simple &IF statement is:

```
&IF 1 EQ 2 &TYPE MATCH FOUND
```

Since the equality specified is false, the executable statement (&TYPE MATCH FOUND) is not executed, and control passes to the next statement in the EXEC procedure.

Although this example is coded correctly, it is not very useful. In fact, most &IF statements establish a comparison between a variable and a constant, or between two or more variables. For example, if a terminal user could properly enter a YES or NO response to a prompting message issued to him, you could set up &IF statements to check the response as follows:

```
&READ ARGS
&IF &1 EQ YES &GOTO -YESANS
&IF &1 EQ NO &GOTO -NOANS
&TYPE &1 IS NOT A VALID RESPONSE (MUST BE YES OR NO)
&EXIT
-YESANS
   .
   .
   .
-NOANS
   .
   .
   .
```

In this example, the branch to -YESANS is taken if the entered argument is YES; otherwise, control passes to the next &IF statement. The branch to -NOANS is then taken if the argument is NO; otherwise, control passes to the &TYPE statement, which types the entered argument in an error message and exits.

The test performed in an &IF statement need not be a simple test of equality between two tokens; other types of comparisons can be tested, and more than two variables can be involved. The tests that can be performed are:

Symbol	Meaning
EQ	A equals B
NE	A does not equal B
LT	A is less than B
LE	A is less than or equal to B (not greater than)
GT	A is greater than B
GE	A is greater than or equal to B (not less than)

The special tokens &$ and &* can be specified to include the entire range of numeric variables &1 through &30, as follows:

- The special token &$ is interpreted as "any of the variables &1, &2, ..., &30." That is, if the value of any one of the numeric variables satisfies the established condition, then the &IF statement is considered to be true. The statement is false only when none of the variables fulfills the specified requirements.
- The special token &* is interpreted as "all of the variables &1, &2, ..., &30." That is, if the value of each of the numeric variables satisfies the established condition, then the &IF statement is considered to be true. The statement is false when at least one of the variables fails to meet the specified requirements.

If an &IF statement specifies a special token (&* or &$) that is null because no values were supplied for any of the numeric variables, the token cannot be successfully compared. The &IF statement is therefore considered a null statement. Execution continues at the next sequential statement.

As an example of using special tokens, suppose you want to make sure that some particular value is passed to the EXEC. You can check to see if any of the arguments satisfy this condition, as follows:

```
&IF &$ EQ PRINT &SKIP 2
&TYPE PARM LIST MUST INCLUDE PRINT
&EXIT
```

In this example, the path to the &TYPE statement is taken only when none of the arguments passed to the EXEC equal the character string PRINT.

The action to be executed as a result of &IF testing is frequently an &GOTO control statement. The next discussion examines the use of &GOTO in conjunction with &IF, as well as by itself as an unconditional branch instruction.

BRANCHING WITH THE &GOTO STATEMENT

The &GOTO options allow you to transfer control within your EXEC procedure in three ways:

1. Directly to the top of the EXEC file (&GOTO TOP).

2. To a particular line within the EXEC file (&GOTO linenum, where linenum specifies the line number to which control is passed).

3. To a specified EXEC label somewhere in the EXEC file (&GOTO label, where label specifies the label to which control is passed).

The scan for a line number or label begins on the line following the &GOTO specification, proceeds to the bottom of the file, then wraps around to the top of the file and continues to the line immediately preceding the &GOTO specification.

If the label or line number is not found during the scan, an error exit from the EXEC procedure is taken and an error message is typed. If the label or line number is found, control is passed to that location and execution continues.

The &GOTO control statement can be coded wherever an executable statement is permitted in an EXEC procedure. One of its common uses is in conjunction with the &IF control statement. For example, in the statement:

```
&IF &INDEX EQ 0 &GOTO -ERROR
```

the branch to the statment labeled -ERROR is taken only when the value of the &INDEX special variable is zero. In all other cases, control passes to the next sequential statement in the EXEC procedure.

Another common use of &GOTO is to specify where to pass control if an error occurs in CMS command processing. You can determine whether or not an error occurred in CMS command processing by examining the special variable &RETCODE, but you may want to perform this analysis in a single subroutine instead of checking the return code after each CMS command completes execution. To pass control to a subroutine labeled -FINDERR, for example, you could code the &ERROR control statement as follows:

```
&ERROR &GOTO -FINDERR
```

The use of the &ERROR control statement is more fully illustrated in the section on "Checking for Execution Errors".

An &GOTO statement can also stand alone as an EXEC control statement. When coded as such, it forces an unconditional branch to the specified location. For example, to pass control unconditionally to the top of the EXEC procedure, simply enter:

```
&GOTO TOP
```

Further examples of &GOTO usage are found throughout the rest of this book.

BRANCHING WITH THE &SKIP STATEMENT

The &SKIP control statement provides you with a second method of passing control to various points in an EXEC procedure. Instead of branching to a named or numbered location in an EXEC procedure, &SKIP passes control a specified number of lines forward or backward in the file.

When you want to pass control to a point that precedes the current line, simply determine how may lines backward you want to go, and code &SKIP with the desired negative value. For example, to use &SKIP for loop control, you could enter statements as follows:

```
&IF &INDEX EQ 0 &EXIT 12
&TYPE COUNT IS &1
&1 = &1 - 1
&IF &1 GT 0 &SKIP -2
```

When this EXEC procedure is invoked, it checks to make sure that at least one argument was passed to it. If an argument is passed, it is assumed to be a number that indicates how many times the loop is to execute. The loop executes until the count in &1 is zero. If the argument entered is zero, the loop executes only once, typing out:

```
COUNT IS 0
```

If no argument is passed, an exit is taken with a return code of 12.

Just as you can pass control backward using &SKIP, you can also pass control forward by specifying how many lines to skip. For example, to handle a conditional exit from an EXEC procedure, you could code the following:

```
&IF &RETCODE EQ 0 &SKIP
&EXIT &RETCODE
```

where the branch around the &EXIT statement is taken whenever the value of &RETCODE equals zero. If the value of &RETCODE does not equal zero, control passes out of the current EXEC procedure with a return code that is the nonzero value in &RETCODE. Note that when no &SKIP operand is specified, a value of 1 is assumed.

USING COUNTERS FOR LOOP CONTROL

A primary consideration in designing a section of an EXEC procedure that is to be executed a number of times is how the number of executions is controlled. One simple way to control the execution of a sequence of instructions is to create a loop that tests and modifies a counter.

Before entering the loop, the counter is initialized to some value. Each time through the loop, the counter is adjusted (up or down) toward some limit value. When the limit value is reached (the counter value is the same as the limit value), control passes out of the loop and it is not executed again.

The example in the previous section, "Branching With the &SKIP Statement," uses a counter to control the execution of the loop. The counter in that example is the numeric variable &1, which is initialized by an argument passed when the EXEC procedure is invoked. Each time through the loop, the value in &1 is decremented by one. When the value of &1 reaches zero, control passes from the loop to the next sequential statement (in the example, control returns to CMS).

There are several ways of setting, adjusting, and testing counters. Some ways to set counters are:

- By reading arguments from a terminal, such as:

```
&READ VARS &COUNT1 &COUNT2
```

- By arbitrary assignment, such as:

```
&COUNTER = 43
```

- By assignment of a variable value or expression, such as:

```
&COUNTS = &INDEX - 1
```

Counters can be adjusted up or down by any increment or decrement that meets your purposes. For example:

```
&COUNTEM = &COUNTEM - &RETCODE
&COUNT1 = &COUNT + 100
```

Counters can be tested by the &IF control statement for a specific value, a range of values, or a simple equality to zero. For example, suppose a counter should contain a value from 5 to 10 inclusive:

```
&IF &COUNT LT 5 &SKIP
&IF &COUNT LE 10 &SKIP
&TYPE &COUNT IS NOT WITHIN RANGE 5-10
```

If the value of &COUNT is less than 5, control passes to the &TYPE control statement, which types out the erroneous value and an explanatory message. If the value of &COUNT is greater than or equal to 5, the next statement checks to see if it is less than or equal to 10. If this is true, then the value is between 5 and 10 inclusive, and the typeout of the error message is skipped.

Further examples of counter usage for loop control are found throughout the rest of this book.

LOOP CONTROL WITH THE &LOOP STATEMENT

A convenient way of controlling execution of a sequence of EXEC statements is with the &LOOP control statement. An &LOOP statement can be set up in four ways:

1. To execute a particular number of statements a specified number of times.
2. To execute a particular number of statements until a specified condition is satisfied.
3. To execute the statements down to (and including) the statement identified by a label a specified number of times.
4. To execute the statements down to (and including) the statement identified by a label until a specified condition is satisfied.

The numbers specified for the number of lines to execute and the number of times through the loop must be positive. In addition, if a label is used to define the limit of the loop, it must follow the &LOOP statement (it cannot precede the &LOOP statement).

The decision as to which form of the &LOOP statement is intended is based on the number of tokens in the statement after scanning and substitution for any EXEC variables. If the conditional form is intended, and the first comparand or the comparator is given in the form of an EXEC variable, then the value of the variable must not be blank at the time the statement is interpreted.

In the conditional form, the tokens forming the conditional phrase are saved in an unscanned format, so that substitution for any EXEC variables can be performed dynamically before each execution of the loop. For example, the statements:

```
&X = 0
&LOOP -END &X EQ 2
&X = &X + 1
-END &TYPE &X
```

are interpreted and executed as follows:

1. The variable &X is assigned a value of 0.

2. The &LOOP statement is interpreted as a conditional form; that is, to loop to -END until the value of &X equals 2.

3. The variable &X is incremented by one and is then typed.

4. Control returns to the head of the loop, where &X is tested to see if it equals 2. If it does not, the loop is executed again. When &X does equal 2, control is passed to the EXEC statement immediately following the end of the loop (in this case, an exit from the EXEC procedure is taken).

The typed lines resulting from a execution of this EXEC are:

```
1
2
```

at which time the value of &X equals 2, and the loop is not executed again.

Another example of conditional loop control is as follows:

```
&Y = &LITERAL A&B
&LOOP 2 .&X EQ &LITERAL .&
&X = &SUBSTR &Y 2 1
&TYPE &X
```

These statements are interpreted and executed as follows:

1. The variable &Y is set to the literal A&B.

2. The two statements following the &LOOP statement are to be executed until the value of &X is &. Notice the periods that precede the values to be compared in the &LOOP statement. The first time this statement is executed, the variable &X has not been initialized to any value and, therefore, is considered a null string. Since the

EXEC interpreter ignores null strings, a period has been added to the &X to give it a recognizable value. A period must also be added to the constant & so the two values can compare correctly. (Any character that is not significant to the EXEC interpreter can be used instead of a period.)

3. The variable &X is set to the value of the second character in the variable &Y, which is a literal ampersand (&).

4. The ampersand is typed once, and the loop does not execute again because the condition that the value of &X be a literal ampersand is met.

Further examples of loop control with the &LOOP statement are found throughout this book.

CONTROLLING EXECUTION OF CMS COMMANDS

CMS commands in an EXEC procedure can be executed sequentially, under control of EXEC control statements, or they can be placed (stacked) in the terminal input buffer for later execution. You already know how to set up an EXEC procedure to control execution of CMS commands, and also how to execute CMS commands sequentially. This section shows you how to stack CMS commands for later execution.

PLACING A COMMAND IN THE CONSOLE STACK

The &STACK control statement allows you to stack a line of tokens in the terminal input buffer (also called the console stack). Normally, the line of tokens comprises a CMS command (or subcommand) and its parameters, although a null line can also be stacked by omitting the tokens. In fact, the data placed in the stack need not be a command at all. See the heading "An Annotated EXEC Procedure" for an example of such stacking.

Lines placed in the console stack by &STACK are normally stacked FIFO (first in, first out), but you can explicitly specify LIFO (last in, first out) by entering LIFO as the first token following &STACK.

Note: The Logical Line End character is a hexadecimal '15'. CMS routines do not process the symbolic line end character (#).

The stacking facilities are especially useful in designing EXEC procedures that use such CMS commands as EDIT. You can use the &STACK control statement to set EDIT subcommands in the console stack, then execute the EDIT command itself. For example, suppose you wanted to define a special filetype that is to contain uppercase and lowercase characters. Since the EDIT command assumes only uppercase characters for all filetypes except MEMO and SCRIPT, you need to specify the EDIT subcommand CASE M each time you edit a file with the special filetype. If the special filetype is called TABLE, you could use the following EXEC procedure to control the CMS EDIT command:

```
&IF &INDEX LT 2 &EXIT 12
&IF &2 NE TABLE &SKIP
&STACK CASE M
EDIT &1 &2 &3 &4 &5 &6
```

When this EXEC is invoked, it expects to find at least two arguments, although up to six may be passed. The first argument must be a filename, and the second a filetype. If fewer than two arguments are passed, an exit with a return code of 12 is taken.

If the filetype specified (&2) is not TABLE, the next sequential statement is skipped. In this case, an EDIT command is issued by specifying the arguments that were passed to the EXEC procedure. An exit is taken on return from CMS.

If the filetype specified is TABLE, control passes to the next sequential statement in the file. This statement stacks the EDIT subcommand CASE M in the terminal input buffer. When the EDIT command is issued (in the next statement), the first time the Editor reads from the terminal buffer, it reads the CASE M subcommand. The person working at the terminal is thus freed from having to enter CASE M when editing a TABLE file.

If you want to stack one or more untokenized lines in the console stack, you can use the &BEGSTACK and &END control statements, which are discussed next.

Inserting More Than One Line

Using the &BEGSTACK and &END control statements, you can place untokenized lines of data in the terminal input buffer (also called the console stack). Normally, a line consists of a CMS command (or subcommand) and its parameters. Since the line is untokenized, no substitution is performed by the EXEC interpreter before the line is placed in the console stack.

As with the &STACK control statement, you can specify whether the lines are to be executed on a FIFO (first in, first out) or LIFO (last in, first out) basis.

Lines of data are normally truncated at column 72, but you can specify the option ALL in the &BEGTYPE statement to extend the data entry area to 80 characters (or to 130 characters if the EDIT command specifies LRECL 130).

Consider another example using subcommands of the CMS EDIT command. This time, in addition to allowing uppercase and lowercase characters, you want to set tabs at columns 1, 10, and 20, and you want the short prompting message to appear when an invalid EDIT subcommand is entered. Using a filetype of TABLE for files that require this special treatment, you could enter the EXEC procedure as follows:

```
&IF &INDEX LT 2 &EXIT 12
&IF &2 NE TABLE &GOTO -EDIT
&BEGSTACK
CASE M
TABSET 1 10 20
SHORT
&END
-EDIT EDIT &1 &2 &3 &4 &5 &6
```

The analysis of the arguments passed is similar to that discussed in the previous example for &STACK. In this EXEC procedure, a branch is taken to the label -EDIT when the filetype specified in the variable &2 is not TABLE. When the filetype is TABLE, control passes to the &BEGSTACK control statement, which places the lines down to &END in the console stack. The EDIT command is then executed. The first three lines read from the terminal buffer are the subcommands that were stacked there. Thus, the person working at the terminal is freed from having to enter the CASE M, TABSET 1 10 20, and SHORT subcommands when editing a TABLE file.

You can check the status of the console stack within your EXEC procedure by examining the &READFLAG special variable, which is discussed next.

Examining the Read Status Flag

During EXEC processing, you may need to know whether or not any data lines are stacked in the terminal input buffer. You can determine this quickly and easily by examining the special variable &READFLAG. If &READFLAG contains the value CONSOLE, the next attempt to read from the terminal obtains a physical line from the real terminal. If &READFLAG contains the value STACK, the next attempt to read from the terminal obtains a line from the terminal input buffer.

Note: You cannot set this variable explicitly, but you can examine it and save its value in some other variable at any point in an EXEC procedure.

For example, if successful completion of an EXEC procedure requires at least one data line to be stacked in the terminal input buffer, you might set up the following test just before leaving the EXEC procedure:

```
&IF &READFLAG EQ STACK &EXIT
&TYPE PROCESS ERROR; NO LINES IN STACK
&EXIT 16
```

If at least one line is in the stack, the normal exit is taken with a return code of zero. When no lines appear in the stack, the value of &READFLAG is not STACK, so control passes to the following &TYPE statement, which types an error message and exits with a return code of 16.

Note: &READFLAG can contain only the two values CONSOLE and STACK.

CHECKING FOR EXECUTION ERRORS

It is usually good programming practice to check for errors during execution of a program. This is as true for EXEC procedures as it is for any formal programming language. Three major types of errors can be detected during EXEC processing:

1. Errors in number, type, or contents of arguments passed to an EXEC procedure. Checking for these kinds of errors has already been discussed under the heading "Passing Arguments to an EXEC Procedure."

2. Errors in CMS command processing. Checking for these kinds of errors is controlled by specifying an &ERROR control statement that passes control to an analysis routine whenever a CMS command error occurs. These techniques are discussed under the following headings, "Identifying Error Handling Routines" and "Checking for CMS Error Return Codes."

3. Errors in EXEC interpreter processing. These kinds of errors usually result in an error exit from the EXEC procedure. The EXEC interpreter types an error message and a return code, and returns to CMS. The messages and return codes are discussed under the heading "Recognizing EXEC Processing Errors."

IDENTIFYING ERROR HANDLING ROUTINES

When an error is detected in processing a CMS command, a return code indicating the severity of the error is passed back to the EXEC interpreter. The EXEC interpreter then activates the &ERROR control statement currently in effect. If none has been specified, &CONTINUE is assumed, and no error processing is performed.

An &ERROR control statement can specify that any executable statement be processed when an error is recognized. However, if the action taken is itself a CMS command that also yields an error return code, the EXEC interpreter types an error message and exits from the EXEC procedure.

Note: An &ERROR control statement must be set up before the CMS commands for which it is to handle errors are executed. Thus, the effect of an &ERROR control statement ranges from the point at which it is entered until the next &ERROR statement is encountered.

A simple error action is to exit from the EXEC procedure, passing the return code from the CMS command back to the user. For example:

```
&ERROR &EXIT &RETCODE
```

In another EXEC procedure, you may want to check the severity of the error before taking any action. For example:

```
&ERROR &IF &RETCODE LT 12 &GOTO -FIX
&EXIT &RETCODE
    .
    .
    .
-FIX    (Analysis routine begins here.)
```

When the value of &RETCODE is 12 or more, the branch to -FIX is not taken, and the &EXIT statement that follows is executed. When the return code is less than 12, the branch to the error analysis routine at -FIX is taken.

Once control passes to the error analysis section of an EXEC procedure, you can test the &RETCODE special variable to determine what kind of error occurred. This technique is discussed next.

CHECKING FOR CMS ERROR RETURN CODES

When an EXEC procedure passes control to an error analysis routine, it is usually because the error discovered is not very serious, and may be one that can be contained by performing some corrective action. Before you can take any corrective action, though, you need to know exactly what the error is. One way to determine the cause of the error is to examine the return code in the special EXEC variable &RETCODE.

For example, suppose you want to set up an analysis routine to identify return codes 1 through 11 (anything greater than 11 is set up to cause an immediate exit). When a return code is identified, control is passed to a corresponding routine that attempts to correct the error. You could set up such an analysis routine as follows:

```
-ERRANAL
&CNT = 0
&LOOP 2 &CNT EQ 12
&IF  &RETCODE EQ &CNT &GOTO -FIX&CNT
&CNT = &CNT + 1
   .
   .
   .
-FIX0 &GOTO -ALLOK
-FIX1...
-FIX2...
   .
   .
   .
-FIX10...
-FIX11...
-ALLOK...
```

When the value of the &CNT variable equals the return code value in &RETCODE, the branch to the corresponding -FIX routine is taken. Each corrective routine performs actions depending on its code.

When you want to pass the value of &RETCODE out of the EXEC procedure, you can simply specify it in the &EXIT statement, as follows:

```
&EXIT &RETCODE
```

Further examples of the use of the &RETCODE special variable are found throughout the rest of this book.

RECOGNIZING EXEC PROCESSING ERRORS

Just as errors can occur in CMS command execution, so can they occur in EXEC control statement processing. When the EXEC interpreter finds an error, it types the message:

```
ERROR IN EXEC FILE fname, LINE nn -- error description
```

where:

fname
is the filename of the EXEC file in which the error was detected.

nn
is the line number in the file at which the error occurred.

error description
is one of the conditions described below. A return code is passed back to the calling program; the return codes are shown with the messages to which they belong.

Return Code	Message and Explanation
802	&SKIP OR &GOTO ERROR An error occurred in attempting to process an &SKIP or &GOTO statement. For &SKIP, a value that specifies a position before the beginning of the EXEC file may have been specified. For &GOTO, the target specification may have been omitted.
804	TOO MANY ARGUMENTS The user has tried to pass more than 30 arguments to the specified EXEC at the line indicated. If more than 30 variables must be used, define your own and initialize them with an &READ VARS control statement.
805	MAX DEPTH OF LOOP NESTING EXCEEDED Up to four nested loops using the &LOOP control statement may be specified. If more than this are required, use another technique for loop control for additional levels of nesting.
806	DISK OR TERMINAL READ ERROR An error occurred while reading from a disk or user terminal. Try the EXEC procedure again. If the error persists, contact you computer center for assistance.
807	INVALID SYNTAX The syntax of the indicated statement is incorrect. Correct the error and try the EXEC procedure again.
808	INVALID FORM OF CONDITION The conditional expression in the indicated statement is invalid. This error can arise due to faulty logic, but when the logic is correct, it occurs most often when some required argument is not passed to the EXEC procedure.
809	INVALID ASSIGNMENT The assignment in the indicated statement is invalid. The types of errors that can cause this message to appear are like those for INVALID FORM OF CONDITION (808).
810	MISUSE OF SPECIAL VARIABLE One of the special EXEC variables was improperly used in the indicated statement. Review the rules for use of the special EXEC variables and correct the error before attempting to use the EXEC procedure again.

Return Code	Message and Explanation
811	ERROR IN &ERROR ACTION The action specified when the indicated &ERROR statement was executed resulted in a processing error. You should correct both errors before attempting to use the EXEC procedure again.
812	CONVERSION ERROR An error occurred in the indicated statement when the EXEC interpreter attempted to convert one type of data to another. This can happen when you try to perform arithmetic operations on alphabetic data. Correct the error and try the EXEC procedure again.
813	TOO MANY TOKENS IN STATEMENT No more than 19 tokens can appear in a single EXEC statement. Reduce the number of tokens in the indicated statement and try the EXEC procedure again.
814	MISUSE OF BUILT-IN FUNCTION One of the EXEC built-in functions was improperly used in the indicated statement. Review the rules for use of the built-in functions and correct the error before attempting to use the EXEC procedure again.
815	EOF FOUND IN LOOP The end of the EXEC file was encountered at the indicated statement while execution was being controlled by an &LOOP control statement. Check for an incorrect limit value or a failure to attain an exit condition during loop execution. When the error is corrected, try the EXEC procedure again.
816	INVALID CONTROL WORD The EXEC interpreter does not recognize a control word in the indicated statement. Locate and correct the error before using the EXEC procedure again.

Note: None of the EXEC processing errors can be corrected dynamically. Depending on the error, you must either correct the indicated error and invoke the EXEC procedure again, or simply invoke the EXEC again with correct arguments.

SPECIAL EXEC FILES

Certain EXEC files have special uses in a CMS virtual machine. They are:

- PROFILE EXEC, which allows a user to set up his own operating environment within CMS.
- CMS EXEC, which is a file of 80-character records created by the CMS LISTFILE command.
- EDIT macros, which are special EXEC files that contain only EXEC control words and EDIT subcommands.

Each of these special EXEC files is discussed in this section under a separate heading.

PROFILE EXEC

A PROFILE EXEC is an EXEC procedure that tailors a CMS virtual machine to the user's specifications. If you usually enter several commands to change your virtual machine after you load CMS, you should set up your own PROFILE EXEC to execute these commands for you. This will save you the trouble of entering these commands every time you IPL CMS.

A PROFILE EXEC can be as simple or as complex as you require. It is a normal EXEC file, and thus it can contain any valid EXEC control statements or CMS commands. The only thing that makes it special is its filename, PROFILE, which causes it to be executed the first time you press the RETURN key after loading CMS.

Usually, the first thing you do after loading CMS is to type a CMS command. When you press the RETURN key to enter this command, CMS searches your A-disk for a file with a filename of PROFILE and a filetype of EXEC. If such a file exists, it is executed before the first CMS command entered is executed. Because you do not do anything special to cause your PROFILE EXEC to execute, you can say that it executes "automatically."

Note: You can prevent your PROFILE EXEC from executing automatically by entering:

```
     ACCESS (NOPROF)
```

as the first CMS command after you IPL CMS. You can enter:

```
     PROFILE
```

at any time during a CMS session to execute the PROFILE EXEC.

For example, if you want to set up a CMS virtual machine that redefines the blip characters and CMS Ready message, you could create the following PROFILE EXEC:

```
     &CONTROL OFF
     SET BLIP *
     SET RDYMSG SMSG
```

In addition to establishing an operating environment, you could also perform such functions as linking to another minidisk, accessing it, and manipulating files that are found on it. An Assembler Language programmer may want to include a GLOBAL command in his PROFILE EXEC to ensure access to the CMS and OS macro libraries, while a PL/I user would want to have the PL/I Program Product libraries available.

An example of a PROFILE EXEC to do some of these things is:

```
     &CONTROL OFF
     LINK PUBS 191 291 RR RDPAS
     ACCESS 291 B/A
     GLOBAL MACLIB CMSLIB OSMACRO
     SET BLIP Z
     SET RDYMSG SMSG
```

Note: You can use &EXIT to specify a return code to be displayed when the PROFILE EXEC finishes processing. If the return code ycu specify in this way depends on the value of &RETCODE, the return code is displayed only if you have executed the PROFILE EXEC by entering

```
PROFILE
```

If the PROFILE EXEC is executed automatically, however, the return code is not displayed. Only the normal CMS ready message (R;) is displayed, to show that CMS has executed the PROFILE EXEC.

For example, suppose your PROFILE EXEC is:

```
&CONTROL OFF
SET RDYMSG SMSG
GLOBAL TXTLIB CDMLIB
&IF &RETCODE NE 0 &EXIT 1
&EXIT
```

If this PROFILE EXEC is executed automatically and CDMLIB cannot be found, the following message is issued:

```
FILE 'CDMLIB TXTLIB' NOT FOUND.
R;
```

If the same EXEC is executed by your entering

```
PROFILE
```

the return code is displayed, as follows:

```
FILE 'CDMLIB TXTLIB' NOT FOUND.
R(00001);
```

If the file is found, the normal CMS ready message is issued in either case.

CMS EXEC

A CMS EXEC file is a special file of 80-character records that is created on a user's primary disk by the CMS LISTFILE command. The format of each record in the file is:

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

The variables &1 and &2 are standard EXEC numeric variables. The filename, filetype, and filemode are those specified in the LISTFILE command. (The LISTFILE command is described in the _VM/370: Command Language Guide for General Users_.)

Judicious use of the LISTFILE command and CMS EXEC can save much repetitive work in certain situations. For example, if you have several files with a filetype of ASSEMBLE that you want punched so that they can be transferred to some other user, you can do this in two ways:

1. You can enter the command:

   ```
   DISK DUMP filename ASSEMBLE
   ```

   for each file to be moved. If many files are involved, this can be a tedious, time-consuming procedure.

2. You can create a CMS EXEC file by entering the command:

```
     LISTFILE * ASSEMBLE * (EXEC)
```

   which creates a file named CMS EXEC with 80-character records in the format described previously. One record is created for each file found with a filetype of ASSEMBLE. You could then issue the command:

```
     CMS DISK DUMP
```

   When the CMS EXEC is executed, the argument DISK is assigned to &1, DUMP is assigned to &2, and a DISK DUMP command is thus created for each record in the file.

   If your virtual punch is spooled to the user to whom you want the files to go, each file is punched to that user.

The CMS EXEC variables &1 and &2 can be assigned any arguments that create a valid CMS command or EXEC control statement. (No arguments need be specified at all, if desired).

A more comprehensive example of the use of the LISTFILE command and CMS EXEC is found in the section entitled "An Annotated EXEC Procedure."

EDIT MACROS

If you have a good knowledge of the CMS EXEC facilities, you can write your own EDIT macros. You must ensure that any EDIT macro you write checks the validity of its operands and displays an error message if necessary.

The conventions followed when creating EDIT macros are:

1. EXEC files that are EDIT macros have a filename that starts with a dollar sign ($) and a filetype of EXEC. These files are referred to as EDIT macro files.

2. An EDIT macro subcommand consists of the name of an EDIT macro file (including the initial $), possibly followed by operands.

3. An EDIT macro file contains only EDIT subcommands and EXEC control statements. EDIT macros can execute only in EDIT mode.

Operands of an EDIT macro must be separated from the macro name, and from each other, by at least one blank. Percent signs (%) cannot be entered as operands, since they have a special meaning to the EXEC interpreter. Operands passed to an EDIT macro are subject to the same rules as any other EXEC file (that is, the length of an operand must not exceed eight characters).

When you create the macro, IMAGE mode must be off if you include tab characters (X'05').

All EDIT subcommands in EDIT macros must be stacked (that is, you must specify &STACK or &BEGSTACK before the EDIT subcommands). If your EDIT macro uses variables, you should use &STACK rather than &BEGSTACK, since &BEGSTACK inhibits substitution of variables.

If an EDIT macro is issued, and the EDIT macro file does not exist, the Editor issues the message ?EDIT:. If an EDIT macro is used incorrectly, the Editor displays a message, and the macro is ignored. If an EDIT macro is assigned to X or Y, it is an error to issue that X or Y subcommand with a numeric operand other than 0 or 1.

Some EDIT macros use the CMS DESBUF command during their execution (for example, $DUP and $MOVE). If stacked lines exist when one of these macros is invoked, the macro deletes the stacked lines and issues the message STACKED LINES CLEARED BY (macro name). This also occurs in user-written macros if the CMS line end character has been used to stack additional subcommands after the macro is issued.

A user-written EDIT macro that uses first-in, first-out (FIFO) stacking should ensure that the stack is initially clear. You do this by including in your EDIT macro the line

```
&IF &READFLAG EQ STACK DESBUF
```

before you stack anything. The DESBUF command clears the console stack. Alternatively, your EDIT macro can use last-in, first-out (LIFO) stacking to avoid having to initially clear the console stack.

If the operation of an EDIT macro is completed without an error, the Editor clears any stacked lines and issues the message STACKED LINES CLEARED. Thus, the macro has no effect on the Editor or its contents.

To avoid having the Editor type during execution of your EDIT macros, you can specify that your EDIT macros operate with verification off. You can accomplish this without losing your setting by stacking PRESERVE and VERIFY OFF for execution first, and RESTORE for execution last.

Do not interrupt the execution of an EDIT macro by pressing the ATTN (attention) key or its equivalent.

An example of an EDIT macro that you could write is the following, which puts a continuation character in column 72 of an Assembler Language source statement. Remember to issue the EDIT subcommand IMAGE OFF before you create this EXEC, because it includes tab characters.

```
&CONTROL OFF
&BEGSTACK
PRESERVE
VERIFY OFF
TRUNC *
TABS 1 72
&END
&STACK REPEAT &1
&BEGSTACK
OVERLAY       C
        (Enter a blank and a tab character between OVERLAY and C.)
RESTORE
&END
```

If you name this file $CONT EXEC, then when you want to put a continuation character in column 72, you can:

1. Enter the Assembler Language source statement (except for the continuation character).

2. Enter a null line to get into EDIT mode (from INPUT mode).

3. Invoke the $CONT EXEC to put a continuation character in column 72.

4. Enter INPUT mode and continue entering source statements.

Two EDIT macros are supplied with VM/370: $DUP and $MOVE. These are described in "Section 4. EDIT Macros" in the *VM/370: EDIT Guide*. "Appendix B. User-Written EDIT Macros" in the *VM/370: EDIT Guide* describes other EDIT macros that you can create and may find useful.

CONTROLLING THE CMS BATCH FACILITY

The EXEC facilities provide you with a convenient way of controlling the CMS Batch Facility. Using EXEC procedures, you can simulate terminal control of your batch job execution, while freeing yourself of the repetitious tasks involved in preparing input for the batch card reader.

Since the CMS Batch Facility executes commands in the same way as a normal CMS virtual machine, you can submit data to it in three primary ways:

1. Invoke EXEC procedures in your virtual machine to punch the necessary commands to the batch card reader.
2. Punch EXEC files to the batch card reader in such a way that the EXEC files are loaded on the batch machine's primary disk, and are then executed in the batch machine.
3. Punch appropriate commands to the batch card reader to cause the batch machine to link to a user disk and invoke an EXEC procedure that resides on that disk.

Of course, any combination of these (or other) methods may serve the purpose of a particular user job. Methods 2 and 3 also show how to make a data file (such as a source file for an assembly) available to the batch machine. You can either punch the data file (preceded by appropriate CMS commands) directly to the batch reader, or punch a series of commands that enable the batch machine to link to a user disk and have access to the data file.

To simulate actual console input, the batch machine truncates trailing blanks from every record it reads from its card reader and determines the length of the line read accordingly. Thus, a blank record is treated as a null line.

The only exception to the normal batch machine handling of /* (end of job) and blank records is that of using /* as an end-of-file indicator when MOVEFILE is executing and reading from the console (batch card reader). The /* is translated to a null line so that blank data records can be read in as data, and so that MOVEFILE can recognize an end-of-file from the console. This is the only time that the batch machine does not interpret /* as an end-of-job indicator, and the only condition when a blank record is treated as a blank card-image record rather than as a null line.

This use of the MOVEFILE command is helpful with input method 2 above, in which the data file may be preceded with two FILEDEF commands and a MOVEFILE command, and followed by a /* record. If the input data file is defined as residing at the console, MOVEFILE reads the data file from the console and recognizes the null line (the /* record translated by the CMS Batch Facility) as the end of the data file.

SAMPLE PROCEDURES FOR BATCH EXECUTION

The sample EXEC procedures in this section can be used to assemble Assembler Language source files under control of the CMS Batch Facility. The three EXEC files are named BATCH EXEC, INPUT EXEC, and ASSEMBLE EXEC.

1. BATCH EXEC

```
*    THIS EXEC SUBMITS ASSEMBLIES/COMPILATIONS TO CMS BATCH
*
*    - PUNCH BATCH JOB CARD;
*    - CALL INPUT EXEC TO PUNCH DATA FILE;
*    - CALL APPROPRIATE LANGUAGE EXEC (&3) TO PUNCH EXECUTABLE COMMANDS
*
     &CONTROL ERROR
     &IF &INDEX GT 2 &SKIP 2
     &TYPE CORRECT FORM IS: BATCH USERID FNAME FTYPE (LANGUAGE)
     &EXIT 100
     &ERROR &GOTO -ERR1
     CP SPOOL D CONT TO BATCHCMS
     &PUNCH /JOB &1 1111 &2
     EXEC INPUT &2 &3
     EXEC &3 &2 &1
     &PUNCH /*
     CP SPOOL D NOCONT
     CP CLOSE D
     CP SPOOL D OFF
     &EXIT
     -ERR1 &EXIT 100
```

2. INPUT EXEC

```
*    CORRECT FORM IS: INPUT FNAME FTYPE
*
*    PUNCH DATA FILE FOR BATCH PROCESSOR; THE /* LINE BEHIND THE
*    DATA FILE IS TRANSLATED TO A NULL LINE BY BATCH SO THAT MOVEFILE
*    RECOGNIZES THE END OF THE DATA SET.
*
     &CONTROL ERROR
     &ERROR &GOTO -ERR3
     &PUNCH FILEDEF INMOVE TERM (BLOCK 80 LRECL 80 RECFM F
     &PUNCH FILEDEF OUTMOVE DISK &1 &2 (BLOCK 80 LRECL 80 RECFM F
     &PUNCH MOVEFILE
     PUNCH &1 &2 * (NOHEADER)
     &PUNCH /*
     &EXIT
     -ERR3 &EXIT 103
```

3. ASSEMBLE EXEC

```
*    CORRECT FORM IS: ASSEMBLE FNAME USERID
*
*    PUNCH COMMANDS TO:
*      - INVOKE CMS ASSEMBLER
*      - RETURN TEXT DECK TO CALLER
*
     &CONTROL ERROR
     &ERROR &GOTO -ERR2
     &PUNCH CP LINK &2 191 199 RR PASS= RPASS
     &BEGPUNCH
     ACCESS 199 B/B
     GLOBAL MACLIB UPLIB CMSLIB OSMACRO
     RELEASE 199
```

```
&END
&PUNCH CP MSG &2 ASMBLING '&1 '
&PUNCH ASSEMBLE &1 (PRINT NOTERM)
&PUNCH CP MSG &2 ASSEMBLY DONE
&PUNCH CP SPOOL D TO &2 NOCONT
&PUNCH PUNCH &1 TEXT A1 (NOHEADER)
&BEGPUNCH
CP CLOSE D
CP SPOOL D OFF
CP DETACH 199
&END
&EXIT
-ERR2 &EXIT 102
```

These EXEC files use both methods of making data files available to the batch machine for job execution, that is, the user's macro library UPLIB is on his 191 disk. Instead of punching the file to the batch card reader and using MOVEFILE to write the file to the batch machine's disk (as he does with the source file), the user enables the batch machine to link to his disk and gain access to the macro library needed for the assembly.

Executing the Sample EXEC Procedure

This section describes the sequence of events that occur when the sample job is submitted to the CMS Batch Facility. The BATCH EXEC is invoked, with arguments, as follows:

Terminal Typeout	Source
batch name payroll assemble	User
PUN FILE 0073 TO BATCHCMS	CP
R;	CMS

At this point, the BATCHCMS reader file contains the following statements (in the same general form as a FIFO console stack):

```
/JOB NAME 1111 PAYROLL
FILEDEF INMOVE TERM (BLOCK 80 LRECL 80 RECFM F
FILEDEF OUTMOVE DISK PAYROLL ASSEMBLE (BLOCK 80 LRECL 80 RECFM F
MOVEFILE
.
.
.
source file (PAYROLL ASSEMBLE)
.
.
.
/*
CP LINK NAME 191 199 RR PASS= RPASS
ACCESS 199 B/B
GLOBAL MACLIB UPLIB CMSLIB OSMACRO
RELEASE 199
CP MSG NAME ASMBLING 'PAYROLL'
ASSEMBLE PAYROLL (PRINT NOTERM)
CP MSG NAME ASSEMBLY DONE
CP SPOOL D TO NAME NOCONT
PUNCH PAYROLL TEXT A1 (NOHEADER)
CP CLOSE D
CP SPOOL D OFF
CP DETACH 199
/*
```

Eventually, the following messages appear on the user console (if connected):

<u>Message</u>	<u>Source</u>
FROM BATCHCMS: JOB 'PAYROLL' STARTED	Batch
FROM BATCHCMS: ASMBLING 'PAYROLL'	User job
FROM BATCHCMS: ASSEMBLY DONE.	User job
PUN FILE 0082 FROM BATCHCMS	CP
FROM BATCHCMS: JOB 'PAYROLL' ENDED	Batch

At this point, the user has the resultant object deck (PAYROLL TEXT) in his card reader.

<u>A Batch EXEC for a Non-CMS User</u>

If an installation is running the CMS Batch Facility for non-CMS users, a series of EXEC files could be stored on the system disk so that each user need only include a card to invoke the system EXEC, which in turn would execute the correct CMS commands to process his data.

For example, if a non-CMS user wanted to compile FORTRAN source files, the following BATFORT EXEC file could be stored on the system residence disk:

```
&CONTROL OFF
FILEDEF INMOVE TERM (RECFM F BLOCK 80 LRECL 80
FILEDEF OUTMOVE DISK &1 FORTRAN A1 (RECFM F LRECL 80 BLOCK 80
MOVFILE IN OUT
GLOBAL TXTLIB FORTRAN
FORTGI &1 (PRINT)
&FORTRET = &RETCODE
&IF &RETCODE NE 0 &GOTO -EXIT
PUNCH &1 TEXT A1 (NOHEADER)
-EXIT &EXIT &FORTRET
```

Using this EXEC, the non-CMS user could place a real card deck in the system card reader (the first job must have a CP userid card). The statements to invoke the BATFORT EXEC would be as follows:

```
/JOB  JOEUSER 1234 JOB10
BATFORT  JOEFORT
    .
    .
source file
    .
    .
/*                                    (end-of-file indicator)
/*                                    (end-of-job indicator)
```

The BATFORT EXEC moves the source file onto the batch machine work disk with a file identification of JOEFORT FORTRAN. The FORTRAN G1 compiler is then invoked, and if the return code is 0, the JOEFORT TEXT file is punched for JOEUSER.

Additional functions may be added to this EXEC procedure, or others may be written and stored on the system disk to provide, for example, a compile, load, and execute facility. These EXEC procedures would allow an installation to accommodate the non-CMS users and maintain common user procedures.

AN ANNOTATED EXEC PROCEDURE

The following EXEC procedure could be used to assemble all the ASSEMBLE files on a user's primary disk (A-disk). The numbers to the left of each statement are not part of the EXEC file, but are simply reference points for the discussion that follows the EXEC procedure.

```
1   &CONTROL ERROR
2   LISTFILE * ASSEMBLE A1 (EXEC)
3   EXEC CMS &STACK
4   GLOBAL MACLIB &2 CMSLIB
5   &LOOP -LOOPEND &READFLAG EQ CONSOLE
6   &READ ARGS
7   ASSEMBLE &1
8   &ASMRET = &RETCODE
9   &IF &RETCODE GT 4 &SKIP 2
10  PRINT &1 LISTING A1
11  ERASE &1 LISTING A1
12  -LOOPEND &CONTINUE
13  &EXIT &ASMRET
```

The function of each of the statements in this sample EXEC procedure is as follows:

1. The &CONTROL statement specifies that only those CMS commands that result in a nonzero return code are to be typed at the user's terminal. In addition, the return code is typed in the CMS ready message.

2. The LISTFILE command creates a CMS EXEC file that contains the names of all the files on the user's A-disk that have a filetype of ASSEMBLE. For each ASSEMBLE file found, a record is created in the CMS EXEC file in the following format:

   ```
   &1 &2 filename ASSEMBLE A1
   ```

3. The CMS EXEC is invoked with the argument &STACK. Thus, each record in the CMS EXEC file is placed in the terminal input buffer in the following format:

   ```
   filename ASSEMBLE A1
   ```

4. The GLOBAL command locates and gains access to the macro libraries required for the assembly. The user can specify any library he wishes by passing an argument to the EXEC procedure in the second argument position.

5. The loop that is created by &LOOP executes each statement down to, and including, the label -LOOPEND, until such time as the special variable &READFLAG has a value of CONSOLE. As long as there are records in the console stack (placed there by statement 3), the loop continues to execute.

6. The &READ statement reads a line from the console stack, assigning it tokens to the numeric variables &1, &2, &3, etc.

7. The ASSEMBLE command causes the file named in the variable &1 to be assembled.

8. This statement saves the return code from the Assembler in the user-defined variable &ASMRET.

9. The &IF statement checks to see if the return code from the Assembler is greater than four. If it is, the next two lines are skipped. If not, no skipping of lines occurs, and control passes to the next sequential statement.

10. The PRINT command causes the LISTING file of assembled source statements to be spooled to the user's virtual printer.

11. When the LISTING file has been spooled, it is erased to make room for the next one (if any).

12. This statement marks the end of the loop defined in statement 5.

13. When the loop is through executing, &EXIT passes control back to CMS with the return code from the most recent assembly (contained in &ASMRET), not from the most recently executed CMS command (which might be ERASE).

Many other combinations and variations are possible in the construction of EXEC procedures. The reader is encouraged to investigate the capabilities of the EXEC facilities in depth. The uses to which EXEC procedures can be put in your VM/370 system should be limited only by your imagination and any performance criteria that must be met.

APPENDIX A: EXEC CONTROL STATEMENT SUMMARY

The charts in this section provide a quick reference summary of the EXEC control statements and built-in functions. Should you need more information than that in the charts, see the appropriate discussion in the body of this book.

EXEC Command Description	Control Statements and Built-in Functions
EXEC Invokes EXEC files.	EXec fn [args...] The formats of the EXEC control statements and built-in functions are as follows:
Defines or redefines arguments in the EXEC file.	&ARGS [arg1 [arg2 ...]]
Punches the following lines of this EXEC file into cards.	&BEGPUNCH [ALL]
Stacks the following lines of this EXEC file into the terminal input buffer.	&BEGSTACK FIFO ALL LIFO
Types the following lines of this EXEC file at the terminal.	&BEGTYPE [ALL]
Concatenates tok1 and tok2 into a single token.	&CONCAT tok1 {tok2 ...}
Used in conjunction with an EXEC label to provide an address for branch statements.	&CONTINUE
Supplies the console printout parameters for the execution phase of the EXEC file.	&CONTROL OFF TIME PACK MSG ERROR NOTIME NOPACK NOMSG CMS ALL
Allows the token to be known from this point on by its composition (that is, numeric or character data).	&DATATYPE tok

EXEC Command Description	Control Statements and Built-in Functions
Indicates the completion of the action started by &BEGPUNCH, &BEG-STACK, or &BEGTYPE.	&END
Provides an execution path for a previous EXEC file statement that resulted in a non-zero return code.	&ERROR action
Exits from the EXEC file with a given return code.	&EXIT $\left[\begin{matrix}\text{return-code}\\ \underline{0}\end{matrix}\right]$
Transfers control to a defined location.	&GOTO $\left\{\begin{matrix}\text{TOP}\\ \text{line-number}\\ \text{label}\end{matrix}\right\}$
Allows statement execution if the comparison is satisfied.	&IF $\left\{\begin{matrix}\text{tok1}\\ \&\$\\ \&*\end{matrix}\right\}\left\{\begin{matrix}\text{EQ}\\ \text{NE}\\ \text{LT}\\ \text{LE}\\ \text{GT}\\ \text{GE}\end{matrix}\right\}\left\{\begin{matrix}\text{tok2}\\ \&\$\\ \&*\end{matrix}\right\}$ executable statement
Indicates the number of nonblank characters in the following token.	&LENGTH tok
Allows the use of the literal value of the token without substitution.	&LITERAL tok
Repetitively executes a sequence of statements a defined number of times or until a specific condition is achieved.	&LOOP $\left\{\begin{matrix}\text{n}\\ \text{label}\end{matrix}\right\}\left\{\begin{matrix}\text{m}\\ \text{condition}\end{matrix}\right\}$
Punches a card with the defined tokens.	&PUNCH tok1 [tok2 ...]
Reads the next line (or lines) from the terminal and treats the data as part of the EXEC file.	&READ $\left[\begin{matrix}\text{n}\\ \underline{1}\\ \text{ARGS}\\ \text{VARS var1 [var2 ...]}\end{matrix}\right]$

EXEC Command Description	Control Statements and Built-in Functions
Skips subsequent statements or transfers control up or down in the EXEC file.	&SKIP [n | 1]
Types blank lines at the terminal.	&SPACE [n | 1]
Stacks a line of tokens in the terminal input buffer.	&STACK [FIFO | LIFO] [tok1 [tok2 ...]
Extracts the desired string of characters from the given token.	&SUBSTR tok i [j]
Types at the terminal time information pertaining to EXEC file execution.	&TIME [ON | OFF] [RESET | TYPE]
Types at the terminal a line containing the indicated tokens.	&TYPE tok1 [tok2 ...]

Index

Trim Along This Line

READER'S COMMENTS

Title: IBM Virtual Machine Facility/370: EXEC User's Guide **Order No.** GC20-1812-1

Please check or fill in the items; adding explanations/comments in the space provided.

Which of the following terms best describes your job?

☐ Customer Engineer	☐ Manager	☐ Programmer	☐ Systems Analyst
☐ Engineer	☐ Mathematician	☐ Sales Representative	☐ Systems Engineer
☐ Instructor	☐ Operator	☐ Student/Trainee	☐ Other (explain below)

How did you use this publication?

☐ Introductory text ☐ Reference manual ☐ Student/ ☐ Instructor text

☐ Other (explain) ____________________

Did you find the material easy to read and understand? ☐ Yes ☐ No (explain below)

Did you find the material organized for convenient use? ☐ Yes ☐ No (explain below)

Specific criticisms (explain below)

Clarifications on pages ____________________

Additions on pages ____________________

Deletions on pages ____________________

Errors on pages ____________________

Explanations and other comments:

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

GC20-1812-1

YOUR COMMENTS PLEASE . . .

This manual is one of a series which serves as a reference source for systems analysts, programmers, and operators of IBM systems. Your comments on the back of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: Requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

Trim Along This Line

FOLD FOLD

FIRST CLASS
PERMIT NO. 172
BURLINGTON, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

IBM CORPORATION
VM/370 PUBLICATIONS
24 NEW ENGLAND EXECUTIVE PARK
BURLINGTON, MASS. 01803

FOLD FOLD

IBM Virt Mach. Facility/370 EXEC User's Guide Printed in U.S.A. GC20-1812-1

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

GC20-1812-1

IBM Virt Mach Facility/370 EXEC User's Guide   Printed in U.S.A.   GC20-1812-1

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)