NASA Contractor Report 178385

The Computational Structural Mechanics Testbed Architecture: Volume II - Directives

Carlos A. Felippa

Lockheed Missiles and Space Company, Inc.
Palo Alto, California

Contract NAS1-18444

February 1989

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Preface

The first three volumes of this five-volume set present a language called CLAMP, an acronym for **Command Language for Applied Mechanics Processors**. As the name suggests, CLAMP is designed to control the flow of execution of Processors written for NICE, the **Network of Interactive Computational Elements**, an integrated software system developed at the Applied Mechanics Laboratory.

The syntax of CLAMP is largely based upon that of a 1969 command language called NIL (NOSTRA Input Language). The language is written in the form of free-field source command records. These records may reside on ordinary text files, be stored as global database text elements, or be directly typed at your terminal. These source commands are read and processed by an interpreter called CLIP, the **Command Language Interface Program**. The output of CLIP does not have meaning *per se*. The Processor that calls CLIP is responsible for translating the decoded commands into specific actions.

NIL and its original interpreter LODREC, which now constitutes the "kernel" of CLIP, has been put to extensive field testing for over a decade. In fact NIL has been the input language used by all application programs developed by the author since 1969 to 1979. (NIL also drives the relational data manager RIM developed by Boeing for NASA Langley Research Center.) During this period many features of varying degree of complexity were tried and about half of them discarded or replaced after extensive experimentation. CLAMP represents a significant enhancement of NIL, particularly as regards to directive processing, interface with database management facilities, and interprocessor control. The current version is therefore believed to be powerful, efficient, and easy to use, and well suited to interactive work.

The present Manual is a greatly expanded version of the original March 1980 version, revised on April 1981. Because of its length, the material has been divided into five volumes, which cater to different user levels.

Volume I (NASA CR 178384) presents the basic elements of the CLAMP language and is intended for all users. Volume II (NASA CR 178385), which covers CLIP directives, is intended for intermediate and advanced users. Volume III (NASA CR 178386) deals with the CLIP-Processor interface and related topics, and is meant only for Processor developers. Volume IV (NASA CR-178387) describes the Global Access Library (GAL) and is intended for all users. Volume V (NASA CR-178388) describes the low-level input/output (I/O) routines.

All volumes are primarily organized as reference documents. Except for feeble at-

tempts here and there (*e.g.*, §3.1 in Volume I and Appendix C in Volume III), the presentation style is not tutorial.

Acknowledgements

The ancestor of CLIP, LODREC, was patterned after the input languages of ATLAS and SAIL, two structural analysis codes that evolved at Boeing in the late 1960s. More modern language capabilities, notably command procedures and macrosymbols, have been strongly influenced by the Unix™ operating system and the C programming language. The Unix "shell/kernel" concept, in fact, permeates the architecture of the NICE system, of which CLIP is a key component.

The author is indebted to the many CLIP users for constructive criticism and suggestions that have resulted in steady improvement of the interpreter, the CLAMP language, and its documentation over the past four years. Special thanks are due John DeRuntz, Don Flaggs, Bill Greene, Stan Jensen, Peter Kellner, Warren Hoskins, Tina Lotts, Ian Mathews, Bill Loden, Charles Perry, Charles Rankin, Jan Schipmolder, Gary Stanley, Brian Stocks, Lyle Swenson, Phil Underwood, Frank Weiler and Jeff Wurtz. Dave Cunningham contributed VAX/VMS environment query routines.

The development of CLIP during the period 1980-1981 was supported by the Advanced Software Architecture Project of the Independent Research Program of Lockheed Missiles and Space Co., Inc. The support received from 1982 to date from MSD's Structures is gratefully acknowledged. The development of several CLIP enhancements reported here has been supported by NASA Langley Research Center on contracts NAS1-17660 and NAS1-18444.

Contents

Contents (continued)

Contents (continued)

Contents (concluded)

THIS PAGE LEFT BLANK INTENTIONALLY.

1

Introduction

§1.1 DIRECTIVES

Directives are special commands that are understood and processed by CLIP, and not passed along to the Processor. (For a definition of the Processor, see §2 of Volume I.) Directives may be entered either by the Processor user or (as messages) by the Processor itself. A directive is to CLIP like an ordinary command is to the Processor.

You may grasp the function of directives by thinking of CLIP as a specialized operating system that supports interactive applications programming. One of the functions of operating systems is to provide services of general usefulness to users. To request a service by your computer operating system you enter a control statement. To request a service by CLIP you enter a directive.

A directive is distinguished from an ordinary command by beginning with a keyword attached to a directive prefix. By default this prefix is the asterisk, and this particular prefix will be assumed throughout the Manual. To illustrate:

```
*LIST INPUT.FIL
 LIST INPUT.FIL
```

The first command is a directive because the first item is prefixed by *. The second one is an ordinary command that is supposed to be interpreted by the Processor that calls CLIP.

The item prefixed by the asterisk must be the first item. For example,

```
DELETE *X.DATA
```

is an ordinary command.

An alphabetized list of the present CLIP directives is provided in Table 1.1. Some directives have commonly used abbreviations that are listed in Table 1.2. Sections 12 and beyond in the present Manual describe the directives in detail. Those sections are ordered in accordance with Table 1.1.

Table 1.1 **CLIP Directives**

Name	*Purpose*	*See*
ABORT	Triggers an abnormal run termination	§12
ADD	Redirects input to script source	§13
ALIAS	Defines a short name for a textstring	§14
CALL	Redirects input to command procedure	§15
CLOSE	Closes data library(ies)	§16
COPY	Copies datasets or records	§17
DEFINE	Defines or redefines macrosymbol(s)	§18
DELETE	Deletes datasets or records	§19
DO	Introduces FORTRAN-like looping block	§20
DUMP	Dumps contents of any system file	§21

Table 1.1 **CLIP Directives (continued)**

Name	*Purpose*	*See*
ELSE	Introduces "else" subblock in IF-THEN-ELSE block	§22
ELSEIF	Introduces ELSEIF subblock in IF-THEN-ELSE block	§23
ENABLE	"Undeletes" dataset(s)	§24
END	Terminates definition of command procedure	§25
ENDDO	Terminates label-less DO block	§26
ENDIF	Terminates IF-THEN-ELSE block	§27
ENDLOG	Terminates command transcription to log file	§28
ENDWHILE	Terminates WHILE-DO block	§29
EOF	Forces end of command source	§30
EOL	Forces end-of-line and clears dataline collector	§31

Table 1.1 **CLIP Directives (continued)**

Name	*Purpose*	*See*
FCLOSE	Closes FORTRAN text file	§32
FIND	Retrieves information on library, dataset or record	§33
FLUB	Flushes buffers of data library(ies)	§34
FOPEN	Opens FORTRAN text file and connects to unit	§35
FPRINT	Prints FORTRAN text file attached using FOPEN	§36
FREWIND	Rewinds FORTRAN file attached using FOPEN	§37
GAL2MAC	Defines a macrosymbol from GAL dataset value(s)	§37a
GENERATE	Generates next command(s) by item incrementation	§38
GET	Gets database entity	§39
HELP	Lists topic-qualified sections of help file	§40
IF	Tests and branches to label, or introduces an IF-THEN-ELSE block	§41
JUMP	Transfers control to label	§42
LIST	Lists file, Text Dataset or Text Group	§43
LOAD	Internalizes dataset(s) from ASCII files	§44
LOCK	Changes dataset access codes	§45

Table 1.1 **CLIP Directives (continued)**

Name	*Purpose*	*See*
LOG	Initiates command transcription to log file	§46
MAC2GAL	Writes a macrosymbol value to a GAL dataset	§46a
OPEN	Opens data library and connects to LDI	§47
PACK	Compresses library	§48
PRINT	Prints database entity	§49
PROCEDURE	Initiates definition of command procedure	§50
PUT	Stores database entity	§51
REMARK	Prints "active comment" line	§52
RENAME	Renames datasets or records	§53
RETURN	Exits from command procedure to caller	§54
RUN	Starts execution of another processor	§55
SET	Sets CLIP environmental parameters	§56
SHOW	Shows CLIP environmental parameters	§57
SPAWN	Initiates detached process under VAX/VMS	§58

Table 1.1 **CLIP Directives (continued)**

Name	*Purpose*	*See*
STOP	Terminates RUN-initiated processor execution	§59
TYPE	Terminal-directed LIST	§60
UNDEFINE	Deletes macrosymbol(s)	§61
UNLOAD	Externalizes dataset(s) to ASCII files	§62
WALLOCATE	Allocates scratch workrecord(s)	§63
WCHANGE	Changes logical size of workrecord(s)	§64
WCLOSE	Closes backed workrecord(s)	§65
WDEALLOCATE	Deallocates workrecord(s)	§66
WDEFINE	Defines macrosymbol(s) from workrecord items	§67
WFLUSH	Backs modified non-scratch records	§68
WGET	Reads database record(s) into workrecord(s)	§69
WHILE	Introduces WHILE-DO block	§70

Table 1.1 **CLIP Directives (concluded)**

Name	*Purpose*	*See*
WMAP	Prints Workpool allocation map	§71
WMARK	Marks workrecord(s)	§72
WOPEN	Opens backed workrecord(s)	§73
WPOOL	Changes Workpool extent	§74
WPRINT	Prints workrecord(s)	§75
WPUT	Writes workrecords to nominal library	§76
WSET	Sets workrecord items to given values	§77

Table 1.2 **CLIP Directive Abbreviations**

Abbreviation	*Full name*
DEC	SHOW DECODED_ITEMS
ECHO	SET ECHO
HFILE	SET HFILE
RAT	PRINT RAT
TOC	PRINT TOC

THIS PAGE LEFT BLANK INTENTIONALLY.

2

Directive Format

§2.1 STANDARD CLAMP FORMAT

This Section deals with the general format of directives. Without exception, directives obey the so-called *standard CLAMP format*, which is covered in §3.1 of Volume I. The following material focuses on the aspects of the standard format that are most significant when you write directives.

The Directive Verb

The first item of a directive is the *directive verb*, which is preceded by the directive prefix (normally the asterisk). The verb identifies in general terms what the directive does. A few *one-item directives* consist only of the verb. Examples:

```
*ABORT
*STOP
*EOF
```

The action verb on many directives has such a wide scope that it has to be followed by a *verb modifier* that circumscribes its meaning. Notable examples of such directives are `PRINT`, `SET` and `SHOW`.

Parameterized Directives

Most directives are *parameterized* in one way or another. For example:

```
*TYPE INPUT.DAT
```

is a parameterized directive that causes the contents of the text file `INPUT.DAT` to be listed on the terminal. Here `TYPE` is the directive verb while `INPUT.DAT` is a parameter. If you look up the description of the `TYPE` directive in the present Manual you will see a format description such as

TYPE *Filename*

This description style obeys the metalanguage rules covered in §10 of Volume I. The key point is that *Filename* is a parameter; that's why it is shown in lower case italics; capitalization of the first letter conventionally indicates that a character string is expected. On the other hand, the directive verb `TYPE` is shown in upper-case typewriter style to indicate a *literal*, *i.e.*, something that you should write exactly as shown.

Parameter Lists

Parameters need not be single items. Some directives take *parameter lists*. A list is a sequence of items separated by commas. Example:

```
*DELETE 2,35
```

The two integers: `2,35` form a parameter list for the `DELETE` directive. In all CLIP directives that admit lists, *the order of the items is relevant*. For the above example,

```
*DELETE 35,2
```

would have a different meaning. The first DELETE directive requests that dataset at sequence 35 in library 2 be deleted. The second one requests that dataset at sequence 2 in library 35 be deleted. As library indices cannot exceed 30, the second directive is in fact illegal.

Assignment Directives

The most general form of a parameterized directive is one in which a parameter, or parameter list, is equated to another parameter, or parameter list. Examples:

```
*SET UNIT PRT = 8
*WSET IFIBO = 1,2,3,5,8,13,21
*COPY 1,2 = 3,6,ABSTRACT
```

This directive form is typically used to "assign" or "instantiate", in some sense, objects named in the right parameter list to the objects named on the left. Think of the general form

Verb *Destination* ⟵ *Source*

or, if you are mathematically minded,

Verb *Left-hand side* ⟵ *Right-hand side*

in which the *Verb* clarifies the operation. These are called *assignment directives.* For some directives the second list is optional, in which case the equals sign (also called the list separator) may be omitted. But if the second list is present, you may not omit the equal sign.

Qualifiers

Directive options are generally specified through *qualifiers.* A qualifier is a word preceded by a special prefix, which in CLAMP is by default the slash. For example, the directive

```
*OPEN /NEW INPUTFIL
```

opens a *new* permanent file called INPUTFIL. The blank before the / is in most cases optional but does no harm.

The key feature of a qualifier is that it is *optional*, which means that there is always a *default interpretation.* If you just type:

```
*OPEN INPUTFIL
```

this must be a legal directive for opening INPUTFIL. (The default interpretation, by the way, is open an old file if it exists, otherwise create a new one.)

In most cases, the position of a qualifier doesn't matter as long as it comes after the verb. Thus

```
*OPEN INPUTFIL /NEW
```

also works in the OPEN directive. This indifference to position is an asset in interactive work, as the need for a qualifier often comes as an afterthought, after one has typed much of the non-default part.

A few directives such as DEFINE and WSET require that qualifiers, if any, appear immediately after the verb.

The slash is the default qualifier in CLAMP directives, but it may be changed to another special character through the use of the SET CHARACTER directive.

Parameterized Qualifiers

Sometimes qualifiers are followed by a parameter or parameter list, to which they are connected by an equal sign. Here is an example: the directive

```
*OPEN INPUTFILE /NEW /LIMIT=4500000
```

specifies the *capacity* of file INPUTFIL, which is the maximum size to which the file may expand after creation; this is necessary on some computer systems. Just using /LIMIT wouldn't work; the computer has to be instructed "how big it can get". But it is perfectly acceptable to have a default capacity, so LIMIT is a directive qualifier and not a directive parameter.

Directive Output using Macrosymbols

Certain directives are designed to retrieve information from the global database, and to make this information available to subsequent commands or directives. This information is stored as the value of variable-like entities called *macrosymbols*. The techniques involved are described in further detail in §4. Here we simply note that such macrosymbols are specified as *parameterized qualifiers*. Example:

```
*FIND RECORD 4,35,STIFFNESS /TYPE=STIFF-TYPE
```

The FIND RECORD directive searches for a database record identified by the parameter list 4,35,STIFFNESS. If the record is found, its type code is returned as the value of macrosymbol STIFF-TYPE, which is created as a result of the directive execution.

3

Input Redirection

§3.1 MOTIVATION FOR SCRIPTS

A beginner CLAMP user soon gets used to typing commands in conversational interaction with a processor. But sooner or later the thought arises, "Boy, wouldn't it be neat if I could prepare all these commands in advance and just tell CLIP where to find them. That would save me the work of retyping many of them when I am rerunning similar problems."

This can in fact be easily done, but first you have to master three terms that are frequently used in this document: *input redirection*, *scripts*, and *command procedures*. Input redirection means simply that CLIP is taking its input from a source other than the standard input file described in §4 of Vol. I. (To save you looking it up there, this file is your terminal when running in the interactive mode or a system-predefined file when running in the batch mode.) Redirection can be made to a *script* or to a *command procedure*. In this Section we deal primarily with scripts. Command procedures, which pertain to a more advanced level, are dealt with in §5.

A script is a set of commands which is prepared *before* you run the program that will process those commands. The script may reside on an ordinary system file, or (less frequently) on a data library accessible by the global database manager GAL-DMS. Usually the script is prepared with a text editor but sometimes you can built a script using the "log file" facility of CLIP.

The ADD directive (alternative name: INCLUDE) is used to instruct CLIP to read a script. For example, if you type

```
*ADD INPUT.DAT
```

CLIP will begin reading commands from file INPUT.DAT. This file may also contain ADD directives and this "script nesting" process may be continued for several levels. The following section gives a tutorial introduction to script preparation.

§3.2 A GENTLE INTRODUCTION TO SCRIPTS

A Tutorial Example

You should go through the following example if you have never prepared a CLIP script file before. All it takes to become a script writer is to be able to operate a text editor. Access your favorite editor and make a file that only contains two directives:

```
*rem Entering script file
*show css
```

Let's say the name of this file is SCRIPT.ADD, which is a legal VMS filename. Now run any NICE Processor and in response to the prompt type

```
*add script.add
```

You should see a response like:

```
Entering script file

<CL> Command Source Stack:
     Un  Lin  Rec Ldi Dsn Rap   Loc E  Name
     35    2    0   0   0   0     0 0  SCRIPT.ADD
      0    5    0   0   0   0     0 0  $term
```

In response to the ADD directive, CLIP has redirected the input to file SCRIPT.ADD and begins reading lines from it. The first line is a REMARK directive that causes CLIP to print the text that follows. The next line is a SHOW directive that asks CLIP to display the Command Source Stack. The first line of the display should show the current source and sure enough, it's SCRIPT.ADD. When CLIP tries to read the third line it detects an end of file and input returns to the previous command source. As the previous source is the terminal, you will see the prompt reappear.

For this simple case, you may visualize the ADD process by imagining that the contents of the SCRIPT.ADD file *replaces* the ADD directive line.

REMARK 3.1

Since you are running the example interactively you won't normally see an echo of the lines read from SCRIPT.ADD. If you would like to see them you should turn the echo on through a SET ECHO directive.

A More Realistic Example

One can hardly justify the work involved in preparing a two-line script file, even if it is used over and over. For interactive work it is just as quick to type the lines directly. But things change as your command streams become more voluminous. For example, the following command stream defines a problem for the example processor studied in Appendix B of Volume III:

```
clear
def segments
   seg=1 b=1,0          e=.9659,.2588
   seg=2 b=.9659,.2588 e=.8660,.5000
   seg=3 b=.8660,.5000 e=.7071,.7071
   seg=4 b=.7071,.7071 e=.5000,.8660
   seg=5 b=.5000,.8660 e=.2588,.9659
   seg=6 b=.2588,.9659 e=0,1
end
def material
   em=7.0E4 ; pr=0.2 ; end
def symmetry
   xsym=0 ; ysym = 0 ; end
def prestress
    sxx0=100 ; end
def field
    line=1 f=1,0 l=6,0 p=9
    line=2 f=0,1 l=0,6 p=9
end
pri seg ; pri mat ; pri bou ; pri symm ; pri pres ; pri field
build ; gen ; sol
pri res ; pri res/field
stop
```

The significance of these commands is not relevant here. What is important is that typing them interactively would be error prone. If you type in one wrong item you will have to begin again. And this is a simple command stream compared to those used in the production analysis of complex engineering problems.

Further Applications

Two important applications of scripts are noted next.

Batch Runs. If you run CLIP supported programs in batch mode, input redirection is normally used because there is no human sitting at a terminal. You simply prepare scripts (or command procedures once you reach that level) and tell CLIP where the input is. Since scripts are usable without change in both interactive and batch modes, it is quite common to perform quick interactive checks before submitting large batch runs.

REMARK 3.2

On many systems you can submit batch runs with the commands typed after the control statement that runs the Processor. This works well for simple input streams but for complicated ones it is better to have a script that can be interactively checked.

Archival. An important attribute of scripts (and of command procedures as well) is that they can be *archived* for documentation and legal purposes. (Much in the same way as programmers used to archive input data decks in the good old days of punched card input.) If you plan to archive important scripts, it is a good idea to document them profusely.

§3.3 MORE ON SCRIPTS

Parameterization

The essential feature that distinguishes scripts from command procedures is the absence of *input parameters* definable as arguments. Script lines are processed "as is". Therefore, you would not expect to see arguments in an ADD directive and in fact no arguments are permitted. You can only give the file name.

What are the practical implications? Consider again the example of §3.2. Let's say you want to run this problem several times with different values of pr in the line that follows **def material**. Since only a single value is involved, it's quite easy to change the file with the text editor before rerunning each case. A slightly more advanced technique is to replace the number by a prompt string; for example

```
em=7.0E4 ; pr="Poissons ratio:  " ; end
```

but this solution is limited to interactive work.

The situation would be less clear if you have several parameters, or, more commonly, if parameterized inputs are interdependent. An example of the latter is be a problem in which you want to change two dimensions, say A and B, and there are many other inputs related to those two, for example 0.5*A, etc. Eventually a point is reached at which it pays to use command procedures rather than scripts.

Scripts vs. Command procedures

The two big differences between scripts and command procedures are:

1. A script cannot be parameterized except through terminal intervention.
2. Processing of script lines is strictly sequential. There can be no branching or cycling directives within a script. It follows that scripts are less flexible than procedures. However, for many applications the additional flexibility is not warranted. Scripts have the advantage of being easier to learn and prepare than procedures, and for many users that is reason enough.

§3.4 LOGGING

Creating Scripts by Logging

Besides the text editor, there is another important way of creating a script: *logging* commands entered during a conversational interactive session. Suppose you start a CLIP-linked

Processor, and in response to the first prompt you type:

```
*log save.dat
```

On receiving the LOG directive, CLIP opens a new card-image file (a log file) called SAVE.DAT. Whatever you type from now on *at the root level* will be transcribed to that file. The transcription will continue until you enter an ENDLOG directive, or the Processor run is terminated. For example, suppose that the commands that follow the *log command are

```
begin nodal data
 node = 1  coordinates = 1.4, -4.9, 0.7
 node = 2  coordinates = 7.8, -8.1, 3.7
 print nodes 1,2
end
*endlog
```

The ENDLOG directive terminates the transcription process, and CLIP closes the log file (i.e., file containing commands entered at the root level). If you would like to check what was stored there, enter a TYPE or LIST directive. For example:

```
*type save.dat
```

and you should get

```
begin nodal data
 node = 1  coordinates = 1.4, -4.9, 0.7
 node = 2  coordinates = 7.8, -8.1, 3.7
 print nodes 1,2
end
```

Note that the ENDLOG directive itself is not stored as long as it appears on a line by itself.

The SAVE.DAT file is a script and so it may be ADDed to reproduce exactly the same commands you had previously. What's the difference with text editor preparation? *Logging* saves commands that were used to run a Processor, and you could observe the response to them. This mode is therefore recommended when you do not have a very clear idea of what the final script will look like.

REMARK 3.3

Logging is also useful when the Processor is not really command driven but asks for specific data using prompts. What you end up with is an input data file that meets the Processor requirements. Although such an input file is normally unreadable, using this procedure usually saves time-consuming manual consultation.

Editing a Log File

In practice the difference between creating scripts by *logging* commands and with a text editor is not drastic because it is quite common to have to edit a log file. For example,

suppose that you enter a wrong command and the Processor gives you an error message. The wrong command will be in the log file because transcription occurs immediately upon entering the command line. You will subsequently edit the file to remove the incorrect line(s); but the overall work may be small compared to the task of building a correct script from scratch and with no Processor feedback.

REMARK 3.4

On VAX/VMS you can edit from a detached process by using the **SPAWN** directive, without having to stop the Processor to start up the editor.

Logging and the Input Level

Suppose that the interactive session goes

```
*log input.dat
begin nodal data
*add nodal.dat
end
begin element data
*add elem.dat
end
*endlog
*type input.dat
```

As a result of the TYPE directive you will see that the contents of the log file are:

```
begin nodal data
*add nodal.dat
end
begin element data
*add elem.dat
end
```

The important thing to note is that the **ADD** directives have been logged, but not the contents of the script files **NODAL.DAT** and **ELEM.DAT** referenced by those **ADD**s. Similarly, if you type a **CALL** to a procedure the **CALL** will be logged but not the procedure lines. This is what the statement "saving lines at root level" means.

REMARK 3.5

In the preparation of complex scripts "cutting and pasting" techniques are commonly used. For example, the **ELEM.DAT** and **NODAL.DAT** files noted previously could have been prepared by selective logging.

REMARK 3.6

It is possible to ask CLIP to log lines that are read regardless of input level by specifying an **ALL** qualifier in the **LOG** directive.

§3.5 DATA LIBRARY RESIDENCE

Scripts do not necessarily have to be ordinary files. You can store scripts in data libraries and access them through the ADD directive.

A script may be stored on a positional data library (DAL or GAL80 form) as a Text Dataset. A Text Dataset is identified by a pair of items: the Logical Device Index (LDI) of the library (which must be open) and the dataset name (or sequence number). For example:

```
*OPEN 13, MODEL.DAT
*ADD 13, MOTOR-CASE.MODEL.14
```

opens positional library file MODEL.DAT, which is connected to Logical Device index 13, and redirects input to Text Dataset MOTOR-CASE.MODEL.14. If this dataset is at sequence number 116, you can also type

```
*ADD 13, 116
```

but the specification by name is generally preferable.

A script may be stored on a nominal (GAL82) library as a Text Group. In this case you have to provide three pieces of identification: the library's LDI, the owner dataset name or sequence number, and the Text Group name. For example:

```
*OPEN 13, MODELNOM.DAT
*ADD 13, MOTOR-CASE.SCRIPTS.6, MODEL14
```

The OPEN directive does the same work as above, but now the ADD redirects input to the script stored as Text Group MODEL14 of dataset MOTOR-CASE.SCRIPTS.6.

Putting and Extracting Scripts

One drawback of library residence is that initial preparation and modification of scripts requires additional steps because you cannot use the text editor on library files.

To prepare a library-resident script, you first make up a file with the text editor. Then you insert the file into a data library with a PUT TEXT directive. The file can then be deleted.

To modify a library-resident script, you have to extract it into an editable file using a GET TEXT directive. The file is edited and reinserted into the library with a PUT TEXT directive, and the external file deleted.

Why Libraries?

Obviously, scripts that reside in data libraries are more cumbersome to use and modify. Why then bother? There are several advantages:

1. *Archivability.* Data libraries provide a convenient "encapsulation" mechanism. This is particularly true on computers that do not have a nice hierarchical file structure.

2. *Portability.* Frequently your scripts may contain references to other scripts. If scripts reside on ordinary files, you may be faced with a file naming problem if you move

them to a different computer. For example

```
*ADD INPUT.DAT
```

works on VAX/VMS, but will not work on the Cray because `INPUT.DAT` is not an acceptable file name. On the other hand, dataset and record names are not machine dependent.

3. *Transportability.* There may be an efficient data library "converter" between two different computers, for example, the Vax-Cray nominal GAL converter written by Frank Weiler. Placing your scripts into data libraries makes input data conversion straightforward as opposed to moving perhaps hundreds of tiny files back and forth with attendant filename incompatibility problems.

4. *Storage compaction.* On many computers, placing many tiny scripts into a data library saves disk storage because of the clustering factor. (With current trends in storage, this is not likely to be an important factor, however.)

§3.6 ADVANCED TOPICS

The following features are occasionally useful for advanced applications, and will be primarily of interest to programmer developers familiar with the interface described in Volume III. Beginner and intermediate CLAMP users may safely skip this section.

Interrupting a Script using EOF

The run of a script is interrupted if an `EOF` directive is found. For example, let us say you prepare the following script:

```
*set echo = on
line 1
line 2
*eof
line 3
line 4
```

If you interactively `ADD` this to the command stream you will see that upon encountering the `EOF` line the command input reverts to the terminal. The last two command lines are ignored.

At first sight this looks rather silly. If you don't want the Processor to read `line 3` and `line 4`, why put them there in the first place? The answer is that the `EOF` line is usually submitted by the Processor as a message, so it is not physically present in the script.

The main application of `EOF` messages is error recovery in preprocessor programs. Let us say that a Processor is reading a problem definition stored as a script, and that serious errors are detected midway. The Processor could abort the run, but that may not be a good strategy in interactive mode. The solution is to send an `EOF` message after notifying

the user about the error. Control then reverts to the previous source, which is usually the terminal, and the user may initiate recovery procedures.

On command procedures, the EOF has the same effect as a RETURN directive.

ADDing a Preconnected Unit

An ordinary ADD-file directive such as

```
*ADD INPUT.DAT
```

involves dynamic file opening and closing. That is, file INPUT.DAT is opened and connected to an internal logical unit number such as 35; when the end of file is detected the file is closed.

Now suppose that INPUT.DAT has been *preconnected* to the run, for example by an operating system control statement, and that it is connected to FORTRAN logical unit 5. If you type

```
*ADD 5
```

CLIP will simply redirect its input to unit 5, and will not attempt to open a file. When an end of file is detected input will revert to the previous source but the unit is not closed.

In interactive mode unit 0 (zero) is assumed to be your terminal. Consequently, the directive

```
*ADD 0
```

redirects input to the terminal. Normally this directive would be inserted in a script or command procedure (or sent by the Processor as message) to request human feedback. Here is an example of Processor triggered intervention:

```
CALL CLPUT ('*remark I am very confused.  What do you want to do?')
                    CALL CLPUT ('*add 0')
```

You may trace the source of an *ADD 0 by entering a SHOW CSS directive. To return input to the previous source, you enter an EOF directive, which acts as a software end of file.

The READ Qualifier

This final topic is admittedly a highly specialized one. It applies to the following scenario:

1. The Processor accesses CLIP only using messages sent using CLPUT;
2. One of the messages submits an ADD directive, and
3. The Processor does not interact with the user.

The ADD submitted using CLPUT should then have a READ qualifier. Example:

```
program CRUNCH
call CLPUT ('*add/read crunch.dat ')
stop
end
```

where file CRUNCH.DAT consists entirely of directives. The qualifier READ forces the file to be read through.

To understand the need for the READ qualifier, consider two alternative constructions.

```
program CRUNCH
call CLPUT ('*add crunch.dat ')
stop
end
```

CLPUT will process the ADD directive by opening PROCESS.DAT and pushing it into the command source stack. But the directives in that file will not get processed, because there is no CLREAD statement to force that to happen. Adding a call to CLREAD, as in

```
program CRUNCH
call CLPUT ('*add crunch.dat ')
call CLREAD (' >', ' ')
stop
end
```

has an undesirable side effect. The file CRUNCH.DAT will be processed ("pulled in") by the CLREAD call. But when the end of file is detected input will revert to the command source stack root, that is, the terminal, and a surprised user will be prompted for input.

In summary, the READ qualifier forces file read-through without the bad side effects of CLREAD.

THIS PAGE LEFT BLANK INTENTIONALLY.

4

Macrosymbols

§4.1 MOTIVATION

The Need for Variable-Like Items

Data and special items allow only immediate specification of values. You have to write down the item value (or values) when you make up commands or directives. For example, if you type

```
SET XYZ = 1.2, 3.5, 8.913
```

you obviously know that the command verb is SET, its parameter is XYZ and the three numeric items that follow are 1.2, 3.5 and 8.13. This is the common case in conversational interactive work when commands are composed "on the spot."

But suppose that the three numeric values depend on other input values, or on the state of the Processor at the time the command is read in. It would be desirable to treat them like variables and be able to type something like

```
SET XYZ = X, Y, Z
```

with the tacit understanding that X, Y and Z are to be *replaced* by numeric values at the time the command is read. Two questions immediately arise:

- How does CLIP know that X is a variable-like item and not the character string 'X'?
- How is CLIP told that X, for example, is to be replaced by the numeric value 1.2?

Resolving these two questions in a sensible fashion requires *extensions* of the command language. The set of extensions that addresses these questions forms the *macrosymbol* facility.

Extending the Language

In general terms, a *macrosymbol* facility is a tool by which a command language can be extended in power and convenience. The CLIP macrosymbol facility has been designed so that there are virtually no limits to these extensions, other than the amount of effort you are willing to spend to learn it.

Through the macrosymbol facility, you can establish variable-like items, perform arithmetic manipulations, access built-in functions such as sine or arctangent, rename directives, abbreviate procedure calls. You can use macrosymbols in conjunction with the command procedure facility described in §6 to set up branching and looping constructs. Finally, macrosymbols used in conjunction with the message (mailbox) capability allow the *processor* to insert values in the command stream for various purposes.

§4.2 presents an overview of the facilities as presently implemented in CLIP. The remaining sections explain in more detail the things you can do with macrosymbols.

§4.2 OVERVIEW

Terminology

The macrosymbol facility provides a mechanism by which a character string called *macrosymbol* is replaced by another string called *replacement text.*

The replacement process is called *macrosymbol expansion.* Expansion is generally followed by an *evaluation* process influenced by the *type* of the macrosymbol. The final result is a *macrosymbol value.* So the complete process can be diagrammed as

$$\textit{Macrosymbol} \xrightarrow{\textit{expansion}} \textit{Expanded macrosymbol} \xrightarrow{\textit{evaluation}} \textit{Value}$$

REMARK 4.1

The term "macrosymbol" is frequently abbreviated to *macro* when there is no risk of confusion with unrelated things such as MacroProcessors. This abbreviation is often used here; thus we often speak of macro expansions, macro values, etc.

Table 4.1 at the end of §4.2 summarizes terminology.

Implementation

A macrosymbol is implemented in CLIP as a data structure that contains the following information:

Name. A string of up to 16 characters that identifies the macrosymbol. The first character must be a letter or a dollar sign. If a dollar sign, the second character must be a letter.

Definition. A character string that governs the expansion process. There are two definitions: external and internal. The external definition is the one you supply when you define a macro name, and so it is naturally called a *macro definition text.* The internal definition is the one saved internally by CLIP; this is called the *replacement text.* The transition from external to internal is accomplished by a process called *pre-evaluation*:

$$\textit{Macro definition text} \xrightarrow{\textit{pre-evaluation}} \textit{Replacement text}$$

Often the replacement text is identical to the macro definition text, in which case the pre-evaluation process reduces to a straight copy.

Type. The macro type plays a role similar (but not identical) to that of the data type of variables in languages like FORTRAN. The basic types are integer, floating-point and character string, with some variations among the last two. The macro type affects the pre-evaluation process, expansion and final evaluation.

Scope. This is an attribute that is important when macrosymbols are used within command procedures. It refers to the "visibility" of macrosymbols across procedure boundaries, and has points of similarity with the visibility of variables across subroutines in languages like FORTRAN or C.

Defining Macrosymbols

Let us assume you start up a CLIP-supported Processor and that the following conditions apply: the Processor is not initiated through SuperCLIP, and the macrosymbol facility is available in the CLIP version tied to the Processor.

Even before you enter the first command, a set of *built-in* macrosymbols (presently 68 of them) is available to you. These provide values or functions of general utility; for example to evaluate the sine or cosine of an angle or to build logical expressions. These macrosymbols are described in the next Section. You should not redefine built-in macrosymbols.

Macrosymbols that are not built-in must be defined before they can be used. You define a new macrosymbol to CLIP by specifying its name, definition, type and scope in a `DEFINE` directive. The basic components of the directive are

`*DEFINE` *Name* = *Definition_text*

with options for type and scope. The same directive is used to *redefine* an existing macrosymbol.

REMARK 4.2

Macrosymbols may also be defined by the Processor by submitting a `DEFINE` message. In addition, macrosymbols may be created implicitly as a result of certain directives other than `DEFINE`.

REMARK 4.3

Note that we begin with two assumptions. As for the first one: if a Processor is started by another using SuperCLIP, all macrosymbols previously defined are available; they don't disappear across Processor boundaries.

REMARK 4.4

Some versions of CLIP are devoid of the macrosymbol facility; this depends on compile-time decisions. You may check whether your version has the facility by entering a `DEFINE` directive. If the CLIP response is "Illegal or ambiguous directive," it doesn't have it.

Using Macrosymbols

When you are ready to use a defined macro in a command or directive, you surround the macrosymbol name with two special characters called *macro identifiers.* By default these are the angle brackets < and >. The occurrence of a name surrounded by these identifiers triggers an expansion and evaluation process. This process is performed *before* the command text is parsed but *after* procedure arguments are substituted in case macros are used within a command procedure.

REMARK 4.5

We shall see that macrosymbols may be nested within macrosymbols virtually *ad infinitum.* If you become proficient in the use of macros, it is important to keep in mind that expansion always proceeds "from the inside out". That is, the name occurring in the innermost pair is processed first, and so on until the outermost pair is reached.

Can Macrosymbols Disappear?

Any macrosymbol that is not built-in may be "undefined" through an UNDEFINE directive, which erases it from the macrosymbol table. More common, however, is *automated undefinition* triggered by the scope-of-validity rules described later. For example, if a command procedure has defined a non-global macrosymbol called SYMBOL, on exit from the procedure SYMBOL becomes undefined.

Built-in macrosymbols may not be undefined.

Finishing the Example

We are now ready to finish the example of §4.1. Three macrosymbols are defined by typing

```
*DEF X = 1.2
*DEF Y = 3.5
*DEF Z = 8.13
```

These directives create three macrosymbols: X, Y and Z, which are defined to be 1.2, 3.5 and 8.13, respectively. According to the default typing rules elaborated upon later, these three macrosymbols are assigned to be the double-precision floating type. So in fact the replacement text internally stored by CLIP is

Name	*Replacement text*
X	1.2D+00
Y	3.5D+00
Z	8.13D+00

(You may see the actual definition saved by CLIP by using the SHOW MACRO directive.) Now you type the example command as

```
SET XYZ = <X>, <Y>, <Z>
```

Since X is surrounded by the macro identifiers, which we assume to be < and >, CLIP knows that X is supposed to be a macrosymbol name, and proceeds to replace it by its definition, namely 1.2D+00. The same thing happens for <Y> and <Z>, so the command upon macro expansion is

```
SET XYZ = 1.2D+00, 3.5D+00, 8.13D+00
```

This is eventually given to the CLIP item parser. The parser of course interprets the last three items as floating-point constants and stores them as such in the Decoded Item Table for use by the Processor.

This explanation answers the two questions raised in §4.1. Macro delimiters avoid confusion with ordinary character strings: <X> is not the same as X. The replace-by-what question is taken care by the DEFINE directives that must precede the appearance of macrosymbol names in commands.

An Experiment

If you have never been exposed to macro expansions before, the following experiment should be instructive. Run interactively a CLIP supported Processor (preferably a do-nothing Processor) and type

```
*DEF X = 1.2
*DEF Y = 3.5
*DEF Z = 8.13
*SHOW MACROS
*SET ECHO = ON, MACRO, DECODE
 SET XYZ = <X>, <Y>, <Z>
```

Then try variations such as

```
*DEF X = (6/5)
*DEF XX = '(6/5)'
*DEF Y = (3+(1/2))
*DEF/F8.1 = 8.13
*SHOW MACROS
```

and try to figure out what happens.

The replace-name-by-string does not look like a particularly advanced concept. Three features of the macrosymbol facility add greatly, however, to the power of this deceptively simple replacement scheme.

1. The macro definition text may contain macro references; thus, the process may be *nested* to virtually any level.

2. Macrosymbols may contain arguments, and even arguments that are also macrosymbols!

3. The presence of powerful built-in macrosymbols that may be used as building blocks for highly complex arithmetic and logical expressions.

Table 4.1 provides a summary of the nomenclature used so far, plus more used later in this section.

REMARK 4.6

The present CLIP macro capability is largely modeled upon the C-language macro processor by D. M. Ritchie, a FORTRAN version of which is described in *Software Tools* by B.W. Kernighan and P.J. Plauger, Addison-Wesley, 1976.

Table 4.1. **Macrosymbol Terminology**

Term	*Definition*
Macro	An abbreviation for macrosymbol.
Macrosymbol name	A string of 1-16 characters that uniquely identifies a macrosymbol
Macrosymbol definition	A character string that replaces a macrosymbol reference in the expansion process. There are two definitions: external and internal.
External definition	The definition you write in the `DEFINE` directive
Internal definition	The definition saved by CLIP in the macrosymbol table.
Macrosymbol reference	The materialization of a macrosymbol in the text of a command by enclosing its name with the macro identifiers, which by default are < and >.
Macrosymbol expansion	The substitution of a macrosymbol reference by the replacement text.
Macrosymbol type	An attribute similar to the data type of variables in FORTRAN. It controls the pre-evaluation and expansion process
Scope of definition	A macro definition may be global, semilocal or local as regards the procedure level in the command source stack.
Global macro	A macrosymbol whose definition is valid everywhere in the command source stream
Semilocal macro	A macrosymbol whose scope of definition is limited to the procedure that declared it and subordinate procedures.
Local macro	A macrosymbol whose scope of definition is limited to the procedure that defined it.
Built-in macro	A macrosymbol predefined by CLIP

§4.3 MACROS WITH SIMPLE DEFINITIONS

In the present Section we deal with the simplest kind of macrosymbols: those that lack arguments and whose definition text does not contain macrosymbols. The material below

describes how such macrosymbols are defined and referenced, as well as the effect of assigning types. The exposition is example oriented, always proceeding from the specific to the general.

Subsequent subsections and those on Section 5 cover increasingly advanced topics: macros with arguments, built-in macros, logical expressions, macro arrays, and implicit macrosymbol definition.

Implicit Typing

The simplest application of macrosymbols entails the straight replacement of its name by a character or numeric value. Example:

```
*DEFINE PI = 3.14159265358979323
```

defines macrosymbol `PI` to be the fairly well known number on the right side. The equal sign after `PI` is mandatory.

Note that the directive does not explicitly specify the macro type. Consequently the default typing rules given in Table 4.2 (at the end of this §) apply. Since the right hand side is numeric and `PI` begins with the letter `P`, the default type is `D24.16`, which means CLIP will view its definition as a double-precision floating-point number with 16 digits of precision. The internally stored definition of `PI` will be

```
3.1415926535897932D+00
```

Another example:

```
*DEFINE NQ = 51
```

This directive defines `NQ` as an *integer* macrosymbol equal to `51`; integer type is assigned because the right-hand side is numeric, the name begins with the letter `N` and no explicit type is given.

A final example, in which the right hand side contains a character string:

```
*DEF FILEX = drc0:[felippa.clip]example.tes
```

The default type here is `A`, which stands for "unprotected character string", a term explained later.

Explicit Typing

If the default typing rules are unsuitable you must declare the macrosymbol type as a directive qualifier. Legal types are listed in Table 4.3. The qualifier must appear *before* the name, i.e., after `*DEFINE`.

The following exemplifies the use of a nonstandard numeric type:

```
*DEF/F12.4 PI_APPROX = 3.14159265
```

This directive defines PI_APPROX as a macrosymbol of data type F12.4, which has an interpretation analogous to a field specification in a FORTRAN FORMAT statement. If you type SHOW MACRO, you will see that the replacement text of PI_APPROX saved by CLIP is

```
3.1416
```

because you have instructed CLIP to keep only 4 digits after the decimal point and no exponent field.

Pre-evaluation of Arithmetic Expressions

Next consider a slightly more complex specification:

```
*DEF/E10.3 RADIUS = (2/3)
```

The right-hand side is a legal arithmetic expression. If you type SHOW MACRO RADIUS after this, you will find that the replacement text on the VAX is

```
.667E+00
```

This indicates that CLIP has, as part of the pre-evaluation feature, in fact processed the arithmetic expression *before* saving the result as replacement text. For *numeric* macrosymbols the process takes two steps:

1. The DEFINE-directive processor captures the right-hand side and sends it to the arithmetic subsystem to evaluate it. The result comes back as a binary number of integer or double-precision floating-point type.
2. The result is *encoded* back into characters according to the macro type, and saved as definition.

REMARK 4.7

The results of the encoding process may vary somewhat from one computer to another. For example, on computers with a wider exponent field, the replacement text may be .667E+000.

REMARK 4.8

In previous versions of this Manual, pre-evaluation was called *immediate evaluation.*

REMARK 4.9

For the text macros discussed in §5, the pre-evaluation generally reduces to conversion from lower to upper case.

Deferred Evaluation

Pre-evaluation works fine in most circumstances but sometimes it has to be inhibited. Such is the case, for example, if the definition text contains as-yet-undefined macrosymbols as explained in §4.4. To inhibit it, the definition text is enclosed within apostrophes, as in

```
*DEF/E10.3 RADIUS = '(2/3)'
```

You may verify that the replacement text saved by CLIP is

```
(2/3)
```

i.e., the directive processor "strips" the apostrophes and saves the definition intact. Now suppose you type

```
SET VALUE = <RADIUS>
```

Macro expansion gives

```
SET VALUE = (2/3)
```

and the last value evaluates to 0.6666666 ... in full double precision. Note that specifying E10.3 as the macrosymbol type is irrelevant; any other floating point type would produce the *same* answer. Therefore, if you choose deferred evaluation for a floating-point constant, there is no point in specifying a type other than D24.16.

Replacement

Here is a sample usage of the example definitions:

```
SET PERIMETER = (2.0 * <PI_APPROX> * <RADIUS> )
```

The macrosymbol occurrences are replaced by the definitions, yielding the macro expansion

```
SET PERIMETER = (2.0* 3.1416 * .6667E+00 ),
```

and the last item is eventually evaluated as a composite floating-point constant, yielding the value 4.18900944, as you may verify.

Table 4.2. **Default Typing Rules**

Right-hand side of DEFINE *is*	*First letter in Name**	*Default type*
Numeric	A-H, a-h, O-Z, o-z	D24.16**
Numeric	I-N, i-n	I
Character string	Any	A

* If the first name character is **$**, the rule applies to the second character.

** E24.16 on Cray or CDC.

Table 4.3. **Macrosymbol Type Identifiers**

Type	*Meaning*
A	Unprotected character string.
D*w.d*	Double-floating (*w*=width, *d*=decimals).
E*w.d*	E-single-floating (*w*=width, *d*=decimals).
F*w.d*	F-single-floating (*w*=width, *d*=decimals).
G*w.d*	G-single-floating (*w*=width, *d*=decimals).
I	Integer.
N	Nearest integer.
P	Protected textstring.
*	Generic (takes type of definition text; used only in some built-in macros)

§4.4 DEFINITION TEXT WITH MACROSYMBOLS

The definition of a macrosymbol may contain references to other macrosymbols, or to the macrosymbol being redefined. Examples:

```
*DEFINE PISQUARE = (<PI>^2)
*DEF N = (<N>+1)
*DEF/F12.4 BELLCURVE = (1/(1+(<X>*<X>)))
```

These three examples involve numeric macrosymbols defined by arithmetic expressions which include macrosymbols; the second one redefines N in terms of itself.

Since the definitions have not been enclosed in apostrophes, pre-evaluation takes place. Therefore, the macrosymbols on the right-hand side *must be defined* at the time these

directives appear. Sometimes this is not convenient or even possible, and if so deferred evaluation must be used:

```
*DEFINE/D PISQUARE = '(<PI>^2)'
*DEF/I N = '(<N>+1)'
*DEF/D BELLCURVE = '(1/(1+(<X>*<X>)))'
```

The definition is then saved "as is" (except that apostrophes are stripped) and the evaluation takes place when the macro reference appears.

REMARK 4.10

The choice of deferred or pre-evaluation can have a profound effect if right-hand-side macrosymbols are redefined in the interim between definition and use of a derived macrosymbol. For example, consider the two sequences:

```
*DEFINE N = 4              *DEFINE N = 4
*DEFINE M = (<N>+1)        *DEFINE/I M = '(<N>+1)'
*DEFINE N = 6              *DEFINE N = 6
 WHAT IS <M>?               WHAT IS <M>?
```

On the left sequence <M> evaluates to 5 whereas on the right it evaluates to 7.

Text Macrosymbols

For text macrosymbols that contain macro references in the definitions, the use of apostrophes is generally mandatory. Example:

```
*DEF/A OPEN3 = '*OPEN <LDI>, <FILEX>'
```

Omitting the apostrophes would cause CLIP to stop scanning after *OPEN because of the blank separator.

THIS PAGE LEFT BLANK INTENTIONALLY.

5

More on Macrosymbols

§5.1 MACROS WITH ARGUMENTS

It is also possible to define macrosymbols with arguments, so that the replacement text depends on the way the macrosymbol is referenced. Up to nine arguments are permitted in user-defined macros (some built-in macros may accept more, up to 19). Arguments are identified positionally, and are specified in the definition text. The definition text should be enclosed with apostrophes to inhibit pre-evaluation. A simple example:

```
*DEFINE POWER = '(($1)^($2))'
```

The replacement text is `(($1)^($2))`. Now consider the following reference:

```
<POWER(2.0;3.0)>
```

This expands to

```
((2.0)^(3.0))
```

which CLIP eventually evaluates to the floating point number `8.0D+00`.

REMARK 5.1

Note the use of possibly redundant parentheses. If you omit the parentheses surrounding **$1** and **$2** by writing

```
*DEFINE POWER = '($1^$2)'
```

you will get the same results for **<POWER(2.0;3.0)>**. But if the arguments are also arithmetic expressions surprises may result; for example **<POWER(4/2;3.0)>** expands to **<(4/2^3.0)>**, which evaluates to **0.5** instead of **8.0** because the exponentiation operator has higher precedence than the division operator.

Argument Specification Rules

The occurrence of an argument is indicated by the dollar sign **$** immediately followed by an unsigned integer in the range 1 through 9. This integer is the argument index; thus **$1** means "replace me by the first argument", **$2** "replace me by the second argument" and so on. The symbol **$0** is also acceptable and calls for replacement by the macrosymbol name itself.

Unlike procedure arguments, which are identified by keywords, macrosymbol arguments are strictly positional. Most implementation rules are nonetheless similar:

1. Macrosymbol arguments are bounded by matching left/right parentheses, and separated by semicolons.

2. Arguments may be individual data items, lists, or symbolic items. In particular, procedure arguments and macrosymbols (with or without arguments) are acceptable. An example of the latter:

```
<POWER(<SIND(30)>^2;<COSD(30)>^2>)>
```

which evaluates to

$$\left(\sin^2 30^\circ\right)^{\cos^2 30^\circ} = 0.25^{0.75} = 0.353553\ldots$$

3. If the macrosymbol type is not P, leading and trailing blanks are stripped, and lowercase letters converted to upper case. To protect blanks or lower case letters, enclose argument text in apostrophes as usual. If the macrosymbol type is P (protected text macrosymbol), blanks are preserved and lower case letters are not converted to upper case.

4. Trailing arguments may be omitted. Associated symbols in the macro definition text then "vanish" (more technically: they are replaced by a zero-length string). Intermediate arguments may be omitted by writing consecutive semicolons; the leading argument may be omitted by following the opening parenthesis with a semicolon, or a closing parenthesis if there is only one argument.

EXAMPLE 5.1

To illustrate the argument replacement mechanism, let us go through an example with the following macro, which defines the Euclidean length of a 3-vector:

```
*def v3length = (((($1)^2+($2)^2+($3)^2)^0.5)
```

In what follows, it is assumed that the definition of <PI> is 3.14159 and that formal procedure argument HALF stands for 0.5:

Reference	*Expansion → evaluation*
`<v3length(1;2;3)>`	`(((1)^2+(2)^2+(3)^2)^0.5)` → $\sqrt{14.}$ = 3.741657387
`<v3length( <PI>^[HALF] ; [HALF]^<PI> ;0)`	`(((3.14159^0.5)^2+(0.5^3.14159)^2+(0)^2)` → 1.776071585

§5.2 SCOPE OF MACROSYMBOLS

We have so far covered names, definitions and type. The fourth attribute of a macrosymbol is its *scope*. This is important only if you do a great deal of work with command procedures (described in §6). If you don't, you may skip this section without great loss.

When you learn a high-level programming language you probably had to adjust to the idea of global versus local variables. For example, in FORTRAN, COMMON variables are global to all subroutines that access their common block name. Local FORTRAN variables are not known outside the subroutine they are used.

There is a similar concept for macrosymbols. The equivalent of a subroutine is the command procedure. Macros may be global, local or semilocal with respect to command procedures.

A *global* macrosymbol is known everywhere in the command source stream. Built-in macrosymbols are global. Any macrosymbol defined at the root level is global unless its first character is a dollar sign, in which case it is local.

A *semilocal* macrosymbol is one that is defined inside a command procedure and is not explicitly declared global or local. A semilocal macrosymbol is known only within the procedure level that declared it or any higher procedure level. For example, if procedure X calls procedure Y, all semilocal macros defined by X before it calls Y are visible to Y. When the procedure that declares a semilocal macrosymbol is exited, the definition disappears.

A *local* macro is one whose name begins with a dollar sign. A local macro is not visible outside of the procedure level that defines it. When the procedure level falls under that of definition, a local macrosymbol disappears.

To force a macrosymbol defined within a procedure to be global, you write a double equals sign after the name in the DEFINE directive.

REMARK 5.2

The concept of *procedure level* is covered in §4 of Volume I. Here is a summary. The procedure level of the root is zero. When a procedure is called (exited) the procedure level is incremented (decremented) by one. ADDing a script does not change the level.

REMARK 5.3

In previous CLIP versions there were only global and semilocal macros. What is now called semilocal was then called local. The new class of local macros was introduced primarily to permit the implementation of the DO directive and thus provide a substitute for the departed register symbols.

§5.3 TEXT MACROSYMBOLS

A text macrosymbol is one whose definition text is to be handled as a character string. We have already encountered some examples in previous sections. In the present section text macrosymbols are examined in more detail.

Text macrosymbols are typed A or P. The former are called unprotected, the latter protected.

Unprotected Text Macrosymbols

If you specify type A, the definition text is treated much like an ordinary character string. It is best to go through an example. Consider

```
*DEF/A FILENAME = input.dat
```

Since you haven't used apostrophes, CLIP pre-evaluates the definition text and stores

```
INPUT.DAT
```

The reference

```
OPEN 3,<FILENAME>
```

expands to

```
OPEN 3,INPUT.DAT
```

If the definition text had been enclosed with apostrophes, the replacement text saved by CLIP would have been `input.dat`. But when expansion of `<FILENAME>` occurs, then the text is converted to upper case. The end result is the same.

A Long Definition Text

Now let's consider a longer definition text that includes several words so apostrophes are mandatory:

```
*DEF/A OTC = '*open 1,file ; *toc 1 ; *cl 1'
```

which incidentally illustrates the fact that the definition text may be a set of commands and/or directives. The text is saved by CLIP "as is". Now suppose you type:

```
<OTC>
```

This expands to

```
*open 1,file ; *toc 1 ; *cl 1
```

This is evaluated as three directives, with a grand total of seven items and two separators.

But sometimes this is not what you have in mind; instead you want to retrieve the definition text *intact*. For example,

```
*DEF/A TITLE = 'This is a plot title'
```

will not work if you want the Processor to retrieve the expansion of `<TITLE>` as a plot title: lower case is converted to upper, and, worst of all, the string is broken up into five items! To "protect" the string you should type `*DEF/P` instead of `*DEF/A` as explained below.

Protected Macrosymbol

For text macrosymbols declared of type `P` (for "protected") CLIP will insert a pair of enclosing parentheses when the name is expanded. Using the last example:

```
*DEF/P TITLE = 'This is a plot title'
```

Suppose that the macro name is used in

```
SET PLOT TITLE = <TITLE>
```

This expands to

```
SET PLOT TITLE = 'This is a plot title'
```

The last item evaluates to the character string `This is a plot title`, which may be retrieved as such by the Processor. Notice that blanks and lower case letters are preserved.

REMARK 5.4

If you define `TITLE` with type `A` and then try

```
SET PLOT TITLE = '<TITLE>'
```

then `<TITLE>` is *not* a macrosymbol occurrence, because it is enclosed in apostrophes! If you have defined `TITLE` with type `A` for some reason, you may force a `P` evaluation by saying

```
SET PLOT TITLE = <TITLE/P>
```

This technique is explained in more detail later in this Section.

§5.4 BUILT-IN MACROSYMBOLS

Concept

CLIP comes equipped with a set of predefined or "built-in" macrosymbols. These macrosymbols define constants, variables or arithmetic or logical operations of general utility that assist in writing complicated expressions. For example, the macrosymbol `PI` so often used previously is actually a built-in macrosymbol that gives π to 16 decimal places; macrosymbol `SIN` evaluates the sine of its angle-in-radians argument, and so on.

There are presently over 60 built-in macrosymbols. Some of these are used in conjunction with the Local Data Manager and are not treated here. The other ones are listed in Table 5.1. Some context-directed macrosymbols, such as `MOD`, appear in the Table twice should it be convenient to distinguish between integer and floating arguments.

Built-in macros can be categorized into various types described briefly below roughly in terms of increasing complexity.

Predefined Constants

These are: `D2R` $= \pi/180$ (degrees-to-radians conversion factor), `TRUE` (integer 1), `FALSE` (integer 0), and `PI`.

Mathematical Functions

The basic ones are: `ACOS` (arccosine), `ASIN` (arcsine), `ATAN2` (two-argument arctangent), `COS` (cosine), `EXP` (exponential), `LOG` (natural logarithm), `SIN` (sine) and `TAN` (tangent).

Variants of these include LOG10 (decimal logarithm), and ATAN2D, COSD, SIND and TAND, which work with degree rather than radian angular units.

There is also the polynomial evaluator POLY, which takes up to five coefficient arguments. Recent additions include SINH (hyperbolic sine), COSH (hyperbolic cosine), TANH (hyperbolic tangent) and QUADRATIC (quadratic equation solver). All return double-precision floating values.

There are four new macros that pertain to random number generation: RANDNORM, RANDSET, RANDTRUN and RANDUNIF. Every time RANDNORM, RANDTRUN or RANDUNIF is reference you get a different value unless the sequence is re-initialized using RANDSET or by setting an optional argument to a nonzero value (see Table 5.1).

REMARK 5.5

ATAN2D, COSD, SIND and TAND were named ATAN2G, COSG, SING and TANG, respectively, in previous CLIP versions. The old names will also work.

Generic Functions

Macros ABS, MAX, MIN, MOD and SIGN furnish the equivalent of the generic FORTRAN functions of the same name. These can take either integer or floating-point arguments and return a result of the same type.

MAX and MIN now take a variable number of arguments (up to 19).

Reserved Variables

Macrosymbol names ERROR_STATUS and RUN_MODE are reserved by CLIP.

Boolean Functions

Macrosymbols AND, OR and NOT provide the intersection, addition and logical-negation Boolean functions. These operate on integer arguments and produce a TRUE (integer one) or FALSE (integer zero) result. An argument which is neither 1 or 0 is treated as TRUE. AND and OR take a variable number of arguments, up to 19.

REMARK 5.6

With the recent provision of the logical macro expressions discussed in §4.10 there is less need for using AND and OR in its primitive form.

Logical Functions

The "IF functions" IFEQ, IFLE, IFLT. IFGE, IFGT and IFNE expand to either TRUE (integer one) or FALSE (integer zero), depending on an algebraic, floating-point comparison of the arguments.

String Catenator

Macrosymbol CAT catenates its string arguments into a single output string. Both argument and result are of protected-string type, which means that lower case is not converted to upper case and that blanks are respected. The primary use of CAT is building legends for graphic displays.

REMARK 5.7

In the present version of CLIP, string catenation can be more easily done with the double-slash operator, just like FORTRAN. For example

```
'The 16-dig value of pi is '//<pi>
```

is equivalent to (but easier to write than)

```
<CAT(The 16-dig value of pi is ;<pi>)>
```

String Matchers

IFELSE and IFDEF are the most powerful built-in macrosymbols presently offered. They are patterned after similar facilities in the C programming language. Both IFDEF and IFELSE treat all arguments as character strings and expand to another character string. Since they are type A, blanks and lower-case letters are *not* significant unless explicitly protected.

IFDEF checks whether its first argument is a defined macrosymbol. If so, it expands to its second argument; otherwise to its third. Example:

```
<IFDEF(LIBFIL; *open 2,<LIBFIL> ;)>
```

This means: if LIBFIL is a defined macrosymbol then open a data library whose name is the macrosymbol definition, otherwise do nothing.

IFELSE compares its first two arguments character-by-character. If they are equal it expands to its third argument; otherwise to its fourth. For example:

```
<IFELSE(<CONVERGENCE>;<FALSE>;*jump :loop ; *return)>
```

This means: if the definition of macrosymbol CONVERGENCE is FALSE (i.e., integer zero), then jump to label LOOP; otherwise return from procedure.

A recent addition is IFCASE, which simplifies the construction of "CASE" statements that depend on comparing a key against values.

Status Macros

The value of these macros depend on the run state. Presently they are BATCH, CSLEVEL and CSNAME, DATE and TIME.

String Functions

Macros INDEX, LEN and SUBSTRING will be provided in the future to mimic similar FORTRAN functions.

Workpool Macros

There is a class of built-in macros whose names start with the letter W. These macros are used to retrieve attributes of local database entities. These macros, which are in an experimental status, are described in Section 9.

Table 5.1. Built-in Macrosymbols

Macrosymbol reference	*Argument type*	*Result type*	*Explanation*
<ABS(k)>	xint	I	Absolute value of argument k.
<ABS(x)>	xfloat	D	Absolute value of argument x.
<ACOS(x)>	float	D	Arc cosine of x; result in radians.
<ACOSD(x)>	float	D	Arc cosine of x; result in degrees.
<AND($k_1;\ldots;k_n$)>	int	I	Logical intersection of arguments $k_1 \ldots k_n$, $2 \leq n \leq 19$.
<ASIN(x)>	float	D	Arc sine of x; result in radians.
<ASIND(x)>	float	D	Arc sine of x; result in degrees.
<ATAN2($x_1;x_2$)>	float	D	Arctangent of x_1/x_2; result in radians.
<ATAN2D($x_1;x_2$)>	float	D	Arctangent of x_1/x_2; result in degrees.
<BATCH>		I	1 (TRUE) if batch run; else 0 (FALSE).
<CAT($s_1;\ldots;s_n$)>	P	P	Catenate protected strings $s_1 \ldots s_n$, $1 \leq n \leq 19$.
<COS(x)>	float	D	Cosine of x; argument in radians.
<COSD(x)>	float	D	Cosine of x; argument in degrees.
<COSH(x)>	float	D	Hyperbolic cosine of x.

See Tables 5.2 and 5.3 for rules on argument and result types.

Table 5.1. Built-in Macrosymbols (Continued)

Macrosymbol reference	*Argument type*	*Result type*	*Explanation*
<CSLEVEL>		I	Current command source level (0 = root level)
<CSNAME>		A	Name of current command source.
<D2R>		D	Degree-to-radians conversion factor $\pi/180$ to 16 places.
<DATE>		A	Date in $mm : dd : yy$ format.
<ERROR_STATUS>		I	Reserved.
<EXP(x)>	float	D	Exponential of x
<FALSE>		I	Integer zero for logical tests.
<IFCASE($s; s_1; \ldots; s_k$)>	A	I	If string s matches s_i ($0 \leq i \leq k$, $k \leq 18$), then i else 0.
<IFDEF($s_1; s_2; s_3$)>	A	A	If s_1 is a defined and accessible macrosymbol, then s_2 else s_3.
<IFELSE($s_1; s_2; s_3; s_4$)>	A	A	If string s_1 matches s_2, then s_3 else s_4.
<IFEQ($x_1; x_2$)>	float	I	If $x_1 = x_2$ then 1 (TRUE) else 0 (FALSE).
<IFEQS($s_1; s_2$)>	A	I	If $s_1 = s_2$ then 1 (TRUE) else 0 (FALSE).
<IFLE($x_1; x_2$)>	float	I	If $x_1 \leq x_2$ then 1 (TRUE) else 0 (FALSE).
<IFLT($x_1; x_2$)>	float	I	If $x_1 < x_2$ then 1 (TRUE) else 0 (FALSE).
<IFGE($x_1; x_2$)>	float	I	If $x_1 \geq x_2$ then 1 (TRUE) else 0 (FALSE).

See Tables 5.2 and 5.3 for rules on argument and result types.

Table 5.1. Built-in Macrosymbols (Continued)

Macrosymbol reference	*Argument type*	*Result type*	*Explanation*
<IFGT(x_1;x_2)>	float	I	If $x_1 > x_2$ then 1 (TRUE) else 0 (FALSE).
<IFNE(x_1;x_2)>	float	I	If $x_1 \neq x_2$ then 1 (TRUE) else 0 (FALSE).
<INDEX(s_1;s_2)>	A	I	FORTRAN-like substring locator. Not implemented.
<LEN(s)>	A	I	Not implemented.
<LOG(x)	float	D	Natural logarithm of x.
<LOG10(x)>	float	D	Decimal logarithm of x.
<MAX(k_1;...;k_n)>	xint	I	Algebraically largest of $k_1 \ldots k_n$, $2 \leq n \leq 19$.
<MAX(x_1;...;x_n)>	xfloat	D	Algebraically largest of $x_1 \ldots x_n$, $2 \leq n \leq 19$.
<MIN(k_1;...;k_n)>	xint	I	Algebraically smallest of $k_1 \ldots k_n$, $2 \leq n \leq 19$.
<MIN(x_1;...;x_n)>	xfloat	D	Algebraically smallest of $x_1 \ldots x_n$, $2 \leq n \leq 19$.
<MOD(k_1;k_2)>	xint	I	Remainder of k_1/k_2.
<MOD(x_1;x_2)>	xfloat	D	Remainder of x_1/x_2.
<NOT(k)>	int	I	If $k = 0$ (FALSE) then 1; if $k = 1$ (TRUE) then 0.
<OR(k_1;...;k_n)>	int	I	Logical addition of arguments $k_1 \ldots k_n$, $2 \leq n \leq 18$.

See Tables 5.2 and 5.3 for rules on argument and result types.

Table 5.1. Built-in Macrosymbols (Continued)

Macrosymbol reference	*Argument type*	*Result type*	*Explanation*
<PI>		D	π to 16 places.
<POLY(x;c_0;...;c_4)>	float	D	Quartic polynomial $c_0 + c_1x + \cdots + c_4x^4$. Omitted trailing arguments are assumed zero.
<QUADRATIC[1](a;b;c)>	float	D	Real part of root x_1 of $ax^2+bx+c = 0$, where $\Re(x_1) \leq \Re(x_2)$.
<QUADRATIC[2](a;b;c)>	float	D	Real part of root x_2 of $ax^2+bx+c = 0$, where $\Re(x_1) \leq \Re(x_2)$.
<QUADRATIC[3](a;b;c)>	float	D	Absolute value of imaginary component of roots of $ax^2 + bx + c = 0$.
<RANDNORM(x_1;x_2;k)>	F,F,I	D	Generate element of a normally-distributed pseudo-random number sequence with mean value x_1 and standard deviation x_2. Optional argument k initializes the sequence if it appears (see RANDSET).
<RANDSET(k)>	I	N	Initializes random number generation sequence in use by RANDxxxx macros. If $k < 0$, initialize sequence from clock reading. If $k > 0$, use this integer value to initialize the sequence.
<RANDTRUN(x_1;x_2;x_3;x_4;k)>	F,F,F,F,I	D	Generate element of a truncated, normally-distributed pseudo-random number sequence with mean value x_1, standard deviation x_2, min value x_3 and max value x_4. Optional argument k initializes the sequence if it appears (see RANDSET).

See Tables 5.2 and 5.3 for rules on argument and result types.

Table 5.1. Built-in Macrosymbols (Concluded)

Macrosymbol reference	*Argum type*	*Result type*	*Explanation*
<RANDUNIF(x_1;x_2;k)>	F,F,I	D	Generate element of a uniformly-distributed pseudo-random number sequence with minimum value x_1 and maximum value x_2. Optional argument k initializes the sequence if it appears (see RANDSET).
<RUN_MODE>		A	Reserved.
<SIGN(k_1;k_2)>	xint	I	Absolute value of k_1 with sign of k_2.
<SIGN(x_1;x_2)>	xfloat	D	Absolute value of x_1 with sign of x_2.
<SIN(x)>	float	D	Sine of x, argument in radians.
<SIND(x)>	float	D	Sine of x, argument in degrees.
<SINH(x)>	float	D	Hyperbolic sine of x.
<SUBSTRING(s;i;j)>	A,I,I	A	Experimental.
<TAN(x)>	float	D	Tangent of x, argument in radians.
<TAND(x)>	float	D	Tangent of x, argument in degrees.
<TANH(x)>	float	D	Hyperbolic tangent of x.
<TIME>		A	Time of day in $hh:mm:ss$ format.
<TRUE>		I	Integer one for logical tests.
<Wxxxxxxx(....)>			Workpool macros; see Table 9.2.

See Tables 5.2 and 5.3 for rules on argument and result types.

Table 5.2. Rules for Argument Typing in Built-in Macros

Argument Type	*Meaning*
`int`	Arguments are assumed to be of integer type. If a floating point item is given, it is truncated to integer. If a character string is given, it is assumed zero.
`float`	Arguments are treated as double-precision floating-point numbers. If an integer appears, it is converted to floating point; for example, `<EXP(1)>` is the same as `<EXP(1.0)>`. If a character string appears, it is treated as zero.
`xfloat`	Explicit floating-point. At least one argument must be written as a floating-point item. This affects macros `ABS`, `MAX`, `MIN`, `MOD` and `SIGN`. For example, `<MAX(2;4.)>` is the same as `<MAX(2.;4.)>` and evaluates to floating-point `4.DO`.
`xint`	Explicit integer. All arguments must be written as integers. This affects macros `ABS`, `MAX`, `MIN`, `MOD` and `SIGN`. For example, `MAX(2;4)` evaluates to integer 4.
`A`	Ordinary character string. Unless explicitly protected with apostrophes, lower case letters are converted to upper case, and leading/trailing blanks are ignored.
`P`	"Protected" character string. Lower case letters are not converted to upper case, and all blanks are respected. There is no need to use apostrophe delimiters.

Table 5.3. Rules for Result Type in Built-in Macros

Result Type	*Meaning*
A	Ordinary character string. May be changed to P by writing qualifier P after macro name.
D	Double precision with ≈16 digit accuracy (Single precision on Cray)
I	Integer.
N	Null: expands to nothing (technically: to a zero length string)
P	Protected character string. CLIP encloses the result string in apostrophes automatically. May be changed to A by writing qualifier A after macro name.

§5.5 MACROSYMBOL ARRAYS

A *macrosymbol* array is a set of macrosymbols with the same "name root" followed by a bracket-enclosed index. For example:

```
DIRCOS[1], DIRCOS[2], DIRCOS[3]
```

might identify the 3 direction cosines of a space vector.

Consecutive elements of a macrosymbol array may be simultaneously defined in one directive as illustrated by

```
*def/i fibo = 1,2,3,5,8
```

This directive defines *five* integer macrosymbols named `fibo[`i`]`, where i is 1 through 5. More precisely, it is equivalent to the five definitions:

```
*def/i fibo[1] = 1
*def/i fibo[2] = 2
*def/i fibo[3] = 3
*def/i fibo[4] = 5
*def/i fibo[5] = 8
```

The integer enclosed in square brackets is called the *macro index*, which can be used as a selector in obvious ways. For example:

```
BEGIN COUNT = <FIBO[4]>
```

or

```
*DO $K = 1,5
  BEGIN COUNT = <FIBO[<$K>]>
*ENDDO
```

Arbitrary Index Base

A macrosymbol array does not have to begin at index 1. If you type

```
*def/i fibo[4] = 5,8,13
```

then you define `fibo[4]=5`, `fibo[5]=8`, and `fibo[6]=13`. A zero index is also permitted, but not a negative one. The above is the same as typing

```
*def/i fibo[4:6] = 5,8,13
```

The general definition may contain an index specification as in

```
*def/i fibo[1:7:3] = 5,8,13
```

which defines `fibo[1]=5`, `fibo[4]=8`, `fibo[7]=13`. The default for the third index is 1 if the first index is less or equal than the second one, or -1 otherwise. For example:

```
*def/i fibo[12:1] = 1:12
```

defines `fibo[12]=1`, ... `fibo[1]=12`.

In the previous examples the list on the right of the equal sign matches the number of expected values. If the list is shorter, the last value is used as a filler. Examples:

```
*def/d b[1:100] = (1/3)
*def/i pow[40:80:2] = 2,4,8
```

The first directive defines the 100 macrosymbols `b[1]`, `b[2]`, ... `b[100]` and assigns the value one-third to each one. The second directive defines `pow[40] = 2`, `pow[42] = 4`, `pow[44] = ...` `pow[80] = 8`.

Reference by Index Range

Several macrosymbols that belong to the same array may be referenced by giving an index range. Example:

```
DELETE ELEMENTS <K[2:4]>
```

is the same as writing

```
DELETE ELEMENTS <K[2]>,<K[3]>,<K[4]>
```

Notice that the expansion is in the form of a list. As in the case of macro definitions, the general index reference is

$$[n_1 : n_2 : n_3]$$

which is interpreted in the usual FORTRAN DO style.

The following default rules apply. If n_3 is omitted, one is assumed. If n_1 is omitted but a colon appears before n_2, the lowest defined index is assumed. If n_2 is omitted but a colon appears after n_1, the largest defined index is assumed. For example, suppose that bb[12] through bb[24] are defined, then

bb[:15]	same as	bb[12:15]
bb[14::2]	same as	bb[14:24:2]
bb[:]	same as	bb[14:24]

Macrosymbol Arrays with Arguments

Nothing prohibits a macrosymbol array from having arguments. Believe it or not there are some applications for that complexity. Here is a definition of normalized direction cosines in space:

```
*def s =  '(($1)^2+($2)^2+($3)^2)^.5)'
*def dc = '(($1)/<s($1;$2;$3)>)', --    defines dc[1]
          '(($2)/<s($1;$2;$3)>)', --    defines dc[2]
          '(($3)/<s($1;$2;$3)>)' .      defines dc[3]
```

It is instructive to figure out what happens if you write

```
<dc[1:3](3;0;4)>
```

This should evaluate to the list 0.6, 0.0, 0.8.

If you manage to reach this level, keep in mind that the arguments are written last, since the index or indices are attached to the macro name.

REMARK 5.8

Built-in macrosymbol **QUADRATIC**, which solves a quadratic equation, is of this type. For example,

```
<QUADRATIC[1:3](1;2;1)>
```

finds the roots of $x^2 + 2x + 1 = 0$ and expands to -1,-1,0 (see Table 5.1).

§5.6 LOGICAL EXPRESSIONS

Another way of using the Boolean built-in macros such as **AND**, **OR**, or the **IFxx** series is through *logical expressions*. The expression form is more readable and easier to remember since it is similar to high-level language constructions. Table 5.4 lists all elementary forms of these expressions. More complex ones are obtained by catenating and nesting.

Comparing Numeric Values

The most commonly used logical expressions result from comparing two numeric values. For example:

```
<4 /GT 3>
```

This cannot be a macrosymbol reference, because the first character after < is not a letter. It cannot be a numeric macro (§5.7) because of the presence of the qualifier **/GT**. So by default CLIP views it as a logical expression, which evaluates to either 1 (for **TRUE**) or 0 (for **FALSE**). CLIP in fact transforms first the above expression to

```
<IFGT(4;3)>
```

which is a built-in macro. Since 4 is greater than 3, the macro evaluates to 1.

Another example that illustrates the use of nesting and internal macro references:

```
< <4 /gt <PI>> /or <2 /le <exp(1)> > >
```

This evaluates first to <1 /or 1> and finally to 1 (**TRUE**).

Boolean operators **/AND** and **/OR** may be used at the same nesting level, but the two types should not be mixed. For example, the expression

```
< 1 /or 1 /or 0 >
```

is correct; CLIP transforms this to <OR(1;1;0)> which evaluates to 1. On the other hand,

```
< 1 /or 1 /and 0>
```

is incorrect because of ambiguity. You have to group it to tell CLIP what you have in mind:

```
<<1 /or 1> /and 0>
<1 /or <1 /and 0>>
```

The first form evaluates to 0 and the second form evaluates to 1.

Comparing Characters

To test two character strings for equality you can use the EQS relational qualifier. Example:

```
<abcde /eqs abcde>
```

This is transformed to

```
<IFEQS(abcde;abcde)>
```

which evaluates to TRUE because ABCDE matches ABCDE (the unprotected strings are converted to upper case before comparing them).

There is no NEQS qualifier; use NOT to get TRUE on unmatched strings. For example,

```
*if <NOT(<<analysis> /eqs dynamic>)> /then
```

Table 5.4. Relational Qualifiers in Logical Expressions

Expression or subexpression	*evaluates to*
e_1 /EQ e_2	TRUE if $e_1 = e_2$, else FALSE
e_1 /LE e_2	TRUE if $e_1 \leq e_2$, else FALSE
e_1 /LT e_2	TRUE if $e_1 < e_2$, else FALSE
e_1 /GE e_2	TRUE if $e_1 \geq e_2$, else FALSE
e_1 /GT e_2	TRUE if $e_1 > e_2$, else FALSE
e_1 /NE e_2	TRUE if $e_1 \neq e_2$, else FALSE
b_1 /AND b_2	TRUE if both b_1 and b_2 are TRUE, else FALSE
b_1 /OR b_2	TRUE if either b_1 or b_2 is TRUE, else FALSE

§5.7 NUMERIC MACROS

There is a final interpretation placed on expressions surrounded by macro delimiters and which cannot be legally classified as macrosymbol references or logical expressions: numeric macros.

Suppose you write an arithmetic expression that looks like a macrosymbol reference:

```
<1/3>
```

CLIP interprets this as the arithmetic expression

```
(1/3)
```

sends it to the item-evaluation subsystem where it gets evaluated to the fraction one third, and encodes the result back with the result that <1/3> ends up replaced by something like 3.333333333333333D-01. An expression such as <1/3> is called a *numeric macrosymbol* reference.

What are numeric macros good for? After all, typing (1/3) is just as easy as typing <1/3> (in fact, parentheses are easier to find on keyboards). There is no difference if the Processor accesses ordinary commands through the CLREAD or CLNEXT entry points described in Volume III; use of any of these entries implies that CLIP item decoding is accepted.

But if the calling program uses CLGET to directly retrieve datalines, there is a significant difference because macro substitution is effected *before* the line is delivered. For example, if you type

```
SET VALUE = (1/3)
```

CLGET delivers exactly that, but if you write

```
SET VALUE = <1/3>
```

then CLGET delivers

```
SET VALUE = 3.3333333333333333D-01
```

This can make a difference if the receiving program does not know what to do with (1/3) but it can deal with 3.3333333333333333D-01.

When does this situation arise? Programs that call CLGET fall into two classes: NICE conforming Processors whose developers prefer to do their own item parsing, and command-driven foreign programs connected to CLIP as card-image feeder.

§5.8 TYPE OVERRIDE AT EXPANSION TIME

One final refinement concerning macrosymbol references is worth mentioning. You can override the macro type in the reference by writing the desired type as a qualifier just after the macrosymbol name. For example:

```
<pi/p>      <exp/p(3)>
```

expand to

```
'3.1415926535897932'      '2.008553692318767D+01'
```

whereas if you remove the /P you don't get the enclosed apostrophes. Note that if the macro has arguments, like EXP, you have to place the type qualifier *before* the opening parenthesis.

As of this writing this feature works correctly only for A and P type specifications, but it will eventually extended to any type.

THIS PAGE LEFT BLANK INTENTIONALLY.

6

Command Procedures

§6.1 BASIC CONCEPTS

The command-procedure capability of CLIP allows the insertion of a set of *predefined command records* at any point in the command source stream. Unlike scripts, selected portions of the inserted commands can be *replaced* by text specified in the procedure reference or "call". Another important difference from scripts is that commands need not be processed sequentially; branching and looping constructions may be implemented with the help of `DO`, `IF` and `WHILE` directives.

CLIP procedures bear faint similarities to FORTRAN subroutines and to assembly-language "templated" macros. It has to be stressed, however, that a command language procedure represents source text to be *interpreted* rather than compiled or assembled. Although CLIP procedures go through a process called here "compilation" for want of a better word, the resulting product is not machine code.

These attributes make command procedures more flexible than ordinary programming languages. Of course, something has to give: the additional flexibility is obtained at the expense of execution efficiency. Consequently, you should not use a procedure to carry out detailed number-crunching activities that are best performed by the Processor. Procedures, like command languages, should be primarily written to define problems and manage activities.

REMARK 6.1

Although procedures are inherently more flexible than scripts, there is a minor physical limitation that must be kept in mind when you begin writing command procedures. Dataline length *is strictly limited to 80 characters*, whereas with script files you can go up to 132 by increasing the maximum line input width. The limitation is tied up to the fixed-record-length nature of callable procedure elements. It follows that you must be prepared to use continuation marks more often should long commands occur.

REMARK 6.2

The command procedure capability must be specifically activated through an MSC key before compilation of CLIP's source code. If CLIP is compiled without the command procedure capability, the directives discussed in this Section are not recognized.

An Illustrative Example

The motivation behind the use of command procedures can be more easily grasped through an illustrative example.

A CLIP-supported processor has to solve a geometrically nonlinear structural problem through a modified Newton-Raphson iteration while the external loads are kept fixed. Let's say that the solution process is controlled by a command sequence such as

```
SET LOADS 1.25, -2.61
FORM STIFFNESS K
FACTOR K
  FORM RHS FI, FA, R
  SOLVE K, R, X
  INCREMENT D, X
  FORCES D, FI
  FORM RHS FI, FA, R
  SOLVE K, R, X
  INCREMENT D, X
  FORCES D, FI
SET LOADS 1.44, -2.85
FORM STIFFNESS K
FACTOR K
  FORM RHS FI, FA, R
  ...
```

Here a SET LOADS command sets the value of two external load parameters. The remaining commands specify stiffness assembly, factorization, and two modified Newton-Raphson iteration cycles per load step. (The items after each command verb are local matrix names.)

The repetitive nature of the process makes it a natural candidate for the use of a command procedure. Let's call it ILMNR2 (for "increment loads and do two modified Newton-Raphson cycles"). Here is the procedure definition:

```
*PROCEDURE ILMNR2 (P1; P2)
 SET LOADS [P1], [P2]
 FORM STIFFNESS K
 FACTOR K
   FORM RHS FI, FA, R
   SOLVE K, R, X
   INCREMENT D, X
   FORCES D, FI
   FORM RHS FI, FA, R
   SOLVE K, R, X
   INCREMENT D, X
   FORCES D, FI
*END
```

Now some basic terminology. The symbols P1 and P2 that appear in the procedure naming line are known as *formal arguments.* These arguments appear in the SET LOADS command as

```
SET LOADS [P1], [P2]
```

where they are later to be replaced by *actual arguments* supplied in the procedure CALL. The bracket delimiters are required to distinguish them from unrelated "local" use of P1 and P2; for example as matrix names.

The procedure definition is presented to CLIP, typically by a script file that has it by using an ADD command. CLIP "compiles" the procedure and puts out a *callable version* into an ordinary data file or a data library, as explained later. This version can be invoked by a CALL directive such as

```
*CALL ILMNR2 (P1=1.25; P2=-2.61)
```

The effect of this call is to reproduce exactly the first ten commands of the example sequence. You can then keep on merrily going:

```
*CALL ILMNR2 (P1=1.25; P2=-2.61)
*CALL ILMNR2 (P1=1.44; P2=-2.74)
*CALL ILMNR2 (P1=1.49; P2=-2.88)
```

The use of a procedure here is justified by its labor saving: ten commands are replaced by one. Plainly the motivation grows stronger as one faces the prospect of executing hundreds or even thousands of commands in a repetitive fashion.

Nonsequential Command Execution

There is another important motivation for using command procedures. Nonsequential command execution that involves *conditional transfers* and *looping* can only be accomplished within the framework of a command procedure. To illustrate this, suppose that you want to make the number of modified Newton-Raphson iterations in our example procedure a variable called NCYCLES. This can be easily implemented as shown:

```
*PROCEDURE ILMNR2 (P1; P2; NCYCLES=2)
 SET LOADS [P1], [P2]
 FORM STIFFNESS K
 FACTOR K
 *DO $N = 1,[NCYCLES]
    FORM RHS FI, FA, R
    SOLVE K, R, X
    INCREMENT D, X
    FORCES D, FI
 *ENDDO
*END
```

Now a reference such as

```
*CALL ILMNR2 (P1=1.25; P2=2.61; NCYCLES=5)
```

will result in the iteration loop being executed five times. (Details on the use of the DO directive to control command loops are given in Section 6.) This example also illustrates the use of *argument default text*, which is the text that follows after NCYCLES=. Suppose that the reference to the procedure happens to be

```
*CALL ILMNR (P1=1.25; P2=2.61)
```

Since NCYCLES is not specified in the calling sequence, the default value of 2 is inserted in [NCYCLES], and the loop is executed twice. No default text is given for P1 and P2; if P1, say, is omitted then [P1] is "erased" when the line that references it is processed.

With the aid of macros and the self-message facility, one could have the *processor* take control of the cycling termination. (For example, if convergence tests performed within the processor are met.) Thus there is virtually no limit on "command language power" afforded through the combined use of these advanced facilities.

§6.2 PROCEDURE MANAGEMENT OVERVIEW

From the example discussed in the previous section it is not difficult to see that a command procedure consist of three components: *header*, *body*, and *terminator*. The header and terminator are provided through the PROCEDURE and END directives, respectively. The body consist of ordinary commands possibly intermixed with directives other than PROCEDURE and END.

When CLIP encounters a PROCEDURE directive, it enters directive mode and does not exit until the END directive is detected. The result of this process is an "object" version of the procedure, known as a *callable procedure element*.

Callable procedure elements may be accessed through CALL directives. You may visualize the effect of a procedure reference by imagining that its body, with argument text substituted as indicated, replaces the CALL directive. Text substitution is controlled by the argument specification mechanism. A procedure body may include CALLs to other procedures, or may even call itself, and the ensuing call tree can extend down several levels. Thus a seemingly innocent procedure reference can potentially trigger a very complex sequence of events that entails the processing of thousands or millions of commands.

Two User Roadblocks

Experience has shown that CLAMP users "arriving" at the procedure level are often baffled by two aspects: the parameter-substitution mechanism, and the physical residence of callable procedure elements.

Parameter substitution may be mildly puzzling to FORTRAN programmers familiar with the call-by-reference argument passing scheme typical of FORTRAN implementations. In a command procedure reference, text is passed instead of addresses to data. The text supplied in the CALL directive *is replaced before the command is interpreted.* This linking scheme is known in computer science as call-by-name, and is characteristic of some strongly-typed programming languages such as Algol.

Another unconventional feature of parameter passing are the default rules. Occurrences of a parameter that is not explicitly specified in the calling sequence are replaced by default text. The default text is specified when you write the procedure source. An example of this is provided in the last version of the example procedure ILMNR2, in which the default value of argument NCYCLES is 2. If default text is not explicitly specified, occurrences of unspecified arguments disappear.

The second understanding difficulty is tied up to the questions

How are procedure definitions entered?

Where does CLIP put callable procedure elements?

This difficulty is more serious in nature and deserves a fairly detailed explanation. This is done in the following two subsections.

Preparing a Procedure Source

Most directives are so simple that entering them from a keyboard terminal is trivial. One could try to define simple procedures in exactly that way: just sit there and type them while you are running a Processor. *Don't.* There are two problems with this:

1. A keyed-in procedure definition is volatile and not saved unless you have opened a log file.

2. Post-facto editing is out; once the return key is pressed, that line is gone! (The next subsection tells where.) These disadvantages become increasingly serious in long or involved procedures, and most real-life ones are so.

The sensible way to create a nontrivial procedure is to use the text editor. Once the procedure source text is ready on a data file, it can be inserted in the command source stream with an ADD directive, and away it goes! If it becomes necessary to change the procedure, the source file can be easily edited as appropriate, and re-ADDed.

Residence of Callable Procedure Elements

CLIP can store a callable procedure element in one of two residence media:

1. An *ordinary direct-access formatted file* created through a FORTRAN 77 OPEN statement. All records of such a file have the same length (80 characters) and contain one data line. The file name is the same as the procedure name.

2. A *data library* managed through the Global Data Manager GAL-DMS. A callable procedure element is stored
   (a) as a Text Dataset if the library is positional (DAL or GAL80 format). The dataset name is the same as the procedure name. The library is identified by the Logical Device Index (LDI) specified in a previous SET PLIB directive.

(b) as a Text Group if the library is nominal (GAL82 format). The procedure name is the same as the Group name. The library LDI and the "owner" dataset name (or sequence number) are specified in a previous SET PLIB directive.

In either case, the callable element is a sequence of fixed length (80 characters) card-image records.

REMARK 6.3

The text of a callable procedure text is basically a copy of the source procedure body, prefaced by three linkage tables. These tables store argument names, argument default text, labels (explicit or generated) and their locations within the body. The copy of the procedure body is terminated by a special marker record.

REMARK 6.4

You may wonder why direct-access files are used in case 1. Wouldn't sequential access files save space? The answer is that procedure records are not necessarily processed sequentially. One of the reasons for using procedures is the ability to branch and loop, and efficient nonsequential processing demands a direct-access organization.

REMARK 6.5

You should never tamper with a callable procedure element (much the same as you wouldn't try to edit object code produced by a compiler). If a procedure has to be changed, edit the source and reprocess it.

How does CLIP know where to store callable procedure elements? The recommended method involves the use of a procedure library specification. It is also possible to store the specification in the procedure declaration itself. Experience has shown that the second method is confusing and even dangerous unless you are quite experienced. Consequently, only the first method is explained below.

Residence on Ordinary Files

Suppose that you have prepared (with the text editor) a card-image file named STOOGES that contains the definition of three procedures:

```
*PROCEDURE LARRY
 *CALL CURLY
*END
*PROCEDURE CURLY
 *CALL MOE
*END
*PROCEDURE MOE
 *CALL LARRY
*END
```

Now run a processor — any CLIP-linked processor will do — and enter the following directive

```
*ADD STOOGES
```

As a result of the ADD, CLIP will read file STOOGES, find three procedures: LARRY, CURLY and MOE, and create three callable procedure elements. Since no residence specification appears in the *PROCEDURE line, these will be three ordinary files named LARRY, CURLY and MOE, respectively, with perhaps some computer-dependent appended extension (see Remark).

REMARK 6.6

On the VAX, the system generated names will be slightly different: LARRY.DAT, CURLY.DAT and MOE.DAT, respectively.

REMARK 6.7

If you would like to see what the callable procedure element files look like, use the TYPE directive; for example

```
*TYPE LARRY
```

REMARK 6.8

What happens if you say *CALL LARRY? Try it and you soon get a "CSS exhausted" catastrophic error termination. Hint: look for an infinite-recursion condition.

Residence on Data Libraries

Next, consider the case in which the callable procedure are to be stored in a *positional* data library called PROCLIB. A possible sequence of directives is

```
*SET PLIB =2
*OPEN 2,PROCLIB
*ADD STOOGES
```

The SET PLIB directive specifies 2 as the LDI of the procedure library (there's nothing magic about 2, any integer in the range 1 through 30 would do equally well). This LDI is linked to PROCLIB through an OPEN directive. If no error messages occur after the ADD, three Text Datasets named LARRY, CURLY and MOE are created. You may verify this by entering a TOC to print the Table of Contents of library PROCLIB. The callable procedure element contents may be printed by saying *LIST 2,LARRY, etc.

If PROCLIB is a *nominal* library the SET PLIB directive is a bit more complicated, since you have to specify a dataset that will hold the callable procedure elements as Text Groups. For example:

```
*SET PLIB =2,THREE.STOOGES
*OPEN 2,PROCLIB
*ADD STOOGES
```

The callable procedure elements then become three Text Groups in dataset THREE.STOOGES in library PROCLIB. If this dataset doesn't exist, it is created.

You are allowed to change PLIB specification at any time during the run. Or you could leave it fixed, say at 2, and connect different data libraries to the same index.

Rationale for Residence Choice

The original design of CLAMP assumed that all callable procedure elements were to reside on data libraries. As implementation progressed, it became evident that an alternative residence medium had to be made available to users.

One of the basic design goals of the NICE architecture, of which CLIP is a part, states that "a user should not be forced to know architectural aspects beyond those required for accomplishing stated needs". Now for many research projects that involve parameter studies, knowledge of command procedures is highly desirable, but knowledge of global database management techniques may be unnecessary. The possibility of storing procedures on ordinary files obviates the problem.

For production-level engineering analysis, use of hundreds of command procedures is expected to be commonplace. For this usage level, ordinary data files are patently inadequate (imagine trying to keep track of hundreds of tiny files that are constantly updated). Data libraries then provide a powerful and convenient "encapsulation" and archival mechanism.

Another (mild) disadvantage of ordinary-file residence is the fact that procedure names are restricted to legal file names, and the latter vary from computer to computer. On the other hand, dataset names and Text Group names are machine independent.

§6.3 ADVANCED TOPICS

Long Replacement Strings

If a replacement string specified in a CALL or (as default text) in a PROCEDURE directive contains internal blanks or lower case letter you want to preserve, you should enclose it in apostrophes. For example,

```
*call plotter (title = This is a plot title)
```

is not recommended; this is passed as THISISAPLOTTITLE, which is probably not what you had in mind. The right way is to say

```
*call plotter (title = 'This is a plot title')
```

The replacement text is now passed as

```
This is a plot title
```

i.e., apostrophes are stripped but the text remains intact. If you want to restore the surrounding apostrophes when TITLE is referenced in the body of the procedure, read on.

Protected Argument Substitution

On some applications (*e.g.*, graphics) you may want to "protect" the argument substitution text with apostrophes to force the item decoder to treat the text as one string, or to preserve blanks or lower case. For example, let us suppose that the replacement text for argument TITLE is `This is a plot title` (see above). If you write

```
[TITLE]
```

in a procedure line, upon substitution this becomes

```
This is a plot title
```

which ends up broken up into five items. What you want is

```
'This is a plot title'
```

But to get those surrounding apostrophes, you *cannot* use

```
'[TITLE]'
```

because argument replacement would be inhibited; recall (from Volume I) that apostrophes hide everything but hyphenation marks. The solution is to put a P ("protect me") qualifier after the argument name:

```
[TITLE/P]
```

No blanks are permitted before and after the slash. Note that this technique is analogous to that described for macrosymbols in §4.12, but only the P qualifier is valid for arguments.

Passing Macros and Quote Strings

Previous versions of CLIP had some restrictions regarding passing macrosymbols and quote strings as argument text (or as default arguments). The current version does not pose any particular restrictions and should work as you expect. Example:

```
*proc plot (title="Enter title:  "; scale="Enter scale:  ")
 set title = [title]
 set scale = [scale]
...
*end
*call plot (title = 'Actual title'; scale= <pi>^2) .  no prompts
*call plot (scale = (2+<pi>) ) .  now you will get one prompt
*call plot .  now you will get two prompts
```

Automatic Deletion of Compiled Procedure Files

If you specify the qualifier DELETE in a CALL directive, and the callable procedure element resides on an ordinary file, the file will be deleted when CLIP exits the procedure. This feature is sometimes of value when running "one shot procedures" to reduce directory file cluttering. For example, suppose that file PROBLEM.SOURCE (a VAX/VMS filename) contains the source

```
*proc analyze.inp
...
(problem   analysis   input)
...
*end
*call/delete analyze.inp
```

Executing an *ADD PROBLEM.SOURCE causes the procedure to be compiled, which produces the file ANALYZE.INP, which is then executed. On exiting from the procedure, the file ANALYZE.INP, is deleted, so you are not left with an extraneous file in the working directory. This strategy should be restricted, however, to straightforward input sequences such as this one.

THIS PAGE LEFT BLANK INTENTIONALLY.

7

Nonsequential Command Processing

§7.1 GENERAL

Command records are normally processed in strict forward sequence. Within a command procedure, however, it is possible to specify nonsequential processing with the help of branching and looping directives:

- *Unconditional branching* is provided by `JUMP` and `RETURN` directives.
- *Conditional branching* is provided by `IF` directives.
- *Looping* is provided by `DO` and `WHILE` directives.

These constructions work only *within a command procedure.* A reader familiar with programming language implementation will immediately understand why: forward or backward control transfers require that the target command be predefined, and its location identified. This implies a "compiling" process which is only available in CLIP as part of the command procedure capability described in §6.

The implementation of control transfers is made with the help of explicit or implicit labels. The following section describes the explicit kind.

REMARK 7.1

In pre-1984 versions of CLIP, `CYCLE` directive provided a primitive implementation (historically the first, back in 1976) of looping. That role is now assumed by the `DO` directive, which is closer in appearance to the FORTRAN `DO` loop and hence easier to understand.

§7.2 LABELS

A label line identifies a location within a procedure. This label can be referenced as a target in control transfer directives such as `JUMP` and `DO`.

A label line is constructed simply as

:Label

where *Label* is the label name. The name is a character string of arbitrary length, but the first eight characters must be unique. The name is prefixed by a colon. The first character of user-defined labels must be a letter. Internal labels generated by the procedure compiler begin with reserved characters.

A label may appear free field, but any text that follows on the same line is ignored. When control transfers to a label, the *next procedure line* is read.

REMARK 7.2

During procedure "compilation" label lines are physically eliminated. The associated information is collected in a table stored at the start of the callable procedure element.

REMARK 7.3

Explicit labels were more prominently used in the old versions of CLIP, which did not feature labelled structured constructs such as `IF-THEN-ELSE` and `WHILE-DO`.

EXAMPLE 7.1

Here is a label construction:

```
:ASK_USER
    SET VALUE = "Please enter correct value:"
    (more commands)
    *JUMP :ASK_USER
```

If control transfers to label ASK_USER, the SET VALUE command is read next.

§7.3 LOOPING

Looping can be specified through the DO and WHILE directives. The first one mimics the FORTRAN DO statement and may involve an explicit label. The second form is block structured and does not use an explicit label. Both forms test at the top.

The DO Loop

The DO directive purportedly resembles the FORTRAN DO statement. Cycling is controlled by a *local integer macrosymbol* that is created and updated as the loop is executed. Use of an explicit label to mark the DO range is optional. The form with an explicit label is

```
*DO :Label  Macro_name = i1,i2,[i3]

    (body)

:Label
```

where *Macro_name* is the name of a local macrosymbol, and i_1, i_2 and i_3 denote integers or integer expressions. If i_3 is omitted, the value 1 is assumed. The maximum level to which DO loops may be nested is 5.

The label may be omitted, in which case the loop body must be terminated by an ENDDO directive:

```
*DO  Macro_name = i1,i2,[i3]

    (body)

*ENDDO
```

In this case the procedure compiler generates appropriate internal labels.

Here's a specific example:

```
*DO :PRINLOOP $N = 1,10
  PRINT REC 1,55,VEL.<$N>
:PRINLOOP
```

The example loop is controlled by macrosymbol $N. This is a *local* macrosymbol because its name begins with a dollar sign; it is invisible outside the procedure. The body of the loop, which consists here of a single command, is executed ten times. In the first pass $N will have a value of 1, the second time the value will be 2, and so on. This value may be *materialized* within the body of the loop as illustrated above.

The WHILE DO Block

A WHILE DO block tests cycling on a logical rather than arithmetic expression. No explicit labels are permitted. The general form is

```
*WHILE logical_expression /DO

    (body)

*ENDWHILE
```

where both the WHILE and ENDWHILE directives must be on a line by themselves. Testing occurs at the top, *i.e.*, at the WHILE directive. If the logical expression is false, control passes to the command that follows the ENDWHILE terminator. Otherwise the body of the loop is traversed and control jumps back to the WHILE line for another test.

This form is converted to logical IFs and unconditional jumps during the procedure compilation. Two internal labels are generated in the process.

§7.4 CONDITIONAL BRANCHING

Conditional branching is provided by the IF directive. Previous CLIP versions offered only an "arithmetic IF" type of directive. The present version offers a "logical IF" with an explicit label and a block-structured IF-THEN-ELSE construction that avoids explicit labels.

The Labelled IF Directive

Conditional control transfer to a specific label may be specified by the labelled IF directive. This is analogous to the LOGICAL IF statement of FORTRAN.

The general format is

```
*IF logical_expression [:Label]
```

The *logical_expression* is in fact an integer that may take on two values: 1 for TRUE or 0 for FALSE. Control transfers to the line that follows *:Label* if the value is TRUE; otherwise it continues on to the next line. For example:

```
*IF 1 :THERE
```

control passes to the line that follows label THERE. In practice a bare statement like this makes little sense; more often than not the 1 or 0 is the product of a logical macro evaluation.

The BLOCK IF Construction

CLIP now offers a "structured" IF directive through which a construction similar to the IF-THEN-ELSE construction of FORTRAN 77 can be specified. Use of this construction eliminates the use of explicit labels and results in more readable procedures. The general form is

```
*IF logical_expression1 /THEN
    (commands)
*ELSEIF logical_expression2 /THEN
    (commands)
    ...
(additional ELSEIFs)
    ...
*ELSE
    (commands)
*ENDIF
```

The Block IF construction is initiated by an IF directive. The form of this directive is similar to that of the labelled IF directive described above, but it has a THEN qualifier instead of a label. If the logical expression given in the directive text evaluates to TRUE, the following commands are processed and then control transfers to the command that follow the ENDIF directive.

One or more ELSEIF directives (or none) may follow. The logic for processing the commands that follow an ELSEIF directive is similar to that followed in FORTRAN 77.

An ELSE directive may appear before the ENDIF directive. Commands between the ELSE and ENDIF are processed if all previous tests are not verified.

The ENDIF directive terminates the Block IF construction and is mandatory. On the other hand, the ELSEIF and ELSE parts may be omitted.

THIS PAGE LEFT BLANK INTENTIONALLY.

8

Global Data Manager Interface

§8.1 GAL DIRECTIVES

A set of directives available in the full CLIP version gives you the ability for interacting with the Global Data Manager GAL. These are collectively known as *GAL directives* and are listed in Table 8.1. The set of GAL directives forms the GAL Interface.

The interface allows the user of NICE Processors to perform global data management functions at the command level, without having to learn the calling sequences available at the FORTRAN level.

The present Section covers interface operations of a general variety, such as opening and closing data libraries, listing the Table of Contents, etc. The section does not discuss record-transfer operations which involve exchanging data between a GAL library and memory; these are covered under the Local Data Manager Interface.

Table 8.1. **GAL Directives**

Directive Name Id	*Database Level*	*Multiple Occurrences?*	*Applicable to Library Forms*
ADD TEXT_DATASET	*Dataset*	*No*	DAL, GAL80
ADD TEXT_GROUP	*Record*	*No*	GAL82
CLOSE	*Library*	*Yes*	All
COPY DATASET	*Dataset*	*Yes*	All
COPY RECORD	*Record*	*Yes*	GAL82
DELETE DATASET	*Dataset*	*Yes*	All
DELETE RECORD	*Record*	*Yes*	GAL82
ENABLE	*Dataset*	*Yes*	All
FIND LIBRARY	*Library*	*No*	All
FIND LIBRARIES	*Library*	*Yes*	All
FIND DATASET	*Dataset*	*No*	All
FIND DATASETS	*Dataset*	*Yes*	All
FIND RECORD	*Record*	*No*	DAL, GAL80
FIND RECORD_KEY	*Record*	*No*	GAL82
FLUB	*Library*	*Yes*	All
GET DATASET	*Dataset*	*No*	All
GET TEXT_DATASET	*Dataset*	*Yes*	DAL, GAL80
GET TEXT_GROUP	*Dataset*	*Yes*	GAL82

Table 8.1. **GAL Directives** (Concluded)

Directive Name Id	*Database Level*	*Multiple Occurrences?*	*Applicable to Library Forms*
LIST TEXT_DATASET	*Dataset*	*Yes*	DAL, GAL80
LIST TEXT_GROUP	*Record*	*Yes*	GAL82
LOCK	*Dataset*	*No*	GAL80,GAL82
OPEN	*Library*	*No*	All
PACK	*Library*	*No*	All
PUT DATASET	*Dataset*	*No*	All
PUT TEXT_DATASET	*Dataset*	*No*	DAL, GAL80
PUT TEXT_GROUP	*Record*	*No*	GAL82
PRINT RECORD	*Record*	*Yes*	All
PRINT RAT	*Record*	*Yes*	GAL82
PRINT TOC	*Dataset*	*Yes*	All
RENAME DATASET	*Dataset*	*Yes*	All
RENAME RECORD	*Record*	*Yes*	GAL82
TYPE TEXT_DATASET	*Dataset*	*Yes*	DAL, GAL80
TYPE TEXT_GROUP	*Record*	*Yes*	GAL82

§8.2 DBM TERMINOLOGY

Readers already familiar with GAL need not dwell on the following material, which is included for completeness.

Hierarchy

An GAL database consists of one or more *data libraries.* Data libraries reside on system files. These files are accessed by GAL through the I/O Manager DMGASP.

The information within a data library is hierarchically organized into *datasets* and *records.* A dataset is a named collection of records. Records contain the actual data used by applications programs.

Library Forms

GAL supports three data library formats, known as DAL, GAL80 and GAL82.

DAL. The DAL library form is used by the structural analysis programs SPAR and EAL, and the DALPRO utility processor. Most DAL datasets are formed by equal-size records, which must be stored contiguously. Records are identified by index.

GAL80. This generalizes the DAL format in the sense that datasets are formed by heterogeneous records, which must nonetheless be stored contiguously. Records are identified by index.

GAL82. Datasets are formed by collections of homogeneous records which may reside anywhere within the library file. Records are identified by name. A record name consists of a key and a cycle. Records of identical size and data type may be logically bound to form a Record Group.

Positional vs. Nominal

Datasets in DAL and GAL80 libraries are called *positional* because records are identified by their position within the dataset.

Dataset in GAL82 libraries are called *nominal* because records are identified by name instead of position.

Because DAL and GAL80 libraries can only hold positional datasets, the term *positional library* is commonly used to collectively refer to both forms. Similarly, GAL82 forms are called *nominal libraries.* Successors to GAL82 will also be of nominal type.

Identifying Database Entities

Directives that access the global database usually need to refer to entities such as libraries, datasets, records and even portions of records. CLIP has a consistent way of referring to such things through a set of syntax rules. These rules are explained in the following subsections. We start with the simplest rules and work our way up to the most complex ones.

§8.3 LIBRARIES

Accessing a Library

Before any library can be used by a Processor, it must be explicitly *opened.* For libraries that reside on permanent files, the open operation links the file name to a Logical Device Index (LDI), which is an integer in the range 1 to 16. For example:

```
*OPEN/R 14, ANALYSIS.LIB
```

is a directive that opens (in read-only mode) an existing library that resides in file ANALYSIS.LIB (a VAX file name) and connects it to Logical Device Index 14. All subsequent references to the library are through the LDI and *not* through the file name. This rule is appropriate because some libraries do not have names; see Remarks 8.2 and 8.3 below.

An open library is said to be *active.* You may get a list of active libraries through the SHOW LIBRARIES directive.

Once you are through with a library you may *close* it by issuing a CLOSE directive. A closed library that resides on a permanent file reverts to the inactive status and may not be accessed by the Processor. If the library resided on a scratch file, it disappears when it is closed.

REMARK 8.1

The LDI plays a role analogous to that of a logical unit in FORTRAN I/O, in the sense of being a pointer to a file. However, an LDI is *not* a logical unit.

REMARK 8.2

Libraries may reside on nameless files (in the sense of having no external name). This happens if you use a *scratch* file to temporarily store a library created with FORTRAN I/O, because the FORTRAN standard says that scratch files have no names. (Scratch files created with Block I/O may have names under some operating systems). But all libraries have an LDI.

REMARK 8.3

Most exotic are "core libraries." These reside on a pseudo-file created in blank common by the I/O Manager DMGASP. As in the previous case, core libraries have an LDI but no file name.

Library Specification

Some GAL directives operate at the library level and so they need only a library identifier. You have seen an example (the OPEN directive) earlier; here are two more:

```
*PACK 14
*CLOSE 14
```

The PACK directive compresses the library connected to Logical Device Index 14. The CLOSE directive performs a close-library operation as discussed above. For these directives it is sufficient to type the LDI as a positive integer.

A Special Convention

For all library directives but OPEN, CLIP will also let you type a 0 instead of the actual LDI to mean "the active library with the highest LDI". For example, suppose that you have two active libraries assigned to LDIs 3 and 8. Then

```
*PACK 0
```

means PACK 8. (If no libraries are open you will get an error diagnostic.) Omitting the LDI is the same as typing a zero.

§8.4 DATASETS

Identification

Datasets are identified by name or (once in the library) by sequence number. The naming conventions are discussed in detail in the GAL Reference Manual but in essence you identify the dataset by two names called *mainkey* and *key extension*, respectively, followed by up to three integers called *cycle numbers*. Any component except the mainkey may be omitted; an omitted extension is assumed blank whereas omitted cycles are assumed zero. Components are separated by periods. Examples:

```
MATRIX
STIFFNESS.MATRIX
DISPLACEMENT.VECTOR.1.3.101
ALUM-7075...3
```

The sequence number of a dataset is the ordinal of its occurrence in the data library. Once assigned, this number cannot change except as a result of a PACK operation.

The identification by name is more general and mnemonic. It also allows the use of masking characters and cycle-range specifications to do operations on several datasets related by name. Using names has the disadvantage that a search of the library Table of Contents (TOC) is required, which can be expensive if a library contains many datasets.

Writing Dataset Directives

To specify a dataset subject of a directive, you write the library LDI, then a comma, and then the dataset name or sequence number. Examples:

```
*TOC 4,FORWARD.STEP
*GET TEXT_DATASET OUTPUT.FIL = 6,35
```

In the TOC directive, the dataset is FORWARD.STEP, which resides in library 4. In the GET directive, the dataset is located at sequence number 35 of library 6.

Multiple Dataset Specification

Many GAL directives let you apply an operation to many datasets. These datasets may be identified by name masking, or by sequence range. Examples:

```
*TOC 4,FORWARD.S*
*TOC 4,45:59
```

The first example says that the TOC operation is to apply to all datasets in library 4 whose mainkey is FORWARD and whose key extension begins with S. The second example specifies the sequence number range 45 through 59 (inclusive).

Relative Sequence Specification

You may replace a sequence number by an explicit 0 to mean *the last sequence number.* Similarly, -1 means the penultimate dataset, and so on. Examples:

```
*ENABLE 7,0
*DELETE 12,-9:0
*COPY 0 = 1,42:0
```

Translation: enable the last dataset in library 7, delete the last 10 datasets in library 12, and copy datasets 42 through last in library 1 to the library with the highest LDI.

REMARK 8.4

The relative specification is experimental and may be removed from future CLIP versions if it is not found particularly useful.

§8.5 RECORDS

Some GAL directives let you do operations at the record level; for example printing record contents. Only specifications for GAL82 libraries, which use named records, are discussed here. Specification for indexed records in DAL or GAL80 libraries (which are becoming obsolescent) are summarized in Table 8.2.

Named Record Identification

An individual record is identified by a *key* and a number called a *cycle.* If the cycle is zero it may be omitted. If both name components are given they must be separated by a period. Examples:

```
FLUID_MASS
SIGMA-XX.45
```

Records with the same key and consecutive cycle numbers are collectively referred to as a Record Group. Each Group record has identical length and type. A Text Group is a special Record Group the records of which are card images. A Record Group subset may be specified by giving the key and a cycle range, as in

```
VELOCITIES.31:65
```

Writing Record Directives

A named record specification requires three pieces of data: owner library, owner dataset and record name. You begin by typing the first two pieces as in the case of a dataset specification; then you type another comma and then the record name. Example:

```
*PRINT RECORD 6,STIFFNESS.MATRIX,DETERMINANT
*PRINT RECORD 6,79,DETERMINANT
```

In the first line the PRINT RECORD directive applies to record DETERMINANT of dataset STIFFNESS.MATRIX in library 6. The second form uses a dataset sequence number and is equivalent to the first one if 79 happens to be the sequence number of STIFFNESS.MATRIX.

Multiple Dataset Specification

Some record-level directives allow you to specify multiple datasets linked by name masking or dataset sequence range. Examples:

```
*PRINT RECORD 6,ST*,DETERMINANT
*PRINT RECORD 6,23:79,DETERMINANT
```

Multiple Record Specification

Some record-level directives, notably PRINT RECORD, may apply to several records linked key masking or cycle specifications. Examples:

```
*PRINT RECORD 6,79,D*
*PRINT RECORD 6,79,*
*PRINT RECORD 6,79,XYZ.31:45
```

The first prints all records under STIFFNESS.MATRIX that begin with D; the second one prints all named records; the last one prints records XYZ.31 through XYZ.45 (inclusive) in Record Group XYZ.

Both multiple-dataset and multiple-record specifications may be combined, as in

```
*PRINT RECORD 6,*,*
```

which prints *all* records in nominal library 6.

§8.6 SUBRECORDS

A few DBM directives let you access parts of records. To specify portion of a record you enclose the data item range, written as $n_1 : n_2$ within squared brackets. Example:

```
*WGET RECEIVER = 5,13,TRANSFORM.6[3:12]
```

accesses items 3 through 12 of record TRANSFORM.6 of dataset 13 in library 5. (The WGET directive, by the way, belongs to the Local Data Manager Interface and thus is not covered here.)

And this is as complicated as it gets.

Table 8.2. **Database Entity Identifiers**

Entity	*Identifier*	*Explanation*
Library	*ldi*	*ldi* is the Logical Device Index. If *ldi=0*, the dataset with highest LDI is assumed except for the `OPEN` directive.
Single dataset	*ldi,Dataset_name*	Name of dataset
Single dataset	*ldi,dsn*	*dsn* is the dataset sequence number. If *dsn* is zero, the last dataset is assumed. If *dsn* is negative and equal to $-n$, the dataset located at n positions from the last one is assumed.
Several datasets	*ldi,Dataset_name*	*Dataset_name* contains masking specifications
Several datasets	*ldi,*dsn_1*:*dsn_2	Datasets with sequence number dsn_1 to dsn_2 inclusive. See above for treatment of zero or negative numbers.
Indexed record	*ldi,dsn,irec*	*irec*: record index
Indexed record	*ldi,Dataset_name,irec*	*irec*: record index
Named record	*ldi,dsn,Record_name*	*Record_name* is record name
Named record	*ldi,Dataset_name,Record_name*	*Record_name* is record name
Record portion	*ldi,dsn,Record name*$[n_1:n_2]$	Record items n_1 through n_2 inclusive. *dsn* may be replaced by a dataset name.

§8.7 MACROSYMBOLS AS INFORMATION CARRIERS

Certain DBM directives allow you to do database queries; the most important one being `FIND`. For applications like writing command procedures it is convenient to have a mechanism for storing the result of such queries as values that can be subsequently used in other

commands. This can be achieved with the help of macrosymbols that are created as a byproduct of the execution of the directive. Consider

```
*OPEN/NEW INPUT.LIB /LDI=INPUT
```

The OPEN directive does not specify the LDI of the library INPUT.LIB. Instead the LDI is picked by GAL; let's say it is 3. Integer macrosymbol INPUT is defined, and assigned the value 3. A subsequent directive or ordinary command can make use of this value by materializing the macrosymbol in the usual fashion. For example:

```
*REMARK INPUT.LIB created and assigned to LDI=<INPUT>
*TOC <INPUT> ; *CLOSE <INPUT>
```

THIS PAGE LEFT BLANK INTENTIONALLY.

9

Local Data Manager Interface

§9.1 GENERAL DESCRIPTION

The Workpool

CLIP contains an experimental (see below) interface to a *local data manager* called the *Workpool Manager*, or WM. The Workpool is an allocatable area of memory that can be used to hold dynamic data structures used by the Processor. The term *work* means that the Processor can access the data structures held in the Workpool to carry out processing functions.

The Workpool resides physically on a numeric common block. The block may be either blank common or block CCLWRK. The selection is governed by an MSC distribution key specified at compile time.

The Workpool area is subdivided into *workrecords*. The allocation is performed as a one-way stack.

Operational Status

The Workpool Manager and directive interface to it is presently on experimental status. Changes may occur without notice. It is expected, however, that the final form of the manager and interface will be fairly close to that presented here.

Workrecords

A *workrecord* is a contiguous area of the Workpool identified by name. A workrecord name has the same structure of a nominal record name: a record key of up to 12 characters optionally followed by a cycle number. The first character of the key must be a letter. If the cycle number is omitted, zero is assumed. The following are legal workrecord names:

```
ZETA  COORDINATES.56  TITLE.8
```

A workrecord has three basic attributes: a data type, its logical size, and its location within the workpool. These and other attributes are listed in Table 9.1.

There are two types of workrecords supported by the Workpool Manager:

1. A *backed* workrecord is linked to a nominal data library through backing information (see Table 9.1). A backed workrecord is created by an *open* operation, at which time an *old* or *new* status is specified.

2. An *unbacked* or *scratch* workrecord does not have backing information. An unbacked workrecord is created by an *allocate* operation. These workrecords are normally used to hold temporary data.

Workgroups

Workrecords with the same key may be grouped over a cycle range to form *workgroups.* All records of a workgroup have the same size and data type. A workgroup is identified by giving the cycle range after the key. Example:

```
ALPHA.1:20
```

ALPHA is a 20-record workgroup.

The concept of workgroup is similar to that of Record Group in a nominal (GAL82) library, but there is an important difference: no cycle gaps are permitted.

§9.2 WORKPOOL OPERATIONS

The main operations that can be performed on workrecords are listed next.

Allocate

The allocate operation sets aside space in the Workpool to store a workrecord or workgroup. Protection keywords are installed before and after the allocation. The record key is entered in the macrosymbol table, and its attributes become the value of the macrosymbol. For an unbacked workrecord, the allocate and open operation are identical. For a backed workrecord, the allocate operation is a subset of the open operation.

Open

The open operation applies to *backed* records and combines allocation with backup and/or initialization. The workrecord or workgroup is given space in the Workpool through an allocate operation. If the status is old, the contents are read from the backing library and the modified flag is cleared. If the status is new, the space is initialized (to zero if data type is numeric, to blank if character) and the modified flag is set.

Change Size

The size of a workrecord may be expanded or reduced at any time. A side effect of the size change is that the Workpool locations of other workrecords are changed.

Mark as Modified

A backed workrecord or workgroup may be flagged as modified so as to force its contents to be written to the backing data library on a flush or close operation. An opened-new operation or a change-size operation forces the modification flag to be set.

Flush

A backed modified workrecord or workgroup marked as modified is written to the backing data library, and the modified flag cleared. The allocation is not altered.

Deallocate

The storage allocation of a workrecord or workgroup is released to the Workpool. Locations of other workrecords may change as a result of this operation.

Close

For backed workrecords, the close operation combines flushing and deallocation. For unbacked (scratch) workrecords, close and deallocate are equivalent operations.

Table 9.1. **Workrecord Attributes**

Attribute	*Explanation*
Name	A record key/cycle pair separated by a dot; a zero cycle may be omitted.
Key	A string of 1 to 16 characters; the first one must be a letter.
Cycle	An integer in the range 0 to 999999.
Logical size	Record length in logical units (also called items)
Physical size	Number of machine words needed to store the logical size.
Type	A one letter type identifier. See Table 63.1.
Pool location	The word address of the first record item in the Workpool.
Modification flag	A flag that indicates whether a backed record has been modified since the last backup operation.
Backing library	Database information (device index, dataset, record) that links a workrecord or workgroup to the database for backup purposes.

C-2

THIS PAGE LEFT BLANK INTENTIONALLY.

a trivial MCP that doesn't do much except looping:

```
*procedure super (nruns=2)
*do $n = 1,[nruns]
  *rem I am inside P1.  The value of $n is <$n>
  *run P2
  *rem I am inside P2.  The value of $n is <$n>
  *stop
*enddo
*end
```

For demonstration purposes P1 and P2 may be "do nothing" 4-line processors such as

```
      program P1
1000    call CLREAD (' P1> ', ' ')
        go to 1000
      end
```

and

```
      program P2
1000    call CLREAD (' P2> ', ' ')
        go to 1000
      end
```

Compile P1 and P2 and link to the NICE library to make executables P1.EXE and P2.EXE. Now run P1 and compile the source file of procedure SUPER by using an *ADD command. Upon typing

```
*call super
```

you should see SuperCLIP in action:

```
I am inside P1.  The value of $n is 1
I am inside P2.  The value of $n is 1
I am inside P1.  The value of $n is 2
I am inside P2.  The value of $n is 2
```

State Preservation

The key point of the example is that the value of macrosymbol $N is known to *both* P1 and P2, just as if they were the same program. Since P1 and P2 are independent programs, a save/restore process has to take place transparently to the user.

It is this *state preservation* that makes all the difference. For example, if you manually stop P1 and start P2, the latter has no way of knowing that macrosymbol $N exists and what its value is, unless you explicitly insert a DEFINE. Similarly, P2 wouldn't know what the value of the procedure argument NRUNS is, and there is no simple way to specify that.

In production-level networks there may be many Processors working together, and hundreds of shared quantities: macrosymbols, procedure arguments, labels, the command source stack, etc. Manually saving and restoring this mass of data makes little sense. If there is a way, let the machine do it.

Machine Dependency

Unfortunately the idea of having a program start up another program is foreign to FORTRAN (as well as to most programming languages). The feasibility of using SuperCLIP is directly related to the operating system used.

1. *Unfeasible.* Either the concept is foreign to the O/S, or the necessary hooks are not provided. Example: CRAY/COS.
2. *Barely Feasible.* A simple mechanism is provided that works although it lacks embellishments such as parameter passing and synchronization. Example: the VAX/VMS implementation described in §10.3, which uses `LIB$RUN_PROCESS`.
3. *Feasible with Enhancements.* This includes systems like Unix in which concurrent process execution and synchronization were basic elements of the design.

REMARK 10.1

Some systems offer more than one way of implementing SuperCLIP. For example, on VAX/VMS you can have a process start another process, or you can have a process "spawn" subordinate subprocesses. The first implementation (1981) of SuperCLIP on VAX/VMS in fact made use of the "spawn" technique. This was changed in the present implementation because spawning may be disallowed (as system-generation option) on some VAX systems, whereas the first form is universal. The subprocess technique has the advantage that one can return to the parent process at exactly the same point at which the subprocess was spawned.

§10.3 VAX/VMS IMPLEMENTATION

Implementation of the `RUN` directive

When you enter a `RUN` directive, CLIP enters the SuperCLIP subsystem, which performs the following steps:

1. *Push PNS.* The name of the Processor specified in the `RUN` directive (or, more exactly, the name of its executable file) is pushed onto a data structure known as the Process Name Stack (PNS).

2. *State Save.* SuperCLIP opens a new, PRU addressable, Block I/O file by calling the I/O Manager DMGASP. The name of the file is `ZZZZZZZ.DAT` on VMS and `ZZZZZZZ` on UNIX. All data structures that govern the state of CLIP are block copied to that file. These structures include the Decoded Item Table, Macrosymbol Table, Command Source Stack, Process Name Stack, control characters, logical unit table, and list of active data libraries. All open libraries are closed.

3. *Process Switch.* Call the VAX/VMS operating system function LIB$RUN_PROGRAM or call the UNIX system function execlp to stop the current process and start the target process.

4. *CLIP Booting.* The target Processor starts. On first entry to CLIP, a "booting" routine is called. One key duty of this routine is to ask: is this Processor run the result of SuperCLIP? Since VAX/VMS does not tell, an indirect query procedure is followed. The existence of the state save file in the current directory (ZZZZZZZ.DAT or ZZZZZZZ) is checked. If the file exists, SuperCLIP is called.

5. *State Restore.* The state save file (ZZZZZZZ.DAT or ZZZZZZZ) is read to restore the CLIP data structures. Non-scratch libraries that were open in the parent processor are re-opened. The Command Source Stack is also reconstructed so it has the same array of open files, and script files are read forward to restore them to the original position. The state save file is closed with delete option, and so it disappears.

REMARK 10.2

Many entities foreign to CLIP are not restored. These include: the memory-resident data other than the data structures listed in step 3, scratch data libraries, logical devices that are not data libraries, and FORTRAN files not part of the command source stack. Basically the automatic restore is concerned with CLIP operation.

REMARK 10.3

There may be a lag between the time a Processor starts and the first entry to CLIP. Since the state restore takes place only upon the latter, some unusual things may happen if the lag sets up parameters that may be overriden by the state restore. It is, therefore, a good idea to make Processors call CLIP as soon as possible in the main program if they may become part of the network. Just calling CLPUT with an empty message would suffice.

REMARK 10.4

If the RUN directive fails because the target Processor name is incorrect, the state save file will be left sitting in your directory. *Please delete such a file if you see one.* If you innocently start a Processor with a $RUN and the Processor finds that file there, it will think it has been started using SuperCLIP and you may see deviant behavior. If you run across such behavior, abort the run and check the current directory to see if the state save file is present.

Implementation of the STOP directive

A STOP directive is implemented very much like a RUN directive. It is sufficient to note the differences:

1. *Pop PNS.* The name of the parent Processor is extracted from Process Name Stack (PNS), which is popped. If the stack is empty, take a normal run termination.

2–5 Same as for the RUN directive.

THIS PAGE LEFT BLANK INTENTIONALLY.

11

Directive Classification

§11.1 CLASSIFICATION

All directives are alphabetically described in Sections 12 through 74. The present Section classifies directives according to their availability in the present version of CLIP.

1. *Core Directives* are those implemented as part of kernel capabilities and consequently are not dependent on the availability of CLIP subsystems.
2. *Subsystem Directives* are part of non-kernel modules such as Macrosymbol and Command Procedure. Availability of these directives depends on which subsystems are extracted from the Master Source Code. For example, the SuperCLIP subsystem is only implemented on VAX/VMS and consequently the `RUN` and `STOP` directives are not available under other operating systems.

§11.2 CORE DIRECTIVES

The following directives listed in Table 11.1 are part of the CLIP "kernel".

Table 11.1 **Core Directives**

Name	*Modifier or Subclass*	*Function*	*Status*
ABORT		Triggers an abnormal run termination	Operational
ADD	*File*	Redirects input to script file	Operational
DUMP		Print the contents of any DMGASP file	Operational
ENDLOG		Terminates command logging	Operational
EOF		Terminates command source	Operational
EOL		Inserts end-of-line in dataline collector	Operational
FCLOSE		Closes FORTRAN unit	Operational
FOPEN		Open card-image FORTRAN unit	Operational
GENERATE		Generates next command(s) by incrementation	Operational
HELP		Lists topic-qualified segments of NICE help file	Operational
LIST	*File*	Lists card-image file on print unit	Operational
LOG		Initiates command transcription to log file	Operational
REMARK		Print remark line	Operational

Table 11.1 **Core Directives** (Concluded)

Name	*Modifier or Subclass*	*Function*	*Status*
SET	CHARACTER	Changes a volatile control character	Operational
SET	CPU_TIME	Sets internal CPU time stopwatch	Operational
SET	ECHO	Sets dataline echo options	Operational
SET	HELP	Sets current help file	Operational
SET	MODE	Sets command processing modes	Operational
SET	RUN	Sets run execution environment	Operational
SET	TERMINAL	Sets terminal environment	Not implemented
SET	UNIT	Sets logical unit number	Operational
SET	VIDEO	Sets CRT-display control parameters	Not implemented
SET	WIDTH	Sets line input or print width	Operational
SET	WINDOW	Sets CRT windowing parameters	Not implemented
SHOW	CHARACTERS	Shows volatile control characters	Operational
SHOW	CSS	Shows Command Source Stack	Operational
SHOW	CPU_TIME	Show elapsed CPU time	Operational
SHOW	DEC	Shows Decoded Item Table	Operational
SHOW	ECHO	Shows dataline echo options	Not implemented
SHOW	HELP	Shows current help file	Operational
SHOW	MODES	Shows command processing modes	Operational
SHOW	RUN	Shows run execution environment	Operational
SHOW	TERMINAL	Shows terminal environment	Not implemented
SHOW	UNITS	Shows logical units	Operational
SHOW	VIDEO	Shows CRT-display control parameters	Not implemented
SHOW	WIDTHS	Shows line input and print widths	Operational
SHOW	WINDOWS	Shows CRT windowing parameters	Not implemented
TYPE	*File*	Lists file on terminal	Operational

§11.3 MACROSYMBOL DIRECTIVES

The following directives listed in Table 11.2 are available as part of the Macrosymbol subsystem described in §4.

Table 11.2 **Macrosymbol Directives**

Name	*Modifier or Subclass*	*Function*	*Status*
ALIAS		Defines an abbreviation for a textstring	Not implemented
DEFINE		Defines or redefines a macrosymbol, or macrosymbol array	Operational
GAL2MAC		Defines or redefines a macrosymbol or macrosymbol array with values obtained from a GAL dataset	Operational
MAC2GAL		Writes values of a macrosymbol or macrosymbol array into a GAL dataset	Operational
SHOW	MACROS	Shows defined macrosymbols	Operational
UNDEFINE		Deletes macrosymbol(s)	Operational

Note: GAL2MAC and MAC2GAL also require the Global Data Manager (GAL) Interface subsystem.

§11.4 COMMAND PROCEDURE DIRECTIVES

The following directives listed in Table 11.3 are available as part of the Command Procedure subsystem described in §5. Directives for nonsequential command processing, which can only be used in procedures, were described in §6.

Table 11.3 **Command Procedure Directives**

Name	*Modifier or Subclass*	*Function*	*Status*
CALL		Redirects input to a callable procedure element	Operational
ELSE		Introduces "else" subblock in IF-THEN-ELSE block	Operational
ELSEIF		Introduces "else if" subblock in IF-THEN-ELSE block	Operational
END		Terminates definition of command procedure	Operational
ENDDO		Terminates a label-less DO block	Operational
ENDIF		Terminates an IF-THEN-ELSE block.	Operational
ENDWHILE		Terminates a WHILE-DO block.	Operational
IF	*Labeled*	Logically tests and transfers to label	Operational
IF	*Label-less*	Introduces an IF-THEN-ELSE block	Operational
JUMP		Transfers control to specified label	Operational

Table 11.3 **Command Procedure Directives** (Concluded)

Name	*Modifier or Subclass*	*Function*	*Status*
PROCEDURE		Initiates definition of command procedure	Operational
RETURN		Forces exit from command procedure	Operational
SET	ARGUMENT	Sets procedure argument replacement text	Not implemented
SET	PLIB	Sets procedure library for residence of callable procedure elements	Operational
SHOW	ARGUMENT	Show procedure argument replacement text	Operational
SHOW	PLIB	Shows procedure library for residence of callable procedure elements	Operational
WHILE		Introduces a WHILE-DO block	Operational

§11.5 GAL DIRECTIVES

The following directives listed in Table 11.4 are available as part of the Global Data Manager (GAL) Interface subsystem described in §7.

Table 11.4 **GAL Directives**

Name	*Modifier or Subclass*	*Function*	*Status*
ADD	TEXT_DATASET	Redirects input to Text Dataset script	Operational
ADD	TEXT_GROUP	Redirects input to Text Group script	Operational
CLOSE		Closes data library(ies)	Operational
COPY	DATASET	Copies and optionally renames dataset(s) from a library to another	Operational
COPY	RECORD	Copies and optionally record(s) from a nominal dataset to another	Experimental
DELETE	DATASET	Delete dataset(s)	Operational
DELETE	RECORD	Delete named record(s)	Experimental
ENABLE		Enable dataset(s)	Operational
FIND	DATASET	Returns information on individual dataset	Experimental
FIND	DATASETS	Returns information on several datasets	Experimental
FIND	LIBRARY	Returns information on library	Experimental
FIND	LIBRARIES	Returns information on all libraries	Experimental
FIND	RECORD	Returns information on indexed record	Experimental
FIND	RECORD_KEY	Returns information on named record key	Experimental
FLUB		Flushes buffers of data library(ies)	Operational

Notes: FIND also requires Macrosymbol facility.

Table 11.4 **GAL Directives** (Concluded)

Name	*Modifier or Subclass*	*Function*	*Status*
GET	TEXT_DATASET	Extracts Text Dataset to file	Operational
GET	TEXT_GROUP	Extracts Text Group to file	Operational
LIST	TEXT_DATASET	Lists Text Dataset on print file	Operational
LIST	TEXT_GROUP	Lists Text Group on print file	Operational
LOAD		Internalizes record(s) from text file	Operational
LOCK		Sets dataset lock code	Not implemented
MAC2GAL		Writes a macrosymbol value or values to a nominal GAL dataset	Operational
OPEN		Opens data library	Operational
PRINT	DATASET	Prints all dataset records	Not implemented
PRINT	RAT	Prints Record Access Table of dataset	Operational
PRINT	RECORD	Prints record contents	Operational
PRINT	TOC	Prints Dataset Table of library	Operational
PUT	TEXT_DATASET	Inserts Text Dataset from card-image file	Operational
PUT	TEXT_GROUP	Inserts Text Group from card-image file	Operational
RENAME	DATASET	Renames dataset(s)	Operational
RENAME	RECORD	Renames record(s)	Experimental
SET	ERR	Sets Error processing options	Not implemented
SHOW	ETS	Shows Error Trace Stack	Experimental
TYPE	TEXT_DATASET	Lists Text Dataset on terminal	Operational
TYPE	TEXT_GROUP	Lists Text Group on terminal	Operational
UNLOAD		Externalizes record(s) to text file	Not implemented

Notes: OPEN requires Macrosymbol facility for qualifier LDI.

§11.6 SUPERCLIP DIRECTIVES

The following directives listed in Table 11.5 are available as part of the SuperCLIP subsystem described in §9.

Table 11.5 **SuperCLIP Directives**

Name	*Modifier or Subclass*	*Function*	*Status*
RUN		Starts execution of another Processor	Operational
STOP		Stops execution of Processor, restarts parent	Operational

§11.7 WORKPOOL DIRECTIVES

A group of directives that begin with the letter W is available as command interface to the local data manager called Workpool Manager. The following directives listed in Table 11.6 are available.

Table 11.6 **Workpool Directives**

Name	*Function*	*Status*
WALLOCATE	Allocate scratch workrecord	Operational
WCHANGE	Change logical size of workrecord	Operational
WDEALLOCATE	Reclaim storage used by workrecord	Operational
WDEFINE	Define macrosymbol from workrecord values	Operational
WDIMENSION	Set the first matrix dimension of workrecord	Operational
WGET	Read database record(s) into workrecord	Operational
WMAP	Give allocation map of the Workpool	Operational
WPRINT	Print contents of workrecord	Operational
WPUT	Write workrecord to nominal library	Operational
WSET	Set workrecord items to specified values	Operational

12
ABORT

§12.1 THE ABORT DIRECTIVE

Purpose

Trigger an abnormal run termination.

Format

```
*ABORT [/BATCH]
```

Word Qualifiers

BATCH	Abort if process is running in batch mode, or else do nothing.

Description

When an ABORT directive is detected, CLIP forces an immediate run abort condition. The way in which this condition is triggered depends on the host computer and operating system, but the end result is the same.

Operational Restrictions

Post-mortem actions will depend on the operating system.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP subsystem(s) required:

None.

Status

Operational.

REMARK 12.1

If the interrupted process was initiated by another (parent) Processor via SuperClip, control does not return to the parent Processor.

REMARK 12.2

Run abort is useful when running on batch mode under certain operating systems which produce post-mortem information such as core dumps. You will rarely need it for interactive work.

EXAMPLE 12.1

```
IF (BAD_NEWS)  CALL CLPUT ('*ABORT')
```

THIS PAGE LEFT BLANK INTENTIONALLY.

13
ADD

§13.1 THE ADD-FILE DIRECTIVE

Purpose

Redirects command input to a script file.

Format

```
*ADD Filename [/CLOSE] [/END] [/READ]
```

Synonyms

INCLUDE and ADD are equivalent directive verbs.

Required Parameters

Filename — The name of the card-image file from which CLIP will begin reading data lines.

Word Qualifiers

CLOSE — Close previous source unless at root level. On ADDing *Filename*, the command source stack stays at same level. This has specialized use on Cray/COS (and in general on any operating system that does not allow read-only multiple connection of the same file to several logical units).

END — If this qualifier appears, CLIP emits the one-word command

END

when the end of the ADDed file is sensed (or an EOF directive detected). This is an ordinary command and is therefore received by the processor, which may take action accordingly.

READ — Forces file read-through when the directive is submitted as a message using CLPUT. Otherwise is has no effect.

Description

When an "ADD file" directive is detected, CLIP opens the script file (in read-only mode if this is allowed by the operating system) and connects it to a logical unit in the range 35-40, as explained in §4 of Volume I. Selection of the logical unit is automatic. Processing of the file is strictly sequential. When the end-of-file is sensed (or an EOF directive detected) command input reverts to the previous source in the Command Source Stack.

The **ADD**ed file may contain **ADD** or **CALL** directives as long as the command source stack capacity is not exceeded (see Remarks 13.1 and 13.2 below).

Dataline Restrictions

This directive must be on a dataline by itself. If the script contains procedure definition(s), line length *must not exceed 80 characters.*

Operational Restrictions

You must have permission to access the file. On systems such as VAX/VMS in which a file may be opened in read-only mode, you may **ADD** files that belong to other users if they give read permission. On systems that do not offer this feature, such as CRAY/COS, the file should be under your ownership.

CLIP subsystem(s) required:

None.

Status

Operational.

REMARK 13.1

The **ADD**ed file may contain any command or directive appropriate to a non-procedural source. In particular, it may contain directives that again redirect the command source input, such as **CALL**s or other **ADD**s. (See next Remark as to **ADD**ing the same file several times.)

REMARK 13.2

If the operating system allows a file to be opened with a read-only option, the same file may be added at different command source stack levels. This means that the same disk file is connected to different FORTRAN logical units. This happens, for instance, if you do "**ADD** recursion". But some operating systems do not allow multiple connection if the read-only option is not provided. In this case the file should appear only once and recursive **ADD**ing is ruled out.

REMARK 13.3

If the file is not found, or cannot be accessed for some reason (*e.g.*, file access denied by owner), the message

Can't open *Filename*

is issued but the run continues normally.

REMARK 13.4

Any text following this directive on the same dataline is ignored.

REMARK 13.5

Univac users may recognize the influence of the @ADD control statement.

EXAMPLE 13.1

```
*ADD JAD:SHOCK.DAT
```

Subsequent commands will be read from file JAD:SHOCK.DAT (a VAX filename).

EXAMPLE 13.2

```
call CLPUT ('*ADD/R JAD:SHOCK.DAT ')
```

File JAD:SHOCK.DAT is ADDed by submitting a message using CLPUT. The READ qualifier means that CLIP will open the file and begin to read from it before returning from CLPUT.

§13.2 THE ADD-TEXT-DATASET DIRECTIVE

Purpose

Redirects command input to a Text Dataset script resident on a positional library.

Format

*ADD *Text_dataset_id* [/END] [/READ]

Synonyms

INCLUDE and ADD are equivalent directive verbs.

Required Parameters

Text_dataset_id — Identifies the Text Dataset to be ADDed. The identification involves two items. The first is the Logical Device Index (LDI) of the source data library, which must be open at the time the ADD is issued. The second item identifies the dataset by name or by sequence number:

ldi, *Dataset_name*

ldi, *dsn*

If the first form is used, *Dataset_name* should not have masking characters or cycle range specifications.

Word Qualifiers

END — Same as in the ADD-file case.

READ — Same as in the ADD-file case.

Description

When an "ADD Text Dataset" directive is detected, CLIP queries the NICE-GAL data manager as to the existence of the dataset. If the dataset exists and is of text type, a channel to it is set and an internal card-image buffer primed. Subsequent dataline requests are taken from the buffer. When the end-of-dataset is sensed (or an EOF directive detected)

command input reverts to the previous source in the Command Source Stack, but the data library is *not* closed.

The ADDed dataset may contain ADD or CALL directives as long as the command source stack capacity is not exceeded.

Dataline Restrictions

This directive must be in a dataline by itself. If the script contains procedure definition(s), line length *must not exceed 80 characters.*

Operational Restrictions

The library must be open when you issue the ADD directive. The status and contents of the source library should not be altered while the ADD is in progress; otherwise very strange things may happen.

CLIP subsystem(s) required:

NICE-DMS Interface.

Status

Operational.

REMARK 13.6

If the LDI is inactive or the dataset is not found, an appropriate error message is given but the run continues normally.

REMARK 13.7

Any text following this directive on the same dataline is ignored.

EXAMPLE 13.3

```
*OPEN 3, HOME:SCRIPT.LIB /R
*ADD  3, OWN_WEIGHT.LOADS
```

Positional data library HOME:SCRIPT.LIB is opened in read-only mode and connected to Logical Device Index 3. CLIP is then told to get command input from Text Dataset OWN_WEIGHT.LOADS.

§13.3 THE ADD-TEXT-GROUP DIRECTIVE

Purpose

Redirects command input to a Text Group script resident on a nominal library.

Format

*ADD *Text_group_id* [/END] [/READ]

Synonyms

INCLUDE and ADD are equivalent directive verbs.

Required Parameters

Text_group_id — Identifies the Text Group to be ADDed. The identification involves three items. The first is the Logical Device Index (LDI) of the source data library, which must be open at the time the ADD is issued. The third item is the Text Group key. The second item identifies the owner dataset by name or by sequence number:

ldi, Dataset_name, Key

ldi, dsn, Key

If the first form is used, *Dataset_name* should not have masking characters or cycle range specifications.

Word Qualifiers

END — Same as in the ADD-file case.

READ — Same as in the ADD-file case.

Description

When an "ADD Text Group" directive is detected, CLIP queries the NICE-GAL data manager as to the existence of the dataset and Text Group. If both entities exist, a channel to it is set and an internal card-image buffer primed. Subsequent dataline requests are taken from the buffer. When the end-of-Text-Group is sensed (or an EOF directive detected) command input reverts to the previous source in the Command Source Stack, but the data library is *not* closed.

Dataline Restrictions

This directive must be in a dataline by itself. If the script contains procedure definition(s), line length *must not exceed 80 characters.*

Operational Restrictions

The library must be open when you issue the ADD directive. The status and contents of the source library should not be altered while the ADD is in progress; otherwise very strange things may happen.

CLIP subsystem(s) required:

NICE-DMS Interface.

Status

Operational.

REMARK 13.8

If the LDI is inactive, or the dataset is not found, or the Text Group is not found, an appropriate error message is given but the run continues normally.

REMARK 13.9

Any text following this directive on the same dataline is ignored.

EXAMPLE 13.4

```
*OPEN 18, HOME:SCRIPT.LIB /R
*ADD  18, OWN_WEIGHT.CASE.12, LOADS
```

Nominal data library HOME:SCRIPT.LIB is opened in read-only mode and connected to Logical Device Index 18. The ADD directive then tells CLIP to get command input from Text Group LOADS that belongs to dataset OWN_WEIGHT.CASE.12.

§13.4 THE ADD-UNIT DIRECTIVE

Purpose

Redirects command input to a preconnected unit, with the users terminal as special case.

Format

*ADD *unit* [/TERMINAL]

Directive Parameters:

unit — Number of a "preconnected" logical unit to which input will be redirected. CLIP *does not* open this unit as in the case of the "ADD file"; but begins immediately to read from it.

A value of zero denotes the users terminal.

Word qualifier

TERMINAL — Not presently implemented.

Description

When an ADD directive of this form is detected, CLIP does not try to open a file as in the case of the ADD file directive. Command input is redirected to the logical unit *unit*, which is assumed to be preconnected. The unit is not rewound. On detecting an end-of-file condition, command input reverts to the previous source but the unit is *not* closed.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP subsystem(s) required:

None.

Status

Operational for **ADD 0**; other forms are less reliable.

REMARK 13.10

This directive is useful in advanced "spectator mode" sequences. To give an example, suppose that the master processor is running under a script or procedure while the user monitors the run. Suddenly the program encounters a situation which requires user intervention. It then sends an ***ADD 0** message through the mail facility, perhaps preceded by an Attention!! message. The user sees the prompt come up on the terminal and enters appropriate commands. To revert to the previous source, the user types a **@** on column 1, which acts as a "terminal end of file", and CLIP will then pop the Command Source Stack and revert to the previous input source.

EXAMPLE 13.5

A processor **SOLVE** is to be run on the VAX in interactive "spectator" mode. A predefined system procedure **RUNSOLVE.COM** contains

```
$assign "_txc4:" for018
$run solver
*add solver.dat
```

14

ALIAS

§14.1 THE ALIAS DIRECTIVE

Purpose

Define abbreviation for a text string.

Format

*ALIAS [/DELIMITER=*Delim*] *Abbreviation* = *Definition_text*

Status

Not implemented.

§14.2 GENERAL DESCRIPTION

An *abbreviation* is a symbol that has a name and a value. The name may be up to 16 characters long and consist of letters, digits, underscores and dollar signs, but the first character must be a letter. The value is a character string that contains from 0 through 255 characters.

When the abbreviation name is found as the *first item* of a command, it is replaced by its value. Replacement takes place before any other line-processing action (e.g., macrosymbol replacement or argument substitution) occurs. If the name is not the first item, the substitution does not take place.

An abbreviation is similar to a protected-string CLIP macrosymbol, except that it does not have to be identified as a macrosymbol by being enclosed in special characters. An abbreviation, on the other hand, is more restricted than an ordinary macrosymbol in several respects; for example, it must appear as the first item, cannot be nested, may not stand for a numeric expression, and cannot have arguments.

To illustrate the differences and similarities between abbreviations and macrosymbols, consider the following directive

```
*ABB LF = 'LIST INPUT.FIL'
```

This establishes LF as an abbreviation for LIST INPUT.FIL. One can now type, for example

```
LF
```

and the effect is the same as entering

```
LIST INPUT.FIL
```

The same effect can be achieved with a macrosymbol by defining

```
*DEF/A LF = 'LIST INPUT.FIL'
```

and one then types <LF>. The macrosymbol is more flexible in that one can put it anywhere; for example

```
*<LF>
```

yields the directive `*LIST INPUT.FIL`. But it involves extra keystrokes when that flexibility is not warranted.

THIS PAGE LEFT BLANK INTENTIONALLY.

15
CALL

§15.1 THE CALL DIRECTIVE

Purpose

Redirects input to a callable procedure element.

Format

```
*CALL [/DELETE] [/READ] Procedure_name [Argument_list]
```

Required Parameters

Procedure_name — The name of the callable procedure element. For rules governing this name, see PROCEDURE section.

Optional Parameters

Argument_list — An optional list of argument specifications may follow the procedure name. The format is:

(*Arg1* = *Text1* ; *Arg2* = *Text2* ; ...)

where *Arg1*, *Arg2* ... are formal argument names, and *Text1*, *Text2*, ... are the corresponding replacement textstrings.

Argument specifications are separated by semicolons.

The opening parenthesis *must be preceded by a blank.*

Arguments may appear in any order. If a formal argument is omitted, occurrences of that name in the procedure body are erased (or, more technically, replaced by a null string) unless default text has been specified in the procedure declaration as discussed in the PROCEDURE directive section.

Word Qualifiers

DELETE — On exit from procedure, delete file that contains the compiled form of it. Only applicable if this form resides on an ordinary file, and the running processor has delete privileges on that file.

READ — Forces procedure read-through when the directive is submitted as a message using CLPUT. Otherwise is has no effect.

Description

When you issue this directive, CLIP will begin reading data lines from the callable procedure element, following a file-open if appropriate. You may visualize the effect of the CALL by imagining that the body of the callable procedure, with formal arguments replaced as indicated, replaces the directive line. (Of course the visualization would be strained if the body contains other CALLs.)

Input reverts to the previous source when the end of procedure is reached, or a **RETURN** or **EOF** directive encountered. If the **DELETE** qualifier is given and the compiled procedure resides on an ordinary file, the file is closed, then re-opened with delete privileges (if possible) and closed with delete option.

Dataline Restrictions

This directive must be in a dataline by itself, although it may be continued over many lines. Qualifiers, if any, must appear between the **CALL** verb and the procedure name.

Operational Restrictions

If the callable procedure element resides on a file, you must have appropriate access permissions to read that file. If the **DELETE** qualifier is given, you must have delete permission.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**.

CLIP Subsystem required

Command Procedure.

Status

Operational.

REMARK 15.1

Leading and trailing blanks that appear around the replacement textstrings are stripped. The same is true of internal blanks unless you protect the text with apostrophes, as in Example 15.5. For additional details see §5.3.

REMARK 15.2

Macrosymbols, prompt strings and formal arguments (the latter if the **CALL** occurs within a procedure) may appear in *Argument_list*.

REMARK 15.3

If an error is detected while processing the **CALL** directive (for example, an undefined argument name is specified), processing of the directive is terminated and the procedure is not called.

EXAMPLE 15.1

```
*CALL OPEN.LDI (LDI = 3; FILE = LIBRARY.GAL; OPT = NEW)
```

The formal argument names are **LDI**, **FILE** and **OPT**. The corresponding replacement texts are: **3**, **LIBRARY.GAL** and **NEW**. Note the use of blanks for readability.

EXAMPLE 15.2

This is the same **CALL** as before, but now it is broken into several lines:

```
*CALL OPEN.LDI (          -- Call OPEN procedure
  LDI = 3 ;               -- Logical Device Index
  FILE = LIBRARY.GAL ;    -- Library filename
  OPT = NEW)              .  Status option
```

All text to the right of the hyphenation marks is treated as comment.

EXAMPLE 15.3

Yet another variant, in which the argument-text portions are replaced by prompt (quote) strings:

```
*CALL OPEN.LDI (
  LDI = "Logical Device Index (1-30)?  " --
  FILE = "Library Filename?  " --
  OPT = "Options (NEW/OLD/SCR)? ")
```

EXAMPLE 15.4

Some CLIP users have raised the question: why are semicolons used as argument list separators? Why not the more familiar commas? Answer: to permit transmission of comma-connected lists as replacement text without forcing apostrophe protection, as in

```
*CALL SHELL.SORT (V = 8,5,1,21,3,13,2 ; ORDER=UP)
```

EXAMPLE 15.5

On a similar topic: can semicolons be included in the replacement text? Yes. Use enclosing apostrophes, as in

```
*CALL OTCLIB (P = '*open 3,newlib ; *toc 3 ; close 3')
```

This is an impressive example although admittedly an unlikely one. But it illustrates the fact that virtually anything can be replacement text.

16
CLOSE

§16.1 THE CLOSE DIRECTIVE

Purpose

Closes a data library or all active libraries.

Format

```
*CLOSE [ldi] [/DELETE]
```

Optional Parameters

ldi	If positive, Logical Device Index of library file to be closed. A negative *ldi* serves a special purpose (see the GAL Manual). If omitted, all active libraries are closed.

Word Qualifiers

DELETE	Delete library file upon close. Use with extreme caution.

Description

You may use the CLOSE directive to close a specific data libraries or all active libraries. A closed data library can no longer be referenced by the Processor, and ceases to exist if it resided on a scratch file, or if the DELETE qualifier is used. For surviving libraries, a flush-buffer operation is performed as part of the close service.

Operational Restrictions

To specify a close-with-delete, you should have delete privileges on the library file(s).

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

REMARK 16.1

No diagnostic is given if the indicated library is not presently active, or if no libraries are active when a no-LDI directive is issued.

REMARK 16.2

You will normally see an informative message issued by the I/O Manager DMGASP, unless such messages have been suppressed.

EXAMPLE 16.1

Close library attached to Logical Device Index 6:

```
*CLOSE 6
```

EXAMPLE 16.2

Close all active libraries:

```
*CLOSE
```

EXAMPLE 16.3

Make the Processor close all libraries near the end of the run.

```
CALL CLPUT ('*CLOSE')
```

THIS PAGE LEFT BLANK INTENTIONALLY.

17

COPY

§17.1 THE COPY-DATASET DIRECTIVE

Purpose

Copies and optionally renames selected datasets from a library to another.

Format

```
*COPY DATASET Output_dataset = Input_dataset [/DELETED]
```

Abbreviation

*COPY DATASET may be abbreviated to *COPY.

Required Parameters

Output_dataset — Identifies the residence and name of output dataset(s). The general specification is

ldid, [*Output_dataset_name*]

where *ldid* is the Logical Device Index of the destination library, and *Output_dataset_name* specifies the name of the output dataset.

The destination library may be the same as the source library. If different, both libraries must be of the same format: positional or nominal (see **Operational Restrictions**).

If the name is omitted, the output dataset(s) will have the same name as the source dataset(s). Restricted masking specifications are allowed for the case in which several datasets are copied: if a name component is replaced by an asterisk the corresponding key or cycle in the input dataset name is substituted.

Input_dataset — The input dataset(s) are identified by one of the constructions

ldis, *dsn1*[:*dsn2*]

ldis, *Source_dataset_name*

where *ldis* is the Logical Device Index of the source library and the following item(s) specify the source dataset(s) to be copied.

The first form specifies that datasets in the range *dsn1* through *dsn2* (inclusive) be copied.

The second form restricts the copy to datasets whose name matches *Input_dataset_name*. Masking and cycle-range specifications are acceptable.

Word Qualifiers

DELETED	Copy only deleted datasets.
	If no qualifiers are given, only active datasets are copied.

Description

The **COPY DATASET** directive tranfers whole datasets or selected nominal dataset records from one library to another (or the same) library. The copy process is more efficient in positional libraries since a blocked copy may be used. In nominal libraries, a record-by-record copy is used, which may be inefficient in the case of many small records.

Operational Restrictions

Both source and destination library must be open at the time the directive is given. The destination library must have write permission. Both libraries must be of the same format. You may copy from a positional library to another positional library, or from a nominal library to another nominal library. But you may not copy from a nominal library to a positional library or vice-versa.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

REMARK 17.1

Deleted datasets can be copied by using the qualifier **DELETED**. These datasets become active in the destination library.

EXAMPLE 17.1

Copy datasets sequenced 3 through 6 from library 11 to library 5.

```
*COPY 5 = 11,3:6
```

EXAMPLE 17.2

Copy all datasets in library 11 whose mainkey is **STRUCTURE** to library 14 with identical names:

```
*COPY 14 = 11, STRUCTURE.*
```

EXAMPLE 17.3

As above, but change the mainkey to **FLUID**:

```
*COPY 14, FLUID.* = 11, STRUCTURE.*
```

§17.2 THE COPY RECORD DIRECTIVE

Purpose

Copies and optionally renames selected records from a nominal dataset to another.

Format

```
*COPY RECORD Output_record = Input_record [/KEY] [/MERGE]
```

Required Parameters

Output_record	Identifies the residence and name of the output record(s).
Input_record	Identifies the residence and name of the output record(s).

Word Qualifiers

KEY	Not implemented.
MERGE	Not implemented.

Description

The COPY RECORD directive transfers one or more named records from one nominal dataset to another nominal dataset. The source and destination datasets may be in the same library or in different libraries.

Operational Restrictions

Both source and destination libraries must be open at the time the directive is given. The destination library must have write permission. Both libraries must be of nominal (GAL82) type.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

EXAMPLE 17.4

Copy record DATA.5 from dataset 16 of library 6 to dataset 18 of the same library, with same name:

```
*COPY REC 5,18 = 5,16,DATA.1
```

EXAMPLE 17.5

Copy record group **DATA.5:18** from dataset 16 of library 6 to dataset **NEWDATA** of library 8, changing the record key name to **DATAX**:

```
*COPY REC 5,NEWDATA,DATAX = 5,16,DATA.5:18
```

EXAMPLE 17.6

Copy last 3 cycles of record group keyed **HISTORY** from dataset 16 of library 6 to the same dataset, renaming them **H.1:3**:

```
*COPY REC 6,16,H.1 = 6,16,HISTORY,H-2:H
```

EXAMPLE 17.7

Copy all records whose key starts with **D** from dataset 16 of library 6 to dataset 4 of library 7, with same name (renaming is unsafe on masked-record copy):

```
*COPY REC 7,4 = 6,16,D*
```

THIS PAGE LEFT BLANK INTENTIONALLY.

18

DEFINE

§18.1 THE DEFINE DIRECTIVE

Purpose

Defines or redefines a macrosymbol.

Format

```
*DEFINE [/Type] [/READ=f_info] [/SCOPE=level]
Macro_name {= | ==} Definition_text
```

Required Parameters

Macro_name

The macrosymbol name. The name may contain up to 16 characters. In the case of a macrosymbol array, the index and enclosing brackets are considered part of the name.

The first character must be a letter or a dollar sign. If the latter, the second character must be a letter. Be careful not to conflict with the built-in macrosymbol names listed in §5.

Definition_text

The macro definition text is a string of up to 420 arbitrary characters. If this text contains delimiters such as blanks or commas, or macrosymbol references, it should be enclosed within apostrophes. You may also use apostrophes to prevent pre-evaluation as explained in §4.

The text may be an item list, in which case it defines a macrosymbol array. Refer to §5 for details.

Macro_name and *Definition_text* must be separated by an equal sign, or an equal sign pair. The latter is used to force global scope.

Qualifiers

Type

A macrosymbol type other than default must be specified as a qualifier immediately following the DEFINE keyword (**never** after the macrosymbol name). Legal types are listed in §4.

Phrase Qualifiers

READ=*f_info*

You may use this specification to define macrosymbol values from data stored in an ASCII file. The specification of *f_info* is:

f_info = *unit_number,starting_line_number,number_of_lines_to_read*

The file must be opened with a previous issue of the directive *FOPEN *unit_number, file.name* (see Chap. 35 in Vol. II) Note that, now the user has the ability to rewind the file and close it with

the *FREWIND and *FCLOSE directives. The *starting_line_number* is the line number in *file.name* at which you wish to begin reading data. The *number_of_lines_to_read* is the number of lines of data to be read. See Examples 18.7 & 18.8. If the READ qualifier is given, it must appear **before** the macrosymbol name.

SCOPE=*level*

You may use this specification to set the macrosymbol scope to a nondefault value. (For a discussion of the macrosymbol scope concept, see §5.) If the SCOPE qualifier is given, it must appear **before** the macrosymbol name.

If *level* is a positive number, the scope is set to this number. If *level* is negative, the scope is set to the current procedural level minus |*level*|, but never less than zero.

If the macrosymbol name begins with $ this qualifier phrase is ignored.

If this qualifier phrase does not appear, the macrosymbol scope is set according to the default rules stated in §5.

Description

The macrosymbol name is checked against the macrosymbol name table and any macrosymbols with the same name and equal or higher level is marked as undefined. If the type is not P, the definition text is pre-evaluated unless protected with apostrophes. The resulting definition text, name, type and scope are stored in the macrosymbol table. If the directive defines a macrosymbol array, the preceding operations are carried out in a loop implicitly controlled by the length of the item list that appears after the equal sign(s).

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

Macrosymbol.

Status

Operational.

REMARK 18.1

If an error is detected while processing a single macrosymbol, that macrosymbol is not defined. A more complicated situation may occur if the error is detected while processing a macrosymbol array definition: processing is discontinued with the possible result that some entries end up defined while others are undefined.

REMARK 18.2

When you define a macrosymbol array by making *Definition_text* a list, comma separators are essential. Redundant commas generate zero items; see Example 18.3.

EXAMPLE 18.1

Define floating-point macrosymbol **XX** at current procedural level, and give it the value π^2:

```
*DEF XX = (<PI>^2)
```

Deferred evaluation is not necessary here because **PI** is a built-in macrosymbol.

EXAMPLE 18.2

As above, but make **XX** global:

```
*DEF XX == (<PI>^2)
```

EXAMPLE 18.3

Define the 9-integer macrosymbol array **KRONECKER** with values **1, 0, 0, 0, 1, 0, 0, 0, 1**:

```
*define Kronecker = 1,,,1,,,1
```

EXAMPLE 18.4

As above, but in **KRONECKER[5:13]**:

```
*define Kronecker[5:13] = 1,,,1,,,1
```

EXAMPLE 18.5

Define macrosymbol array **REC_ID** as a list of three character macrosymbols **LIBRARY**, **DATASET** and **RECORD**:

```
*DEF/A REC_ID = '<LIBRARY>','<DATASET>','<RECORD>'
```

EXAMPLE 18.6

Define integer macrosymbol array **LEVEL** as 20 values equal to 12:

```
*def/i level[1:20] = 12
```

EXAMPLE 18.7

An ASCII file **data.numbers** contains

```
1, 5
7
13
```

To define four indexed (array) integer macrosymbols from this data you issue the first directive to open the file, the second to define the macrosymbols, and the third to close the file.

```
*fopen 17, data.numbers

*define/i /read=17,1,3 yoursym =

*fclose 17
```

This will produce:
```
YOURSYM[1] = 1
YOURSYM[2] = 5
YOURSYM[3] = 7
YOURSYM[4] = 13
```

Note that, the *Type* qualifier is required and the final '=' sign is REQUIRED in the *define.

EXAMPLE 18.8

An ASCII file **data2.numbers** contains

```
1  5
7
13
```

To define four indexed (array) integer macrosymbols from this data you issue the first directive to open the file, the second to define the macrosymbols, and the third to close the file.

```
*fopen 17, data2.numbers

*define/i /read=17,1,3 yoursym[1:4] =

*fclose 17
```

This will produce:
```
YOURSYM[1] = 1
YOURSYM[2] = 5
YOURSYM[3] = 7
YOURSYM[4] = 13
```

Note that, the *Type* qualifier is required and the final '=' sign is REQUIRED in the *define.

And the [1:4] is required to get CLIP to properly read the first line with 2 numbers and no comma separator. However, if each number is a separate line in the file then you can omit the **[m:n]** and you will get the number of indexed macrosymbols equal to the *number_of_lines_to_read* you requested.

REMARK 18.3

Actually, the **/READ** phrase qualifier data works the same as the present macro *define directive works. That is, if you leave out the commas and don't ask for indices you don't get indices or anything except the first number read.

I would be very careful with strings, occasionally **/READ** option does some strange things with **/A** and **/P** types. So let's say the **/READ** option is operational for numerics.

THIS PAGE LEFT BLANK INTENTIONALLY.

19

DELETE

§19.1 THE DELETE DATASET DIRECTIVE

Purpose

Deletes dataset(s) from a data library.

Format

```
*DELETE DATASET Dataset_id
```

Abbreviation

***DELETE DATASET** may be abbreviated to ***DELETE**.

Required Parameters

Dataset_id — Identifies the dataset(s) to be deleted. The specification may be by name or sequence number range:

ldi, *Dataset_name*

ldi, *dsn1* [:*dsn2*]

The first form specifies that datasets in library *ldi* whose name matches *Dataset_name* are to be deleted. The name may contain masking or cycle range specifications.

The second form specifies that datasets in library *ldi* whose sequence number falls in the range *dsn1* through *dsn2* (inclusive) are to be deleted. If *dsn2* is omitted, *dsn2* = *dsn1* is assumed.

Description

You may use the **DELETE DATASET** directive to mark datasets in a data library as deleted. The datasets may be identified by name or by sequence range. Deleted datasets are not physically removed from the library until a **PACK** operation is performed. Until then, a deleted dataset may be restored to active status through an **ENABLE** operation.

Operational Restrictions

Data library file must be open and have write permission. Datasets must not be locked against deletion.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

REMARK 19.1

No diagnostics are given if dataset(s) that match the input specifications are already deleted.

EXAMPLE 19.1

Delete datasets 21 through 32 (inclusive) from library 7:

```
*DELETE 7, 21:32
```

EXAMPLE 19.2

Delete from library 16 all datasets whose extension begins with TEMP:

```
*DELETE 16, *.TEMP*
```

§19.2 THE DELETE RECORD DIRECTIVE

Purpose

Deletes named record(s) from a nominal dataset.

Format

```
*DELETE RECORD Record_id [/COMPRESS] [/KEY]
```

Required Parameters

Record_id — Identifies the record(s) to be deleted. The dataset specification may be by name or sequence number range:

ldi, *dsn1*[:*dsn2*], *Record_name*

ldi, *Dataset_name*, *Record_name*

Record_name identifies the records to be deleted in the specified datasets. If the qualifier KEY is not given, the most general form of *Record_name* is that befitting a Record Group, viz.

Key.low_cycle:high_cycle

If this form is used, *Key* should not have masking characters. See also Remark 19.2.

If the KEY qualifier is given, the cycle specifications should be omitted and *Key* may contain masking characters.

Word Qualifiers

COMPRESS — Not implemented.

KEY — Delete all records that match the input key, regardless of cycle.

Description

You may use the **DELETE RECORD** directive to mark one or more named records as deleted. Unlike datasets, a deleted record cannot be restored to an active status. Although the record itself is kept in the library, the record pointer is often erased.

Operational Restrictions

Data library must be open and have write permission. Owner datasets must not be locked against deletion.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Experimental; format subject to change. Avoid use until fully operational since library damage may occur.

REMARK 19.2

It is possible to delete selective records from both "ends" of a Record Group. Deletion of intermediate cycles, however, may result in the "Conflicting Record Group delete" error condition. For example, suppose that you have Record Group **GGG.7:40** using a single Record Access Packet and that you try to delete **GGG.15:22**. The operation will fail because the remainder, namely **GGG.7:14** and **GGG.23:40**, would use up two Record Access Packets, and presently the Global Data Manager lacks the necessary splitting ability. On the other hand, deleting **GGG.7:15** or **GGG.22:40** would be reasonable.

EXAMPLE 19.3

Delete records named **CONTENTS** in all datasets of library 7:

```
*DELETE RECORD 7,*,CONTENTS
```

EXAMPLE 19.4

Delete the lowest 4 cycles of Record Group **HISTORY** from dataset **TRANSIENT.RESPONSE** in library 16:

```
*DELETE RECORDS 16,TRANSIENT.RESPONSE, HISTORY.L:L+3
```

EXAMPLE 19.5

Delete from dataset at sequence 56 of library 3 all records whose key starts with T:

```
*DELETE RECORDS /KEY 3, 56, T*
```

20

DO

§20.1 THE DO DIRECTIVE

Purpose

Introduces a FORTRAN-like looping construction.

Format

```
*DO [:Label] Macro_name = i1, i2 [,i3]
```

Required Parameters

Macro_name — The name of a *local* macrosymbol that will serve as control variable. The first character of the name must be a dollar sign; the second must be a letter. Integer type is assumed regardless of the letter.

The macro is defined when the DO begins execution. On the first pass, it has the value i_1, on the second pass it is $i_1 + i_3$, and so on. On exit from the loop, the macrosymbol retains the last value assigned to it. The local scope has implications as to materialization and "export" of the macrosymbol value.

The cycle/exit condition is identical to that used in the FORTRAN 77 language.

The equal sign after *Macro_name* is mandatory.

i_1 — An integer or integer expression that specifies the initial value of the control variable.

i_2 — An integer or integer expression that specifies the final value of the control variable.

Optional Parameters

Label — The label that closes the loop. If omitted, the loop is terminated by a matching ENDDO directive, which has to appear on a line by itself.

i_3 — An integer or integer expression that specifies the increment of the control variable. The value may be positive or negative, but not zero.

If i_3 is not given, a value of $\pm$ 1 is assumed, depending on the values of i_1 and i_2. A zero value is treated as a non-given value and transformed to +1. This strategy aims to protect against infinite loops.

Description

DO blocks can only appear in command procedures. When the procedure compiler encounters a DO line, it generates an "UNDEFINE control variable" directive, an internal loopback label, and a transformed DO line. The UNDEFINE directive makes sure that the control variable macrosymbol is freshly defined regardless of previous execution flow.

When the callable procedure element is read and a DO line detected, CLIP queries the macrosymbol facility as to whether the control variable is defined or undefined. If the latter, CLIP knows that this is the first loop traversal and defines the control variable with a value i_1. If defined, a jumpback has necessarily occurred, (recall that the generated jumpback label is placed *after* the UNDEFINE), therefore CLIP redefines the control variable by incrementing its previous value by i_3.

The control variable value is compared against i_2. If the exit condition is satisfied, control passes to a generated label beyond the loop closure. Otherwise the loop is executed.

Dataline Restrictions

This directive must be in a dataline by itself.

Operational Restrictions

DO loops may be nested up to a level of 16 maximum. The control variable is not normally accessible from other procedures, but may be "exported" through macrosymbols or formal arguments. See Example 20.1.

Processor Reference

Not applicable.

CLIP Subsystem(s) required

Command Procedure.

Status

Operational but not thoroughly tested.

REMARK 20.1

The DO directive supersedes the CYCLE directive of previous versions of CLIP and its ancestor LODREC. DO has important advantages over CYCLE: advantages: it tests at the top, it works like a FORTRAN DO with which many users are familiar, and the control variable is a macrosymbol rather than a register. Since the last remaining use of registers in CLIP was to support CYCLE, elimination of the latter means that registers may also be eliminated.

REMARK 20.2

There is another looping construction: the WHILE-DO, which is specified by directives WHILE and ENDWHILE. Like DO, the WHILE-DO tests at the top, but cycling is controlled by a logical rather than an arithmetic condition. The WHILE-DO construct is more along the lines of similar statements in C and Pascal, and may be consequently preferred by non-FORTRAN programmers.

REMARK 20.3

The use of an explicit label in the **DO** line is usually a matter of style. An explicit label comes handy, however, when you want to write explicit transfers (using **JUMP**) to the end of the loop.

EXAMPLE 20.1

The following sample procedures **DO** and **INNER** illustrate nesting, control variable materialization, and control variable "export" to higher procedural levels:

```
*proc do (n1=1;n2=4;n3=1)
     *do $i = [n1],[n2],[n3]
       *remark outer loop counter is <$i>
          *do $j = 1,<$i>
            *remark    middle loop counter is <$j>
               *do $k = 1,<$j>
                 *remark      innermost loop counter is <$k>
                 *call inner (i=<$i>;j=<$j>;k=<$k>)
               *enddo
          *enddo
     *enddo
*end
*proc inner (i;j;k)
  *remark     proc INNER entered, arguments:  [i], [j], [k]
*end
```

It is instructive to compile these procedures and then ***CALL DO**.

21
DUMP

§21.1 THE DUMP DIRECTIVE

Purpose

Print the contents of any VAX/VMS file in item-by-item format.

Format

```
*DUMP Filename [item_range] [/Format] [/OUT=unit]
```

Required Parameters

Filename — The name of the file to be dumped. This file is opened by the I/O Manager (DMGASP) in read-only mode and closed when the dump operation is complete.

Optional Parameters

item_range — An optional item-range specification of the form

$$n_1 : n_2$$

restricts file dump to item range n_1 through n_2 inclusive. (Note that these are *logical* values; for example, if the dump format is A, n_1 and n_2 denote characters.)

If omitted, 1 through the end-of-file (as seen by DMGASP) is assumed. (For large files this can be a lot of print, so a restricted range specification is recommended.)

Word Qualifiers

Format — One-letter qualifier that specifies dump format, as listed in Table 21.1. If not given, hexadecimal (Z) format is assumed.

Phrase Qualifiers

OUT=*unit* — Write output to logical unit *unit*. If not given, write output to the current CLIP print file (normally logical unit 6).

Description

The DUMP directive may be used to print the contents of any VAX/VMS file on a item-by-item basis, in alphanumeric (A), floating (C,D,E), integer (I) or hexadecimal (Z) format. The file is open by the I/O Manager DMGASP in read-only mode. The specified item print range is accessed using Block I/O and written to the output file according to the specified format. Once finished, the file is closed.

Operational Restrictions

You cannot dump files currently opened for write or files for which the owner has denied read access. In particular, you cannot dump a scratch file, or currently active data libraries.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface and Block I/O.

Status

Operational on VAX only. Format subject to change.

REMARK 21.1

A marvelous debugging aid.

EXAMPLE 21.1

Hex dump of file INPUT.DAT, words 100 through 252 (inclusive):

```
*DUMP INPUT.DAT 100:252
```

EXAMPLE 21.2

Character dump of first 300 bytes of file IMAGE.DAT:

```
*DUMP/A IMAGE.DAT 1:300
```

Table 21.1. **Item Dump Formats**

Qualifier	*Item Dump Format*
A	alphanumeric
D	double precision floating point
I	integer
E,F,G	single precision floating point
C	complex (single precision)
Z	hexadecimal
none	Z assumed

22

ELSE

§22.1 THE ELSE DIRECTIVE

Purpose

Introduces an "else" subgroup within an IF-THEN-ELSE block.

Format

```
*ELSE
```

Description

This directive introduces, within a IF-THEN-ELSE construction, one or more commands which are to be processed if all previous IF and ELSEIF expression assertions fail.

When the procedure compiler encounters an ELSE directive. it generates a "catch all" label and removes the ELSE line.

Dataline Restrictions

This directive must be in a dataline by itself.

Operational Restrictions

Works only within command procedures. Only one ELSE is permitted per IF-THEN-ELSE.

Processor Reference

Not applicable.

CLIP Subsystem(s) Required

Command Procedure.

Status

Operational.

EXAMPLE 22.1

See examples under IF and ELSEIF.

23
ELSEIF

§23.1 THE ELSEIF DIRECTIVE

Purpose

Introduce an "else if" condition within an IF-THEN-ELSE block.

Format

```
*ELSEIF logical_expression /THEN
```

Required Parameters

logical_expression An expression that evaluates to integer 0 for FALSE or 1 for TRUE. More generally, a nonzero value is also interpreted as TRUE.

Control reaches this directive line if all previous ELSEIF tests as well as the initial IF test, on this IF-THEN-ELSE block, have failed. If the expression is TRUE, the following commands are processed and control then transfers to the closing ENDIF line. If the expression is FALSE, control passes to the next ELSEIF, ELSE or ENDIF in this IF-THEN-ELSE block.

Description

The ELSEIF directive introduces, within an IF-THEN-ELSE block, commands whose execution is contingent upon the verification of its asserted numerical expression, and failure of all preceding IF and ELSEIF assertions.

When the procedure compiler encounters an ELSEIF it generates a transfer label to it, and maps the ELSEIF line into a labelled IF directive.

Dataline Restrictions

This directive must be in a dataline (or several datalines) by itself. The THEN qualifier must be in the same line as the ELSEIF. If the *logical_expression* is so long that it requires continuation lines, you should place the THEN immediately after the ELSEIF.

Operational Restrictions

Should be used only within command procedures. No more than 32 ELSEIFs may be subordinate to one IF.

Processor Reference

Not applicable.

CLIP Subsystem(s) Required

Command Procedure.

Status

Operational.

REMARK 23.1

The **THEN** qualifier may in fact be omitted and the procedure compiler will insert one for you.

EXAMPLE 23.1

Consider the following sample procedure

```
*proc if (a)
*def a = [a]
*if < <a> /gt 0> /then
  *rem The value <a> is positive.
*elseif < <a> /eq 0> /then
  *rem The value <a> is zero.
*else
  *rem The value <a> is negative.
*endif
*end
```

If you say ***CALL IF (A=0)** the **REMARK** directive following the **ELSEIF** will be processed.

THIS PAGE LEFT BLANK INTENTIONALLY.

24

ENABLE

§24.1 THE ENABLE DIRECTIVE

Purpose

Returns deleted dataset(s) to active status.

Format

```
*ENABLE Dataset_id
```

Synonym

UNDELETE is the same as ENABLE.

Required Parameters

Dataset_id — Identifies the dataset(s) to be enabled. The specification may be by name or sequence number range:

ldi, *Dataset_name*

ldi, *dsn1*[:*dsn2*]

The first form specifies that datasets in library *ldi* whose name matches *Dataset_name* are to be enabled. The name may contain masking or cycle range specifications.

The second form specifies that datasets in library *ldi* whose sequence number falls in the range *dsn1* through *dsn2* (inclusive) are to be enabled. If *dsn2* is omitted, *dsn2* = *dsn1* is assumed.

Description

You may use the ENABLE directive to revert datasets in a data library to the active status. Enabling a dataset may cause currently active datasets that have the same name to be marked as deleted because active dataset names must be unique.

Operational Restrictions

Data library must be open and have write permission. Datasets that may have to be deleted as a byproduct of this operation must not be locked against deletion.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

EXAMPLE 24.1

Enable dataset at sequence number 132 in library 7:

```
*ENABLE 7, 132
```

THIS PAGE LEFT BLANK INTENTIONALLY.

25

END

§25.1 THE END DIRECTIVE

Purpose

Terminates the definition of a command procedure.

Format

```
*END
```

Description

This directive terminates the definition of a command procedure introduced by a PROCEDURE directive. It is illegal in all other circumstances.

Operational Restrictions

Only works as part of a command procedure definition.

Dataline Restrictions

This directive must be in a dataline by itself.

Processor Reference

Not applicable.

Status

Operational.

CLIP Subsystem(s) Required

Command Procedure.

EXAMPLE 25.1

See examples in **PROCEDURE** Section and in §5.

26
ENDDO

§26.1 THE ENDDO DIRECTIVE

Purpose

Terminates a label-less DO loop.

Format

```
*ENDDO
```

Description

The ENDDO directive closes a command block headed by a matching label-less DO directive. The procedure compiler transforms this line into an unconditional JUMP back to the internal label that precedes the DO lines, and a generated exit label.

Dataline Restrictions

This directive must be in a dataline by itself.

Operational Restrictions

Works only within command procedures.

Processor Reference

Not applicable.

CLIP Subsystem(s) Required

Command Procedure.

Status

Operational.

EXAMPLE 26.1

See examples under DO.

27

ENDIF

§27.1 THE ENDIF DIRECTIVE

Purpose

Terminates an **IF-THEN-ELSE** block.

Format

```
*ENDIF
```

Description

This directive must appear as the last line of a **IF-THEN-ELSE** command block. The procedure compiler transforms the **ENDIF** line into a "collective" exit label for the block.

Dataline Restrictions

This directive must be on a dataline by itself.

Operational Restrictions

Works only within a command procedure.

Processor Reference

Not applicable.

CLIP Subsystem(s) Required

Command Procedure.

Status

Operational.

EXAMPLE 27.1

See the Sections on **IF** and **ENDIF**.

28
ENDLOG

§28.1 THE ENDLOG DIRECTIVE

Purpose

Terminates command logging.

Format

```
*ENDLOG
```

Description

This directive terminates the saving of dataline input on the command log file opened through a previous LOG directive. This directive must be on an isolated dataline, which is not saved. The log file is closed.

Dataline Restrictions

This directive must be in a dataline by itself.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

REMARK 28.1

Upon entering this directive, you may want to TYPE the log file to see what it contains.

EXAMPLE 28.1

See the LOG Section and §3.

29

ENDWHILE

§29.1 THE ENDWHILE DIRECTIVE

Purpose

Terminates a WHILE-DO block.

Format

```
*ENDWHILE
```

Description

A WHILE-DO block introduced by a WHILE directive must be terminated by a matching ENDWHILE directive. The directive line is processed by the procedure compiler analogously to an ENDDO.

Dataline Restrictions

Must appear in a dataline by itself.

Operational Restrictions

Works only within a command procedure.

Processor Reference

Not applicable.

CLIP Subsystem(s) Required

Command Procedure.

Status

Operational.

EXAMPLE 29.1

See examples under WHILE.

30

EOF

§30.1 THE EOF DIRECTIVE

Purpose

Endfile command source.

Format

```
*EOF
```

Description

This directive causes the processing of a command source to be interrupted just like if CLIP had detected an end-of-file condition. It is most widely used for scripts since for command procedures the **RETURN** directive provides an equivalent function.

If this directive is issued at the root level, a normal run termination is taken, *i.e.*, it has the same effect as typing a **©** sentinel (or equivalent control character).

Dataline Restrictions

This directive must be in a dataline by itself.

Processor Reference

This directive may be (and usually is) submitted through the message entry point **CLPUT**.

CLIP Subsystem(s) Required

None.

Status

Operational.

REMARK 30.1

Normally this directive is inserted by the running processor using a **CLPUT** call, as in

```
CALL CLPUT ('*eof')
```

in response to a critical event (e.g., irrecoverable input error). More exotic is the use of the **IFELSE** built-in macrosymbol:

```
<ifelse (<error>;SERIOUS;*eof;)>  .  Got it?
```

EXAMPLE 30.1

See examples in §3.

31
EOL

§31.1 THE EOL DIRECTIVE

Purpose

Inserts an end-of-line (EOL) terminator in the dataline collector.

Format

```
*EOL
```

Description

This directive inserts an end-of-line (EOL) terminator after the current command in the dataline collector. Commands, if any, waiting in the collector are erased. Has specialized uses in highly interactive programming and message based systems. Rarely entered as user directive; it is normally submitted as message using CLPUT (see Remark 31.1).

Dataline Restrictions

When submitted as a message, it should not be followed by other commands or directives in the same line.

Processor Reference

This directive may be (and usually is) submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

REMARK 31.1

Normally this directive is inserted by the running processor using a CLPUT call:

```
CALL CLPUT ('*EOL')
```

when it wants to discard all commands that may follow in the same line, if any. Normally issued after certain types of error conditions in essentially conversational programs such as interactive graphic pre- and post-processors. Flushing the dataline collector helps to control error cascading effects and simplifies error recovery procedures. If input is coming from a script or procedure source, the EOL message is often followed by an EOF message.

32

FCLOSE

§32.1 THE FCLOSE DIRECTIVE

Purpose

Closes a FORTRAN logical unit.

Format

```
*FCLOSE  unit
```

Required Parameters

unit Logical unit number of the file to be closed.

Description

The FCLOSE directive closes a FORTRAN logical unit assigned to the run. The unit is normally connected to a card-image file explicitly opened with a FOPEN directive, or implicitly opened as nonstandard print output unit (e.g., with an OUT=*unit* qualifier). The interactive user may want to close such units to TYPE them without interrupting the Processor run; see Example below.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 32.1

The following illustrates a typical use of FCLOSE in an interactive session:

```
*FOPEN 12,TOC.LIS
*OPEN 1,PROBLEM.GAL
*TOC/OUT=12 1
*FCLOSE 12
*TYPE TOC.LIS
```

File TOC.LIS is opened to receive the TOC of library PROBLEM.GAL so it can be eventually sent to a printer. Before exiting the Processor, however, the user checks the contents of FOR.LIS using TYPE. But for TYPE (or LIST) to work the file must be closed first (because it was opened with write permission).

33

FIND

§33.1 THE FIND DATASET DIRECTIVE

Purpose

Returns information on individual dataset.

Format

```
*FIND DATASET Dataset_id [/BEGIN=dsn0] [/KEYS=Macro_name]
[/MKEY=max_keys] [/MRAP=Macro_name] [/NAME=Macro_name]
[/NOK=Macro_name] [/NOR=Macro_name] [/SEQ=Macro_name]
```

Synonyms

FIND and INQUIRE are equivalent directive verbs.

Abbreviation

*FIND DATASET may be abbreviated to *FIND.

Required Parameters

Dataset_id — Identifies the dataset to be queried. The identification involves two items. The first is the Logical Device Index (LDI) of the source data library, which must be open at the time the directive is issued. The second item identifies the dataset by name or by sequence number:

ldi, *Dataset_name*

ldi, *dsn*

If the first form is used, *Dataset_name* may have masking characters or cycle range specifications, but information is retrieved only for the first dataset that matches the name.

Phrase qualifiers

BEGIN=*dsn0* — Applicable if *Dataset_id* is specified by name. Begin dataset search at sequence number *dsn0+1*.

If omitted, search starts at sequence number 1.

KEYS=*Macro_name* — Not implemented.

MKEY=*max_keys* — Not implemented.

NAME=*Macro_name* — Return the full dataset name as value of macrosymbol specified after qualifier NAME. The macrosymbol, of type A, is defined by error-free execution of the directive. If the library identification or dataset specification is incorrect, the macrosymbol is not defined.

NOK=*Macro_name*	Number of keys.
NOR=*Macro_name*	Number of records.
SEQ=*Macro_name*	If *Dataset_id* is specified by name and a matching dataset name is found, return its sequence number as value of macrosymbol specified after qualifier SEQ. If not found, return the value zero. The macrosymbol, of type I, is defined by error-free execution of the directive. If the library identification or dataset specification is incorrect, the macrosymbol is not defined.

Description

This directive is designed to collect information about an individual dataset. Such information may be obtained from the Dataset Table (Table of Contents) of a given library. For example: get the dataset name given its sequence number, or get the sequence number given the dataset name. Information is returned in the form of indirectly defined macrosymbols. Some of the information retrieval options pertain only to nominal datasets.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface and Macrosymbol.

Status

Experimental; format subject to change. More qualifier options are likely.

REMARK 33.1

If this directive appears in a procedure, macrosymbols created from its execution will be semilocal in scope if one equals sign appears after the qualifier and the macro name starts with a letter. To force global scope, use a double equal sign after the qualifier. If the macro name starts with a dollar sign, it will be local in scope regardless of the number of equal signs.

REMARK 33.2

The type (integer or alpha) of indirectly defined macrosymbols is specified by the qualifier. It is not affected by the first letter of the name as is the case for explicitly defined macrosymbols.

EXAMPLE 33.1

```
*open/old 21,funlib
*find 21,matrix.data /seq=matrix-seq
*if < <matrix-seq> /eq 0> /then
 *remark MATRIX.DATA dataset not found in FUNLIB
*endif
```

§33.2 THE FIND DATASETS DIRECTIVE

Purpose

Returns information on multiple datasets.

Format

```
*FIND DATASETS Dataset_id [/BEGIN=dsn0]
[/NSEQ=Macro_name] [/SEQ=Macro_name]
```

Synonyms

FIND and INQUIRE are equivalent directive verbs.

Required Parameters

Dataset_id — Identifies the datasets to be queried in the form

ldi, *Dataset_name*

where *ldi* is the library's LDI, and *Dataset_name* usually contains masking characters or cycle range specifications.

Phrase qualifiers

BEGIN=*dsn0* — Begin dataset search at sequence number *dsn0+1*.

If omitted, search starts at sequence number 1.

NSEQ=*Macro_name* — Not implemented.

SEQ=*Macro_name* — Return the dataset sequence numbers of up to *max_seq* datasets that match the input name, as found in a forward library search, and assign these values to macrosymbol named after qualifier SEQ. This macrosymbol, of type I, is defined by the error-free execution of the directive. The macrosymbol will be an array if qualifier MSEQ is given, as it usually the case; otherwise, it will be an ordinary one. If no dataset is found, only one value with the value zero is created. If the library identification or dataset name is incorrect, no macrosymbol is created.

Description

This directive is designed to search a library for all datasets that match a given (usually masked) name. Search starts at the first dataset unless this is overridden with a qualifier, and proceeds until a maximum number of datasets is found or the end of the dataset table reached. The information is returned in the form of indirectly defined macrosymbols.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface and Macrosymbol.

Status

Experimental; format subject to change.

REMARK 33.3

Remarks 33.1 and 33.2 also apply here.

§33.3 THE FIND LIBRARY DIRECTIVE

Purpose

Returns information on specific data library.

Format

```
*FIND LIBRARY ldi [/FORM=Macro_name] [/NAME=Macro_name]
      [/NODS=Macro_name]
```

Synonyms

FIND and INQUIRE are equivalent directive verbs.

Required Parameters

ldi — Logical Device Index of the data library to be queried.

Phrase qualifiers

FORM=*Macro_name* — Return library form identifier as value of macrosymbol specified after qualifier FORM. This macrosymbol, of type A, is defined by an error-free execution of the directive. The value returned is the string DAL, GAL80 or GAL82. If *ldi* is not connected to a library or is out of range, no macrosymbol is created.

NAME=*Macro_name* — Not implemented.

NODS=*Macro_name* — Return number of datasets as value of macrosymbol specified after qualifier NODS. This macrosymbol, of type I, is defined by the error-free execution of the directive. The returned count includes deleted datasets. If *ldi* is not connected to a library or is out of range, no macrosymbol is created.

Description

This directive is used to request general information pertaining to a specific data library identified by its LDI. The information is returned in the form of indirectly defined macrosymbols.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP subsystem(s) required:

NICE-DMS Interface and Macrosymbol.

Status

Experimental; format subject to change.

REMARK 33.4

Macros created by this directive will be semilocal or local in scope if a single equal sign appears after the qualifier. To force global scope, use two equal signs, as in the Example below.

REMARK 33.5

Remarks 33.1 and 33.2 also apply here.

EXAMPLE 33.2

```
*OPEN/OLD 7,INLIB
 *FIND LIBRARY 7 /FORM==INLIB-FORM /NODS==INLIB-DSETS
```

§33.4 THE FIND LIBRARIES DIRECTIVE

Purpose

Return information about all active libraries.

Format

```
*FIND LIBRARIES [/LDI=Macro_name] [/MOL=Macro_name]
```

Synonyms

FIND and INQUIRE are equivalent directive verbs.

Phrase qualifiers

LDI=*Macro_name* Return the Logical Device Indices of all libraries currently open and assign them as values of the entries of a macrosymbol array whose

name is specified after qualifier LDI. The macrosymbol entries will be of type I. If no libraries are open, the macrosymbol is not defined.

NOL=*Macro_name* Return number of open libraries as value of macrosymbol specified after qualifier NOL. This macrosymbol, of type I, is defined by the directive execution. If no libraries are open the macrosymbol is assigned the value zero.

Description

The NICE-GAL manager is queried as to the number of open libraries and their LDIs. The requested information is returned in the form of indirectly defined macrosymbols.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP subsystem(s) required:

NICE-DMS Interface and Macrosymbol.

Status

Experimental; format subject to change.

REMARK 33.6

Remarks 33.1 and 33.2 also apply here.

§33.5 THE FIND RECORD DIRECTIVE

Purpose

Return information on indexed record.

Format

```
*FIND RECORD ldi,dsn,irec [/SIZE=Macro name]
```

Required Parameters

ldi Logical Device Index of *positional* library file.

dsn The sequence number of the owner dataset.

irec The record index.

Phrase qualifiers

SIZE=*Macro_name* On search completion, create integer macrosymbol named as qualifier parameter, and assign the record size in words to it. If the record was not found, assign the value zero.

Description

For positional libraries the FIND RECORD directive is used to search for an indexed record and to return size information on it. The information is returned in the form of an indirectly defined macrosymbol.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface and Macrosymbol.

Status

Experimental; format subject to change.

§33.6 THE FIND RECORD DIRECTIVE FOR NOMINAL LIBRARIES

Purpose

Returns information on named record key.

Format

```
 *FIND RECORD_KEY Record_key_id [/CYC=Macro_name] [/DIM=Macro_name]
/ [/NOR=Macro_name] [/SIZE=Macro_name] [/TYPE=Macro_name]
```

Required Parameters

Record_key_id Identifies the nominal record key to be queried. The owner dataset may be specified by name or by sequence number:

ldi, Dataset_name, Key

ldi, dsn1, Key

Phrase qualifiers

CYC=*Macro_name* Returns a macrosymbol with Macro_name[1] equal to the low cycle number and Macro_name[2] equal to the high cycle number. If the library identification or dataset specification is incorrect, the macrosymbol is not defined.

DIM=*Macro_name* Returns a macrosymbol with Macro_name equal to the "first matrix dimension." If the library identification or dataset specification is incorrect, the macrosymbol is not defined.

NOR=*Macro_name* Returns a macrosymbol with Macro_name equal to the number of records. If the library identification or dataset specification is incorrect, the macrosymbol is not defined.

SIZE=*Macro_name* Returns the record size in logical units as the value of the macrosymbol specified after the qualifier SIZE. If the key is not found, return zero. The macrosymbol, of type I, is defined by error-free execution of the directive. If the library identification or dataset specification is incorrect, the macrosymbol is not defined.

TYPE=*Type* Returns the record type code letter (A, I, etc.) as the value of the macrosymbol specified after the qualifier TYPE. If the record key is not found, return U. The macrosymbol, of type A, is defined by error-free execution of the directive. If the library identification or dataset specification is incorrect, the macrosymbol is not defined.

Description

This directive is used to inquire about record keys in a nominal dataset. Information of interest includes low and high cycles, number of records, size and type. The information is returned as value of indirectly defined macrosymbols.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface and Macrosymbol.

Status

Experimental; format subject to change.

REMARK 33.7

Remarks 33.1 and 33.2 also apply here.

THIS PAGE LEFT BLANK INTENTIONALLY.

34

FLUB

§34.1 THE FLUB DIRECTIVE

Purpose

Flushes buffers of a data library or all active libraries.

Format

```
*FLUB [ldi]
```

Optional Parameters

ldi	If positive, Logical Device Index of library file to be flushed (1-16). If omitted (or zero), all active libraries are flushed.

Description

This directive forces core-resident library tables flagged as altered by the data manager to be written to the library file. This "flushing" operation ensures conformity of library contents and protects against catastrophic run aborts. The operation may involve a specific library or all active libraries. It does not apply to scratch libraries or to libraries opened in read-only mode.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

EXAMPLE 34.1

Flush library attached to Logical Device Index 6:

```
*FLUB 6
```

35

FOPEN

§35.1 THE FOPEN DIRECTIVE

Purpose

Opens a card-image file connected to a FORTRAN unit.

Format

```
*FOPEN  unit,Filename [/NEW] [/OLD]
```

Required Parameters

unit	Logical unit number assigned to the file opened by this directive. If omitted, 1 is assumed. Some care should be exercised to avoid clashing with other logical units in use by CLIP or the Processor.
Filename	Name of the file to be opened. There are no default names.

Word Qualifiers

NEW	Open a new file.
OLD	Open an existing file.
	If neither NEW or OLD is specified, a new file.

Description

The FOPEN directive opens a new, sequential access, card-image file through a FORTRAN OPEN statement, and connects the file to a specified logical unit. This file is often used to receive print information produced by directives such as PRINT or SHOW, or coded database information produced by the UNLOAD directive. The file may be printed, rewound or closed using the FPRINT, FREWIND and FCLOSE directives.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 35.1

The following example (also given for **FCLOSE**) illustrate typical use of **FOPEN** in an interactive session:

```
*FOPEN 12,TOC.LIS
*OPEN 1,PROBLEM.GAL
*TOC/OUT=12 1
*FCLOSE 12
*TYPE TOC.LIS
```

For uses in conjunction with database transfer operations, see the sections on **LOAD** and **UNLOAD**.

THIS PAGE LEFT BLANK INTENTIONALLY.

36

FPRINT

§36.1 THE FPRINT DIRECTIVE

Purpose

Prints lines of card-image file connected to a FORTRAN unit.

Format

```
*FPRINT unit, [n]
```

Required Parameters

unit	Logical unit number assigned to the file to be printed. Normally established using a FOPEN directive.
n	Number of lines to be printed; default one. Print starts at the current location of the file.

Description

The FPRINT directive prints lines of a new, sequential access, card-image file connected through a FORTRAN OPEN statement. This is useful for previewing a connected text file.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 36.1

The following example illustrates typical use of FPRINT for an interactive session:

```
*FOPEN/NEW  12,TOC.LIS . open new text file
*OPEN  2, DATA.LIB
*TOC/OUT=12 2 .  write TOC of library 2 to text file
*FREWIND 12 .  rewind it
*FPRINT 12,10 .  print first 10 lines
*FPRINT 12,20 .  print next 20 lines
```

37

FREWIND

§37.1 THE FREWIND DIRECTIVE

Purpose

Rewinds file connected to a FORTRAN unit.

Format

```
*FREWIND [unit]
```

Required Parameters

unit — Logical unit number of the file to be rewound. Normally established using a FOPEN directive.

Description

The FREWIND directive rewinds a sequential access, card-image file connected through an FOPEN directive. This is occasionally useful in conjunction with FPRINT.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 37.1

The following example illustrate typical use of FREWIND in an interactive session:

```
*FOPEN 12,TOC.LIS . opens TOC.LIS and connects to unit 12
*FPRINT 12,50 .  print first 50 lines
*FREWIND 12 .  rewind *FREWIND 12,10 .  print first 10 lines again
*FCLOSE 12 .  close
```

37a

GAL2MAC

§37a.1 THE GAL2MAC DIRECTIVE

Purpose

Defines a Macrosymbol with a value obtained from a GAL Dataset.

Format

```
*GAL2MAC /Name{=|==}Macrosymbol [/Type=type]
[/Ioff=ioff] [/Maxn=max items] Record id
```

Abbreviation

*GAL2MAC may be abbreviated to *G2M.

Required Parameters

Name{=|==} *Macrosymbol*	The name of a Macrosymbol that will receive the values of the GAL Dataset(s) extracted.
Record_id	Identifies the record(s) to be extracted. The identification consists of three items separated by commas. The first item is the Logical Device Index (LDI) of the source data library, which must be open at the time the *GAL2MAC is issued. The second item identifies the owner dataset by name or by sequence number. The third item is the record index (for one record) or a record index range (for several records): *ldi, Dataset_name, irec1[:irec2]* *ldi, dsn irec1[:irec2]* If the first form is used, the dataset name may contain masking and cycle-range specifications.

Phrase Qualifiers

Type=*type*	Enter the *type* of the macrosymbol. See §4 for a description of macrosymbol types. If type not specified, the record is checked and type returned; if non-existent, nothing written.
Ioff=*ioff*	Integer offset by *ioff* in the record to be read. *Ioff* is equivalent to IOFF in the GMGETx calling sequence. See the GAL-DBM Manual §9.5.
Maxn=*max_items*	Read only *max_items* from the record. Note that, *max_items* is used for the value of N in the call to GMGETx and LENGTH is a qualifier, so *max_items* can be a positive or negative number. See the GAL-DBM

Manual §9.5. *Max_items* is limited to 100 integers or single precision floating numbers, 50 double precision floating point numbers, or 400 characters.

Description

This directive is used to move data from nominal GAL Datasets into CLIP macrosymbols.

CLIP Subsystem(s) Required

Macrosymbol and NICE-DMS Interface.

Status

Operational.

REMARK 37a.1

The converse operation is performed by the ***MAC2GAL** directive.

EXAMPLE 37a.1

Read three values from the record DENSITY from dataset 1 on GAL library 1 and create the three macrosymbols DENSITY[1], DENSITY[2], and DENSITY[3].

```
*g2m /Name=DENSITY   1,1,DENSITY.1:3
```

THIS PAGE LEFT BLANK INTENTIONALLY.

38

GENERATE

§38.1 THE GENERATE DIRECTIVE

Purpose

Generates subsequent commands through an item incrementation mechanism.

Format

```
*GENERATE  [/k]  [increment_list]
```

Optional Parameters

increment_list — A list of numeric increments. The list must not exceed 100 items.

These increments are considered in one-to-one correspondence with numeric items in the previous ordinary command, character strings being ignored for purposes of this correspondence. Omitted trailing increments are considered zero.

Word Qualifiers

k — The number of commands to be generated (default 1). If it appears it must follows the directive keyword.

Description

The GENERATE directive causes subsequent command(s) to be generated through a numeric-item incrementation mechanism. It is sort of a relic of the first CLIP implementation (1969) at Boeing, but once in a while it comes in handy.

The formal description is rather messy, but the example below should be sufficient to illustrate how it works.

CLIP Subsystem(s) required

None.

Status

Operational since 1969.

REMARK 38.1

In previous versions of CLIP and its LODREC ancestor, this directive was called DO. This name now applies to the directive that introduces a FORTRAN-like loop.

REMARK 38.2

This directive cannot be used to generate directives.

REMARK 38.3

Generated commands are not printed even if command echo is on, because the generation proceeds directly at the Decoded Item Table level. The generation process may be visualized, however, by turning on the "Decoded Command" echo option by saying ***SHOW ECHO = DEC**.

EXAMPLE 38.1

The following two lines

```
SET XYZ = 1.0, 2.0, 3.0
*GEN /4 -1, 1.5
```

are equivalent to the five lines

```
SET XYZ =  1.0, 2.0, 3.0
SET XYZ =  0.0, 3.5, 3.0
SET XYZ = -1.0, 5.0, 3.0
SET XYZ = -2.0, 6.5, 3.0
SET XYZ = -3.0, 8.0, 3.0
```

The **GENERATE** directive has generated the last 4 **SET XYZ** commands.

THIS PAGE LEFT BLANK INTENTIONALLY.

39

GET

§39.1 THE GET TEXT-DATASET DIRECTIVE

Purpose

Extracts Text Dataset(s) to card-image file.

Format

```
*GET TEXT_DATASET Filename = Dataset_id
```

Required Parameters

Filename — The name of a FORTRAN card-image file that will receive the contents of the Text Dataset(s) extracted. This file is created by execution of the directive.

Dataset_id — Identifies the Text Dataset to be extracted by the Logical Device Index followed by the dataset name or sequence range:

ldi, *Dataset_name*

ldi, *dsn1* [:*dsn2*]

The first form specifies that Text Datasets in library *ldi* whose name matches *Dataset_name* are to be extracted. The name may contain masking or cycle range specifications.

The second form specifies that Text Datasets in library *ldi* whose sequence number falls in the range *dsn1* through *dsn2* (inclusive) are to be extracted. If *dsn2* is omitted, *dsn2* = *dsn1* is assumed.

Description

This directive is used to extract one or more Text Datasets resident in a positional data library. The output is written to a card-image FORTRAN file, which is created by execution of this directive.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

REMARK 39.1

The converse operation is performed by the **PUT TEXT_DATASET** directive.

EXAMPLE 39.1

Extract Text Dataset **CARD.DECK** from library 7 to output file **CARDS.DCK**:

```
*get text cards.dck = 7,card.deck
```

EXAMPLE 39.2

Extracts all Text Datasets in data library 12 whose extension is **DECK** and put the output in file **BIGDECK**:

```
*get text bigdeck = 12,*.deck
```

§39.2 THE GET TEXT-GROUP DIRECTIVE

Purpose

Extracts Text Group(s) to card-image file.

Format

*GET TEXT_GROUP *Filename* = *Text_group_id*

Required Parameters

Filename — The name of a FORTRAN card-image file that will receive the Text Dataset contents. This file is created by execution of the directive.

Text_group_id — Identifies the Text Group(s) to be extracted. The applicable forms are:

ldi, *dsn1*[:*dsn2*], *Key*

ldi, *Dataset_name*, *Key*

The first form specifies that Text Groups named *Key* in library *ldi* that belong to datasets whose sequence number range *dsn1* through *dsn2* (inclusive) are to be extracted. If *dsn2* is omitted, *dsn2* = *dsn1* is assumed. *Key* may contain masking characters.

The second form specifies that Text Groups named *Key* that reside in library *ldi* and belong to datasets whose name matches *Dataset_name* are to be extracted. The dataset name may contain masking or cycle range specifications. *Key* may contain masking characters.

Description

This directive performs essentially the same functions as the **GET TEXT_DATASET**, but for Text Groups.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational for single Text Group only.

REMARK 39.2

The converse operation is performed by the **PUT TEXT_GROUP** directive.

EXAMPLE 39.3

Extract Text Group **CONTENTS** that resides in dataset **NODAL.COORDINATES.4** of library 22 into output file **CONTENTS.LIS**:

```
*get text contents.lis = 22, nodal.coordinates.4, contents
```

40

HELP

§40.1 THE HELP DIRECTIVE

Purpose

Lists topic-qualified sections of a NICE help file.

Format

```
*HELP  [Topic [Subtopic ...]]
```

Optional Parameters

Topic ... — A list of topics that specifies which segment of the current help file is to be listed. If omitted, the "root" segment is listed.

Description

The HELP directive causes topic-qualified segments of NICE-formatted help files to be listed. The help file is designated through the SET HFILE directive. The default help file is the one that explains CLIP directives. For details and application examples see §6.10 and Appendix H of the CLIP Reference Manual, Volume III.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 40.1

```
*HELP ADD
```

41

IF

§41.1 THE LABELED IF DIRECTIVE

Purpose

Conditionally branch to target label on logical expression.

Format

```
*IF logical_expression [:Label]
```

Required Parameters

logical_expression — An expression that evaluates to integer 0 for FALSE or 1 for TRUE. More generally, a nonzero value is also interpreted as TRUE. The expression is usually constructed through ordinary or logical macrosymbols (see §4 and examples).

Optional Parameters

Label — The name of the target label. Control passes to the target label (or, more precisely, the command that follows it) if the logical expression is TRUE; otherwise the next command is processed.

If the label is omitted, a return from procedure is assumed.

Description

When an IF directive of this type is encountered, CLIP evaluates the logical expression. If the value is TRUE (integer one or, more generally, any nonzero integer) the "next dataline to read" counter in the command source stack is set to that associated with the label. If the expression is FALSE, nothing happens and the next dataline is fetched.

Dataline Restrictions

This directive must be in a dataline (or several datalines) by itself. The label must be in the same line as the IF. If the *logical_expression* is so long that it requires continuation lines, you should place the label immediately after the IF.

Operational Restrictions

Works only inside a command procedure.

Processor Reference

Not applicable.

CLIP Subsystem(s) Required

Command Procedure.

Status

Operational.

REMARK 41.1

The present labeled **IF** directive is an extension of the previous **IF** directive, which only accepted very limited arithmetic expressions. The old forms will work but procedure writers are encouraged to switch to the more FORTRAN-like logical expression forms.

EXAMPLE 41.1

This illustrates a simple arithmetic comparison:

```
*IF < <pi> /gt 3.1 > :B1256
```

where **PI** is the built-in macro that defines π to 16 decimal places. Since $\pi > 3.1$, the expression is **TRUE** and control will pass to the command that follows label **B1256**.

EXAMPLE 41.2

This is a more complex expression:

```
*IF < < <x1> /ge 0 > /and < <x2> /ge 0> /and < <x3> /ge 0> > :GO-AHEAD
```

This says: if the value of the three numeric macrosymbols **X1**, **X2** and **X3** is nonnegative, jump to **GO-AHEAD**. The expression may be put in more readable form by defining three logical switches:

```
*def/i x1nonneg = < <x1> /ge 0>
*def/i x2nonneg = < <x2> /ge 0>
*def/i x3nonneg = < <x3> /ge 0>
*IF < <x1nonneg> /and <x2nonneg> /and <x3nonneg> > :GO-AHEAD
```

Which form is better is largely a matter of personal preference.

REMARK 41.2

It is generally best to avoid complicated expressions as these are harder to visualize in the debugging phase.

EXAMPLE 41.3

If you have a logical expression that involves both **AND**s and **OR**s, be sure to make your intentions clear by proper nesting. For example:

```
*IF < <<x1nonneg> /or <x2nonneg>> /and <x3nonneg> > :GO-AHEAD
```

is not the same as

```
*IF < <x1nonneg> /or <<x2nonneg> /and <x3nonneg>> > :GO-AHEAD
```

EXAMPLE 41.4

The following form is for instructional purposes only:

```
*IF 1
```

Can you guess what happens? Since 1 is **TRUE**, control transfers to the implied label, which is the end of the procedure. So in fact the above is equivalent to a ***RETURN** !

§41.2 THE BLOCK IF DIRECTIVE

Purpose

Introduces an IF-THEN-ELSE block.

Format

```
*IF logical_expression /THEN
```

Required Parameters

logical_expression — An expression that evaluates to integer 0 for FALSE or 1 for TRUE. More generally, a nonzero value is also interpreted as TRUE.

If the expression is TRUE, control drops to the command that follows. If the expression is FALSE, control jumps to the next ELSEIF, ELSE or ENDIF directive that pertains to this IF-THEN-ELSE block.

Description

When the procedure compiler an IF encounters a directive of this form, it transforms it to a labeled IF that transfers to a generated label. The THEN qualifier is removed and a tag appended to the compiled IF to let CLIP known that the normal interpretation of the labeled IF (jump if TRUE) should be reversed (jump if FALSE).

Dataline Restrictions

This directive must be in a dataline (or several datalines) by itself. The THEN qualifier must be in the same line as the IF. If the *logical_expression* is so long that it requires continuation lines, you should place the THEN immediately after the IF.

Operational Restrictions

Works only inside a command procedure.

Processor Reference

Not applicable.

CLIP Subsystem(s) Required

Command Procedure.

Status

Operational.

REMARK 41.3

If you forget the qualifier THEN, the procedure compiler assumes that this is a labeled IF with implied "exit of procedure" label, and you are likely to get some nasty error messages later. The same thing will occur if the THEN is on a continuation line; see **Dataline Restrictions** above.

EXAMPLE 41.5

Consider the procedure

```
*proc blockif (value=10)
*if < [value] /gt 0 > /then
  *rem The decimal log of [value] is <log10([value])>.
*elseif < [value] /eq 0> /then
  *rem The input <value> is zero; its log is -infinity.
*else
  *rem The input <value> is negative; it has no real log.
*endif
*end
```

If after compiling you say ***CALL BLOCKIF (VALUE=2)** you should get the printout

```
The decimal log of 2 is 3.010299956639813D-01.
```

For other examples, see §6.

THIS PAGE LEFT BLANK INTENTIONALLY.

42

JUMP

§42.1 THE JUMP DIRECTIVE

Purpose

Transfers control to specified procedure label.

Format

```
*JUMP [:Label]
```

Optional Parameters

Label — The target label. If omitted, control transfers to the end of the procedure (equivalent to a **RETURN** directive).

Description

This directive unconditionally transfers control to a specified target label in a procedure. The dataline that follows the label line will be processed next.

Operational Restrictions

Works only inside a command procedure.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**, although such an event would be highly unlikely.

CLIP Subsystem(s) Required

Command Procedure.

Status

Operational.

REMARK 42.1

Note that CLAMP is a truly structured language. It has no **GOTO** statement.

EXAMPLE 42.1

Unadorned **JUMP**:

```
*JUMP :LOOP
```

EXAMPLE 42.2

More readable form with "noise" word:

```
*JUMP TO :LOOP
```

EXAMPLE 42.3

Conditional form:

```
.
.  If EXECUTE = 'YES' jump to :EXECUTE else exit procedure
.
<ifelse ( <execute> ; yes ; *jump to :execute ; *jump )>
```

The unlabeled **JUMP** in the fourth macro argument plays the same role as a **RETURN**.

THIS PAGE LEFT BLANK INTENTIONALLY.

43
LIST

§43.1 THE LIST FILE DIRECTIVE

Purpose

List card-image file on CLIP print output file.

Format

```
*LIST Filename [/HEAD] [/OUT=unit]
```

Required Parameters

Filename — The name of the card-image file to be listed. Masking is not permitted.

Word Qualifiers

HEAD — Write a header line that gives the name of the file.

Phrase qualifiers

OUT=*unit* — Write output to logical unit *unit*. If not given, output is written to the CLIP print file (normally logical unit 6).

Description

The LIST-file directive lists the contents of a card-image source file. Output goes to the CLIP print output file (that shown as Cpr: in response to a SHOW UNITS directive) unless overridden by the OUT qualifier phrase. The file is accessed by a FORTRAN OPEN statement. Read-only mode is used if allowed by the host computer. Once the listing is completed, the file is closed.

Operational Restrictions

You cannot list files that are currently open for write, or files where read access is denied by the owner. This directive should not be on unformatted or direct-access files (a "Cannot open file" error message will appear).

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

REMARK 43.1

The print record length is normally limited to 80 characters. This is the default line print width, which may be increased up to 132 characters, if desired, through the **SET WIDTH** directive.

EXAMPLE 43.1

```
*LIST PROC:FORPRC.MSC
```

File **PROC:FORPRC.MSC** (a VAX filename) is to be listed.

§43.2 THE LIST TEXT DATASET DIRECTIVE

Purpose

List Text Dataset on CLIP print file.

Format

```
*LIST Text_dataset_id [/HEAD] [/OUT=unit]
```

Required Parameters

Text_dataset_id — Identifies the Text Dataset(s) to be listed. The dataset may be specified by name or sequence number:

ldi, *Dataset_name*

ldi, *dsn1*[:*dsn2*]

The first form specifies that Text Datasets in library *ldi* whose name matches *Dataset_name* are to be listed. The name may contain masking or cycle range specifications.

The second form specifies that Text Datasets in library *ldi* whose sequence number falls in the range *dsn1* through *dsn2* (inclusive) are to be listed. If *dsn2* is omitted, *dsn2* = *dsn1* is assumed. Datasets that are not of text type are ignored.

Word Qualifiers

HEAD — Write a header line that gives the dataset name.

Phrase qualifiers

OUT=*unit* — Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

This form of the LIST directive is similar to the list-file form, but the source is a Text Dataset that resides on an active positional (DAL or GAL80) library.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface for listing Text Datasets or Text Groups.

Status

Operational.

REMARK 43.2

Remark 43.1 also applies here.

EXAMPLE 43.2

```
*LIST/H 1,4
*LIST/H 1,*
```

§43.3 THE LIST TEXT GROUP DIRECTIVE

Purpose

List Text Group on CLIP print file.

Format

```
*LIST Text_group_id [/HEAD] [/OUT=unit]
```

Required Parameters

Text_group_id — Identifies the Text Group(s) to be listed. The applicable forms are:

ldi, *Dataset_name*, *Key*

ldi, *dsn1*[:*dsn2*], *Key*

The first form specifies that Text Groups named *Key* that reside in library *ldi* and belong to datasets whose name matches *Dataset_name* are to be listed. The dataset name may contain masking or cycle range specifications. *Key* may contain masking characters.

The second form specifies that Text Groups named *Key* in library *ldi* that belong to datasets whose sequence number range *dsn1* through *dsn2* (inclusive) are to be listed. If *dsn2* is omitted, *dsn2* = *dsn1* is assumed. *Key* may contain masking characters.

Word Qualifiers

HEAD	Write a header line that gives the library index, dataset name and Text Group name.

Phrase qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

The LIST Text Group directive causes CLIP to search the library for the owner dataset(s) specified in it. For each dataset, the Record Access Table is searched for the given Text Group name (= Record Group key). When found, the equivalent of the "get text" operation is invoked with the CLIP print unit (or the OUT-specified unit) as output. The library must be opened at the time the LIST is requested and is not closed.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational except for *Key* masking.

REMARK 43.3

Remark 40.1 also applies here.

EXAMPLE 43.3

```
*LIST/H 1,4,CONTENTS
*LIST 2,*,ABSTRACT /OUT=2
```

THIS PAGE LEFT BLANK INTENTIONALLY.

44

LOAD

§44.1 THE LOAD DIRECTIVE

Purpose

Internalize nominal records from UNLOAD-created text file to data library.

Format

```
*LOAD ldi [[,Dataset_id],Record_id] = unit
```

Optional Parameters

ldi — Logical Device Index (LDI) of the library that will receive the load data. Must be open at the time the directive is issued, have write permission, and be of GAL82 (nominal) format.

unit — Logical unit number of the text file containing loading data. This file is normally created by an UNLOAD directive, but not necessarily on the same computer. The file must be opened with the old option before the LOAD directive is issued. The unit-file connection is usually performed by an FOPEN directive (see examples). The file is rewound and scanned until end of file, but is not closed (see Description).

Optional Parameters

Dataset_id — A generally masked dataset name that may be used to select datasets to be loaded. If omitted, datasets in the load file are not filtered.

Record_id — A generally masked record name that may be used to select records to be loaded. If omitted, records names are not filtered.

Description

The load file is rewound, and then scanned forward until the end of file is detected. When a dataset name is encountered, CLIP tests when it is to be loaded. If so, it checks the data library for a match; if none a new dataset is installed. Then all subordinate records are examined. If a record is to be loaded, CLIP checks whether the record key already exists. If it exists and the new record has the same length and data type, it is overwritten or appended. If the key does not exist, it is installed and records are then appended.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

EXAMPLE 44.1

Load all database data in file **RESPONSE.VAL** into data library **RESPONSE.LIB**:

```
*open/new 1,response.lib
*fopen/old 3,response.val
*load 1= 3
*fclose 3
*toc ; *rat
```

The use of 1 and 3 for LDI and logical unit, respectively, is incidental.

EXAMPLE 44.2

As above, but now load only datasets whose name starts with **H** into existing data library **RESPONSE.LIB**:

```
*open 1,response.lib
*fopen/old 3,response.val
*load 1,H* = 3
*fclose 3
*toc ; *rat
```

THIS PAGE LEFT BLANK INTENTIONALLY.

45

LOCK

§45.1 THE LOCK DIRECTIVE

Purpose

Change locking status of dataset(s).

Format

```
*LOCK Dataset_id = Lock_code
```

Status

Not implemented.

46
LOG

§46.1 THE LOG DIRECTIVE

Purpose

Initiates command logging.

Format

```
*LOG [Filename] [/ALL]
```

Optional Parameters

Filename — Name to be given to the command log file.

If omitted, the name CLIPLOG.DAT is assumed on the VAX running under VAX/VMS; CLIPLOG on other systems.

Word Qualifiers

ALL — Log lines read from all command source stack levels.

If this qualifier is omitted, only lines entered at the command source stack root level are logged; this is precisely what is needed in applications in which the log file is to be ADDed in a subsequent run.

Description

This directive opens a new card-image file called the "command log" file, which is connected to an internal unit. Subsequent input lines entered from the terminal are copied "as is" to this file. The logging process may be terminated by an ENDLOG directive. Refer to §3.4 for additional details.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

Dataline Restrictions

This directive must be in a dataline by itself.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 46.1

Consider the conversational sequence

```
*log newinput.dat
*add oldinput.dat
solve equations
print results
*endlog
*type newinput.dat
```

The TYPE directive should show that file NEWINPUT contains the following 3 lines:

```
*add oldinput
solve equations
print results
```

Note that the lines from OLDINPUT are not logged; only the ADD directive.

THIS PAGE LEFT BLANK INTENTIONALLY.

46a

MAC2GAL

§46a.1 THE MAC2GAL DIRECTIVE

Purpose

Writes a Macrosymbol value or Macrosymbol values to a nominal GAL Dataset.

Format

```
*MAC2GAL /Name{=|==}Macrosymbol [/Type=type]
[/Ioff=ioff] [/Maxn=max_items] Record_id
```

Abbreviation

*MAC2GAL may be abbreviated to *M2G.

Required Parameters

Name{=|==} *Macrosymbol*	The name of a Macrosymbol that contains the value(s) to be written into a GAL Dataset.
Record_id	Identifies the record(s) to be written to. The identification consists of three items separated by commas. The first item is the Logical Device Index (LDI) of the source data library, which must be open at the time the *GAL2MAC is issued. The second item identifies the owner dataset by name or by sequence number. The third item is the record index (for one record) or a record index range (for several records): *ldi, Dataset_name, irec1[:irec2]* *ldi, dsn irec1[:irec2]* If the first form is used, the dataset name may contain masking and cycle-range specifications.

Phrase Qualifiers

Type=*type*	Enter the *type* of the macrosymbol. See §4 for a description of macrosymbol types. If *type* not specified, the record is checked and *type* returned; if nonexistent, nothing is written.
Ioff=*ioff*	Integer offset by *ioff* in the record to be read. *Ioff* is equivalent to IOFF in the GMPUTx calling sequence. See the GAL-DBM Manual §9.9.
Maxn=*max_items*	Read only *max_items* from the record. Note that, *max_items* is used for the value of N in the call to GMPUTx, so *max_items* can be a positive or negative number. See the GAL-DBM Manual §9.9.

Max_items is limited to 100 integers or single precision floating numbers, 50 double precision floating point numbers, or 400 characters.

Description

This directive is used to move data from CLIP macrosymbols into nominal GAL Datasets.

CLIP Subsystem(s) Required

Macrosymbol and NICE-DMS Interface.

Status

Operational.

REMARK 46a.1

The converse operation is performed by the ***GAL2MAC** directive.

EXAMPLE 46a.1

Write the values from the three macrosymbols DENSITY[1], DENSITY[2], and DENSITY[3] into the record DENSITY in dataset 1 on GAL library 1.

```
*m2g /Name=DENSITY[1:3]    1,1,DENSITY.1:3
```

THIS PAGE LEFT BLANK INTENTIONALLY.

47

OPEN

§47.1 THE OPEN DIRECTIVE

Purpose

Opens a data library.

Format

```
*OPEN [ldi,][Filename] [/Qualifiers] [/LIMIT=limit] [/PRU=xpru]
[/LDI=Macro_name]
```

Optional Parameters

ldi — If positive, Logical Device Index (LDI) to be connected to the library file.

A negative *ldi* specification is used for special purposes discussed in the GAL-DMS Manual.

If this parameter is omitted, GAL-DMS searches for the first free LDI slot. The value selected may be materialized as a macrosymbol through an LDI phrase qualifier, as explained below.

Filename — External name of permanent library file. Should comply with file naming conventions for host system, as well as with external device names for the I/O manager DMGASP.

For libraries resident on scratch files or blank common, this name is irrelevant and may be omitted. DMGASP then selects an appropriate internal name.

Word Qualifiers

See Table 47.1.

Phrase qualifiers

LIMIT=*limit* — Applies only to newly created libraries. Declares the maximum capacity of the library device in *words*, overwriting a DMGASP-assumed default (usually ≈10 million words). Only useful for very large libraries.

LDI=*Macro_name* — If the open operation succeeds, create an integer macrosymbol whose name is the qualifier parameter and whose value is the LDI. Primarily useful in command procedures when parameter *ldi* is omitted.

Description

This directive opens a data library, and links the file name to a Logical Device Index (LDI). If the latter is not specified, it may be retrieved in the form of an indirectly created macrosymbol.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

REMARK 47.1

Remarks 32.1 and 32.2 are also applicable here to the LDI qualifier.

EXAMPLE 47.1

Open a new GAL82 library in file PROBLEM.GAL (a VAX file name) and connect it to Logical Device Index 4:

```
*OPEN/NEW/NOMINAL 4,PROBLEM.GAL
```

EXAMPLE 47.2

As above, but now let GAL-DMS pick the LDI, which is to be returned as global macrosymbol LIBLDI:

```
*OPEN/NEW/NOMINAL PROBLEM.GAL /LDI==LIBLDI
```

Table 47.1. **One-item Qualifiers for OPEN Directive**

(a) Qualifiers related to file existence and accessibility

Qualifier	*Explanation*
BC	Create "core resident library" in blank common.
COLD	Conditional: as OLD if file exists, otherwise create NEW.
NEW	Create new file, which is to survive upon close.
OLD	Open existing library file, and allow writes.
PRIVATE	Used with NEW: Library file will be private-access (Only meaningful on some operating systems e.g., UNIVAC)
ROLD	Open existing library file read-only.
SCRATCH	Open scratch library, which disappears upon close.
none of above	COLD is assumed.

(b) Word qualifiers related to library form

Qualifier	*Explanation*
DAL	Library will be of DAL form.
GAL80	Library will be of GAL80 (positional dataset) form.
GAL82	Library will be of GAL82 (nominal dataset) form.
NOMINAL	Same as GAL82.
none of above	If NEW or SCRATCH are specified, created library will be of GAL82 form unless either DAL or GAL80 appears. The same applies if library creation results from a COLD qualifier. If an existing library is attached, these qualifiers are ignored.

Table 47.1. **One-item Qualifiers for `OPEN` Directive (Concluded)**

(c) Word qualifiers related to file access method

Qualifier	*Explanation*
`BIO`	Use Block I/O (default if available).
`FIOS`	Use FORTRAN I/O device with small internal PRU.
`FIOL`	Use FORTRAN I/O device with large internal PRU.

(d) Word qualifiers related to internal I/O operation

Qualifier	*Explanation*
`PAGED`	Use Paged I/O if buffer declared (GAL only).

THIS PAGE LEFT BLANK INTENTIONALLY.

48

PACK

§48.1 THE PACK DIRECTIVE

Purpose

Compress data library removing all deleted datasets.

Format

```
*PACK ldi
```

Required Parameters

ldi Logical Device Index of library file to be packed.

Description

This directive packs an active data library. The packed library contains no deleted datasets. Dataset sequence numbers may change as a result of this operation. GAL performs packing by copying active datasets to a scratch library, which is copied back (in block mode) to the original file. The scratch library is then closed.

Operational Restrictions

The library to be packed must have write permission.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

EXAMPLE 48.1

Pack library connected to Logical Device Index 12:

```
*PACK 12
```

49

PRINT

§49.1 THE PRINT DATASET-TABLE DIRECTIVE

Purpose

Prints the Dataset Table (≡ Table of Contents) of a library.

Format

```
*PRINT DATASET_TABLE  [ldi [,Dataset_range]] [/Format] [/OUT=unit]
```

Abbreviations

*PRINT DATASET_TABLE may be abbreviated to *PRINT TOC or *TOC. The last abbreviation is by far the most commonly used.

Optional Parameters

ldi

Logical Device Index (LDI) of the library whose Dataset Table is to be listed.

If the LDI is omitted or zero, the library with highest LDI (this is usually the last one open) is assumed.

If the LDI is omitted the *Dataset_range* specification must not be present.

Dataset_range

This specification may be used to restrict the Dataset Table listing to datasets identified by name or sequence number range:

Dataset_name

dsn1 [: *dsn2*]

In the first form, *Dataset_name* usually contains masking characters or cycle range specifications. Listing is restricted to datasets that match this name.

In the second form, listing is limited to datasets whose sequence number is *dsn1* to *dsn2* inclusive. Having *dsn2* < *dsn1* is permissible and listing proceeds backwards. If *dsn2* is omitted and *dsn1* is positive, only the *dsn1*-th dataset is listed. If *dsn2* is omitted and *dsn1* is negative, the last |*dsn1*| datasets are listed in reverse order.

If the *Dataset_range* specification is not given, the entire table is listed.

Word Qualifiers

Format

A letter that specifies a display format other than standard, as per Table 49.1.

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output to the current print file (normally logical unit 6).

Description

This directive is usually issued as *TOC, which historically means Table of Contents of a positional library (DAL or GAL80). Nominal libraries (GAL82) have a Dataset Table and several Record Tables, and by extension the name TOC applies to the former.

Operational Restrictions

Library must be open. Any form is acceptable.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) required:

NICE-DMS Interface.

Status

Operational.

EXAMPLE 49.1

List complete Dataset Table of library at LDI 17 in the standard format:

```
*TOC 17
```

EXAMPLE 49.2

As above, but only the last 10 datasets:

```
*TOC 17,-10
```

EXAMPLE 49.3

As above, but restrict the listing to datasets whose extension starts with FORCE:

```
*TOC 17,*.FORCE*.*
```

EXAMPLE 49.4

List datasets in range 33 to 49 (inclusive) of library 11 in P (physical) display format and direct output to logical unit 46:

```
*toc/p 11,33:49 /out=46
```

Table 49.1. **Dataset Table (TOC) Print Formats**

Letter	*Table Display Format*
B	Brief: shows only dataset name and sequence number.
D	Dated: as B plus creation date and time (will also show last update in a forthcoming GAL-DMS version).
E	SPAR format. Primarily intended for DAL files.
M	Matrix-oriented format as used by the DALPRO code. Primarily intended for DAL files.
P	Shows physical storage details.
S	SPAR: same as E.

§49.2 THE PRINT RECORD DIRECTIVE FOR POSITIONAL DATASETS

Purpose

Prints contents of record(s) in a positional library.

Format

```
*PRINT RECORD Record_id [/Format] [/M=max_items]
[/LIST=Option_letters] [/OUT=unit]
```

Abbreviation

*PRINT RECORD may be abbreviated to *PRINT.

Required Parameters

Record_id — Identifies the record(s) to be printed. The identification consists of three items separated by commas. The first item is the Logical Device Index (LDI) of the source data library, which must be open

at the time the PRINT is issued. The second item identifies the owner dataset by name or by sequence number. The third item is the record index (for one record) or a record index range (for several records):

ldi, *Dataset_name*, *irec1*[:*irec2*]

ldi, *dsn* *irec1*[:*irec2*]

If the first form is used, the dataset name may contain masking and cycle-range specifications.

Listing the descriptor record of GAL80 files requires special care.

Word Qualifiers

Format — An item print format may be specified as a qualifier. For example, /F12.5. If the format is not given, record items are printed in hexadecimal or octal format, depending in the host computer. Some abbreviated (one-letter) print formats are available and are listed in Table 49.2.

Phrase Qualifiers

M=*max_items* — Limit number of items printed per record to *max_items*.

LIST=*Option_letters* — These provide additional control on the appearance of the print. In alpha test; will be described more fully in downstream versions.

OUT=*unit* — Write output to logical unit *unit*. If not given, output to the current print file (normally logical unit 6).

Description

This directive prints the contents of indexed records in an open positional library (DAL or GAL80). Records are read into an internal buffer and items printed in string form.

Operational Restrictions

Library must be open and be of DAL or GAL80 form.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) required:

NICE-DMS Interface.

Status

Operational.

REMARK 49.1

Explicit format specifications are more important for indexed records than for nominal records, because the record type of the latter is retained in the database. There are two other problems that may lead to unsightly output: mixing items of different type in indexed records is permitted, and the same format applies to all records printed.

EXAMPLE 49.5

Print indexed record 3 in dataset NODAL.CONNECTIVITY of library 14 in integer format:

```
*print record 14,NODAL.CONNECTIVITY,4 /i
```

EXAMPLE 49.6

Print indexed records 3 to 7 in dataset at sequence number 61 of library 7 in E11.2 format:

```
*print record 7,61,3:7 /e11.2
```

Table 49.1. **Abbreviated Format Specs for PRINT RECORD Directive**

Letter	*Item Print Format*
A	Alphanumeric (line-oriented, bounded by output line width
D	Same as 1PD24.16
E	Same as 1PE12.4
F	Same as 1PF12.4
G	Same as 1PG12.4
I	Same as I12
O	Octal; width and availability are machine dependent.
Z	Hexadecimal; width and availability are machine dependent.

§49.3 THE PRINT RECORD DIRECTIVE FOR NOMINAL DATASETS

Purpose

Prints contents of record(s) in a nominal library.

Format

```
*PRINT RECORD Record_id [/Format] [/M=max_items]
[/LIST=Option_letters] [/OUT=unit]
```

Abbreviation

*PRINT RECORD may be abbreviated to *PRINT.

Required Parameters

Record_id — Identifies the record(s) to be printed. The identification consists of three items separated by commas. The first item is the Logical Device Index (LDI) of the source data library, which must be open at the time the PRINT is issued. The second item identifies the

owner dataset by name or by sequence number. The third item is the name of the record or Record Group to be printed.

ldi, Dataset_name, Record_name

ldi, dsn Record_name

If the first form is used, the dataset name may contain masking and cycle-range specifications.

Word Qualifiers

Format — An item print format may be specified as a qualifier. For example, /F12.5. If the format is not given, record items are printed in accordance to the record type. The abbreviated (one-letter) print formats of Table 49.2 are also available for named records.

Phrase Qualifiers

M=*max_items* — Limit number of items printed per record to *max_items*.

LIST=*Option_letters* — These provide additional control on the appearance of the print. In alpha test; will be described more fully in downstream versions.

OUT=*unit* — Write output to logical unit *unit*. If not given, output to the current print file (normally logical unit 6).

Description

This directive prints the contents of named records in an open nominal library (GAL82). Records are read into an internal buffer and items printed in string form.

Operational Restrictions

Library must be open and be of nominal (GAL82) form.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) required:

NICE-DMS Interface.

Status

Operational.

REMARK 49.2

Explicit format specifications are relatively unimportant for named records. The default format usually works well. Note that if you print several records with one directive, the default setting is recommended if the records have different types (e.g., integers and reals) because the print routine can chose the format according to type.

EXAMPLE 49.7

Print record LINKS.32 in dataset NODAL.FREEDOMS of library 14 in default format:

```
*print record 14,NODAL.FREEDOMS,LINKS.32
```

EXAMPLE 49.8

Print Record Group ANGLES.1:24 in dataset at sequence number 61 of library 7 in G12.4 format:

```
*print record 7,61,ANGLES.1:24 /g
```

§49.4 THE PRINT RECORD_TABLE DIRECTIVE

Purpose

Print the Record Access Table (RAT) of nominal dataset(s).

Format

*PRINT RECORD_TABLE *Dataset_id* [/*Format*] [/OUT=*unit*]

Abbreviations

*PRINT RECORD_TABLE may be abbreviated to *PRINT RAT or *RAT. The last abbreviation is the most commonly used.

Required Parameters

Dataset_id

Identifies the dataset(s) whose Record Table is(are) to be printed. The specification may be by name or sequence number range:

ldi, *Dataset_name*

ldi, *dsn1* [:*dsn2*]

ldi

If the first form is used, the dataset name may have masking and cycle-range specifications. If the last form is used, all Record Tables in library *ldi* are printed.

Word Qualifiers

Format

A letter that specifies a display format other than standard. Presently only F, which produces a "fuller" display, is implemented.

Phrase Qualifiers

OUT=*unit*

Write output to logical unit *unit*. If not given, output to the current print file (normally logical unit 6).

Description

This directive lists the RAT of one or more nominal datasets.

Operational Restrictions

Library must be open and be of nominal (GAL82) form.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**.

CLIP Subsystem(s) required:

NICE-DMS Interface.

Status

Operational.

EXAMPLE 49.9

Print all Record Tables in library 17:

```
*RAT 17
```

EXAMPLE 49.10

As above, but in the full format:

```
*RAT/F 17
```

EXAMPLE 49.11

Print the Record Table of dataset **MY.DATA** in library 9:

```
*RAT 9,MY.DATA
```

§49.5 PRINT DIRECTIVES FOR GAL DEBUGGING

Purpose

Print GAL-DMS internal data structures.

Format

*PRINT *Entity_id*

Required Parameters

Entity_id One of the following specifications:

BALL Print internal buffer allocations.

`HEADER` *ldi*	Print header of library *ldi*.
`HASH` *ldi*	Print hash table for library *ldi*.
`DAP` *ldi*, *dsn*	Print Dataset Access Packet for dataset at sequence *dsn* in library *ldi*.
`RAP` *ldi*, *dsn*, *irap*	Print Record Access Packet numbered *irap* for dataset at sequence *dsn* in nominal library *ldi*.
`RAHEAD`	Print Record Header in-core table.

Phrase Qualifiers

`OUT=`*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

These directives are not for ordinary users since a deep knowledge of the internal workings of the global database manager is required for proper interpretation. They are applicable to debugging, general troubleshooting and I/O optimization.

Processor Reference

These directives may be submitted through the message entry point `CLPUT`.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational but availability depends on the master source code extraction keys.

THIS PAGE LEFT BLANK INTENTIONALLY.

50

PROCEDURE

§50.1 THE PROCEDURE DIRECTIVE

Purpose

Initiates the definition of a command procedure.

Format

```
*PROCEDURE Procedure_name [Argument_list]
```

Required Parameters

Procedure_name — The name of the command procedure. As discussed in §6, this name, in conjunction with the last **SET PLIB** specification, designates where the callable procedure element (CPE) will reside.

Legal procedure names vary according to callable procedure element residence. For example, if the callable procedure element is to be an ordinary file, the name must comply with the file naming restrictions of the host operating system. If the callable procedure element is to be a Text Group, the name must be a legal record key, and so on.

Optional Parameters

Argument_list — A list of up to 25 formal arguments may follow the procedure name. The list format is

(*Arg1* [= *Default_text1*] ; *Arg2* [= *Default_text2*] ; ...)

In this format *Arg1*, *Arg2* ... are formal argument names, and *Default_text1*, *Default_text2*, ... are optional default-replacement-text specifications.

Argument specifications are separated by semicolons.

The opening parenthesis *must be preceded by a blank* (else CLIP will make it part of the procedure name)

Formal argument names are characters strings of up to 16 characters; the first character must be a letter. Upper case and lower case letters are equivalent.

Each formal argument may be followed by default text. This text (a character string of up to 256 characters) is used as replacement text if the formal argument is not explicitly named in the **CALL** directive that subsequently activates the procedure. If the default text for an argument is missing, it is considered to be a null string.

Refer to §6.3 for a detailed explanation of argument passing.

Description

When CLIP detects this directive it enters the directive mode and does not exit until the **END** directive is detected. The procedure body is "compiled" to generate an object version known as a Callable Procedure Element (CPE). The callable procedure element may be subsequently activated through a **CALL** directive. The effect of a **PROCEDURE** definition is totally passive as far as the Processor is concerned because the entire process is carried out in the directive mode.

If you define a procedure interactively (only recommended for demonstration purposes) you will get a **Body>** prompt until you type an **END** directive.

Dataline Restrictions

This directive must be in a dataline by itself. If continued into several lines (common) the last line should not be followed by another command.

Operational Restrictions

If the callable procedure element is to reside on an ordinary file, you must have permission to create such a file. If the callable procedure element is to reside on a data library as a Text Dataset or Text Group, the library must be open and have write permission at the time the **PROCEDURE** directive is encountered.

Processor Reference

The Processor may create a procedure in advanced network operations. Since a procedure definition necessarily requires several lines, it has to be submitted through the multiline message entry point **CLPUTM**.

CLIP Subsystem(s) Required

Command Procedure, plus NICE-DMS Interface if callable procedure element is to reside on a data library.

Status

Operational.

REMARK 50.1

Leading and trailing blanks that appear around the default textstrings (if given) are stripped. If you want to preserve such blanks, use apostrophes. Intermediate blanks are protected.

REMARK 50.2

If an error is detected while processing this directive, CLIP merely scans for the **END** directive after issuing an appropriate diagnostic. The callable procedure element is not produced.

EXAMPLE 50.1

This illustrates a procedure that is to reside in a Text Dataset of a positional data library:

```
*SET PLIB = 2
*PROCEDURE MELLOW.YELLOW
    (body)
*END
```

The procedure name is MELLOW.YELLOW, which is a legal GAL dataset name. The procedure contains no arguments. The callable element will reside on the positional data library connected to Logical Device Index 2.

EXAMPLE 50.2

The following directives declare a procedure whose callable procedure element version is to reside on an ordinary file:

```
*SET PLIB = 0
*PROCEDURE SURPRISE (WHO;IS;THERE)
    (body)
*END
```

The procedure name is SURPRISE, which is a legal file name on most computers (CDC and Cray excepted, as it exceeds 7 characters). The procedure has three formal arguments identified by keywords WHO, IS and THERE.

EXAMPLE 50.3

The following illustrate residence on a nominal library connected to Logical Device Index 7:

```
*SET PLIB = 7, EINSTEINS.INTERSECTION.N
*PROC FIND (E ; M ; C)
    (body)
*END
```

The procedure name is FIND, which is a legal Text Group name. The procedure has three formal arguments: E, M and C. The callable element will reside on the dataset named

EINSTEINS.INTERSECTION.*cycle1*

where *cycle1* denotes the highest first cycle of active datasets whose mainkey and extension are EINSTEINS.INTERSECTION.

EXAMPLE 50.4

The following example illustrates the specification of default argument-replacement text.

```
*PROCEDURE INITIALIZE (GDB=NICE.GAL ; CYCLE)
    (body)
*END
```

If argument GDB is not specified in the CALL to this procedure, as in

```
*CALL INITIALIZE (CYCLE=6)
```

then all references to argument **GDB** in the procedure body are replaced by the default text, namely, **NICE.GAL**. On the other hand, should argument **CYCLE** be omitted from the **CALL**, references to it are erased (more technically: all occurrences of **[CYCLE]** in the procedure body are replaced by a null string).

THIS PAGE LEFT BLANK INTENTIONALLY.

51

PUT

§51.1 THE PUT DATASET DIRECTIVE

Purpose

Install dataset name in library.

Format

```
*PUT DATASET_NAME ldi, Dataset_name [/CONDITIONAL] [/MRAT=mrat]
 [/SEQ=Macro_name]
```

Abbreviation

*PUT DATASET_NAME may be abbreviated to *PUT.

Required Parameters

ldi	Logical Device Index of library file.
Dataset_name	Name of dataset to be inserted.

Word Qualifiers

CONDITIONAL	Not implemented.

Phrase qualifiers

MRAT=*mrat*	For nominal libraries only. Set maximum RAT-packets that dataset may use to *mrat*. If omitted, *mrat* = 64 is assumed, which is normally adequate.
SEQ=*Macro_name*	Not implemented.

Description

The PUT DATASET_NAME directive creates a new dataset by installing its name in the Table of Contents. Any dataset with the same name is deleted.

Operational Restrictions

Data library file must be open and have write permission. Existing datasets with the same name should not be locked against deletion.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational except for the CONDITIONAL and SEQUENCE qualifiers.

EXAMPLE 51.1

```
.          *PUT 2, DISPLACEMENT.ITERATE.N
```

§51.2 THE PUT TEXT-DATASET DIRECTIVE

Purpose

Create Text Dataset from card-image file.

Format

*PUT TEXT_DATASET *Dataset_id* = *Filename*

Required Parameters

Dataset_id	Identifies the Text Dataset by Logical Device Index linked to a sequence number or dataset name and record key.
Filename	The name of an existing FORTRAN card-image file.

Description

This directive creates a Text Dataset by copying the contents of a card image file. The file is open (in read only mode if this is allowed by the operating system) and its records read until the end of file is detected. Then it is closed.

Operational Restrictions

Data library file must be open and have write permission.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

EXAMPLE 51.2

```
*PUT 2, DISPLACEMENT.ITERATE.H, CONTENTS = CONTENTS.LIS
```

§51.3 THE PUT TEXT-GROUP DIRECTIVE

Purpose

Create Text Group from card-image file.

Format

```
*PUT TEXT_GROUP Text_group_id = Filename
```

Required Parameters

Filename — The name of an existing FORTRAN card-image file.

Text_group_id — Identifies the Text Group to be created. The applicable formats are:

ldi, *dsn1*, *Key*

ldi, *Dataset_name*, *Key*

Description

The PUT TEXT_GROUP directive performs a file-to-library function similar to that of the PUT TEXT_DATASET directive, but it creates a Text Group in a nominal library.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

52

REMARK

§52.1 THE REMARK DIRECTIVE

Purpose

Identifies "active comment" line.

Format

```
*REMARK [/OUT=unit] Text
```

Required Parameters

Text — The text to be printed. The text is assumed to extend from the end of the directive keyword (or the end of the directive qualifier if one appears) to the last nonblank character. The printed text will reflect macro and/or argument expansions.

Leading and intermediate blanks are respected and lower case characters are not converted to upper case. Consequently, it is unnecessary to protect the text with apostrophes except in the rare case noted in the second Remark below.

Phrase Qualifiers

OUT=*unit* — Write output to logical unit *unit*. If not given, output to the CLIP print file (normally logical unit 6). Setting *unit* to zero while in interactive mode forces output to go to the terminal.

Placement Restrictions. This qualifier phrase must appear immediately after the directive verb.

Not implemented.

Description

A REMARK differs from a comment line in two respects:

1. The remark text is written to a file regardless of echo-print settings. A comment line is printed only if echo is on.
2. Macrosymbols and/or formal parameters occurring in the remark text are expanded before the text is printed.

These two features justify the name "active comment" used in the **Purpose** description.

See also Remark below.

CLIP Subsystem(s) Required

None.

Status

Operational.

REMARK 52.1

This directive is trapped by CLIP's "early warning system" as soon as it is read in. It is therefore spared the normal processing of the other directives as far as parsing text, lower to upper case letter conversion, etc.

REMARK 52.2

If you want to inhibit macrosymbol expansion and formal argument substitution, you should protect the remark text with apostrophes. These apostrophes will be shown in the print output.

EXAMPLE 52.1

One common application of **REMARK** is displaying macrosymbol value(s) in more legible form than that provided by **SHOW MACROS**:

```
*rem The current value of macrosymbol FREQUENCY is <FREQUENCY>.
*rem The value of pi-square is < <pi>^2>.
*rem/v Today is <date>.  At the beep the time will be <time>.
```

EXAMPLE 52.2

Another application is to show formal argument values on entry to a command procedure:

```
*proc solve (input;output;tolerance;status)
 *rem  Entering SOLVE with INPUT = [INPUT], OUTPUT = [OUTPUT]
 *rem  and TOLERANCE = [TOLERANCE].
    ...
*end
```

THIS PAGE LEFT BLANK INTENTIONALLY.

53

RENAME

§53.1 THE RENAME DATASET DIRECTIVE

Purpose

Changes dataset name(s).

Format

```
*RENAME DATASET Dataset_id = New_name
```

Abbreviation

*RENAME DATASET may be abbreviated to *RENAME.

Required Parameters

Dataset_id

Identifies the dataset(s) to be renamed. The specification may be by name or sequence number range:

ldi, *Dataset_name*

ldi, *dsn1* [:*dsn2*]

The first form specifies that datasets in library *ldi* whose name matches *Dataset_name* are to be renamed. The name may contain masking or cycle range specifications.

The second form specifies that datasets in library *ldi* whose sequence number falls in the range *dsn1* through *dsn2* (inclusive) are to be renamed. If *dsn2* is omitted, *dsn2* = *dsn1* is assumed.

New_name

Specifies the new dataset name(s).

If *New_name* contains no masking specification, then *Dataset_id* should identify only *one* dataset.

If *Dataset_id* specifies more than one dataset, *New_name* should have at least one component masking specification. If a name component in *New_name* is specified by an asterisk, the corresponding old name component is retained.

Description

You may use RENAME DATASET to change the name of one or more datasets. You should be aware, however, that the before inserting the new name the data manager deletes all existing datasets of which names are equal to the new name. In particular, beware of renaming several datasets to a common name.

Operational Restrictions

Library must be open and have write permission.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

EXAMPLE 53.1

Change the name of the dataset at sequence number 171 of library 23 to **BORN.AGAIN**:

```
*ren 23,171 = born.again
```

EXAMPLE 53.2

Change the mainkey of all datasets in library 14 from **STRUCTURE** to **FLUID**:

```
*ren 14, structure.* = fluid.*
```

EXAMPLE 53.3

For all datasets in the range 15 to 40 (inclusive) in library 6, change the second cycle to 1 and the extension to **DATA**, leaving the other components unchanged:

```
*ren 6,15:40 = *.data.*.1.*
```

§53.2 THE RENAME RECORD DIRECTIVE

Purpose

Changes the key or cycle of nominal record(s).

Format

*RENAME RECORD *Record_id* = *New_record_name*

Required Parameters

Record_id — Identifies the nominal record(s) to be renamed. The complete specification may specify the owner dataset by name or sequence number range:

ldi, *Dataset_name*, *Record_name*

ldi, *dsn1* [:*dsn2*], *Record_name*

The two forms correspond to those discussed in the **RENAME DATASET** directive, but now are followed by a record identifier. The record identifier may be

Key

Key.cycle

New_record_name Specifies the new record name(s). The specification may be *Key* or *Key.cycle*, and should correspond to the *Record_name* in *Record_id.*

Description

You may use **RENAME RECORD** to change the name of one or more nominal records.

Operational Restrictions

You must have write access to the library file.

Processor Reference

This directive may be submitted through the message entry point **CLPUT.**

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

EXAMPLE 53.4

Rename record **HISTORY** in dataset **TIME.RESPONSE** of library 12 to **TRANSIENT**:

```
*ren 12, TIME.RESPONSE, HISTORY = TRANSIENT
```

EXAMPLE 53.5

There are several datasets called **MOTOR.GEOMETRY.***cycle* with *cycle* is 1 to 15, in library 7. Each dataset has a Record Group identified as **COORDINATES.1:240.** It is desired to change the Record Group key to **XYZ** in all datasets:

```
*ren  7, motor.geometry.1:15, coordinates = xyz
```

EXAMPLE 53.6

As above, but with two modifications: the change should apply only to the 2 datasets with highest *cycle* numbers, and the record cycle numbers should be incremented by 10 so that it becomes **XYZ.11:251**:

```
*ren  7, motor.geometry.h, coordinates.1 = xyz.11
```

54

RETURN

§54.1 THE RETURN DIRECTIVE

Purpose

Forces exit from command procedure.

Format

```
*RETURN
```

Description

This directive unconditionally transfers control to the end of a command procedure. The next dataline is taken from the previous command source.

Dataline Restrictions

Commands that follow this directive in the same line are ignored.

Operational Restrictions

Works only within command procedures.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**. Note, however, that sending an **EOF** message achieves the same effect and that **EOF** works also for script input.

CLIP subsystem(s) required:

Command Procedure.

Status

Operational.

REMARK 54.1

It is unnecessary to put a **RETURN** directive before the **END** directive.

55
RUN

§55.1 THE RUN DIRECTIVE

Purpose

Starts the execution of another Processor.

Format

*RUN [/*Qualifier*] *Processor* [*Text*]

Required Parameters

Processor — The name of the Processor to be started. (More precisely, the name of the file that contains the executable image.)

If this name is wrong, the system will usually abort the run. Then on the VAX you should delete state save file ZZZZZZZ.DAT, which is created by SuperCLIP in anticipation of a process save/restore as described in §10. (This file is automatically deleted if the directive is executed normally.)

Optional Parameters

Text — Presently ignored.

Word Qualifiers

none — Run a CLIP-supported Processor and update Process Name Stack (so that a STOP directive restarts the parent Processor).

CHAIN — Run a CLIP-supported Processor but don't update Process Name Stack (this precludes use of a downstream STOP directive).

FOREIGN — Run program not supported by CLIP.

Description

This directive, together with STOP, forms the basis of SuperCLIP.

Operational Restrictions

This directive is only presently available under VAX/VMS and Unix-based systems.

Dataline Restrictions

This directive must be in a dataline by itself.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

SuperCLIP.

Status

Operational.

EXAMPLE 55.1

See examples in §10.

THIS PAGE LEFT BLANK INTENTIONALLY.

56
SET

§56.1 THE SET ARGUMENTS DIRECTIVE

Purpose

Changes the value of a procedure argument.

Format

```
*SET ARGUMENT = Text
```

CLIP Subsystem(s) Required

Command Procedure.

Status

Not implemented.

§56.2 THE SET CHARACTER DIRECTIVE

Purpose

Changes a volatile control character.

Format

```
*SET CHARACTER Char_name=Char
```

Required Parameters

Char_name — The name of the control character to be replaced. The most important ones (from the user's standpoint) are listed in Table 56.1. A complete list may be obtained by entering a **SHOW CHARACTERS** directive.

Char — The new control character.

Extremely important. The specification is *not* free field. There must be an equal sign immediately after *Name*, and the character must immediately follow the equal sign.

Description

The **SET CHARACTERS** changes a volatile control character. Examples of such characters are the qualifier prefix (default value = slash) and the item repetitor (default value = at-sign).

Replacement characters are not arbitrary. If you try to replace a control character by something like a letter or a number, you will get an error message and the replacement is not performed.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**. In fact it is rarely entered by an external command.

CLIP Subsystem(s) Required

None.

Status

Operational. Should be considered dangerous and will be phased out.

EXAMPLE 56.1

Have the Processor change the item-repetitor character from the default @ to * as in the old CLIP and show the replacement:

```
CALL CLPUT ('*set char repeat=* ; *show char')
```

After verifying the Processor the ***show char** directive may be removed from the message text.

Table 56.1 **Control Character Names for SET CHARACTER Directive**

Name	*Function*	*Default*
ARGBEG	Left formal-argument delimiter in procedure body	[
ARGEND	Right formal-argument delimiter in procedure body	]
DIRPRE	Directive prefix	*
ENDSRC	End-of-command-source sentinel	@
EOL1	End of line terminator #1 (also comment sentinel #1)	.
EOL2	End of line terminator #2 (also comment sentinel #2)	$
HYPHEN	Item hyphenator	--
MACBEG	Left macrosymbol delimiter	<
MACEND	Right macrosymbol delimiter	>
MACPAR	Macro definition argument marker	%
QUAPRE	Qualifier prefix	/
RECSEP	Same-line record separator	;
REPEAT	Item repetitor	@

§56.3 THE SET CPU_TIME DIRECTIVE

Purpose

Set internal CPU time stopwatch.

Format

```
*SET CPU_TIME
```

Description

The CPU system clock is read and its value stored internally. Elapsed times with respect to this reading may be printed through the SHOW CPU_TIME directive

Processor Reference

This directive may be submitted through the message entry point CLPUT.

Operational Restrictions

Operational on VAX/VMS and Unix-based systems.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 56.2
Show CPU time spent running NICE Processor SKYPUL:

```
*set cpu
*run nice$exe:skypul
*show cpu
```

§56.4 THE SET ECHO DIRECTIVE

Purpose

Set dataline echo options.

Format

```
*SET ECHO [ = Option_list]
```

Abbreviation

*SET ECHO may be abbreviated to *ECHO.

Optional Parameters

Option_list — A list of keywords that specifies echo options, as shown in Table 56.2.

If no option keys are given the default settings indicated in Table 56.3 are put in effect. These settings are those in force at run start.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 56.3

```
*SET ECHO = ON,VERBOSE,MACRO
```

Table 56.2. **Options for SET ECHO Directive**

Option	*Effect*
CONCISE	Set "concise mode" thus negating the effect of a previous VERBOSE option.
DECODE	Print the Decoded Item Table after each ordinary command. (Equivalent to entering a *SHOW DEC after each command.)
DSPACE	Double space data line echoprint (if echo is ON)
MACRO	Display final effect of macro substitution (if echo is ON)
MDETAIL	Display all stages of macro substitution (if echo is ON)
NDECODE	Turn off the effect of a previous DECODE option.
NHEADER	Suppress print of data line header (if echo is ON).
ON	Turn on data line echo.
OFF	Turn off data line echo.
TERMINAL	Give immediate "terminal echo" of data lines.
VERBOSE	Turn on "verbose mode" so that splash lines supplied in the second argument of CLREAD or CLNEXT (Volume III) are shown when terminal input is requested.

Table 56.3. **Default Echo Settings**

Run Mode	*Defaults at Runstart*
Batch	Data line images are immediately echoed, single spaced, on the CLIP print file. The image echo is preceded by a short header. Procedure line images with substituted arguments are printed. Effects of macro substitution are not printed. Decoded Item Table is not printed after each command.
Interactive	No echoprint of any form.

§56.5 THE SET ERROR_TRACE DIRECTIVE

Purpose

Set NICE-DMS Error Trace display options.

Format

```
*SET ERROR TRACE [ = Option_list]
```

Optional Parameters

Option_list Format to be determined.

Description

This directive sets error processing options pertaining to NICE-DMS.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Experimental; format subject to change.

§56.6 THE SET HELP_FILE DIRECTIVE

Purpose

Set current help file.

Format

```
*SET HELP_FILE [= Filename]
```

Abbreviation

*SET HELP_FILE may be abbreviated to *HFILE.

Optional Parameters

Filename	The name of the help file. If omitted, the help file for CLIP directives is assumed.

Description

This directive sets the help file that will be accessed by subsequent HELP directives. Proper operation depends on installation options.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

§56.7 THE SET MODE DIRECTIVE

Purpose

Sets mode parameters that affect command processing.

Format

```
*SET MODE = [Mode_key]
```

Optional Parameters

Mode_key	One of the keywords listed in Table 56.4.
	If no keyword is given, all default modes are set.

Description

This directive sets certain parameters that affect certain aspects of command processing.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Experimental; format subject to change.

Table 56.4 **Keywords for SET MODE Directive**

Mode_key	*Effect*	*Default setting*
EMPTY	Accept empty input lines that return a zero item count	Ignore empty input lines
LC	Same as NOUPCASE	
MACROPROCESSOR	Not implemented	
NOUPCASE	Suppress automatic conversion of all unprotected strings to upper case	Convert lower to upper
NOMACRO	Suppress macrosymbol expansion	Expand macrosymbols

§56.8 THE SET PROCEDURE_LIBRARY DIRECTIVE

Purpose

Identifies residence of callable procedure elements.

Format

```
*SET PLIB = [ldi [,Dataset]]
```

*SET PROCEDURE_LIBRARY is abbreviated to *SET PLIB.

Optional Parameters

ldi

If nonzero, Logical Device Index of procedure library in which Callable procedure elements (CPEs) for subsequent CALL or PROCEDURE directives are assumed to reside.

If zero (or if the entire specification is omitted), residence is on ordinary files (default setting).

Refer to §5 for details.

Dataset_name

If *ldi* is nonzero and points to a nominal (GAL82) library, name of the owner dataset in which callable procedure elements are assumed to reside.

Description

The directive sets the residence of callable procedure elements. Refer to §5 for details.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

Command Procedure and NICE-DMS Interface for data library residence.

Status

Operational.

§56.9 THE SET RUN_MODE DIRECTIVE

Purpose

Sets the run mode environment overriding detected mode.

Format

```
*SET RUN_MODE = { BATCH | INTERACTIVE | CONVERSATIONAL }
```

Required Parameters

BATCH	Declares a batch environment.
INTERACTIVE	Declares interactive environment.
CONVERSATIONAL	Declares conversational environment.

Description

The SET RUN_MODE directive overrides the environment detected by CLIP on first entry.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

§56.10 THE SET TERMINAL DIRECTIVE

Purpose

Sets characteristics of your terminal device.

Format

```
*SET TERMINAL = Options
```

CLIP Subsystem(s) Required

None.

Status

Not implemented.

§56.11 THE SET UNIT DIRECTIVE

Purpose

Set a CLIP logical unit.

Format

```
*SET UNIT Unit_name [ = unit]
```

Required Parameters

Unit_name Keyword that identifies the unit. See Table 56.5.

Optional Parameters

unit New unit number. If omitted, the default value is set.

Description

The SET UNIT directive resets the value of a logical unit used by CLIP. These values may be displayed through the SHOW UNITS directive.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

Table 56.5 **Logical Unit Names for SET UNIT Directive**

Argument key	*Description*	*Default value*
CIN	Logical unit number of the command source file from which CLIP is reading data lines. If zero, the default input device is assumed.	0
ECH	Logical unit number of dataline echo file.	6
ERR	Logical unit number of the error print file if greater than zero. If zero, error messages go to the default print file (the terminal in interactive mode).	0
LOG	Logical unit number of the command log file if one is currently open, otherwise zero.	0
PRT	Logical unit number of the bulk print file.	6

§56.12 THE SET VIDEO DIRECTIVE

Purpose

Sets attributes that affect display appearance on video terminals.

Format

```
*SET VIDEO = Options
```

CLIP Subsystem(s) Required

None.

Status

Not implemented.

§56.13 THE SET WIDTH DIRECTIVE

Purpose

Sets the maximum line input width or line print width.

Format

```
*SET {LIW | LPW } = width
```

Abbreviations

*SET WIDTH LIW must be abbreviated to *SET LIW and *SET WIDTH LPW must be abbreviated to *SET LPW.

Required Parameters

LIW	To set Line Input Width.
LPW	To set Line Print Width.
width	Width in characters. If value is outside admissible range, the closest legal value is assumed.

Description

The SET WIDTH directive may be used to reset the maximum input line width and maximum line print width over the default values. The default values depend on run mode and computer host, but normally are (80,80) for interactive mode and (80,128) for batch mode.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

§56.14 THE SET WINDOWS DIRECTIVE

Purpose

Show attributes that affect screen-windowing displays.

Format

```
*SET WINDOWS Key [= Option_list ]
```

CLIP Subsystem(s) Required

None.

Status

Not implemented.

57

SHOW

§57.1 THE SHOW ARGUMENTS DIRECTIVE

Purpose

List procedure arguments and their replacement text.

Format

```
*SHOW ARGUMENTS [/OUT=unit]
```

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

The SHOW ARGUMENTS directive lists the formal arguments and replacement text of the command procedure pertaining to the present procedure level. If the procedural level is zero this directive is ignored.

Operational Restrictions

Only operates if the command stream is at procedural level 1 or higher. Normally inserted in the body of the command procedure itself, but works even in script files (or terminals) ADDed by the procedure.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

Command Procedure.

Status

Operational.

EXAMPLE 57.1

List arguments of procedure DO on entry to it:

```
*proc do (n1=1;n2=4;n3=1)
*show arguments
     ...
*end
```

EXAMPLE 57.2

This is a bit fancier. A procedure queries the terminal on entry:

```
*proc solve (input ; output; options)
"SOLVE entered.  Any instructions, boss?  "
     ...
*end
```

If you type ***SHOW ARG** in response to the prompt, you will be shown arguments **INPUT, OUTPUT** and **OPTIONS**.

§57.2 THE SHOW CHARACTERS DIRECTIVE

Purpose

Show CLIP control characters.

Format

```
*SHOW CHARACTERS [/OUT=unit]
```

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

The **SHOW CHARACTERS** directive displays CLIP's volatile control characters and their present values. Examples of such characters are the qualifier prefix (default value = slash) and the item repetitor (default value = at-sign). These characters may be reset using the **SET CHARACTERS** directive, and **SHOW CHARACTERS** provides a way to check whether the reset has been performed correctly.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 57.3

Have the Processor change the item-repetitor character from the default @ to * as in the old CLIP and show the replacement:

```
CALL CLPUT ('*set char repeat=* ; *show char')
```

After verifying the Processor the ***SHOW CHAR** continuation record may be removed.

C-4

§57.3 THE SHOW COMMAND_SOURCE_STACK DIRECTIVE

Purpose

Show the configuration of the Command Source Stack.

Format

```
*SHOW CSS [/OUT=unit]
```

Abbreviation

*SHOW COMMAND_SOURCE_STACK is abbreviated to *SHOW CSS.

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

This directive shows the configuration of the command source stack (CSS) described in §4 of Volume I. The output format is illustrated by the following example.

```
<CL> Command Source Stack:
     Un   Lin   Rec Ldi Dsn Rap   Loc E  Name
      0     5     0   0   0   0     0 0  $term
     35     2     0   0   0   0     0 0  CALLDO.ADD
     -1    25    43   1   1   1   647 0  DO
     -1     2    20   1   1   2  1647 0  INNER
```

The meaning of the columns is

Un	If >0, logical unit number of source file. Zero means the standard input source (your terminal if run is interactive). Negative means that input comes from a data library whose LDI is the absolute value of that shown.
Lin	Number of last line read from source.
Rec	Meaningful only if reading from a data library. If so, it shows the record number of the last line read.
Ldi	If reading from a data library, its LDI.
Dsn	If reading from a positional data library, sequence number of Text Dataset. If reading from a nominal data library, sequence number of dataset that owns the Text Group. Otherwise zero.
Rap	If reading from a nominal data library, record access packet index of Text Group, otherwise zero.

Loc	If input comes from a data library, device location of Text Dataset or Text Group start.
E	1 if source activated using *add/eof, else zero.
Name	The name of the command source. If the source is unit zero, the source is either $term or $root, as explained in Volume I. If reading from a FORTRAN file, positional library, or nominal library, the name is the file name, Text Dataset name, or Text Group name, respectively.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 57.4

If procedure level is 3 or higher, display command source stack.

```
*if < <cslevel> /ge 3> /then
*show css
*endif
```

§57.4 THE SHOW CPU_TIME DIRECTIVE

Purpose

Show CPU time elapsed since previous SET CPU.

Format

```
*SHOW CPU_TIME [/OUT=unit]
```

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

The CPU system clock is read. The time is converted to seconds if necessary, subtracted from the internal stopwatch time, and printed.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

Operational Restrictions

Operational on VAX/VMS and Unix-based systems.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 57.5
Show CPU time spent running NICE Processor SKYPUL.

```
*set cpu
*run nice$exe:skypul
*show cpu
```

§57.5 THE SHOW DECODED_ITEMS DIRECTIVE

Purpose

Show Decoded Item Table.

Format

```
*SHOW DECODED_ITEMS [/OUT=unit]
```

Abbreviation

*SHOW DECODED_ITEMS may be abbreviated to *DEC.

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

This directive shows the present contents of the Decoded Item Table described in Volume III. This table contains the results of the processing of the last *ordinary* command acquired as a result of a Processor reference to CLREAD or CLNEXT.

It is an invaluable command for interface debugging. Consequently, there is the abbreviated form *DEC. An echo mode in which this table is printed after each ordinary command can be set through a SET ECHO directive.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

§57.6 THE SHOW ECHO DIRECTIVE

Purpose

Show command echo settings.

Format

```
*SHOW ECHO [/OUT=unit]
```

CLIP Subsystem(s) Required

None.

Status

Not implemented.

§57.7 THE SHOW ERROR_TRACE_STACK DIRECTIVE

Purpose

Show Error Trace Stack of NICE-DMS.

Format

```
*SHOW ERROR_TRACE_STACK [/OUT=unit]
```

Abbreviation

*SHOW ERROR_TRACE_STACK may be abbreviated to *SHOW ETS.

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

This directive shows the present contents of the Error Trace Stack (ETS) of NICE-DMS. The ETS records the sequence of internal calls in the last reference to a GAL-DMS entry point. The last trace line identifies the Processor subroutine that issued the call. It is primarily useful after an error condition has been reported by GAL-DMS or DMGASP.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 57.6

An error message occurs after an OPEN.

```
*open input.lib
(error message)
*show ets
```

§57.8 THE SHOW HELP DIRECTIVE

Purpose

Show current help file and video parameters.

Format

```
*SHOW HELP [/OUT=unit]
```

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

This directive shows the present values of parameters that affect help file access and display. These parameters are the current help file name, windowing and video settings.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

§57.9 THE SHOW LIBRARIES DIRECTIVE

Purpose

Show active data libraries.

Format

```
*SHOW LIBRARIES [/OUT=unit]
```

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

This directive causes a list of all data libraries presently active to be listed. The list includes Logical Device Index (LDI), library form (i.e., DAL, GAL80 or GAL82) and file name.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

§57.10 THE SHOW LOGICAL_DEVICE_TABLE DIRECTIVE

Purpose

Print Logical Device Table of the I/O Manager DMGASP.

Format

```
*SHOW LOGICAL_DEVICE_TABLE [/FULL] [/OUT=unit]
```

Abbreviation

*SHOW LOGICAL_DEVICE_TABLE may be abbreviated to *SHOW LDT.

Word Qualifiers

FULL	Show complete table, including all legal devices. If omitted, only the active devices are shown.

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

In response to a SHOW LOGICAL_DEVICE_TABLE directive, CLIP calls the I/O Manager DMGASP with a list-LDT request. This directive is primarily used for I/O debugging and for program optimization, since the I/O Manager level is not for the casual user.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

§57.11 THE SHOW MACROSYMBOLS DIRECTIVE

Purpose

Show defined macrosymbol(s).

Format

```
*SHOW MACROSYMBOLS [Name] [/BUILT_IN] [/OUT=unit]
```

Optional Parameters

Name — Restrict print to macrosymbols that match this name.

A trailing asterisk may be used for name masking and is useful for printing macrosymbol arrays.

If *Name* is omitted and qualifier *BUILT_IN* is missing (given), all user-defined (built-in) macrosymbols are shown.

Word Qualifiers

`BUILT_IN` — Restrict print to built-in macros. If this qualifier is omitted, print is restricted to user-defined macros.

Phrase Qualifiers

`OUT=`*unit* — Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

In response to a `SHOW LOGICAL_DEVICE_TABLE` directive, CLIP calls the I/O Manager DM-GASP with a list-LDT request. This directive is primarily used for I/O debugging and for program optimization, since the I/O Manager level is not for the casual user.

Processor Reference

This directive may be submitted through the message entry point `CLPUT`.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

EXAMPLE 57.7

Show all user-defined macrosymbols.

```
*SHOW MAC
```

Show all built-in macrosymbols.

```
*SHOW MAC/B
```

EXAMPLE 57.8

Show all user-defined macrosymbols that begin with **X**

```
*SHOW MAC X*
```

EXAMPLE 57.9

Show all entries of macrosymbol array **FIBO**:

```
*SHOW MAC FIBO[*
```

§57.12 THE SHOW MODE DIRECTIVE

Purpose

Show CLIP modal parameters.

Format

```
*SHOW MODES [/OUT=unit]
```

Description

The modal parameters shown by this directive are those that may be set through the **SET MODE** directive. These parameters affect certain facets of command processing.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**.

CLIP Subsystem(s) Required

None.

Status

Experimental.

§57.13 THE SHOW PROCEDURE_LIBRARY DIRECTIVE

Purpose

Show callable procedure library.

Format

```
*SHOW PROCEDURE_LIBRARY [/OUT=unit]
```

Abbreviation

*SHOW PROCEDURE_LIBRARY may be abbreviated to *SHOW PLIB.

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

The directive shows the presently assumed callable procedure library. This library is identified by two components: a Logical Device Index (LDI) and a optionally a dataset name.

If the LDI is greater than zero, it points to a data library connected to that LDI. If the library is nominal, the dataset name points to the owner of the callable procedure elements stored as Text Groups. If the library is positional, the dataset specification is ignored.

If the LDI is zero, the callable procedure elements reside on ordinary files. This is the default on Processor start. The pointers may be changed through the SET PROCEDURE_LIBRARY directive.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

Command Procedure for data library residence.

Status

Operational.

§57.14 THE SHOW PROCESS_NAME_STACK DIRECTIVE

Purpose

Show configuration of Process Name Stack of SuperClip.

Format

```
*SHOW PROCESS_NAME_STACK [/OUT=unit]
```

Abbreviation

*SHOW PROCESS_NAME_STACK may be abbreviated to *SHOW PNS.

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

The directive shows the present configuration of the Process Name Stack (PNS) maintained by SuperClip.

Operational Restrictions

Available only on VAX/VMS.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

SuperClip.

Status

Operational.

REMARK 57.1

Do not confuse with **SHOW MODES**, which pertains to command processing modes. **SHOW RUN_MODE** pertains to the Processor execution environment.

§57.15 THE SHOW RUN_MODE DIRECTIVE

Purpose

Show the run mode.

Format

```
*SHOW RUN_MODE [/OUT=unit]
```

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

The SHOW RUN_MODE directive shows whether the run is batch or interactive.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

§57.16 THE SHOW TERMINAL DIRECTIVE

Purpose

Show characteristics of your terminal device.

Format

```
*SHOW TERMINAL [/OUT=unit]
```

CLIP Subsystem(s) Required

None.

Status

Not implemented.

§57.17 THE SHOW TIME DIRECTIVE

Purpose

Show current date and time of day.

Format

```
*SHOW TIME [/OUT=unit]
```

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

The SHOW TIME directive shows the date and time of day as read from the system clock.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

§57.18 THE SHOW VIDEO DIRECTIVE

Purpose

Show attributes that affect display appearance on video terminals.

Format

```
*SHOW VIDEO [/OUT=unit]
```

CLIP Subsystem(s) Required

None.

Status

Not implemented.

§57.19 THE SHOW UNITS DIRECTIVE

Purpose

List CLIP logical units.

Format

```
*SHOW UNITS [/OUT=unit]
```

Phrase Qualifiers

`OUT=`*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

The `SHOW UNITS` directive lists the logical units accessible to CLIP, the value of which may be reset through a `SET UNIT` directive. The display format is as illustrated below.

```
<CL> Units: Add:  0, Cin:  0, Cpr:  6, Ech:  6, Err:  0, Hpr:  0
            Lis: 33, Log:  0, Plt:  6, Prt:  6, Qin:  0, Qlo:  0
```

where the meaning of the labels is given in the following table.

Label	*Unit for*	*Description*
`Add`	Add file	For connection to script files
`Cin`	CLIP input	Active unit in command source stack
`Cpr`	CLIP print	Receives nonspecial CLIP print output
`Ech`	CLIP echo	Receives dataline echo if on
`Err`	Error file	Receives error messages; 0 =terminal
`Hpr`	Help file	Receives help file listing; 0 =terminal
`Lis`	Roving list file	Available for `LIST` and `TYPE`
`Log`	Logging file	Receives command logging activated by `LOG`
`Qin`	Query-input file	Advanced applications
`Qlo`	Query-log file	Advanced applications

Processor Reference

This directive may be submitted through the message entry point `CLPUT`.

CLIP Subsystem(s) Required

None.

Status

Operational.

§57.20 THE SHOW WIDTH DIRECTIVE

Purpose

Shows the maximum line input and print widths in effect.

Format

```
*SHOW LIW[/OUT=unit]
```

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

The SHOW LINE_INPUT_WIDTH directive prints the value of the maximum width of CLIP input lines. By default the width is 80 but may be reset using SET LINE_INPUT_WIDTH directive to a value in the range 60 to 132. (Procedure line width is not subjected to change, however; it is always 80.)

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

§57.21 THE SHOW WINDOWS DIRECTIVE

Purpose

Show attributes that affect screen-windowing displays.

Format

```
*SHOW WINDOWS [/OUT=unit]
```

CLIP Subsystem(s) Required

None.

Status

Not implemented.

REMARK 57.2
Primary applications will be on bit-mapped workstations.

§57.22 I/O DEBUGGING

Purpose

Show DMGASP data structures for debugging or code optimization.

Format

```
*SHOW IOM Entity
```

Required Parameters

Entity		One of the following keywords:
	OSD	Show operation status descriptors.
	PIO	Show Paged I/O statistics.
	PBT	Show Page Buffer Table for Paged I/O.
	PKT	Show Block I/O Packet (a communications area).

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Description

These directives are not for ordinary users since a deep knowledge of the I/O Manager level of NICE-DMS is required for proper interpretation.

Operational Restrictions

Entity = PKT, are only available if Block I/O is implemented. *Entity* = PIO and PBT are only available if Paged I/O is implemented.

Processor Reference

These directives may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

58

SPAWN

§58.1 THE SPAWN DIRECTIVE

Purpose

Spawns a subprocess.

Format

```
*SPAWN [System_command]
```

Optional Parameters

System_command — A VAX/VMS DCL command (without the dollar sign). This command will be executed by the spawned process and control will then return to the parent process. If omitted, the spawned process will await your commands; to get back to the parent process you will have to log out.

Description

The spawned subprocess is initiated by a call to **LIB$SPAWN** with the system command, if any, as argument.

Dataline Restrictions

This directive must be in a dataline by itself.

CLIP Subsystem(s) Required

None.

Status

Operational.

EXAMPLE 58.1

Spawn a subprocess to edit the file **TEXT.DAT**:

```
*SPAWN EDT TEXT.DAT
```

EXAMPLE 58.2

Spawn a subprocess that will receive several DCL commands:

```
*SPAWN
```

To get back, you type **LOG**.

59

STOP

§59.1 THE STOP DIRECTIVE

Purpose

Stops RUN-initiated execution and restarts the parent Processor.

Format

```
*STOP
```

Description

Refer to §10 for functional and implementation details.

Operational Restrictions

Applicable if Processor being interrupted was initiated through a RUN directive. Otherwise the STOP produces a normal run stop.

Dataline Restrictions

This directive must be in a dataline by itself.

CLIP Subsystem(s) Required

SuperCLIP.

Status

Operational.

60

TYPE

§60.1 THE TYPE FILE DIRECTIVE

Purpose

List card-image file on the terminal.

Format

```
*TYPE Filename [/HEAD]
```

Required Parameters

Filename — The name of the card-image file to be listed. Masking is not permitted.

Word Qualifiers

HEAD — Write a header line that gives the name of the file.

Description

The TYPE-file directive is analogous to the LIST-file directive but output goes to the user's terminal. If running on VT100 or VT100-compatible terminal, line images are shown in reverse video.

Operational Restrictions

Same as for LIST.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

None.

Status

Operational.

REMARK 60.1

The print record length is normally limited to 80 characters. This is the default line print width, which may be increased up to 132 characters, if desired, through the SET WIDTH directive.

EXAMPLE 60.1

```
*TYPE PROC:FORPRC.MSC
```

File PROC:FORPRC.MSC (a VAX filename) is to be listed on the terminal.

§60.2 THE TYPE TEXT DATASET DIRECTIVE

Purpose

List Text Dataset on terminal.

Format

```
*TYPE Text_dataset [/HEAD]
```

Required Parameters

Text_dataset — Identifies the Text Dataset(s) to be listed. See the LIST dataset directive for details.

Word Qualifiers

HEAD — Write a header line that identifies the Text Dataset.

Description

The TYPE directive is analogous to the LIST Text Dataset directive with two differences: the output always goes to the user's terminal if running in interactive mode, and reverse video is used if the terminal is VT100 or VT100 compatible.

Operational Restrictions

Same as for LIST.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required NICE-DMS Interface.

Status

Operational.

EXAMPLE 60.2

```
*TYPE/H 1,4
*TYPE/H 1,*
```

§60.3 THE TYPE TEXT GROUP DIRECTIVE

Purpose

List Text Group on terminal.

Format

```
*TYPE Text_group [/HEAD]
```

Required Parameters

Text_group — Identifies the Text Group(s) to be listed. Refer to the **LIST** Text Group directive for details.

Word Qualifiers

HEAD — Write a header line that identifies the Text Group.

Description

The **TYPE** Text Group directive is analogous to the Text Group directive with two differences: the output always goes to the user's terminal if running in interactive mode, and reverse video is used if the terminal is VT100 or VT100 compatible.

Operational Restrictions

Same as for **LIST**.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

EXAMPLE 60.3

```
*TYPE/H 1,4,CONTENTS
 *TYPE 2,*,ABSTRACT
```

61

UNDEFINE

§61.1 THE UNDEFINE DIRECTIVE

Purpose

Delete macrosymbol(s).

Format

```
*UNDEFINE [/GLOBAL] Name_list
```

Required Parameters

Name_list — A list of up to eight macrosymbol names to be removed from the macrosymbol name table. Masking characters may be used to on any of these names to specify removal of name-related macrosymbols. For example,

```
*UNDEF XINV*
```

removes all symbols whose name starts with XINV whereas

```
*UNDEF *
```

removes all names within the scope of this directive.

To undefine macrosymbol arrays a masking specification is needed (see Example below). Using a range specification would be more logical, but it has not been implemented.

Word qualifier

GLOBAL — Delete up to the global level. If this qualifier is omitted, macrosymbols under the current procedural level are not deleted.

Description

This directive, complementary to DEFINE, removes macrosymbols from the macro tables. The removal is specified by name and scope.

Processor Reference

This directive may be submitted through the message entry point CLPUT.

CLIP Subsystem(s) Required

Macrosymbol.

Status

Operational except for GLOBAL qualifier.

REMARK 61.1

This directive should not be applied to DO loop control variables (directly, or indirectly using masking), for execution of the loop may be affected in strange ways.

EXAMPLE 61.1

Undefine macrosymbols **ALPHA**, **BETA** and **GAMMA** at the current procedural level:

```
*UNDEFINE ALPHA,BETA,GAMMA
```

EXAMPLE 61.2

Undefine all macrosymbols that begin with **X** at all levels:

```
*UND/G X*
```

EXAMPLE 61.3

Undefine all entries of macrosymbol array **LIST**:

```
*UND LIST[*
```

THIS PAGE LEFT BLANK INTENTIONALLY.

62

UNLOAD

§62.1 THE UNLOAD DIRECTIVE

Purpose

Externalizes nominal records from data library to ASCII file.

Format

```
*UNLOAD unit = ldi [,Dataset_id [,Record_id]]
```

Required Parameters

unit — Logical unit number of FORTRAN text file that will receive the unloaded records. The file must be opened new before the directive is issued. The unit-file connection is usually performed by an **FOPEN** directive (see examples). If the file is connected but is not a new file, I/O errors will likely result as the file is written sequentially.

ldi — Logical Device Index (LDI) of the source library that will receive that supplies the unloading data. Must be open at the time the directive is issued and be of GAL82 (nominal) format.

Optional Parameters

Dataset_id — A generally masked dataset name that may be used to select datasets to be unloaded. If omitted, all datasets in the load file are not filtered.

Record_id — A generally masked record name that may be used to select records to be unloaded. If omitted, records names are not filtered.

Description

The source data library is scanned dataset by dataset. For each active dataset, CLIP tests on whether it is to be unloaded (in case selective unloading is specified by giving a *Dataset_id*). If so, appropriate header data are written to the receiving file, and all records that belong to that dataset are examined. If a record is to be unloaded, appropriate information is written to the file, followed by the character encoded values of the record items. The process continues until the data library is exhausted.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**.

CLIP Subsystem(s) Required

NICE-DMS Interface.

Status

Operational.

EXAMPLE 62.1

Unload the complete data library **RESPONSE.LIB** to file **RESPONSE.VAL**:

```
*open 4,response.lib
*fopen/new 12,response.val
*unload 12 = 4
*fclose 12
*type response.val
```

The use of 4 and 12 for LDI and logical unit, respectively, is incidental. The **TYPE** directive is used to view the contents of the unload file; this has to be preceded by a ***close** command. Another way of viewing the data is using **FREWIND** and **FPRINT**:

```
*open 4,response.lib
*fopen/new 12,response.val
*unload 12 = 4
*frew 12 ; *fprint 12,100000
```

EXAMPLE 62.2

As above, but unload only datasets whose name starts with **H**:

```
*open 4,response.lib
*fopen/old 12,response.val
*load 12 = 4,H*
*fclose 12
*type response.val
```

EXAMPLE 62.3

As above, but unload only the complete record group **HISTORY** of dataset **VELOCITIES**:

```
*open 4,response.lib
*fopen/old 12,response.val
*load 12 = 4,velocities,history.l:h
*fclose 12
*type response.val
```

THIS PAGE LEFT BLANK INTENTIONALLY.

63

WALLOCATE

§63.1 THE WALLOCATE DIRECTIVE

Purpose

Allocate scratch (unbacked) workrecord(s).

Format

```
*WALLOCATE Workrecord [size] [/Type] [/DIM=dim]
```

Required Parameters

Workrecord — The name of the record, or record group, to be allocated in the Workpool.

size — If allocating a single record, logical (item) size of record. If omitted, one item is assumed.

If allocating a record group:

If positive, total number of items to allocate.

If negative, total numbers of items per record.

If zero, one is assumed.

Word Qualifiers

Type — One-letter data type. See Table 63.1. If omitted, integer type is assumed if the workrecord name begins with one of the letters I through N, single floating otherwise.

Phrase Qualifiers

DIM=*dim* — Specifies first matrix dimension. Not presently implemented.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

The allocate operation enters the workrecord name in the Workpool, reserves space to hold items, installs protection keywords, and initializes the reserved space to zero or blank, depending on whether the group is of numeric or character type, respectively. Records allocated by this directive are of scratch status; that is, not backed up by database copies. The allocation may be removed by entering a WDEALLOCATE directive.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Operational.

EXAMPLE 63.1

Allocate floating record TEMP with 250 items:

```
*WALL TEMP 250
```

Table 63.1. **Workrecord Type Identifiers**

Type	*Meaning*
A	Character string (stored as Hollerith).
D	Double-precision floating point.
F or S	Single-precision floating point.
I	Integer.

64

WCHANGE

§64.1 THE WCHANGE DIRECTIVE

Purpose

Change logical size of workrecord.

Format

```
*WCHANGE Workkey = [size] [/BEGIN=item]
```

Required Parameters

Workkey	The key of the workrecord or workgroup whose logical size is to be changed.
size	New logical record size.

Phrase Qualifiers

BEGIN=*item*	If the new size is bigger, start new space allocation at *item+1*; *item* = 0 is acceptable. If this phrase does not appear, the new allocation is appended to the existing record.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

The change operation modifies the logical size of a workrecord or workgroup. The new size may be smaller or larger than the previous size. If larger, space is appended to the record or records, unless the BEGIN qualifier specifies a different location. If smaller, the record tail is truncated. The operation will in general change the pool location of all following records.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Operational.

EXAMPLE 64.1

Change the size of each record of workgroup GPLOT.10:40 to 125 items:

```
*WCHA TEMP = 125
```

65
WCLOSE

§65.1 THE WCLOSE DIRECTIVE

Purpose

Close backup workrecord.

Format

```
*WCLOSE Workrecord
```

Required Parameters

Workkey	The name of the workrecord or workgroup to be closed. May contain backing characters.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

The close operation is intended for backed workrecords. Each record marked as modified is written back to the backing library, then it is deallocated from the Workpool. A close operation applied to a scratch (unbacked) record functions as a deallocate operation.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Not implemented.

EXAMPLE 65.1

Close all records that begin with G: 125 items:

```
*WCLOSE G*
```

66

WDEALLOCATE

§66.1 THE WDEALLOCATE DIRECTIVE

Purpose

Reclaim storage used by workrecord.

Format

```
*WDEALLOCATE Workrecord
```

Required Parameters

Workkey The name of the workrecord or workgroup to be deallocated.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

The deallocate operation differs from the close operation in that no backup operation is performed. The storage taken by the workrecord(s) is released to the Workpool. It is primarily intended for scratch workrecords.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Operational.

EXAMPLE 66.1

Deallocate all records that begin with G: 125 items:

```
*WDEALLOCATE G*
```

67

WDEFINE

§67.1 THE WDEFINE DIRECTIVE

Purpose

Define macrosymbol(s) from workrecord values.

Format

```
*WDEFINE [/Type] Macro_name = Workrecord
```

Required Parameters

Macro_name	The name of the macrosymbol or macrosymbol array to be defined. Same rules as for the DEFINE directive.
Workrecord	The workrecord whose items provide values for the macro definition. Item selection may be specified with indexing within square brackets.

Word Qualifiers

Type	Macro type; same as for the DEFINE directive except for default rule. If omitted, the workrecord type determines the macro type.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

The define operation defines a macrosymbol or a macrosymbol array. It differs from the DEFINE directive in that the values are extracted from a workrecord rather than being specified in the directive itself.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Operational.

EXAMPLE 67.1

Define float macro **WORK** and take the 3rd item of workrecord **TRABAJO.6** as its value:

EXAMPLE 67.2

Define macrosymbol array **VEC** using all values in workrecord **V**:

THIS PAGE LEFT BLANK INTENTIONALLY.

67a
WDIMENSION

§67a.1 THE WDIMENSION DIRECTIVE

Purpose

Change the first matrix dimension of a workrecord.

Format

```
*WDIMENSION Workrecord = [dim]
```

Required Parameters

Workrecord	The name of the record, or record group, whose first matrix dimension is to be changed.
dim	The new first matrix dimension.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

The dimension operation modifies the first matrix dimension size of a workrecord or workrecord group. The new dimension may be smaller or larger than the previous dimension, with a minimum value of one.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Operational.

EXAMPLE 67a.1

Change the first matrix dimension of record TEMP to 50:

```
*WDIM  TEMP = 50
```

THIS PAGE LEFT BLANK INTENTIONALLY.

68

WFLUSH

§68.1 THE WFLUSH DIRECTIVE

Purpose

Backup modified non-scratch workrecords.

Format

```
*WFLUSH Workrecord
```

Required Parameters

Workrecord	Name of workrecords to be flushed. Name masking is allowed and frequently used.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

The workrecords specified in the directive are examined one by one. If a record is backed and modified, it is written to the database and the modified flag cleared.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Not implemented.

EXAMPLE 68.1

Flush all modified backed records in the Workpool:

```
*WFLU *
```

69

WGET

§69.1 THE WGET DIRECTIVE

Purpose

Read database record(s) into workrecord(s).

Format

```
*WGET Workrecord = Record_id
```

Required Parameters

Workrecord	Name of the workrecord or workgroup that will receive the record(s).
Record_id	A nominal record specification of the form *ldi, dsn, Record_name*

If the record name is not specified, the workrecord name is assumed.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

The database records specified in the directive are accessed using GMGETC ot GMGETN and read into the specified workrecord locations.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Operational.

EXAMPLE 69.1

```
*WGET X = 4,16,XXX[4:8]
```

70

WHILE

§70.1 THE WHILE DIRECTIVE

Purpose

Introduces a WHILE-DO block.

Format

```
*WHILE logical_expression /DO
```

Required Parameters

logical_expression An expression that evaluates to integer 0 for FALSE or 1 for TRUE. More generally, a nonzero value is also interpreted as TRUE. The expression is usually constructed through ordinary or logical macrosymbols (see §4 and examples).

If the expression is TRUE, the commands that follows the WHILE directive are executed; when the matching ENDWHILE is reached, control jumps back to the WHILE directive for another test. If the expression is FALSE, control passes to the command that follows the matching ENDWHILE.

Description

A WHILE directive introduces a WHILE-DO block. The block must be terminated by a matching ENDWHILE directive. Execution of the intervening commands is contingent upon the value of the logical expression written in the WHILE line.

The procedure compiler transforms a WHILE line into a labeled IF directive. The label is generated and assigned to the line that immediately follows the matching ENDWHILE. When the compiler reaches the ENDWHILE, it generates a jump back to another generated label that points to the WHILE line.

The WHILE-DO body may contain IF-THEN-ELSE blocks, DO loops and other WHILE-DO blocks, as long as they are properly nested.

Dataline Restrictions

This directive must be in a dataline by itself. The DO qualifier must be in the same line as the WHILE. If the *logical_expression* is so long that it requires continuation lines, you should place the DO immediately after the WHILE.

Operational Restrictions

Works only inside a command procedure.

Processor Reference

Not applicable.

CLIP Subsystem(s) Required

Command Procedure.

Status

Operational.

EXAMPLE 70.1

The following illustrates a typical application in iterative problem solving:

```
*DEF tolerance = "Enter tolerance:  "
*DEF error = 1.E+10
*WHILE < <error> /gt <tolerance> > /THEN
  *CALL ITERATE (solution=<x> ; residual=<error>)
*ENDWHILE
```

THIS PAGE LEFT BLANK INTENTIONALLY.

71

WMAP

§71.1 THE WMAP DIRECTIVE

Purpose

Give a allocation map of the Workpool.

Format

```
*WMAP [/BACK]
```

Work Qualifiers

BACK	List backup information.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

This directive lists the workrecords that are presently in the Workpool, and the attributes of these workrecords.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Operational.

72

WMARK

§72.1 THE WMARK DIRECTIVE

Purpose

Mark workrecord(s) as modified.

Format

```
*WMARK Workrecord
```

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

The workrecords specified in this directive are marked as modified if they are backed.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Not implemented.

73

WOPEN

§73.1 THE WOPEN DIRECTIVE

Purpose

Open database-backed workrecord(s).

Format

```
*WOPEN Workrecord [size] [/Type] = ldi, [Dataset_name[,Record_name]]
[/OLD] [/NEW] [/DIM=dim]
```

Required Parameters

Workrecord	The name of the record, or record group, to be allocated in the Workpool.
size	If allocating a single record, logical (item) size of record. If omitted, one item is assumed. If allocating a record group: If positive, total number of items to allocate. If negative, total numbers of items per record. If zero, one is assumed.
ldi,...	Database linkage information for initialization/backup.

Word Qualifiers

Type	One-letter data type. See Table 63.1. If omitted, integer type is assumed if the workrecord name begins with one of the letters I through N, single floating otherwise.
NEW	Allocate new workrecord and initialize space.
OLD	Allocate old workrecord and read contents from database. If neither OLD nor NEW is given, OLD is assumed.

Phrase Qualifiers

DIM=*dim*	Specifies first matrix dimension. Not presently implemented.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

The allocate operation enters the workrecord name in the Workpool, reserves space to hold items, installs protection keywords, and initializes the reserved space to zero if new or reads the database records if old.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Experimental.

EXAMPLE 73.1

Allocate floating record **CAMP with 250 items:**

```
*WOPEN CAMP 250 = 3, HI, CAMP /NEW
```

THIS PAGE LEFT BLANK INTENTIONALLY.

74

WPOOL

§74.1 THE WPOOL DIRECTIVE

Purpose

Redefine extent and offset of Workpool.

Format

```
*WPOOL extent [/OFFSET=offset]
```

Required Parameters

extent	New size of Workpool in machine words.

Phrase Qualifiers

OFFSET= *offset*	Offset in machine words from the start of common block. Assumed zero if not specified.

Processor Reference

This directive may be submitted through the message entry point **CLPUT**. A direct FORTRAN call is described in Volume III.

Description

This directive changes the default extent of the Workpool and possibly its offset with respect to common block start. This operation is permitted only on a virgin Workpool.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Not implemented.

EXAMPLE 74.1

Set the extent of the Workpool to 100000 words:

```
*WPOOL 100000
```

75

WPRINT

§75.1 THE WPRINT DIRECTIVE

Purpose

Print contents of workrecord(s).

Format

```
*WPRINT Workrecord [/Format] [/MATRIX] [/OUT=unit]
```

Required Parameters

Workrecord	Name of the workrecord or workgroup to be printed. Masking specifications are allowed.

Word Qualifiers

Format	An item print format identification string similar to FORTRAN; for example E12.5. If omitted, a default print format related to the workrecord type will be used.
MATRIX	Print record in matrix format using the matrix dimension attribute as number of rows. Not implemented.

Phrase Qualifiers

OUT=*unit*	Write output to logical unit *unit*. If not given, output goes to the CLIP print file (normally logical unit 6).

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

This print operation shows the contents of workrecords. The print is performed using a vector-print utility routine unless the qualifier MATRIX is specified, in which case rectangular matrix print utilities are invoked.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Operational.

EXAMPLE 75.1

Print floating-point record **QLOAD.6** in **E14.6** format:

```
*WPRINT QLOAD.6 /E14.6
```

THIS PAGE LEFT BLANK INTENTIONALLY.

76

WPUT

§76.1 THE WPUT DIRECTIVE

Purpose

Write workrecord(s) to nominal library.

Format

```
*WPUT Record_id = Workrecord
```

Required Parameters

Record_id — A nominal record specification of the form

ldi, dsn, Record_name

If the record name is omitted, the workrecord name is used.

Workrecord — Name of the workrecord or workgroup that contains the record(s) to be written.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

The workrecord(s) specified in the directive are written using GMPUTC to GMPUTN to the specified data library location.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Operational.

EXAMPLE 76.1

Write workrecord XXX.3 to dataset QDATA in library 4:

```
*WPUT 4,QDATA = XXX.3
```

EXAMPLE 76.2

Write workgroup G.3:12 to dataset sequenced 57 in library 4, and change the record key to GIGI:

```
*WPUT 4,57,GIGI = GG.3:12
```

77

WSET

§77.1 THE WSET DIRECTIVE

Purpose

Set workrecord items to specified values.

Format

```
*WSET Workrecord = value_list
```

Required Parameters

Workrecord — The name of the workrecord whose items are to be set. If no item index specification is given, values are stored beginning at the first item.

value_list — A list of numeric values. The number of values in the list determines how many items will be stored. Data type conversion is performed as necessary.

Processor Reference

This directive may be submitted through the message entry point CLPUT. A direct FORTRAN call is described in Volume III.

Description

This operation sets all or part of a workrecord to to specified values supplied in the item list.

CLIP Subsystem(s) Required

Workpool Manager.

Status

Operational.

EXAMPLE 77.1

Set the first 10 items of workrecord LIST to 1,2,3, ... 10:

```
*WSET LIST = 1:10
```

EXAMPLE 77.2

Set item 45 of workrecord ANGLES to the sine of 45 :

```
*WSET ANGLES[45] = <SIND(45)>
```

EXAMPLE 77.3

Set items 20 through 50 of workrecord PONY to the fraction one third:

```
*WSET PONY[20:50] = 31@(1/3)
```

THIS PAGE LEFT BLANK INTENTIONALLY.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-178385		

4. Title and Subtitle
The Computational Structural Mechanics Testbed Architecture
Volume II - Directives

5. Report Date
February 1989

6. Performing Organization Code

7. Author(s)
Carlos A. Felippa †

8. Performing Organization Report No.
LMSC-D878511

9. Performing Organization Name and Address
Lockheed Missiles and Space Company, Inc.
Research and Development Division
3251 Hanover Street Palo Alto, California 94304

10. Work Unit No.
505-63-01-10

11. Contract or Grant No.
NAS1-18444

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

13. Type of Report and Period Covered
Contractor Report

14. Sponsoring Agency Code

15. Supplementary Notes
† Current affiliation: Department of Aerospace Engineering and Center for Space Structures and Controls, University of Colorado, Boulder, CO 80309-0429

Langley Technical Monitor: W. Jefferson Stroud

16. Abstract
This is the second of a set of five volumes which describe the software architecture for the Computational Structural Mechanics Testbed. Derived from NICE, an integrated software system developed at Lockheed Palo Alto Research Laboratory, the architecture is composed of the command language (CLAMP), the command language interpreter (CLIP), and the data manager (GAL). Volumes I, II, and III (NASA CR's 178384, 178385, and 178386, respectively) describe CLAMP and CLIP and the CLIP-processor interface. Volumes IV and V (NASA CR's 178387 and 178388, respectively) describe GAL and its low-level I/O. CLAMP, an acronym for Command Language for Applied Mechanics Processors, is designed to control the flow of execution of processors written for NICE. Volume II describes the CLIP directives in detail. It is intended for intermediate and advanced users.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
Structural analysis software Command language interface software Data management software	Unclassified—Unlimited Subject Category 39

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	405	A18

NASA FORM 1626 OCT 86
For sale by the National Technical Information Service, Springfield, Virginia 22161-2171