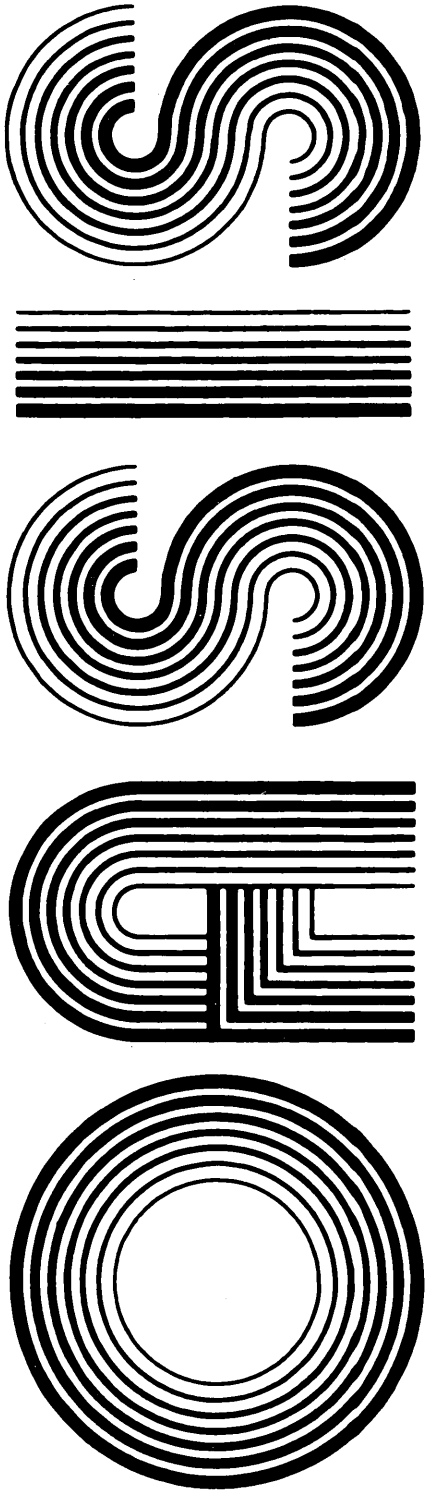


EXEC LANGUAGE REFERENCE MANUAL

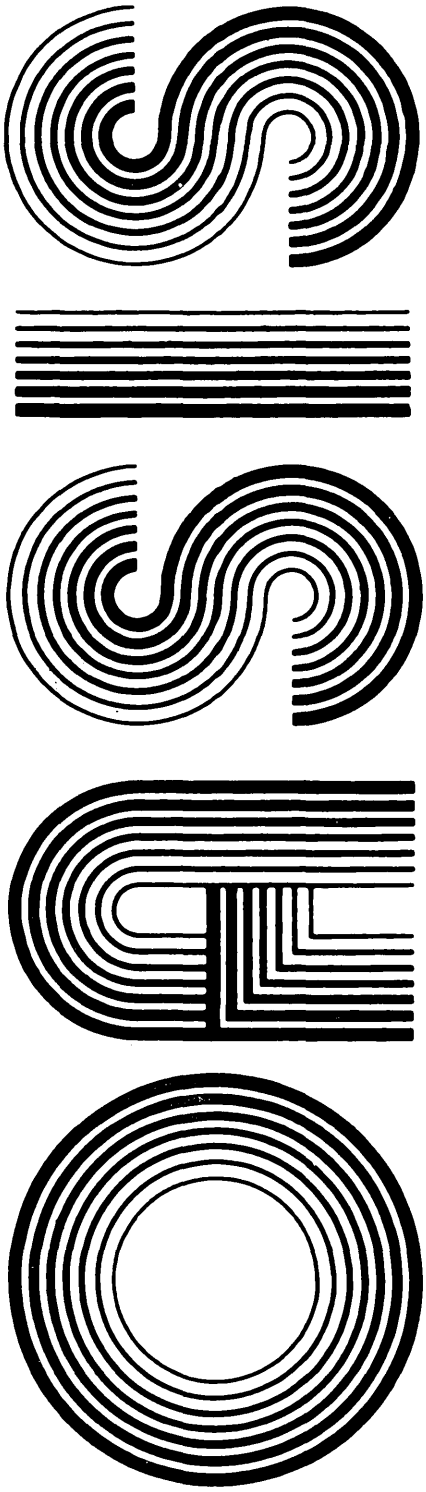
OPERATING SYSTEM SOFTWARE

MAKES MICROS RUN LIKE MINIS

From

PHASE ONE SYSTEMS, INC.

OAKLAND, CALIFORNIA



EXEC LANGUAGE REFERENCE MANUAL

Second Edition

Documentation by: C. P. Williams
Software by: Timothy S. Williams

OPERATING SYSTEM SOFTWARE

MAKES MICROS RUN LIKE MINIS

From
PHASE ONE SYSTEMS, INC.
OAKLAND, CALIFORNIA
7700 Edgewater Drive, Suite 830
Oakland, California 94621
Telephone (415) 562-8085
TWX 910-366-7139

Second edition, first printing: March, 1980

PROPRIETARY NOTICE

P R E F A C E

This manual describes the OASIS system EXEC Language. It provides sufficiently detailed information necessary for the use of this language product in conjunction with the OASIS Operating System.

This manual, named EXEC, like all OASIS documentation manuals, has the manual name and revision number in the lower, inside corner of each page of the body of the manual. In most chapters of the manual the last primary subject being discussed on a page will be identified in the lower outside corner of the page.

This manual describes some features that are only available with the multi-user version of the OASIS operating system. For documentation purposes, any information that pertains to multi-user only will be denoted by the vertical bar character in both margins, similar to this paragraph.

Related Documentation

The following publications provides additional information required in the use of the OASIS EXEC Language:

OASIS System Reference Manual

OASIS Text Editor Reference Manual

TABLE OF CONTENTS

Section **Page**

CHAPTER 1

INTRODUCTION

The OASIS EXEC Language is a process control language used to perform repetitive functions of the operating system. It can be compared to the JCL (Job Control Language) used on larger computers with the added capabilities of operator interaction and conditional execution. With EXEC programs an operator can perform tasks with less chance for error and with less knowledge about the operating system. For example an EXEC program might be written to perform the daily disk backups. The operator would merely type the name of the EXEC program (it might even be executed automatically by another EXEC program) and the program would instruct him about which disks to load, where they can be found, what to do with them afterwards and perform the actual copying.

The format of the EXEC command is:

[EXEC] fn [arg1 [arg2 ... [arg16]]]

Where:

fn Indicates the file name of an EXECutive procedure file.

arg1 Indicates the first argument to be passed to the EXEC file. This argument may contain all ASCII characters except the space character. Only the first eight characters of the argument are actully passed to the EXEC file.

arg2 Indicates the second argument to be passed to the EXEC file.

arg16 Indicates the sixteenth argument to be passed to the EXEC file.

The EXECutive procedure language processor executes an EXEC program as if an operator were entering programs to be executed from the console keyboard with the added ability to include conditional execution (decision processing) and iterative execution (repeat a process several times). With this ability you can write EXEC programs for procedures and routines that are executed frequently or for executing a sequence of programs without operator attention.

An EXECutive procedure file is a sequential file, created by you with the system Editor or BUILD, containing CSI commands, EXEC instructions, data for a user program, data for another EXEC file, etc.

The EXECutive procedure processor is an interpretive type processor in that the commands and data are not analyzed until they are executed by the processor. It is the EXEC command that invokes this processor.

The arguments, arg1 through arg16, are optional but when specified they are truncated to the first eight characters and passed to the EXECutive procedure processor as command variables. An argument of a percent sign only (%) indicates that the argument is empty and is used when other arguments follow this argument.

An EXEC file is a high level language program. Each record or line in the file is an instruction to the EXECutive procedure processor. An EXEC instruction is composed of various elements, discussed in subsequent sections.

1.1 System Cancel during EXEC execution

When the System Cancel key is entered from the console while an EXEC is in control the system will cancel the current command program and display the message:

Cancel EXEC (Y/N)?

You may respond with a Y or an N. A Y response will cause the current EXEC to be canceled returning control to the Command String Interpreter. An N response will cause the current EXEC to continue execution. (The return code will be set to 254 in either case.)

1.2 The EXEC IPL and User Logon File

An important feature of OASIS is the automatic execution of an EXEC program when the system is turned on or when an operator logs on to a new account.

As explained in the chapter on "System Communications" in the OASIS System Reference Manual, the OASIS Operating System will execute an EXEC program named IPL.EXEC after the date and time is entered during initial system start-up. This EXEC program file may belong to the IPL account (or its synonym) or the system account and should contain the sequence of commands that you want executed when the system is turned on. Normally these commands would include a LOGON to the account that contains the programs and data to be processed at the start of a day. When no IPL account exists the system will ask you to LOGON to an account.

When an account is first logged on, the system will search that account's directory for an EXEC file with the file name equal to the account name. If a file is found it will be executed automatically, similar to the IPL.EXEC file during system start-up. For more information refer to the OASIS System Reference Manual chapters on "System Communication" and the "LOGON Command".

1.3 The EXECn File

An OASIS system disk contains a file named SYSTEM.EXEC1. This file is only required by the EXEC language processor. Whenever an EXEC program executes another system or user program (including another EXEC program) the variables currently in use are saved in the EXEC1 file.

If your system disk does not have an EXEC1 file you may create one, using the CREATE command. The EXEC1 file is a direct file with records of length 512 bytes each. Each record corresponds to one level of nesting. Up to 255 records may be specified when the file is created.

| Multi-user OASIS note: The SYSTEM.EXEC1 file is named SYSTEM.EXEC1, SYSTEM.EXEC2, |
| etc., one for each user partition. |

1.4 Variables and Constants

A constant in an EXEC program is an unquoted string, not preceded by an ampersand character (&). If a constant contains only numeric characters then it is a numeric constant and has a numeric value. When a constant contains any non numeric characters then it is a string constant and has no numeric value.

An EXEC programmer has three forms of variables available to him. These three forms include true variables (value can be changed during program execution by the

program), command variables (value determined by the Command String Interpreter when the EXECutive procedure is invoked), and reserved variables (value determined by conditions outside of the program).

True variables are identified by an ampersand followed by a alphabetic character, optionally followed by alphanumeric characters. The number of characters in a true variable name is technically unlimited, although the first eight characters (including the ampersand) must be unique. True variable names may not be the same as any of the EXEC keywords. The value of a true variable is determined by an assignment or READ instruction. Only sixteen true variables may be in use in any one EXEC program. Exceeding this limit causes a symbol overflow error.

Command variables are identified by an ampersand followed by one or two digits. The value of command variables is determined by the Command String Interpreter and may not be changed during the execution of the EXEC program. The value of a specific command variable is determined by a one to one relation with the arguments in the EXEC command as defined in the syntax above. For instance &1 has the value of the first argument, &3 has the value of the third argument. There are only sixteen (16) command variables available to the programmer. The value of a command variable that has no matching argument (that is, fewer arguments were entered than the number of this command variable) is null or empty.

Reserved variables are identified by their reserved names. The value of a reserved variable is determined differently for each reserved variable but may not be changed by the program itself.

&INDEX Numeric value indicating the number of command variables.

&LINE Numeric value indicating the ATTACHed linesize of the console device.

&NULL String value indicating an empty string (length = 0).

&PAGE Numeric value indicating the ATTACHed pagesize of the console device.

&RETCODE Numeric value indicating the system return code. This value is set by each program executed.

Examples:

ABCD	String constant
TDEDWEFD	String constant
1245	Numeric constant
34	Numeric constant
124	Numeric constant
&VALUE	True variable
&A	True variable
&4	Command variable
&EDIT	True variable
&INDEX	Reserved variable
&RESERVED	True variable (Only &RESERVE is used)
&156	Invalid
&RETCODE	Reserved variable

1.5 Labels

An EXEC program may have labels to be used as comments or as a reference point for branching instructions. A label is identified by a negative sign (-) followed by an alphabetic character and optionally followed by alphanumeric characters. The length of a label name is limited to eight characters (including the negative sign).

Labels, when used, must start in column one of the line (no leading spaces other than the single space following a line number).

Examples:

```
-BEGIN
-END
-OPTION1
-BEGININPUT                    Causes an error
```

1.6 Operators

The EXEC language allows a minimal set of arithmetic and comparison operations to be performed. To perform an operation an operator must be used. There are three forms of operators: numeric operators, string operators, and comparison operators.

A numeric operator indicates that arithmetic is to be performed between two variables or constants. The value of the variable or constant must be numeric in type. The valid numeric operators include:

```
+ (addition)
- (subtraction)
* (multiplication)
/ (division)
```

The string operators available in the EXEC language include:

```
| vertical bar - concatenate two variables or constants
&CAT           - concatenate two variables or constants
&SUB           - substring of following variable
&TYP           - type of following variable (Alpha or Numeric)
&LIT           - following characters are not to be tokenized
&LEN           - length of following variable
```

Numeric and string operators are only allowed in the assignment instruction.

Comparison operators are used between two variables or constants to indicate a relationship. Comparison operators are only allowed in the &IF, &WHILE, &UNTIL instructions. There are six comparison operators:

EQ	or	=	Equality
NE	or	<>	Not equal
LT	or	<	Less than
GT	or	>	Greater than
LE	or	<=	Less than or equal
GE	or	>=	Greater than or equal

Relations allowed in these instruction must be simple relations:

<variable>|<constant> <operator> <variable>|<constant>

In order to test a complex relationship the programmer must use the assignment instruction to create a single value for an expression or use multiple &IF instructions.

Relational expressions may use two reserved keywords that may not be used elsewhere. These keywords are &* and &$.

&* Keyword

The &* keyword is a variable indicating "any of the command variables (&1 thru &16)".

&$ Keyword

The &$ keyword is a variable indicating "all of the command variables (&1 thru &16)".

Examples:

```
&IF &A GT 5 &GOTO -LABEL1
&IF &INDEX EQ 0 &GOTO -ERROR
&IF &RETCODE NE 0 &TYPE Error in last program.
&IF &VAR = &NULL &IF &A = &NULL &QUIT
```

The following instruction tests all of the command variables to determine if any of them are equal to the literal PRINT.

```
&IF &* = PRINT &OUTDEV = PRINTER1
```

The following instruction tests all of the command variables to determine if they are all unequal to a left parentheses.

```
&IF &$ NE ( &SKIP 4
```

1.7 Comments

Comments may be inserted in an EXEC program by using the semi-colon (;) character. Comments may be placed on the same line as an instruction with the exception of the instructions:. &CONTROL, &TYPE, and assignment instruction.

It is permissible to use spaces at the beginning of a line (except lines defining labels) or between tokens of an instruction to make the program easier to read but tabs may not be used.

CHAPTER 2

EXEC INSTRUCTIONS

The EXEC language has only a few instructions available to it but taken with the fact that all of the OASIS commands and user written programs may be executed from the EXEC environment the language is very powerful.

The following instructions are presented in alphabetic sequence.

It is important to keep in mind that the space character is a delimiting character and may not be used as part of a variable name or contents.

2.1 Variable Assignment Instruction

The assignment instruction allows you to change or set the value of a true variable. The format of the instruction is:

[line #] [label] true-variable = expression

Where:

true-variable Indicates any valid true variable name as defined previously.

expression Indicates any valid expression. Since this is the only instruction that allows an expression it will be defined here:

<variable>|<const> [<numeric operator> <expression>] ...

or

<variable>|<const> [<string operator> <variable>|<const>]

The expression on the right side of the operator must match in type to the variable or constant on the left side of the operator.

Any of the numeric or string operators may be used but they must be separated from the other elements by at least one space.

Any arithemetic performed is in signed binary integers. The range of value for a numeric expression is -32768 to 32767. Expression exceeding this range will be converted to a value within the range by modulo arithmetic. (32770 is converted to 3, -32770 is converted to +32766, etc.) All sub-expressions are computed, integerized and converted to modulo 32768 before remaining expressions are computed. This means that 1024 * 64 + 1 is equal to 1. (1024 * 64 = 65536, 65536 modulo 32767 = 0.)

&CAT Keyword

The &CAT keyword may be used in an assignment instruction to combine two string variables together. Optionally the vertical bar character (|) may be used. The format of an assignment instruction using the &CAT keyword is:

<true var> = &CAT <var> <var>

or

<true var> = <var> | <var>

&LEN Keyword

The &LEN keyword may be used in an assignment instruction to determine the length of the contents of another variable. The format of an assignment instruction using the &LEN keyword is:

<true var> = &LEN <var>

The <true var> will contain a numeric value indicating the length of the contents of <var>. The &LEN keyword must precede a variable name and only one variable name may be specified.

&LIT Keyword

The &LIT keyword may be used in an assignment instruction before a string to indicate that the string is not to be tokenized. This is especially useful when the string looks like a reserved variable name. The format of an assignment instruction using the &LIT keyword is:

<true var> = &LIT <string>

&SUB Keyword

The &SUB keyword may be used in an assignment instruction to access a portion of a variable or constant. The format of an assignment instruction using the &SUB keyword is:

<true var> = &SUB <token> <1st char> [<last char>]

When the &SUB is evaluated the token following is tokenized. The resulting string is then used - the characters between the <1st character> and the <last character> are extracted and assigned to the <true variable>. When <last character> is not specified the characters from <1st character> position through the end of the string are assigned to the <true variable>.

&TYP Keyword

The &TYP keyword is used in an assignment instruction to determine the variable type of a variable. The format of an assignment instruction using the &TYP keyword is:

<true var> = &TYP <var>

The <var> is evaluated and the variable type (A or N) is assigned to the <true variable>.

Examples:

```
&A = ABCDEFGH                  ; &A receives 'ABCDEFGH'
&ALPHA = &A | EFGH             ; &ALPHA receives 'ABCDEFGHEFGH'
                               ; which is tokenized to 'ABCDEFGH'
&A1 = &SUB &ALPHA 3 4          ; &A1 receives 'CD'
&A2 = &LEN &ALPHA              ; &A2 receives an 8
&X = &TYP &ALPHA               ; &X receives 'A' (alpha)
&BETA = &KAPPA + 123 / &DELTA
```

2.2 BEGSTACK Instruction

The BEGSTACK instruction allows you to to create lines of data to be used by programs executed from the EXEC program. The format of the instruction is:

[line #] [label] &BEGSTACK [LIFO|FIFO]

Where:

label Indicates any valid label.

LIFO Indicates that the lines of data following the instruction are to be placed on the stack in a Last-In-First-Out manner.

FIFO Indicates that the lines of data following the instruction are to be placed on the stack in a First-In-First-Out manner. This is the default option.

A stack may be created using both LIFO and FIFO elements by using multiple BEGSTACK or STACK instructions.

The BEGSTACK instruction is followed by the lines of text or data that is to be placed on the stack. The data is placed on the stack with no analysis (that is, if a variable name is specified then the variable name is placed on the stack and not the contents of it). The data is terminated with the &END instruction.

The information that is placed in the stack by this instruction (or the &STACK instruction discussed later) is accessible by the next program that is executed by this EXEC program. Programs executed from the EXEC environment that require console keyboard input will receive any data in the stack instead of the keyboard.

After the information stored in the stack has been retrieved by a program, future requests for console input will receive data from the keyboard directly. Any information not retrieved from the stack will be lost when the program is terminated and control returns to the EXEC program.

2.3 BEGTYPE Instruction

The BEGTYPE instruction allows you to display information on the console display. The format of the instruction is:

[line #] [label] &BEGTYPE

The BEGTYPE instruction is followed by the lines of information that are to be displayed on the console display. Similar to the BEGSTACK instruction, the data is not analyzed before display and the information is terminated with the &END instruction.

Each line of information following the BEGTYPE instruction is displayed on a separate line of the console, one after the other.

2.4 CONTROL Instruction

The CONTROL instruction allows you the ability to turn on or off the display of any commands executed from the EXEC program. The format of the instruction is:

[line #] [label] &CONTROL ON | OFF | TRACE | STACK | NOSTACK

Where:

ON Indicates that CSI commands executed from the EXEC program are to be displayed on the console. This is the condition that exists when the EXEC program is first entered.

OFF Indicates that CSI commands executed from the EXEC program are to be executed "silently", that is, the display of the command itself is to be inhibited.

TRACE Indicates that all CSI commands and EXEC instructions are to be displayed on the console after tokenization, before execution. The line number of EXEC instruction is displayed surrounded with angle brackes <> followed by the result of the execution of the instruction.

STACK Indicates that information retrieved from the stack is to be displayed on the console, just as if it had come from the keyboard. This is the default condition when the EXEC program is first entered.

NOSTACK Indicates that information retrieved from the stack is not to be displayed on the console. In addition, when information is in the stack all output to the console device is suppressed.

2.5 CRT Instruction

The CRT instruction allows you to position the cursor on the console output device to any position or to perform screen control functions. The format of the instruction is:

[line #] [label] &CRT <column# variable> <line# variable>

or

[line #] [label] &CRT <variable>|<constant>

Where:

column# variable Contains the value of the column number that you wish the cursor positioned to. This must be a numeric value.

line# variable Contains the value of the line number that you wish the cursor positioned to. This must be a numeric value. Caution: addressing a line greater than the screen actually allows produces unpredictable results.

variable Contains the screen control function to be performed. The specific functions allowed vary from terminal to terminal and are controlled by the class code that the console was ATTACHed as.

constant Is the literal specifying the screen control function to be performed.

The various screen control functions allowed by the system include the following:

```
HOME      Move cursor to upper left corner
CLEAR     Clear screen
EOS       Erase to end of screen
EOL       Erase to end of line
UP        Move cursor up one line
DOWN      Move cursor down one line
LEFT      Move cursor one position to the left
RIGHT     Move cursor one position to the right
PON       Following charcters are to be screen protected
POFF      Following characes are not screen protected
KON       Keyboard unlock
KOFF      Keyboard lock
FON       Format on
FOFF      Format off
RVON      Reverse vidio on
RVOFF     Reverse vidio off
ULON      Underline on
ULOFF     Underline off
```

Examples:

```
&A = 5
&CRT 15 &A          ; Position to line 5, column 15
&CRT 3 4            ; Position to line 4, column 3
&CRT &A &A          ; Position to line 5, column 5
&CRT CLEAR            ; Clear screen
```

2.6 CSI command

All OASIS commands and user written programs may be executed from an EXEC program. The format of the CSI command is:

[line #] [label] <command-text>

The <command-text> must specify any parameters or options desired. Before the text specifying the command is passed to the Command String Interpreter it is analyzed for variables, and substitutions are made as applicable.

No restriction is placed on the programs that may be executed in this manner. In fact the program may specify another EXEC program to be executed. This is called nesting. The maximum level of "nesting" of EXEC program calls is determined by the number of records that were created for the SYSTEM.EXEC1 file. This maximum, as stated earlier, is 255. When an EXEC program is executed in this manner the current program will be suspended, control is transferred to the specified EXEC program, and, upon termination of that program, control returns to the "calling" program. The current &CONTROL status is retained when the called EXEC is executed. If that EXEC changes the &CONTROL the change will be in effect until changed again

Examples:

```
LIST FILE DATA S (PRINT
BASIC &1
FILELIST * &A &B ( &C
FILELIST &A (FN
```

It is best to use the normal, complete spelling of command names in an EXEC due to the reduction in search time.

2.7 END Instruction

The END instruction provides a means of terminating the data list following the BEGTYPE or BEGSTACK instruction. The format of the instruction is:

[line #] [label] &END

The END instruction must appear on a line by itself and must be used to terminate the data lines of a BEGSTACK or BEGTYPE.

2.8 ERROR Instruction

The ERROR instruction allows you to specify an instruction to be executed in the event that an error is detected by a CSI command. The format of the instruction is:

[line #] [label] &ERROR <instruction>

Where:

instruction Indicates any valid EXEC instruction described in this manual. Omitting <instruction> indicates that the system return code will not cause any EXEC instruction to be executed when the return code is non-zero.

When the ERROR instruction is executed the <instruction>, when specified, is saved. After any CSI command is executed by the EXEC the system return code is tested and, if non-zero, the saved instruction is automatically executed. This has the same result as if the program had an IF instruction following each CSI command, that tested the &RETCODE reserved variable for a non-zero condition. The ERROR instruction has the added versatility of changing the instruction to be executed when an error is detected for all subsequent CSI commands.

Examples:

```
&ERROR &GOTO -LABEL
&ERROR &IF &RETCODE GT 200 &GOTO -LABEL
&ERROR
&ERROR &TYPE Error &RETCODE detected -- Program aborted!
```

2.9 ESC Instruction

The ESC instruction allows you to perform the actions of the system control keys (the ones that have a lead in of an ESC character) from within an EXEC program. The format of the instruction is:

[line #] [label] &ESC <data>

Where:

data Indicates the literal character or variable that contains the character to be given to the operating system as if it had been typed from the console

keyboard following an escape character.

This instruction might be used to cause the display of the EXEC program to be output to the printer (&ESC P) or to slow down the display of the console (&ESC B), etc.

Note that the command operates just as if the characters were entered from the keyboard. This means that execution of the instruction ESC P causes the status of the printer echo switch to be changed from OFF to ON or ON to OFF, depending upon the current status.

2.10 GOTO Instruction

The GOTO instruction allows you to branch to another portion of the EXEC program. The format of the instruction is:

[line #] [label] &GOTO label

Where:

label Indicates a valid label in the program.

Examples:

```
&GOTO -BEGIN
-LOOP1 &GOTO -RESTART
&GOTO -POINT1
&GOTO -LABEL10
```

2.11 IF Instruction

The IF instruction allows you to test variables and, depending upon the results of the test, execute an instruction. The format of the instruction is:

[line #] [label] &IF <relation> <instruction>

Where:

relation Indicates a simple relation as described previously.

instruction Indicates any of the EXEC instruction described in this section.

The IF instruction evaluates the relation and, if the relation is true, executes the instruction specified. If the relation is false the instruction on the following line is executed and the instruction following the relation is ignored.

2.12 QUIT Instruction

The QUIT instruction allows you to specify the termination of the logic in an EXEC program. The format of the instruction is:

[line #] [label] &QUIT [value]

Where:

value Indicates the optional value that the return code is to be set to. If

this value is not specified the return code is set to zero. This value may be a numeric literal or any variable with numeric contents.

The QUIT instruction unconditionally terminates execution of the current EXEC program. If the EXEC program was invoked from a keyboard command then control will return to the Command String Interpreter. If the EXEC program was invoked from another EXEC then control will return to the instruction following the one invoking this program in the calling program.

2.13 READ Instruction

The READ instruction allows you to request input from the keyboard during the execution of an EXEC program. The format of the instruction is:

[line #] [label] &READ [true-variable] ...

When the READ instruction is encountered execution of the program is interrupted. The EXEC prompt character (:) is displayed on the console at the current cursor position and data is accepted from the keyboard. If any data is currently in the stack, whether loaded by this EXEC or a previous EXEC "calling" this program, the required information will be retrieved from the stack. Each element of the data entered is truncated to eight characters and assigned to the list of variables in a one to one relationship. When more variables are specified than data entered the "extra" variables are set to null or empty. When more data is entered than variables specified the "extra" data is ignored. The elements of the data entered must be separated by at least one space and terminated by a carriage return.

When the READ instruction doesn't specify a variable for the input to be assigned to, the data entered will be executed as if were part of the EXEC program.

Examples:

```
&READ &ALPHA &BETA &CHARLIE &DELTA
&READ &NEVADA
&READ &A &B
&READ
```

2.14 REPEAT Instruction

The REPEAT instruction, used in conjunction with the UNTIL or WHILE instruction, provides a method of loop control. The format of the instruction is:

[line #] [label] &REPEAT

When the REPEAT instruction is executed control of the program is transferred to the first UNTIL or WHILE instruction that precedes the REPEAT instruction.

For examples of the REPEAT instruction see the UNTIL and WHILE instructions.

2.15 SKIP Instruction

The SKIP instruction allows you to "jump relative" to the current instruction. This instruction is usually used as the instruction in an IF instruction or the instruction following the IF instruction. The format of the instruction is:

[line #] [label] &SKIP count

Where:

count Indicates the number of lines, relative to the current instruction, that are to be skipped.

Examples:

```
&SKIP 1
&SKIP 4
&SKIP -2
```

2.16 SPACE Instruction

The SPACE instruction allows you to display multiple carriage return, line feeds on the console. The format of the instruction is:

[line #] [label] &SPACE [count]

Where:

count Indicates the number of lines that are to be advanced on the console. If count is omitted one line will be advanced.

Examples:

```
&SPACE 5
&SPACE &A
&SPACE
```

2.17 STACK Instruction

The STACK instruction allows you to place one line of analyzed data on the stack. The format of the instruction is:

[line #] [label] &STACK [FIFO|LIFO] data

Where:

FIFO Is as described for BEGSTACK.

LIFO Is as described for BEGSTACK.

data Indicates the information to be placed on the stack. This data may include constants or variables. Each variable encountered by the EXEC processor during analysis of the data will be replaced by its current value.

Examples:

```
&STACK HELLO
&STACK LIFO &A &BETA HI
&STACK
```

2.18 TYPE Instruction

The TYPE instruction allows you to display one line of analyzed data on the console. The format of the instruction is:

[line #] [label] &TYPE data ... [\]

Where:

data Is as described for the STACK instruction.

\ Indicates that the data is to be displayed with no automatic carriage return, line feed following. This option is usually used when the displayed text is the prompting message for a READ instruction.

Examples:

```
&TYPE This is a message.
&TYPE This is another message &A
&TYPE And so is this
&TYPE &ALPHA &BETA HELLO &CHARLIE &ETC
```

2.19 UNTIL Instruction

The UNTIL instruction provides a method for conditional loop control. The format of the instruction is:

[line #] [label] &UNTIL <relation>

Where:

relation Is the same as for the IF instruction.

When the UNTIL instruction is executed the <relation> is analyzed and, if false, the intructions following are executed. When the relation is true, the instruction following the next REPEAT instruction is executed.

It is easiest to remember the function of the UNTIL instruction if you think of it as: 'execute following instruction until <relation> is true'.

Example:

```
&A = 123
&UNTIL &A LT 100
&A = &A - 1
&TYPE &A
&REPEAT
&TYPE Done
 .
 .
 .
```

The above example will execute the decrement and type instructions twenty-four times, until the variable &A is less than one hundred, at which point the literal 'Done' will be displayed on the screen.

2.20 WAIT Instruction

The WAIT instruction causes the program to pause until the operator types a key. The format of this instruction is:

[line #] [label] &WAIT

When this instruction is executed it invokes the same process that the system programs use when displaying information to the screen. When encountered by the EXEC processor an up arrow character (^) is displayed at the bottom left corner of the screen and processing is suspended until the operator enters a key. The function of this instruction may be suppressed if the Console Screen-wait key has been used to disable the screen wait function (see System Control Keys in the System Reference Manual).

2.21 WHILE Instruction

The WHILE instruction provides an alternate means of loop control. The format of the instruction is:

[line #] [label] &WHILE <relation>

Where:

relation Is the same as for the IF instruction.

When the WHILE instruction is executed the <relation> is analyzed, and, if true, the following instructions are executed. When the <relation> is false control transfers to the instruction immediately following the next REPEAT instruction.

It is easiest to understand the function of the WHILE instruction if you think of it as: 'execute following instructions while the <relation> is true'.

Example:

```
&WHILE &RETCODE = 0
&TYPE Command to execute
&READ
&REPEAT
 .
 .
 .
```

The above example will ask the operator for a command name to execute and execute that command as long as the system return code is zero. Notice that the READ instruction is not followed by a variable name. In this situation the data read from the console (or stack, if present), is treated as an EXEC instruction, and executed.

2.22 Tokenizing

The EXEC language processor analyzes each instruction line in a right to left manner, reducing each element to the most elementary form. This process of reduction is called tokenizing. An element is tokenized until a space is encountered or column one is encountered. For example, if &A1 contains a 25, &B2 contains 1, and &C contains 2, the element &A&B&C is tokenized in the following

manner:

```
&A&B&C          &C is replaced with 2
&A&B2           &B2 is replaced with 1
&A1             &A1 is replaced with 25
25
```

The end result of a tokenized element is always truncated to eight characters. This process is performed on all elements of an instruction. The following instructions will execute the BACKUP command:

```
0010 &A = BACKUP
0020 &A
```

The following example will also execute the BACKUP command:

```
0010 &A1 = BACKUP
0020 &B = 1
0030 &A&B
```

The &A&B element is tokenized in the following manner:

```
&A&B            &B is replaced with 1
&A1             &A1 is replaced with BACKUP
BACKUP          The command is executed because it does not
                start with an "&" character.
```

APPENDIX A

EXEC EXAMPLES

A.1 Example 1 - Abbreviate BASIC Program

```
0010 &BEGSTACK
0020 CHANGE /PRINT/PRI/* *
0030 TOP
0040 CHANGE /GOTO/GO/* *
0050 TOP
0060 CHANGE /GOSUB/GOS/* *
0070 TOP
0080 CHANGE / = /=/* *
0090 TOP
0100 CHANGE /  //* *
0110 TOP
0120 PAGE
0130 &END
0140 EDIT &1 &2 &3
0150 &QUIT

>EXEC EXAMPLE1 PROGRAM1 BASIC A
```

The above example will perform an edit of the BASIC program "PROGRAM1.BASIC:A", changing all occurrences of PRINT to PRI, all occurances of GOTO to GO, etc. After the five global changes have been made to the file the first page is displayed. At this time the Edit program will have exhausted the stack and further input will come from the keyboard.

A.2 Example 2 - IPL.EXEC

```
0010 &CONTROL OFF
0020 -AGAIN
0030 RUN MAINPROG.BASIC:S
0040 &GOTO -AGAIN
```

The above example is a typical system start EXEC program that executes a BASIC application program until the system cancel key is used or the system is turned off. This type of a program will "isolate" the user from using the OASIS commands. Alternately this program might have displayed a list of options for the user such as making backups of the application diskettes, etc.

A.3 Example 3 - SELECTED.EXEC

```
>FILELIST * BASIC (EXEC

>EXEC SELECTED LIST (PRINT
```

This example will create an EXEC program containing command variables and file descriptions of all of the BASIC programs on the attached diskettes (see the FILELIST command). When the EXEC program is executed with the above specification all of these BASIC program files will be listed on the printer. "LIST" replaces all &1, "(" replaces all &2, "PRINT" replaces all &3. Similarly a group of files could be copied, erased, renamed, etc. The EXEC program created from the FILELIST command is normally edited to remove or add file descriptions or EXEC instructions.

A.4 Example 4 - ASM.EXEC

The following example is a listing of the ASM.EXEC, distributed with the OASIS operating system. Its purpose is to assemble and link an assembly language source program using the MACRO and LINK commands.

```
>LIST ASM.EXEC:S

0010 ; ASM procedure
0020 ;
0030 ; Copyright (C) 1979 by
0040 ; Timothy S. Williams
0050 ;
0060 ;
0070 &IF &INDEX EQ 0 &GOTO -NOFILE
0080 &IF &INDEX EQ 1 &IF &1 EQ HELP &GOTO -HELP
0090 &OPT = 0
0100 &IF &1 EQ ( &GOTO -NOFILE
0110 &FN = &1
0120 &IF &INDEX EQ 1 &GOTO -ASM
0130 &IF &2 EQ ( &GOTO -OPTASM
0140 &FT = &2
0150 &IF &INDEX EQ 2 &GOTO -ASM
0160 &IF &3 EQ ( &GOTO -OPTASM
0170 &FD = &3
0180 &IF &INDEX EQ 3 &GOTO -ASM
0190 -OPTASM
0200 &CT = 0
0210 -LOOP1
0220 &CT = &CT + 1
0230 &IF &CT GT &INDEX &GOTO -ASM
0240 &IF &&CT NE ( &GOTO -LOOP1
0250 -LOOP2
0260 &CT = &CT + 1
0270 &IF &CT GT &INDEX &GOTO -ASM
0280 &IF &&CT EQ / &GOTO -ASM
0290 &OPT = &OPT + 1
0300 &OPT&OPT = &&CT
0310 &GOTO -LOOP2
0320 -ASM
0330 &PAREN =
0340 &IF &OPT NE 0 &PAREN = (
0350 MACRO &FN &FT &FD &PAREN &OPT1 &OPT2 &OPT3 &OPT4 &OPT5 &OPT6 &OPT7 &OPT8
0360 &IF &RETCODE NE 0 &QUIT &RETCODE
0370 &IF &* EQ NOOBJ &QUIT
0380 &OPT = 0
0390 &CT = 1
0400 -LOOP3
0410 &CT = &CT + 1
0420 &IF &CT GT &INDEX &GOTO -LINK
0430 &IF &&CT NE / &GOTO -LOOP3
0440 -LOOP4
0450 &CT = &CT + 1
0460 &IF &CT GT &INDEX &GOTO -LINK
0470 &OPT = &OPT + 1
0480 &OPT&OPT = &&CT
```

```
0490 &GOTO -LOOP4
0500 -LINK
0510 &IF &OPT EQ 1 &IF &OPT1 EQ NOLINK &QUIT
0520 &IF &OPT LT 2 &GOTO -NOREN
0530 &IF &OPT1 NE RENAME &GOTO -NOREN
0540 &FN = &OPT2
0550 ERASE &FN OBJECT * ( NOQUERY
0560 RENAME &1 OBJECT * &FN = = ( NOQUERY
0570 &CT = 0
0580 &CT2 = 2
0590 &OPT = &OPT - 2
0600 -LOOP5
0610 &CT = &CT + 1
0620 &CT2 = &CT2 + 1
0630 &IF &CT GT &OPT &GOTO -NOREN
0640 &OPT&CT = &OPT&CT2
0650 &GOTO -LOOP5
0660 -NOREN
0670 &PAREN =
0680 &IF &OPT NE 0 &PAREN = (
0690 &CT = &OPT
0700 -LOOP6
0710 &CT = &CT + 1
0720 &IF &CT GT 9 &GOTO -LINKIT
0730 &OPT&CT =
0740 &GOTO -LOOP6
0750 -LINKIT
0760 LINK &FN &PAREN &OPT1 &OPT2 &OPT3 &OPT4 &OPT5 &OPT6 &OPT7 &OPT8 &OPT9
0770 &IF &RETCODE NE 0 &QUIT &RETCODE
0780 ERASE &FN OBJECT * ( NOQUERY
0790 &QUIT
0800 -NOFILE
0810 &BEGTYPE
0820 Filename missing!
0830
0840 Use: ASM HELP to display syntax.
0850 &END
0860 &QUIT 255
0870 -HELP
0880 &BEGTYPE
0890 Function: To assemble and link a source program.
0900
0910 Syntax:   ASM fn [ ft [ fd ]] [ ( macro options [ / link options ] ) ]
0920 &END
0930 &QUIT
```

A.5 Example 5 - CLEANUP.EXEC

```
0010 &CONTROL OFF
0020 ; Default drive is * (all)
0030 &DRV = *
0040 &IF &INDEX EQ 0 &GOTO -DEFAULT
0050 &IF &INDEX EQ 1 &IF &1 EQ HELP &GOTO -HELP
0060 &IF &INDEX GT 1 &IF &1 EQ HELP &GOTO -ERROR
0070 &IF &1 NE ( &DRV = &1
0080 &IF &1 NE ( &IF &INDEX EQ 1 &GOTO -DEFAULT
0090 &IDX = 0
0100 &WHILE &IDX LT &INDEX
0110      &IDX = &IDX + 1
0120      &IF &&IDX EQ ( &GOTO -OPTIONS
0130      &REPEAT
0140 -ERROR
0150      &TYPE Syntax error
0160      &QUIT
0170 ;
0180 ; One or more options specified
0190 ;
0200 -OPTIONS
0210      &WHILE &IDX LT &INDEX
0220           &IDX = &IDX + 1
0230           &IF &OPTION EQ 1 &GOTO -NAMES
0240           &IF &OPTION EQ 2 &GOTO -TYPES
0250           &IF &&IDX EQ NAME &GOTO -NAME
0260           &IF &&IDX EQ TYPE &GOTO -TYPE
0270           &GOTO -CONTINU
0280 ;
0290 ; Option NAME specified
0300 ;
0310 -NAME
0320           &OPTION = 1
0330           &IDX = &IDX + 1
0340 ;
0350 ; Still in option NAME
0360 ;
0370 -NAMES
0380           &IF &IDX GT &INDEX &QUIT
0390           &IF &&IDX EQ TYPE &GOTO -TYPE
0400           ERASE &&IDX * &DRV (NOQUERY) ; Erase files with NAME
0410           &GOTO -CONTINU
0420 ;
0430 ; Option TYPE specified
0440 ;
0450 -TYPE
0460           &OPTION = 2
0470           &IDX = &IDX + 1
0480 ;
0490 ; Still in option TYPE
0500 ;
0510 -TYPES
0520           &IF &IDX GT &INDEX &QUIT
0530           &IF &&IDX EQ NAME &GOTO -NAME
0540           ERASE * &&IDX &DRV (NOQUERY) ; Erase files with TYPE
```

```
0550 -CONTINU
0560           &REPEAT
0570 &QUIT
0580 ; Erase only BACKUP files
0590 -DEFAULT
0600      ERASE * BACKUP &DRV (NOQUERY)
0610      &QUIT
0620 -HELP
0630 &BEGTYPE
0640
0650 Function: Erase groups of files from disk(s)
0660
0670 Syntax:   CLEANUP [fd] [(options[])]
0680
0690 Where:
0700   fd  -   Drive to be cleaned (default = all)
0710
0720 Options:
0730   NAME    Following arguments are names to be erased
0740             multiple file names may be specified.
0750   TYPE    Following arguments are types to be erased
0760             multiple file types may be specified.
0770
0780   When no options are specified all BACKUP files will
0790   be erased from the specified disk(s).
0800 &END
0810 &QUIT
```

APPENDIX B

EXEC KEYWORD SUMMARY

Keyword	Function
&BEGSTACK	Stacks the lines following in the console input buffer without tokenizing them.
&BEGTYPE	Types the lines following on the console output device without tokenizing them.
&CAT	Indicates a concatenation of two variables or tokens without the result being tokenized.
&CONTROL	Sets switch indicating whether commands executed are displayed on the console device.
&CRT	Positions cursor on console device or perform screen function.
&END	Terminates the data following a &BEGSTACK or &BEGTYPE.
&ERROR	Designate instruction to be executed in event of error.
&ESC	Generates System Control Key sequences.
&GOTO	Unconditionally branches to a label in the EXEC file.
&IF	Tests variables and/or arguments.
&INDEX	Variable indicating the number of arguments passed to the EXEC processor.
&LEN	Computes value of the length of the variable following.
&LINE	Variable containing the value of the console line length.
&LIT	Causes the following character to not be evaluated as a token. Characters are truncated to eight.
&NULL	Variable containing no characters.
&PAGE	Variable containing the value of the console page length.
&QUIT	Transfers control out of the current EXEC file.
&READ	Accepts input from the console input device. The data read is assigned to the internal variables.
&REPEAT	Execute previous loop again (used in conjunction with UNTIL or WHILE).
&RETCODE	Variable indicating the return code passed to the EXEC processor.
&SKIP	Unconditionally branches to a line relative to this statement.
&SPACE	Types blank lines on the console output device.
&STACK	Stacks a line in the console input buffer. Line is tokenized.
&SUB	Indicates a sub-string of a variable or token.
&TYP	Causes the variable type of the variable following to be determined.
&TYPE	Prints a line at the console output device. Line is tokenized.
&UNTIL	Execute loop until condition is true.
&WAIT	Causes program to pause until operator responds.
&WHILE	Execute loop while condition is true.
&*	Indicates a test of any tokens matching specified comparison.
&$	Indicates a test of all tokens matching specified comparison.

Reader's Comments

Name _____ Date ___/___/___
Organization _____
Street _____
City _____ State ____ Zip _____

Name of manual: _____

Did you find errors in this manual? If so, specify with page number.

_____
_____
_____
_____
_____
_____
_____

Did you find this manual understandable, usable, and well-organized?
Please make suggestions for improvement.

_____
_____
_____
_____
_____
_____
_____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

_____
_____
_____
_____
_____
_____
_____

Indicate the type of user/reader that you most nearly represent:

__ Assembly language programmer
__ Higher-level language programmer (BASIC, FORTRAN, etc.)
__ Occasional programmer (experienced)
__ User with little programming experience
__ Student programmer
__ Non-programmer interested in computer concepts and capabilities
__ Data entry operator

Mail to: OASIS Documentation
Phase One Systems, Inc.
7700 Edgewater Drive #830
Oakland, CA 94621