

# **INTERFACE**

An interface design system

Version 2.0

Copyright (c) 1991, 1992, 1993, 1997

David A. Pearlman

All rights reserved.

## **NOTICE!**

The INTERFACE program and all accompanying documentation,  
including this manual are

Copyright (c) 1991,1992,1993, 1997  
David A. Pearlman  
All Rights Reserved.

Except as permitted under the United States Copyright Act of 1976, or by the licensing agreement under which this manual and related software were provided, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database retrieval system, without the prior written permission of the publisher.

The software described in this document is provided pursuant to a license agreement, and may only be used, copied, or otherwise disclosed in accordance with the terms of the license.

This document contains proprietary information made available only under contract to specified parties. It may not be copied, disclosed, or transferred to others without the express written permission of the author.

# Table of Contents

## Table of Contents

Copyright Notice .....	1
Table of Contents .....	2
1) About INTERFACE.....	4
I) Introduction .....	4
II) Using INTERFACE .....	6
2) Special Capabilities of Interface .....	7
I) In-Line Variable Substitution in INTERFACE.....	7
II) Data Structures in INTERFACE.....	8
III) Control structures supported by INTERFACE.....	9
IV) Writing Files in INTERFACE.....	9
V) Error Checking in INTERFACE.....	10
3) Features and Considerations .....	11
I) Input to INTERFACE.....	11
II) Creating the Interface Programming Script (IPS).....	12
III) The User Command (UC) file.....	14
IV) Case Sensitivity .....	14
V) Abbreviations .....	14
VI) Structure requirements for the IPS and UC file scripts .....	15
VII) Intrinsic commands for the IPS .....	15
VIII) Intrinsic commands for the UC file.....	15
IX) Algebraic and Character Expressions .....	16
X) Intrinsic Arithmetic Functions Provided in Interface .....	17
XI) Logical expressions .....	18
XII) In-Line Variable Substitution in INTERFACE.....	19
XIII) Internal Variable Specifiers (IVS) .....	20
Referencing Variable Ranges using IVS pointers.....	23
XIV) Variable names .....	24
XV) Rules for determining command-name matches.....	25
XVI) Changing the option/argument specification syntax in the UC file.....	26
XVII) Character storage requirements .....	27
XVIII) Numerical storage requirements .....	27
XIX) Special file_equivalence_names.....	27
XX) Speed considerations.....	28
XXI) Machine independence .....	29
4) Command Definitions.....	30
Conventions Used .....	30
Commands Recognized in both the IPS and UC files.....	32
Control Constructs .....	32
DO .....	32
DO WHILE .....	34
EXIT .....	35
GOTO.....	36
IF .....	37

## Table of Contents

Other Commands.....	39
ASSIGN .....	39
CASSIGN.....	40
GASSIGN.....	41
NVASSIGN, NVCASSIGN, NVGASSIGN.....	42
REDIRECT .....	43
STOP .....	45
VASSIGN, VCASSIGN, VGASSIGN .....	46
Commands Recognized only in the UC File.....	47
GETTEMPLATE .....	47
HELP.....	49
MANUAL .....	50
Commands Recognized only in the IPS File.....	51
ALIAS .....	51
ARGUMENTS .....	53
BOUNDS .....	58
CALLED .....	60
CLEARMEM .....	61
CLOSE .....	67
COMMAND.....	68
COPY .....	70
DEFER .....	73
DEFHELP .....	78
DEFINE.....	82
Table of values accessible through DEFINE .....	82
ECHO .....	86
EXCLUSIVE.....	89
EXTERNAL.....	91
FORMAT SPECIFICATION.....	92
MARKMEM .....	95
MAXCALL .....	96
NOT_CALLED.....	98
OPEN .....	100
File types supported by INTERFACE .....	100
OPTION .....	105
POINT .....	107
READ.....	108
REFORM .....	116
WRITE .....	118
5) Design and Debugging Tips.....	120
Common IPS Programming Mistakes.....	121
Debugging Tips .....	123
Error Report Format .....	124
Speed Considerations .....	124
Appendices .....	126
APPENDIX I) A calculator program. ....	126
APPENDIX II) Providing additional "intrinsic" commands in the UC file.....	128
APPENDIX III) A simple DELETE script	

## *Table of Contents*

Interfacing to the host operating system. ....	129
APPENDIX IV) Allowing the interactive user to change the prompt string .....	130
APPENDIX V) A command to return the names of files in a directory to the UC level.....	131
APPENDIX VI) Creating implied do-lists /multiple dimensional arrays.....	134

---

# Chapter 1

## About INTERFACE

### **I) Introduction**

A huge number of programs have been developed since the start of the computer age. Many of these perform functions targeted at a non-computer-proficient audience. Still, a good portion of these programs are used exclusively by computer-knowledgeable experts, rather than the people to whom they are targeted (and the people who would most directly benefit from using them). Why? Because these programs are often unnecessarily difficult to use. In many cases, they rely on a non-pneumonic and cumbersome "values in columns"-type formatted input. This type of input can appear daunting to the computer neophyte, and can be confusing even to the computer literate.

It is not difficult to understand why programs have been written in such a non user-friendly manner: Development of a convenient, flexible, and friendly interface for input of any complexity can be quite difficult. This is because the parsing functionalities needed for such an interface do not exist as standard functions in commonly-used programming languages; they must be built up using long runs of complex, and often application-specific, instructions. To write a user-interface of any value in such a language can often take much longer than writing the accompanying program!

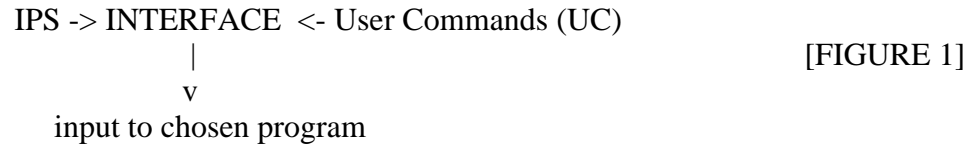
It was recognition of this problem that lead to the development of a new program, INTERFACE. The objective was to design an easy-to-use programming language, that combined intrinsic high-level parsing functionalities with various other control structures (do-loops, if...then, etc.), so that it could be generally applied to design user-friendly interfaces for programs that lacked them. Since many of the programs for which such an interface is desired have already been written, the goal was not to produce a library of subroutines to be incorporated during initial code development. Instead, a program was designed which reads the Interface Programming Script (IPS) and from the IPS generates a friendly environment for the end user. In a sense, INTERFACE and the IPS allow the programmer to define an "operating system" for specific applications. The IPS associates pneumonic command names with the various non-intuitive input otherwise required as input for a program. Since the IPS is flexible, it can also be used to write involved programs (although since it is an interpreted rather than compiled language, heavy computation using such a script is not recommended).

With two localized exceptions,<sup>1</sup> INTERFACE is written completely in ANSI-standard FORTRAN 77, which means that an environment programmed using an IPS can be reproduced identically, and without any required IPS or INTERFACE changes, on any computer with a FORTRAN compiler which adheres to the ANSI standard (provided sufficient memory is available).

---

<sup>1</sup>The two exceptions are 1) the code used to implement the “EXTERNAL” command. This command sends the passed string to the host operating system for execution, and is necessarily machine-dependent; and 2) the code used on machines supporting a Unix or VMS operating system to allow specification of input files at runtime.

The following illustrates the flow in a typical usage of INTERFACE:



The Interface Programming Script (IPS) is designed by the knowledgeable programmer, and gives the rules of the "operating system" to be implemented by INTERFACE. When INTERFACE is invoked, this script is specified and parsed. INTERFACE then reads input from the user (UC). This input can either be specified interactively or read from a file, as a series of commands. In either case, each command is sequentially parsed according to the rules set up in the IPS. The IPS also specifies what action(s) are to be taken for each command, what arguments should accompany the command, etc. Actions taken based on a command read at the User Command level may include reads and writes to any chosen file(s), numerical calculations, etc. The IPS may specify interdependencies among possible commands, e.g. command A cannot precede command B, or if command C precedes command D, wait until command E to take a specific action. Control structures, such as IF...ELSE...END IF blocks, DO loops, and GOTO's, are recognized both in the IPS and at the User Command level. These, and the other intrinsic functions recognized in the IPS, make this a very powerful programming environment.

As a simple example, the IPS could contain the following definition:

```

(...)
COMMAND = SCALE
  ARGUMENTS (2F) /EQUIVALENCE = ^/NAMES = (num1,num2)
  ASSIGN num3 = num1/num2
  ECHO "<num1> divided by <num2> gives <num3>."
END COMMAND
`
(...)
```

Then the at the UC level, the user could issue the commands

```

SCALE      3.0,    2.0
SCALE      2.0**3, 8.0
```

which would result in the following being "echoed" to the user:

```

3.0 divided by 2.0 gives 1.5000.
8.0 divided by 8.0 gives 1.0000.
```

The IPS and UC files, and the programming script language elements recognized by INTERFACE will be described in detail in the following chapters.

---

## Chapter 2

# Special Capabilities of Interface

Any command given in either the IPS or UC file can contain strings enclosed by diamond brackets  $\langle \rangle$ . Any such string is evaluated by the function evaluator before the string is parsed, and  $\langle \text{expression} \rangle$  is replaced by this evaluated value. The expression contained between the  $\langle \rangle$  construct can be any numerical or character constant, variable, intrinsic function (e.g. COS, EXP, ABS, etc.), or compound function.

For example, the lines

```
ECHO COS45 = <COS(45.*3.14/180.)>
ASSIGN RNUM<2*3> = 18.
```

would result in parsing of the lines

```
ECHO COS45 = 0.707
ASSIGN RNUM6 = 18.
```

Nested  $\langle \rangle$ 's can be used to do multiple level evaluation. For example the third line below,

```
ASSIGN CHARVR = "CNAME"
ASSIGN CNAME = 4
ECHO "CNAME = " <<CHARVR>>
```

would result in

```
ECHO "CNAME = " 4 .
```

The  $\langle \text{expression} \rangle$  construct makes the IPS and UC very powerful, and allows the easy construction of programs which would otherwise be very difficult to implement. For example, a calculator program can be written in 14 lines of IPS code, as shown in Example 1 of the Appendix.

Whenever a variable is read in INTERFACE, either as one of the arguments associated with a command at the UC level, or during a file-read operation, it is stored along with a flag which indicates its data type. INTERFACE recognizes character, real, and integer type data. By default, each variable can be referenced by a Internal Variable Specifier (IVS) of the format:

command\_name#command\_number#option\_name#line\_no#argument\_no

where

command\_name = the name of command issued at the UC level when this variable was read

command\_number = how many times command\_name had been invoked when this variable was read

option\_name = the name of the option of command\_name which was being parsed when this variable was read.

line\_no = the line of data corresponding to command\_name and option\_name which was being read when this variable was read.

argument\_no = the value on the line referred to by the previous four fields which is to be used. 1 is the first argument on the line, 2 is the second argument, etc.

Character variables assume the length of the string to be stored. Integer and real variables are stored using machine single precision (typically 4 bytes, e.g. R\*4 or I\*4). Use of the IVS means that the user does not have to explicitly assign a name to any variable, if not desired.

INTERFACE also supports user-defined variable names. Any variable, including variables read as described above, can be assigned any alphanumeric name for mnemonic reference. This can be carried out using either the ASSIGN command, or the /NAMES= option of read statements (such as READ and ARGUMENTS).

Any field of an internal variable specifier can be omitted, with an appropriate default then used. In addition, ranges can be specified for the command\_number, line\_no and argument\_no fields, and each of these fields can be specified as a relative, rather than absolute, value. Complete details are given in section XII of the following chapter, which describes the IVS in detail.

In addition to numerous intrinsic commands (to be discussed below), INTERFACE supports several familiar control structures at both the IPS and UC level which make programming much easier. In particular:

- 1) IF (expression) THEN  
    ELSE IF (expression) THEN  
    ELSE  
    END IF
- 2) IF (expression) command
- 3) DO variable\_name = ibeg, iend, increment  
    END DO
- 4) DO WHILE (expression)  
    END DO
- 5) GOTO label\_name  
    label\_name: command

In addition, EXIT control statements exist for exiting DO, IF, and other constructs.

The control structures are described in more detail under their own sections.

INTERFACE supports several modes of file writing:

- 1) Normal sequential writing. In this mode, each line of output is written to the line following the most recently written line. This is the mode most frequently used in programs written in "standard" languages.
- 2) Direct access writing. In this mode, a line number is attached to each output statement, and the output appears on that line of the output file.
- 3) "Buffered" writing. Like direct access, a line number is attached to each output statement. But in addition, a check can be made before each line is written that data is not being overwritten. Or, if desired, one can request that if any new line of data is being written to a pre-existing line, that the contents of the two lines be "merged". And when the file is

closed, it will be a standard sequential access file (which can be used directly as input to a chosen program). This mode of file access is unique to INTERFACE.

Where appropriate, INTERFACE performs extensive error checking on syntax, data type, etc. Errors follow a standard informative format, where both the IPS and UC lines are reported to the user, along with a description of the error and (often) a more specific indication of the context of the error.

In addition to intrinsic error messages, the user can in some instances specify an alternative error message in the design of the IPS. The availability of this option is noted for those commands which support it (e.g. READ, ARGUMENTS).

The actual format of error reports, and a description of the constituent parts, is presented in Chapter 5 (Design and Debugging Tips).

---

## Chapter 3

# Features and Considerations

### I) Input to INTERFACE

As shown in Figure 1 above, there are two types of standard input to INTERFACE. The first is the Interface Programming Script (IPS), which effectively sets up the set of commands which will be recognized and parsed at the User-Command (UC) level. In a sense, the IPS is used to design an operating system, and you run INTERFACE to effect this operating system.

When you run INTERFACE, you will get the prompt

```
INTERFACE (?=HELP) >
```

At this point, you are expected to specify the required files using the syntax:

```
IN = user_command_file # TEMPLATE = template_file #  
DEF = command_definition_file # OVERWRITE
```

The specifications are separated by pound symbols (#). Do not enter a carriage return until all have been specified. The file names can contain any character which is valid on the host system except for the pound symbol (#). The options can be specified in any order, and each can be abbreviated to the first four characters. Both upper and lowercase are recognized for the flag names. The meanings of these options are:

**DEF:** The file containing the IPS. This is the file written by the INTERFACE programmer, and is only specified if a TEMPLATE file has not yet been created from this IPS. This file is frequently referred to as the IPS file.

**TEMPLATE:** A file to contain the commands of the IPS rearranged and reformatted, as required by INTERFACE. This file is created from the DEF file by INTERFACE. Once the TEMPLATE file is created, the DEF file is no longer used by INTERFACE. If the TEMPLATE file corresponding to the current IPS already exists, only the TEMPLATE name is given; do not specify the DEF file in this case.

**IN:** File from which user-commands (UC) are to be read. If IN = STDIN is specified, UC input will be read interactively (from unit 5), rather than from a file. IN must be specified.

OVERWRITE: By default, you cannot create a new template file with the same name as one which already exists. If you specify OVERWRITE, then if you are generating a new TEMPLATE file (both DEF = and TEMPLATE = are specified), and the specified template file name corresponds to an already existing file, the existing file will be deleted, and a new template file with the same name will replace it.

On UNIX machines (only) you can alternatively supply all the required information in the command line. The accepted syntax is

```

interface      -i(nput)                user_command_file
\
               -t(emplate)            template_file
\
               -d(ef)
command_definitions_file\
               -ov(erwrite)

```

The meanings of the various options are the same as above. Each of the flags can be abbreviated to the one or two letters not enclosed in parentheses.

For example, suppose you have just created an IPS script, “program.def”, and you now want to Interface to interpret the set of commands contained in commands.inp according to the definitions in your IPS. The you would specify to the INTERFACE (?=HELP)> prompt:

```

      IN = commands.inp # TEMP = program.tmp # DEF =
program.def

```

If you subsequently wished to run a second file of commands, “commands2.inp” through Interface, you would specify

```

      IN = commands2.inp # TEMP = program.tmp

```

Note that we do not need to specify DEF = once the template file has been created. If we subsequently changed the IPS script, we would need to create a new template file. In this case, we could specify

```

      IN = commands.inp # TEMP = program.tmp # DEF = program.def #OVERWRITE

```

Note that we specify OVERWRITE so that we can overwrite the existing program.tmp file with a new version.

In some cases, it will be desired to read input from the standard input unit (usually a terminal) rather than from a file. For the above example, we would then specify

```

      IN = STDIN # TEMP = program.tmp

```

Specifying IN=STDIN will result in the desired behavior.

## **II) Creating the Interface Programming Script (IPS).**

The file assigned to DEF in the input, the IPS file (see section I) contains a description of the commands which will be recognized at the User Command (UC) level, and the action(s) to be taken if these commands are issued. In addition, it is possible to specify commands which will always be executed when INTERFACE is invoked, regardless of the UC input.

The IPS file is a formatted sequential access file. Allowable commands, with appropriate syntaxes, are described in the following chapter. In-line variable substitution (" $\langle \rangle$ ") is also supported. By default, each line is assumed to contain a new command. If the last non-blank character of a line is the continuation character (usually "\"), then the next line is treated as a continuation of the current line. The IPS file is terminated when the end\_of\_file is reached.

The general format of the IPS file is:

```

instruction_a1
instruction_a2
    (...)
COMMAND = command_name
    instruction_b1
    instruction_b2
    (...)
END COMMAND
instruction_c1
instruction_c2
    (...)
COMMAND = command_name
    instruction_d1
    instruction_d2
    (...)
END COMMAND
    (...)
```

where "instruction" is any valid command (see next chapter), and (...) indicates an arbitrary number of additional commands. Instructions appearing in COMMAND...END COMMAND blocks are executed when the specified command is given. Commands appearing outside of COMMAND definition blocks are executed, in order of definition, when INTERFACE is started. These can be interspersed with COMMAND definitions, but *all* are executed before input from the UC file is read. For example, in the above construct, the command sequence

```

instruction_a1
instruction_a2
    (...)
instruction_c1
instruction_c2
    (...)
```

would be first executed. Then the UC file would be read, and any COMMAND blocks corresponding to specified commands would be executed, in the order the commands were encountered in the UC file.

There is one special point about IPS script files: While the DEFINE command can appear anywhere, DEFINE commands which appear before *any* other lines (even comment or blank lines) are executed *before* any other processing of the file occurs. So, for example, if you wanted to change the comment character for an IPS file to "#", you could issue a DEFINE CMT = "#" command at the beginning of the file.

### **III) The User Command (UC) file:**

The UC file contains a list of commands and control structures which are interpreted using the definitions given in the IPS. Commands recognized in the UC file are those which are either A) defined in the IPS; B) part of an intrinsic control structure, such as IF...ELSE...END IF or DO...END DO; or C) one a small number of other intrinsic commands, such as ASSIGN and STOP. In-line variable substitution (" $\langle \rangle$ ") constructs are also supported. In essence, the UC file contains a "program", with the programming language defined in the IPS.

Commands in the UC file are parsed as they are encountered. By default, each line is assumed to contain a new command. If the last non-blank character of a line is the continuation character (usually "\"), then the next line is treated as a continuation of the current line. Processing of the UC file is terminated when the end\_of\_file is reached.

### **IV) Case Sensitivity:**

By default, command and option names defined in the IPS and all variable names are case insensitive. All IPS-defined commands and all variable names are converted to upper case before a match is attempted. The result is that any combination of lower and upper-case letters will result in a match. User-defined command names, option names, and variable names can be made case sensitive by appropriately setting the ICASE flag using the DEFINE command (see DEFINE in the next chapter).

Character strings appearing between double quotes are never case converted. Intrinsic arithmetic function names (section X) and intrinsic commands (all commands defined in the next chapter) are case independent in all cases.

### **V) Abbreviations:**

All intrinsic commands, and their associated options, can be abbreviated to as few as the first four characters of the name. Intrinsic command names of less than four characters must be specified in full.

Legal abbreviations for commands defined by the user in the IPS are specified at definition time, and can be of any length. Acceptable command name abbreviations can be used in the command and option fields of an internal variable specifier (IVS) (described below).

Variable names, intrinsic function names, and statement labels cannot be abbreviated.

## **VI) Structure requirements for the IPS and UC file scripts:**

Both the IPS and the UC are expected to be read from formatted, sequential access files (or from standard input, if so specified). Commands are read using the following rules:

- 1) A command can start in any column, and can end in any column.
- 2) If a label is included on a line (e.g. LABEL: COMMAND), the label can start in any column.
- 3) By default, it is assumed that each command uses only one line. If the continuation character (default = "\") appears as the last non-blank character on a line, it is assumed that the next line is a continuation of the current command. As many continuation lines may be used as desired.
- 4) Any line (except continuation lines) which begins with the comment character (default = "!") in the first column is treated as a comment line, and is not processed. Blank lines are also treated as comment lines.
- 5) The file is read and interpreted until the end-of-file is reached, until a condition is flagged within the IPS script which causes the program to stop, or until an error is encountered.

## **VII) Intrinsic commands for the IPS**

The IPS recognizes all the commands and control structure directives defined in the following chapter (except GETTEMPLATE, HELP, and MANUAL), as well as the in-line variable substitution construct (" $\langle \rangle$ ").

## **VIII) Intrinsic commands for the UC file:**

A subset of the commands recognized by the IPS are automatically included as part of the language recognized in the UC file. These include most of the control structure directives, a small number of the other commands, and the in-line variable substitution construct (" $\langle \rangle$ "). Specifically,

Control structure directives recognized in the UC file:

```
IF (...) THEN, ELSE IF (...) THEN, ELSE, END IF
IF (...) command
DO, END DO
```

```
DO WHILE, END DO
EXIT (IF, DO, etc.)
GOTO label, label:
```

Other commands intrinsic to the UC language:

```
ASSIGN
REDIRECT
* GETTEMPLATE
STOP
* HELP
* MANUAL
"<>" (in line variable substitution)
all intrinsic arithmetic functions (see section X below)
```

The commands and required syntaxes are described below. The commands marked with a "\*" are only recognized in the UC file.

## **IX) Algebraic and Character Expressions:**

Many of the intrinsic commands, as well as the in line-variable substitution construct (" $\langle \rangle$ "), accept and evaluate algebraic and/or character expressions. Such an expression can be a constant, or a function of constants and variables. The arithmetic operators recognized for integer and real values of "a" and "b" are

```
a + b      (add a to b)
a - b      (subtract b from a)
a * b      (multiply a times b)
a / b      (divide a by b)
a **b      (raise a to the power of b).
```

"a" and "b" can be numerical constants (e.g. 2.0, 1.0E+4, 3), character constants (e.g. "yes", "hello there") or variable names. Character constants must be surrounded by double quotes. Variable names can be either assigned names (e.g. J, rdist) or Internal Variable Specifiers (e.g. START#1###1; see section XIII).

By default, it is illegal to use a variable name (or IVS) which has not previously been defined. You can use the DEFINE command to change this default by setting flag INOMT (see the DEFINE command description in the following chapter).

The following order is used in evaluating an expression:

- 1) Exponentiations are carried out, right to left.
- 2) / and \* operations are evaluated, left to right.
- 3) Remaining procedures are evaluated, left to right.

This is the same hierarchy as used in the ANSI FORTRAN standard. Left-right parentheses ("()") pairs can be used to force an expression to be evaluated in a particular manner. The

data type of the result of each operation is integer if both operands are integers; otherwise, it is real.

Character constants in expressions must be surrounded by double quotes ("). If the character expression itself is to include double quotes, represent these as a consecutive pair of double quotes within the string. The only arithmetic operator valid between two character strings "a" and "b" is "+", which results in concatenation of strings "a" and "b". (Note this differs from FORTRAN, where the operator "//" is used for the same purpose). No arithmetic operation is valid between a character value and a numerical value.

A large variety of intrinsic functions, such as SIN, COS, EXP, etc. are also recognized. These are defined in the following section.

Examples:

Expression	Value
3.0*2**4	48.0
"far"+"out"	"farout"
2*INT(3.5)	6
"data" - "here"	>Illegal operation for character data <
9.*(i+2)	27
	(where i previously defined = 1)

## **X) Intrinsic Arithmetic Functions Provided in Interface**

The INTERFACE program supports a variety of intrinsic functions as part of an algebraic/character expression. These include most of the important functions typically provided in programming languages, plus some additional functions of use in parsing (e.g. BLSTR, UCASE). These functions are listed below. In the descriptions, N, N1, N2, etc. represent numerical arguments (either integer or real). C, C1, C2, etc. represent character arguments. Function names are case independent.

Function Syntax	Description
INT(N)	Integer portion of argument N.
FLOAT(N)	Convert N to floating representation.
NINT(N)	Integer nearest to argument N.
ABS(N)	Absolute value of argument N.
MOD(N1,N2)	Remainder after N1 is divided by N2
SIGN(N1,N2)	Return the magnitude of N1 with the sign of N2.
MAX(N1,N2,N3...Ni)	Return the maximum of N1,N2,N3...Ni (0 < i < 26)
MIN(N1,N2,N3...Ni)	Return the minimum of N1,N2,N3...Ni (0 < i < 26)
SQRT(N)	Return the square-root of N.
EXP(N)	Return "e" (2.718...) raised to the power N.
LOG(N)	Return the natural logarithm of N1 (log base e of N).
LOG10(N)	Return the logarithm, base 10, of N.

SIN(N)	Return the SIN of N. (N in radians).
COS(N)	Return the COS of N. (N in radians).
TAN(N)	Return the TAN of N. (N in radians).
ASIN(N)	Return the ArcSIN of N (in radians).
ACOS(N)	Return the ArcCOS of N (in radians).
ATAN(N)	Return the ArcTAN of N (in radians).
ATAN2(N1, N2)	Returns the ArcTAN of (N1/N2) (in radians).
SINH(N)	Returns the hyperbolic Sine of N.
COSH(N)	Returns the hyperbolic Cosine of N.
TANH(N)	Returns the hyperbolic Tangent of N.
RAN(N1, N2)	Returns a random number on the inclusive interval (N1,N2).The seed for the random number generator can be changed using the DEFINE command.
INDEX(C1, C2)	Returns the position of the character substring C2 in the character string C1. The returned value is the position of the beginning of the substring. If the substring is not found, a value of 0 is returned.
LEN(C1)	Returns the length of the character string C1.
BLSTR(C1)	Returns the string C1 with all leading and trailing blanks stripped off.
UCASE(C1)	Returns the upper-case equivalent of string C1 (only alphabetic characters are changed).
UPCASE(C1)	Alternative recognized name for the UCASE function.
IASVAL(C1)	If the variable named C1 has been assigned a value, IASVAL will return 1,2, or 3, depending on whether C1 refers to an integer, real, or character value. If the variable has not been assigned, IASVAL will return 0. IVS variable descriptors may be used. Be sure to surround a variable name you wish to pass as the argument by double quotes.
COMSTR(N)	If N=1, returns the character string corresponding to the last IPS command parsed. If N=2, returns the character string corresponding to the last UC command parsed. N=3 and N=4 return the same strings as N=1 and N=2, respectively, but with any quote characters in the string doubled. All other values of N are disallowed.
C1(N1:N2)	Returns the substring from position N1 to position N2 of character string C1. NOTE: For this construct, C1 must be a character string or name of a character variable. A substring specifier cannot follow a right parenthesis ")".

### **XI) Logical expressions:**

INTERFACE recognizes the following relational operations on integer, real, or character variables V:

V1	.EQ.	V2	true if V1=V2
V1	.NE.	V2	true if V1 does not equal V2
V1	.GT.	V2	true if V1 > V2
V1	.LT.	V2	true if V1 < V2
V1	.GE.	V2	true if V1 > V2 or if V1 = V2
V1	.LE.	V2	true if V1 < V2 or if V1 = V2

The following logical operators can appear between two logical expressions:

V1	.AND.	V2	true if V1 is true and V2 is true
V1	.OR.	V2	true if V1 is true or if V2 is true

Logical expressions appear as parts of certain commands. Note that INTERFACE does not support logical variable types. This means you cannot ASSIGN a variable name to a logical expression.

In evaluating a logical expression, the following order of evaluation is used:

- 1) Relational operators, left to right
- 2) Logical operators, left to right.

Parentheses may be used to vary the order of evaluation.

Examples:

1.LT.2	.AND.	8.LT.8	would be false
"YES"(1:1)	.EQ.	"Y"	would be true
1	.AND.	6	would cause an error; logical operator must appear only between two logical expressions.

## **XII) In-Line Variable Substitution in INTERFACE**

Any command given in either the IPS or at the UC level can contain strings enclosed by “diamond” brackets <><sup>2</sup>. Any such string is evaluated by the function evaluator before the string is parsed, and <expression> is replaced by this evaluated value. The length of the replacement string is the actual length of the string if <expression> is either an integer or character. If <expression> is a real value, it is by default replaced by a real string in the format G16.5. (The format used for a real string can be changed by setting FORMEC using the DEFINE command). "expression" can be any numerical or character constant, variable, intrinsic arithmetic function (e.g. COS, EXP, ABS, etc.), or compound function.

The <expression> construct can appear anywhere in a command line, with two exceptions: 1) The <expression> construct cannot appear in the statement label, if any; and 2) The

---

<sup>2</sup>There are two exceptions: <>'s cannot be used in statement labels (see GOTO) or a REDIRECT command in the IPS file

<expression> construct cannot appear as part of the command-name itself (the first non-blank field on the line, or the first non-blank field following the label, if a label is specified).

In addition, the following restriction to using the <> construct applies: the expression within the <>'s must not depend on undefined values (unless INOMT has been reassigned using the DEFINE command).

If you wish to use < and/or > as a literal character (not part of an in-line variable substitution construct), precede it with a single backslash. E.g.

```
ECHO "This is what brackets look like: \< >"
```

would return

```
This is what brackets look like: < >
```

A special case can arise when you wish to use <>'s in an

```
IF (expression) command
```

construct. If "command" depends on a variable which may not be defined when the IF statement is parsed, you should precede the <>'s with backslash characters. Consider:

```
DO I = 1,2
  IF (I.EQ.2) ECHO <VARI>
  IF (I.EQ.1) ASSIGN VARI = "HI THERE"
END DO
```

On the first iteration, the logical expression in the first IF statement will be false. But in-line variable substitution occurs before the IF statement is evaluated. When the program tries to evaluate <VARI>, it will determine that VARI is undefined and stop with an error. However, if we replace the second line above with

```
IF (I.EQ.2) ECHO \< <VARI\ >
```

then the program will operate correctly, and without problem. (Alternatively, one can use the DEFINE command to set INOMT to 0, which will default undefined variables to a given value, rather than flagging an error).

Some other examples:

```
ECHO COS45 = <COS(45.*3.14/180.)>
ASSIGN RNUM<2*3> = 18.
```

would result in parsing of the lines

```
ECHO COS45 = 0.707
ASSIGN RNUM6 = 18. .
```

While INTERFACE does not directly support named arrays, the functionality of an array can be implemented using <>'s, e.g.

```
DO I = 1,10
    ASSIGN JUNK<I> = FLOAT(I)
END DO
```

would assign the series of reals 1.,2.,3.,...10. to the variables named JUNK1, JUNK2, JUNK3...JUNK10. These could be subsequently accessed in a similar manner, such as

```
DO J = 1,10
    ECHO "JUNK(<J>) = " <JUNK<J>>
END DO
```

which would sequentially echo a series of lines to the user:

```
JUNK(1) = 1.000
JUNK(2) = 2.000
      . . .
JUNK(10) = 10.000 .
```

(note use of nested <<>>'s in the second example).

### **XIII) Internal Variable Specifiers (IVS)**

Any variable read by INTERFACE, either during an explicit READ/COPY/REFORM command, or as an argument to a user-defined command, is assigned an Internal Variable Specifier (IVS) by which that variable can be subsequently referenced. If desired, a user-chosen name can also be assigned to read variables, but this is not required. An IVS reference can be used in either the IPS or UC file.

Use of an IVS system is made possible because variable type is assigned implicitly to each variable; that is, the type of any variable is the type of expression assigned to that variable the last time it was defined. Because typing is implicit (rather than explicit, such as it is in FORTRAN or C), variable names do not need to be declared at the beginning of the program.

The IVS system can simplify programming in many instances. Not only does it free the user from coming up with a variety of often less-descriptive variable names, but it can make keeping track of variables in a complex IPS script much simpler.

The IVS takes the form:

```
command # call # option # line # argument
```

Each field refers to the current value of respective value at the time the variable was read. Any field can be omitted (or alternatively, for the integer fields, set to 0), resulting in defaults

as described below. The fields are separated from each other by pound symbols, as shown. (The additional spaces shown are not required).

command = The name of the command. Valid abbreviations are acceptable. To specify data read by commands in the IPS which do not appear as part of a command definition, set the command field = XMAIN1.

Default: The command currently being parsed

call = The sequential number of the invocation of the command. E.g. call = 3 means the third time this command was issued.

Default: The most recent invocation of the command.

option = The name of the option of the specified command. Valid abbreviations are acceptable.

If option = \$ is specified:

If command = the command currently being parsed:

Data corresponding to the last option specified for this command will be used (or data for the command itself, if no options have yet been parsed).

If command = any command other than the current one:

Data corresponding to the last option *which resulted in the IVS storage of data for the specified command* will be used (or data for the command itself, if no options resulted in data reads).

Default: Reference is to data read by the main-level command, not an option.

line = The number of line of data read during the specified command and option.

Default: The last line read.

argument= The number of the argument on the line specified by the previous fields. (E.g. 3 means the 3rd argument on that line).

Default: The first argument on the line.

Valid syntaxes for the numerical fields (call, line, argument): If they are non-blank, these fields must contain a numerical constant or expression which evaluates to a number. If the number is a real, the integer portion is used. In addition, the syntaxes "\$+N" and "\$-N" are recognized, where N is a numerical expression. In this case, N represents a relative displacement from the *last* call of the given command, or a displacement from the last line or argument read for the given command and option. For example, line=\$-1 would result in using the second from last line (\$ is the last line).

If the call or line field contains any characters other than \$ and integer constants, or if the argument field contains any characters other than integer constants, the IVS should be

enclosed in square brackets []. (Alternatively, variable expressions in the call and line fields may be enclosed by <> brackets instead. But the "\$" portion of the field must not appear between the <> brackets). This will result in evaluation of the variable expression fields before the IVS reference is resolved. For example:

```
command # # option # $-6 # 1
[command # # option # $-6 # $]
[command # # option # $- 3*J + 1 # $]
command # # option # $-<3*J + 1> # 1
```

are valid, whereas

```
command # # option # $-6 # $
command # # option # $- 3*J + 1 # 1
command # # option # <$- 3*J + 1> # 1
```

would result in errors.

By default, if the IVS refers to a nonexistent variable (one which was not read), an error will occur. This default behavior can be changed using the INOMT option of the DEFINE command.

Some sample IVS references:

##\$##	would reference the first argument on the last line read during the "current" command and option.
[##\$##\$]	would reference the last argument on the last line read during the "current" command and option. (Note [] brackets are necessary because the arguments field is not an integer constant)
command##option##	would reference the first argument on the last line read in processing the "option" qualifier of command "command".
command#2##\$-2#3	would reference the 3rd variable on the 3rd from last line read during the second invocation of command "command" (excluding lines read by options).
##\$#<I+2>#<J>	would reference the Jth variable on the I+2'th line read during the "current" command and option.
[##\$#I+2#J]	Square brackets outside IVS alleviate the necessity for individual <> brackets for non-constant expressions. This specification would reference the same value as that above.

Referencing Variable Ranges using IVS pointers:

In a WRITE output statement (and *only* in a WRITE statement), it is possible to specify a range of variables with a single IVS reference. The appropriate format is:

```
[command # call1:call2 # option # line1:line2 # argument1:argument2]
```

The square brackets [] are *required* when a range is specified for any field. The meanings of each of the fields is the same as described above, except that a range of values for the call, line, and argument fields can optionally be specified. A colon (:) in any of these fields indicates that a range is being provided for the respective field.

Variable references are expanded from the IVS, with the rightmost field (argument) varying most quickly. This can be thought of as a nested "do loop" with call as the outer loop, line as the middle loop, and argument as the inner loop. E.g. the specifier

```
command # 1:2 # option # 1:2 # 1:2
```

would be expanded (and output) in the order:

```
command # 1 # option # 1 # 1
command # 1 # option # 1 # 2
command # 1 # option # 2 # 1
command # 1 # option # 2 # 2
command # 2 # option # 1 # 1
command # 2 # option # 1 # 2
etc...
```

Any or all of the three fields may be specified as a range. If the colon (:) is provided, but part of the range is omitted, e.g. ":call2" or "arg1:", the omitted value is set to the default described above for the relevant field. If the first value in a range is greater than the second, the indices for this field will be looped through in reverse (decreasing) order.

The "\$+N" and "\$-N" relative position descriptors may be used as part of any range specifier, and work as described above. *However*, the relative position descriptors may only be used in the arguments field if neither the call nor the line field has been specified as a range. They can be used without limitation in the call and line fields.

If the \$ descriptor is used in the line field, and if a range has been specified in the call field, one should ensure that separate calls to the specified command result in the same number of lines being read. The resulting behavior may not be as desired otherwise.

An IVS with any field specified as a range may appear in a WRITE statement as part of an algebraic expression. In that case, a series of values corresponding to the expression with the IVS replaced in turn by each of the values in the IVS expansion is output. Note that only one IVS requiring [] outer brackets can appear as part of an expression appearing in the WRITE output list.

An IVS with any field specified as a range must not appear between arithmetic replacement <> brackets. To do so will result in an error. In addition the character substring operator (string(i:j)) must not appear between the [] brackets delimiting the IVS.

Sample IVS range references:

<code>[command##option#1:\$#2:8]</code>	would reference arguments 2-8 on each line read in processing the "option" qualifier of command "command". The arguments for each line would be output successively, proceeding from the first line to the last line.
<code>[command##option#1:\$#2:8]</code>	Same as above, except the lines would be looped through from last to first.
<code>3*[command##option#1:\$#2:8]+2</code>	Same as the first example above, except that 3 times each value+2 would be output. (Note that this will result in error if any of the IVS's references a value that is not of numerical type).

#### **XIV) Variable names**

If desired, a user-specified name can be attached to any variable. This can be done either with an ASSIGN statement, or with the /NAMES= option for READ/COPY/REFORM/ARGUMENTS commands. If the variable had an IVS specifier, this can still be used, as well. For example,

```
ASSIGN BIGNUM = command##option##
```

would associate BIGNUM with the same variable as is referenced by IVS `command##option##`. Subsequently, both

```
3*BIGNUM
and
3*command##option##
```

would give the same result. Note, however, that if the user-specified name is later attached to a second value, the value attached to the IVS does not change. So, if the above ASSIGN statement was subsequently followed by

```
ASSIGN BIGNUM = BIGNUM + 1
```

BIGNUM and `command##option##` would no longer refer to the same value. The value referred to by an IVS cannot be changed by the user.

Variable names can be of any length and can consist of any combination of alphanumeric or "\_" (underscore) characters. The first character of a variable name must be a letter.

### **XV) Rules for determining command-name matches**

A match between a specified command name and a command whose definition is known by INTERFACE (because it is either defined in the IPS file, or because it is an intrinsic command) will occur when:

- 1) The specified command name contains the "required" portion of the known command in its entirety. For a command defined in the IPS, this is the portion of the command before the optional asterisk "\*" (see the documentation for COMMAND and OPTION). For an "intrinsic" command, this is the first four characters of the command name (or the whole command name for commands shorter than four characters).
- 2) Any additional characters in the specified command match exactly the remaining optional portion of the known command

Matching continues until a string terminator is found. This is either a space, an equivalence symbol (usually "="), or a special delimiter symbol (usually "/"). If the end of the known command name is encountered before a string terminator is found, the match will fail. For example

COMMAND will match the following definitions:

```
COMM*AND
COMMAND*S
COMMAND*ONE
```

but will not match the following:

```
COMM*AN
COMMANDS
COMMANDO*NE
```

### **XVI) Changing the option/argument specification syntax in the UC file**

By default, a forward slash (/) delimits options from the main command, and any command or option which takes an argument list is separated from the list by an equals sign (=). In some applications, it may be desirable to change this syntax in the UC file. For example, one might wish to provide a syntax closer to that in the Unix operating system, where options are delimited by hyphens (-), and are separated from their accompanying argument by spaces. This can easily be accomplished by changing OSPU and EQVU, the default values of the delimiter character and equivalence character. These are modified using the DEFINE command (see DEFINE in the next chapter).

As an example, suppose we have defined the command INSERT:

```
INSERT /PAGE = 14
```

The default syntax for delimiting the option and specifying the argument (14) are shown. We could issue the commands in the IPS file:

```
DEFINE OSPU = "-"  
DEFINE EQVU = " "
```

Then the same command would be specified as

```
INSERT -PAGE 14
```

which is a more “Unix-like” syntax.

Some important points to note:

- 1) Remember that you must protect any delimiter characters which are not to be interpreted as delimiters. For example, with the second syntax above, a character string containing a hyphen would have to be either enclosed in quote characters, or surrounded by left-right parentheses. An arithmetic argument containing a minus sign would have to be enclosed in left-right parentheses. Thus, for example, if we wanted to specify the argument in the above command as the difference 16-2 (14), we would need to give the command as

```
INSERT -PAGE (16-2)
```

- 2) Changing the values of OSPU and EQVU only changes parsing of commands which are defined in the IPS file, and only when they are issued in the UC file. OSPU and EQVU do not affect the syntaxes of commands given in the IPS file, or of any intrinsic commands in the UC file (e.g. DO, IF, REDIRECT, GETTEMPLATE, etc.).

## **XVII) Character storage requirements**

All character variables, variable names, and command names (but not option names) are stored in character memory. The amount of available memory is allocated before INTERFACE is compiled.

Each time a new variable name is defined, this requires character memory. Every new command name (or alias name) which is defined requires character memory. Thus, on systems where memory is limited, it may be beneficial to avoid using particularly long variable/command names.

More importantly, every time a character variable is stored, this requires character memory. Character variable storage is not dynamically reallocated, so, for example, the command sequence

```

ASSIGN NAME = "string"
ASSIGN NAME = "new string"

```

would use the total amount of character storage required for both "string1" and "new string". This may be a consideration on memory constrained systems. The READ/REFORM/COPY commands offer a /NOSTORE option. By using this option, data can be read without permanently using any character (or numerical) storage. In addition, the CLEARMEM and MARKMEM commands can be used to reclaim memory (see the descriptions of CLEARMEM and MARKMEM in the following chapter).

### **XVIII) Numerical storage requirements**

Each numerical variable is stored in numerical storage. Any time a numerical value is read, or any time an ASSIGN statement is used to associate a name with a numerical value, one cell of numerical storage is used. Unlike character variables, numerical variable storage is reused. For example, the sequence

```

ASSIGN NUM = 3
ASSIGN NUM = 2.5

```

would require no more storage than issuing only the first ASSIGN command.

### **XIX) Special file equivalence names**

Whenever a read or write operation is performed by INTERFACE, a file equivalence name is required. This is a mnemonic name which is associated with a particular file. These are defined using the OPEN command.

There are special file\_equivalence names which are automatically defined. These are

name	purpose
STDIN	Usually refers to "unit 5" input. On most computers, this is input to be read from the terminal.
STDOUT	Usually refers to "unit 6" output. On most computers, this is output which will appear on the terminal. If not otherwise specified, most commands which write text will write the text to STDOUT.
ERROUT	The file_equivalence_name used in writing all program generated error messages. This is by default assigned to unit 6.
&FIVE	Equivalence name which refers to unit 5.
&SIX	Equivalence name which refers to unit 6.

Any of these names can be used with in any field that requires a file\_equivalence\_name.

Using the /ATTACH option of the OPEN command, each of these equivalence names can be reassigned so that they specify a user-chosen file. &FIVE and &SIX are provided so that after assigning STDIN, STDOUT, or ERROUT to a non-default file, the equivalence\_name can subsequently be reassigned to unit five or six. See the OPEN command and associated examples for more information.

### **XX) Speed considerations:**

INTERFACE offers enough function flexibility to make it appealing for a wide range of applications in addition to designing program interfaces. It is important to realize, however, that the IPS script used by INTERFACE is an *interpreted* language. That is, each instruction is interpreted and carried out sequentially. In contrast, languages like FORTRAN and C are compiled languages, and can execute considerably faster. For this reason, the user must carefully evaluate how compute-intensive a particular application may be, and how important it will be to use INTERFACE for that application. Some procedures, especially those which are "compute bound", will typically be better handled by a compiled language.

See Chapter 5 for additional discussion.

### **XXI) Machine independence:**

With two exceptions, INTERFACE is written entirely in ANSI-standard FORTRAN 77. For this reason, it should compile and run without problem on any computer with an ANSI standard FORTRAN compiler. One result is that the IPS-directed INTERFACE environment should be identical on different computers--a important advantage when designing a user-friendly interface.

There are two routines of INTERFACE which is not machine independent. The first is the routine which carries out EXTERNAL "string" command. The EXTERNAL command sends the supplied string to the host operating system, for execution according to the specific rules of that computer. There is no standard FORTRAN to carry out such a request. A dummy routine can be used in place of this routine, with the effect that all of INTERFACE will be machine independent, but then the EXTERNAL command will not work.

The second routine which is machine dependent allows one to specify the names of input files for INTERFACE on the command line which invokes the program. This routine is only used for Unix(r) computers, and can be replaced by a dummy routine, in which case the required filenames are specified in response to the initial prompt from INTERFACE, just as on any other machine.

---

## Chapter 4

# Command Definitions

The following defines the recognized commands and options, and the appropriate syntax for each. In the descriptions, the following conventions are used:

- 1) any item in curved brackets "{}" is optional;
- 2) "string" refers to any valid character string. A string can be enclosed by quote characters (which are, by default, double quotes ").

M refers to an integer constant or algebraic expression which evaluates to an integer. In most cases, if a real is supplied for M, the integer portion will be used.

R refers to a real constant or algebraic expression which evaluates to a real. In most cases, if an integer is supplied for R, the floating representation of the number will be used.

- 3) (...) indicates an arbitrary number of valid command lines.
- 4) Options separated by a "pipe" character (|) are mutually exclusive; that is, pick one and only one.
- 5) Options marked with a (D) are default values, which are used if no alternative is specified.
- 6) Spacing between command and options, on both sides of an equivalence marker, and in any unquoted text string is arbitrary, unless otherwise noted. Spacing within a quoted text string will be maintained.
- 7) options (e.g. /START=M) can appear in any order in the actual command, but all options must follow any other required information. For example,

```
ECHO "Text string"/START=2/COLUMNS=60
      and
ECHO "Text string"/COLUMNS=60/START=2
```

are acceptable, but

```
ECHO/START=2/COLUMNS=60 "Text String"
```

is not.

Commands are listed in three groups. The first group contains all the commands which are recognized in both the IPS and the UC file. These include most of the control structures, plus a few assorted other commands. The second group contains commands which are only

recognized in the UC file. The third group contains commands which are only recognized in the IPS file. Many of the control constructs of the first group are similar to constructs in FORTRAN, and the unacquainted user should probably consult a FORTRAN manual for further details. Commands which read and write data support FORTRAN format statements. If you wish to use formatted input and are not familiar with such format statements, consult any standard FORTRAN manual for further details.

## *Commands Recognized in Both the IPS and UC Files: Control Constructs*

### **DO**

Syntax:

```
DO variable_name = M1, M2, M3
    (...any series of valid commands...)
END DO
```

Description: Do Control loop. The commands between initial opening DO statement and the END DO statement are executed as many times as indicated by the DO loop control parameters, M1, M2 and M3. M1, M2, and M3 are all integer (or real) constants, variables, or expressions. On each iteration, the given variable\_name is set to the current control value. The first control value is M1, and on each iteration of the loop, M3 is added to variable\_name until variable\_name > M2 (if M3 > 0) or variable\_name < M2 (if M3 < 0).

By default, M3 = 1. M1 and M2 must be specified. The total number of iterations of the DO loop is given by

$$\text{MAX}(\text{INT}((M2-M1+M3)/M3), 0).$$

Note that if M1 > M2 and M3 > 0, or M1 < M2 and M3 < 0, then the commands within the DO loop will not be executed at all.

When the DO loop finishes executing, variable\_name will equal the control value on the last iteration where the contents of the DO loop were executed. (E.g. variable\_name=M2, if M3 is omitted). Note that this differs from the FORTRAN 77 standard, where variable\_name = the control value on the last iteration + M3 (E.g. variable\_name = M2+1, if M3 is omitted).

DO loops can be nested within other DO loops, or within other control structures.

A DO loop cannot contain the following commands: OPTION, CALLED, NOT\_CALLED.

The space between END and DO is required.

*Examples:*

```
ASSIGN IBEG = 1
ASSIGN IEND = 3
ASSIGN INC = 2

DO I = IBEG, IEND, INC
    ECHO "I =" <I>
END DO
ECHO "I at end of loop = " I
```

would produce:

```
I = 1
I = 3
I at end of loop = 3
```

**DO WHILE**

Syntax:

```
DO WHILE (logical expression)
    (...any series of valid commands...)
END DO
```

Description: The DO WHILE construct will repeatedly execute the commands within the DO WHILE ... END DO construct until the provided logical expression is false. The parentheses surrounding the logical expression are required. Any of the logical operators described above (section XI) can appear in a logical expression.

The DO WHILE construct can appear nested within other DO WHILE constructs, or within other control structures.

A DO WHILE loop cannot contain the following commands: OPTION, CALLED, NOT\_CALLED.

The space between END and DO is required.

*Example:*

```
ASSIGN I = 1
DO WHILE (I.LT.3)
    ECHO "I =" <I>
    ASSIGN I = I + 1
END DO
```

would produce

```
I = 1
I = 2
```

**EXIT**

Syntax:

```
* EXIT DO
* EXIT IF
  EXIT OPTION
  EXIT DEFER
  EXIT CALLED
  EXIT NOT_CALLED
  EXIT ALL
```

Description: The EXIT command effects an exit from the type of command block specified, by skipping commands until the appropriate END (DO, IF, OPTION, DEFER, CALLED, or NOT\_CALLED) command is found. For example, EXIT DO will cause an exit of a DO loop, transferring control to the first line outside the appropriate END DO command.

EXIT ALL will effect an exit from all control structures currently in effect: COMMAND, DEFER, OPTION, etc. It can be used, e.g. to end processing of a particular command.

Only those EXIT command marked with a \* are recognized in the UC file. All the EXIT commands are recognized in the IPS file.

Any IF/DO block initiated in a DEFER block must be terminated *before* the EXIT DEFER command is issued.

*Example:*

```
DO I = 1,3
    ECHO "I =" <I>
    IF (I.EQ.2) EXIT DO
END DO
```

would produce

```
I = 1
I = 2
```

**GOTO**

Syntax:

```
GOTO label
    (...any valid series of commands...)
label: command
```

Description: The GOTO command effects a transfer to the line which begins with the specified label. A label may contain any string of alphanumeric characters, plus the underscore character "\_". Any other characters or blanks in the label will cause an error.

The label itself must be immediately followed by a colon (:), and must be the first non-blank element on the line. The referenced label may either precede or follow the GOTO statement. If the GOTO statement references a nonexistent label, an error will occur.

A label may optionally be followed by a valid command on the same line.

A GOTO must refer to a label which is defined in the current "unit". A unit is any of the following: A DEFER block; an OPTION block; an explicit CALLED block; an implicit CALLED block; a NOT\_CALLED block. If file redirection has been requested, the GOTO label must be contained in the current file.

A GOTO must not refer to a label which results in a transfer into a DO-loop, IF block, OPTION block, DEFER block, NOT\_CALLED block or explicit CALLED block. A transfer into an IF or DO block which was defined before the GOTO label was encountered will result in an "unmatched DO...END DO" or "unmatched IF...END IF" type message.

The optional command following a label cannot be an "END" or "ELSE" command.

*Examples:*

```

        ASSIGN I = 0
START:  ASSIGN I = I+1
        ECHO "I =" <I>
        IF (I.LT.3) GOTO START
        IF (I.EQ.4) GOTO 1_END
        ECHO "Not 4 yet"
        GOTO START
1_END:
        ECHO "I = 4 now"
```

would produce

```
I = 1  
I = 2  
I = 3  
Not 4 yet  
I = 4  
I = 4 now
```

**IF**

Syntax:

```
(I)   IF (logical expression) THEN
        (...)
      {ELSE IF (logical expression) THEN}
        (...)
      {ELSE}
        (...)
      END IF

(II)  IF (logical expression) command
```

Description:

Type (I): The IF () THEN ... ELSE IF () THEN ... ELSE ... END IF construct allows sections of code to be conditionally executed, depending on the values of specified logical expressions. The parentheses surrounding the logical expressions are required.

First, the logical expression in the initial IF () THEN statement is evaluated. If this expression is true, the following block of commands is executed until the another element of the IF block (ELSE IF, ELSE) is encountered, at which point all subsequent commands will be skipped until the END IF statement is found. If no ELSE IF or ELSE statements were specified, all commands between the IF () THEN and the END IF will be executed.

If the logical expression in the initial IF () THEN statement is false, subsequent commands are skipped until either A) an ELSE IF () THEN statement with a logical expression that evaluates as true is found, at which point behavior is as described above for the initial logical expression evaluating to true; B) an ELSE statement is found, at which point all commands will be executed until the terminating END IF command is encountered; C) the IF block terminating END IF statement is found.

"ELSE IF () THEN" and "ELSE" statements are optional parts of the IF block construct. The "END IF" statement is required. As many "ELSE IF () THEN" statements may be used as are necessary, but only one "ELSE" statement be used, and must follow any "ELSE IF () THEN" elements.

An IF...THEN...ELSE... block cannot contain the following commands: OPTION, CALLED, NOT\_CALLED.

Type (II): The IF (logical expression) command construct will conditionally execute the specified command if the provided logical expression is true. If the logical expression is false, no further action is taken. "command" can be any valid command (and associated arguments/options), but should not typically be part of a multi-line construct (e.g an END DO command).

The parentheses surrounding the logical expression are required. The space between END and IF is required.

*Examples:*

Type (I):

```
DO I = 1,3
  IF (I.EQ.1) THEN
    ECHO "First pass"
  ELSE IF (I.EQ.2) THEN
    ECHO "Second pass"
  ELSE
    ECHO "Third pass"
  END IF
END DO
```

would produce

```
First Pass
Second Pass
Third Pass
```

-----  
Type (II):

```
DO I = 1,3
  ECHO "I =" <I>
  IF (I.EQ.2) ECHO "I equals 2"
END DO
```

would produce

```
I = 1
I = 2
I equals 2
I = 3
```

## Commands Recognized in Both the IPS and UC Files: Other Commands

### ASSIGN

Syntax:

```
ASSIGN variable_name = expression
```

Description: ASSIGN associates the given variable\_name with the expression on the right of the equals sign. "expression" can be a numerical or character constant, variable, or variable expression. variable\_name must adhere to the rules for variable\_name specification (must start with a letter; must contain only alphanumeric or "\_" characters). If "expression" is a character expression, it must be enclosed in character delimiters (usually double quotes ("")).

If variable variable\_name has already been assigned, the previous assignment is replaced after the expression on the right of the equals sign is evaluated. Since variable names are not declared with an explicit type, a variable name associated with one data type can be reassigned to a different data type without error. The current data type of any variable is maintained internally.

Assignments made with the ASSIGN command are local to the level at which they are made. That is, assignments made in the IPS file do not affect variables in the UC file, and vice-versa. To make an assignment to a variable\_name which can be accessed by both the IPS and UC files, use the GASSIGN command. To make an assignment to a variable\_name which only affects the value of the variable at the level complementary to that at which the assignment is made, use the CASSIGN command.

An ASSIGN performed in the UC file is by default non-volatile to memory clearance (same as NVASSIGN). An ASSIGN in the IPS file is by default volatile to memory clearance (same as VASSIGN).

*Examples:*

Statement	assignment made
ASSIGN A_VAL = 3*(1+6)	A_VAL = 27
ASSIGN A_VAL = A_VAL + 3.	A_VAL = 30.0
ASSIGN CHRNAM = "char data"	CHRNAM = "char data"
ASSIGN R = MOD(2.1,2.0)	R = 0.1

ASSIGN TR = 2.GT.1

--error-- only character and numerical data types can be assigned in an ASSIGN statement

**CASSIGN**

## CASSIGN

Syntax:

```
CASSIGN variable_name = expression
```

Description: CASSIGN performs the same function as ASSIGN, except that the assigned variable name can subsequently be accessed only at the level which is *complementary* to the level at which the assignment was made. Thus, an assignment made at the IPS level using a CASSIGN statement would only modify the value of the variable at the UC level. Analogously, an assignment made at the UC level using a CASSIGN statement would only modify the value of the variable at the IPS level.

See the description of ASSIGN for more syntactical details.

Examples:

The IPS script

```
command = multiply
  arguments (f) /equivalence = ^/name=arg
  option = newval
    arguments (a)/name=newarg
    cassign <newarg> = 3.0*arg
  end option
end command
```

Could be used to define a UC command which multiplies the supplied argument by 3.0 and returns the result in a variable name supplied by the user. E.g.the UC command

```
multiply 12.0 /newval=value2
```

would result in value2 being set to 36.0. Note that the name being supplied for option newval would be irrelevant to the IPS script, since the value would be assigned only at the UC level (using the CASSIGN command).

## **GASSIGN**

Syntax:

```
GASSIGN variable_name = expression
```

Description: GASSIGN performs the same function as ASSIGN, except that the assigned variable name can be accessed by commands in both the IPS file and the UC file. Using the ASSIGN command, the variable name assignment is only made for the type of file where the ASSIGN was encountered (*IPS or UC*).

See the description of ASSIGN for more syntactical details.

**NVASSIGN,**  
NVCASSIGN,  
NVGASSIGN

Syntax:

```
NVASSIGN variable_name = expression  
NVCASSIGN variable_name = expression  
NVGASSIGN variable_name = expression
```

Description: NVASSIGN, NVCASSIGN, and NVGASSIGN are analagous to ASSIGN, GASSIGN and CASSIGN, respectively. The NV prefix requests that the variable be stored as “non-volatile.” This means the variable assignment will remain even when a CLEARMEM command is issued to clear stored memory. A “volatile” variable may become undefined when a CLEARMEM is effected. See the description of CLEARMEM for more details. Once a variable is defined as “non-volatile,” it will remain non-volatile even if the value assigned to the variable\_name changes. A non-volatile variable becomes undefined only when a CLEARMEM/NONVOLATILE command is issued which clears the value of the variable.

See the descriptions of ASSIGN, CASSIGN, and GASSIGN for more details of the various types of assignments.

At the UC level, all assignments are by default performed as non-volatile, and ASSIGN, CASSIGN, and GASSIGN are completely equivalent to NVASSIGN, NVCASSIGN, and NVGASSIGN.

In the IPS script, all assignments are by default performed as volatile, and one must explicitly use NVASSIGN, NVCASSIGN and NVGASSIGN to effect a non-volatile assignment.

The complements of the NVASSIGN commands are the VASSIGN set of commands. (See VASSIGN).

**REDIRECT**

Syntax:

```
REDIRECT = filename
          {/BEG = M}
          {/END = M}
          {/NOTENOUGH = stop | continue (D)}
          {/DEFAULT}
```

Description: REDIRECT allows redirection of input for either the IPS or UC file. The specified number of lines are read from the indicated file. Control then reverts to the IPS/UC file in which the REDIRECT was issued.

REDIRECT commands can appear nested in files being read by a REDIRECT command. A maximum of 10 levels of such redirection are allowed.

A REDIRECT command appearing in an IPS file is resolved when the template file is created, not when the template file is used.

A REDIRECT command appearing in a UC file is resolved when the file is read (at run-time)

Redirection can only be made to formatted sequential access files.

If the filename contains slashes (/), the entire name must be surrounded by double quotes.

Note: In line variable substitution constructs (<>) are not valid in a REDIRECT command appearing in the IPS file. They *may* be used in REDIRECT commands in the UC file. The DEFDIR:filename construct (see below) *is* valid in both the IPS and UC files.

**filename:** The name of the file to be used in the redirection. Any name valid on the host computer can be used, but the name must correspond to a formatted, sequential access file (default type of file). This is the actual file name (not an alias).

The filename specification may contain the special construct

DEFDIR:filename

“DEFDIR:” is a literal string, and must precede the filename. If this construct is provided, DEFDIR: is replaced by a definition previously provided by the user for DEFDIR. Thus, for example, one could specify all redirection in the IPS file as DEFDIR:filename. Then, if later it became desirable to move the location of all the

IPS files, only one change would have to be made the IPS file: a modification of the directory pointer contained in DEFDIR.

The special filename "&STDIN" can be used in UC files to indicate that redirected input is to come from the standard input source (usually unit 5 == terminal input). No special filename is allowed in IPS files.

**BEG = M:** Specifies the number of the line at which reading should begin in the the given file. The value may optionally be specified as "\$+M" or "\$-M", which will be translated into a relative displacement from the end of the file (e.g. \$-1 is the last line in the file).

By default, reading begins at the first line of the file.

**END = M:** Specified the number of the line at which reading should end in the given file. The value may optionally be specified as "\$+M" or "\$-M" (see above).

By default, reading ends at the end of the file.

**NOTENOUGH = stop | continue (D):** Specifies the behavior if the /END = M option is given and the specified line exceeds the last line in the actual file. By default, the program will close the file, and control will revert to the file where the REDIRECT command was encountered. If /NOTENOUGH = stop is specified, the program will stop.

**DEFAULT:** When DEFAULT is specified, the "filename" string provided is defined as DEFDIR for all subsequent REDIRECT commands. When DEFAULT is specified, no actual redirection occurs with this REDIRECT command, and any other qualifiers, if specified, are ignored. Once DEFDIR has been defined by a REDIRECT/DEFAULT command, it can be reassigned by a subsequent REDIRECT/DEFAULT command.

DEFDIR can also be defined using the DEFINE command.

*Example:*

```
REDIRECT = "/usr/misc/def/" /DEFAULT
          (...any series of commands...)
REDIRECT = DEFDIR:IPS1.DEF
```

The first REDIRECT command would set a value for DEFDIR:. It would be set to a directory specification. This definition is maintained in non-volatile memory and can subsequently be used at any time. Later, we specify redirection to a file given as DEFDIR:IPS1.DEF. The actual redirection will be to file /usr/misc/def/IPS1.DEF.

Note that we must enclose the directory specification in double quotes because it contains slash characters which would otherwise be interpreted as option delimiters.

**STOP**

Syntax:

```
STOP {"string"}  
      {/OUTFILE = file_alias_name}
```

Description: The STOP command will stop program execution, reporting an optional text string, if specified.

**"string"**: Optional text string to be output before the program is stopped. If the string has any embedded slash (/) characters, it must be surrounded by quote characters ("). Text is output starting in column 1, ending in column 80, and is broken into multiple lines at appropriate break points if too long to fit on one line.

**OUTFILE = file\_alias\_name**: By default, text, if any, is output to STDOUT (typically unit 6). The OUTFILE option can be used to specify that the text be output to a different file. The file must already have been opened using an OPEN command, and file\_alias\_name is the alias that was defined when the file was opened.

file\_alias\_name cannot refer to an unformatted file.

If file\_alias\_name refers to a file opened as "BUFFERED", the text will be output starting at the "current" line, and may overwrite contents of the file regardless of whether or not the "/OVERWRITE" flag was specified when opening the file.

*Example:*

```
STOP "Program stopped" /OUTFILE=ERRROUT
```

would cause program execution to stop, and the message "Program stopped" to be echoed to STDOUT.

**VASSIGN,**  
VCASSIGN,  
VGASSIGN

Syntax:

```
VASSIGN variable_name = expression  
VCASSIGN variable_name = expression  
VGASSIGN variable_name = expression
```

Description: VASSIGN, VCASSIGN, and VGASSIGN are analogous to ASSIGN, GASSIGN and CASSIGN, respectively. The V prefix requests that the variable be stored as “volatile.” This means the variable being assigned may become undefined if a CLEARMEM command is issued to clear the block of memory used to store the variable. See the description of CLEARMEM for more details.

See the descriptions of ASSIGN, CASSIGN, and GASSIGN for more details of the various types of assignments.

At the UC level, all assignments are by default performed as non-volatile. Using VASSIGN, VCASSIGN, and VGASSIGN will make these variable assignments subject to becoming undefined with a CLEARMEM issued in the IPS file. *In most applications you would **not** use VASSIGN, VCASSIGN and VGASSIGN at the UC level.*

In the IPS script, all assignments are by default performed as volatile, and VASSIGN, VCASSIGN and VGASSIGN are completely equivalent to ASSIGN, CASSIGN, and GASSIGN.

The complements of the VASSIGN commands are the NVASSIGN set of commands. (See NVASSIGN).

# Commands Recognized only in the UC File

## **GETTEMPLATE**

Syntax:

```
GETTEMPLATE TEMPLATE = template_file
                        {# DEF = command_definition_file}
                        {#OVERWRITE}
```

Description: GETTEMPLATE causes the reading of an IPS command definition file, or of a template file derived from the IPS. Any commands read in this IPS/template file are added to any commands read in the IPS/template file specified when INTERFACE was started. Any commands in the IPS which do not lie within a COMMAND definition or DEFER block will be executed immediately when the template file is read.

The `command_definition_file` is the file containing the unmodified IPS. The `template_file` is a modified and processed version of the IPS, created either on a previous run of INTERFACE, or when GETTEMPLATE is specified. It is this file which is actually used when INTERFACE is running. (See section I under Features and Considerations). If the `template_file` has already been created, do not specify the `command_definition_file` (IPS script). If the `template_file` has not already been created, you must specify both the `command_definition_file`, and the name of the `template_file` which will be created from it. The `template_file` will remain when INTEFACE exits.

Files can be specified with any path name valid for the host computer.

The `TEMPLATE=` and `DEF=` options are separated by a mandatory pound sign (#). We use this non-standard separator for this command (only), because the default slash (/) separator is commonly used in file names.

GETTEMPLATE is only recognized in the UC file. It is not a valid command in the IPS file. In the IPS file, the REDIRECT command can be used to read additional IPS/template files.

**OVERWRITE:** If the IPS file (`command_definition_file`) is modified after creation of a corresponding `template_file`, a new version of the template file should be generated. Use the OVERWRITE flag to allow a new version with the same name to overwrite the old version.

*Examples:*

```
GETTEMPLATE DEF = program.ips # TEMPLATE =  
program.tmp
```

would read the IPS in file program.ips, and create a template file named program.tmp. The commands in program.ips would be added to those defined in the TEMPLATE file specified when INTERFACE was started.

On a subsequent run of INTERFACE, one could specify

```
GETTEMPLATE TEMPLATE = program.tmp
```

since the template file corresponding to program.ips would already exist.

## **HELP**

Syntax:

```
HELP {command_name} {/option_name}
```

Description: The HELP command will produce a list of commands or options for a command which have been defined in the IPS file. In addition to the listing, any informative HELP message text which has been supplied in the IPS file (using DEFHELP commands) will be echoed to the user.

The HELP command is only valid in the UC file (not the IPS script).

When the HELP command is specified by itself, a listing of all commands which have been defined in the IPS file will be echoed to STDOUT (usually unit 6). By default, this list will be alphabetized. If you wish the list to reflect the order in which the commands were defined, you can define the variable IHLALP to be 1 using the DEFINE command.

When HELP is specified with a `command_name`, but no `option_name(s)`, HELP will produce a listing of all options which are defined for the specified `command_name`. In addition, any descriptive text supplied by DEFHELP commands in the main level command will be echoed to the user.

When HELP is specified with both a command name and `option_name(s)`, any descriptive messages defined in the IPS file using DEFHELP commands for the specified command and option will be echoed to the user. The special character "\*" may be specified in the `option_name` field. In this case, help on all available options for the specified command will be reported.

Note that alias names (defined using the ALIAS command) will not appear in the help command listings, but may be specified as explicit arguments to HELP to retrieve the appropriate DEFHELP descriptive messages.

If `command_name` or `option_name` is specified, but does not correspond to a valid name, a warning will be reported, and the program will continue.

**command\_name:** Name of a command defined using a COMMAND block in the IPS file.

**option\_name:** Name of an option defined for the corresponding `command_name` in the IPS file.

*Example:*

```
HELP command1/option2
```

would retrieve all information about option2 with parent command1, as defined by DEFHELP commands in the IPS file.

HELP

would produce a listing of all commands which were defined in the IPS file.

**MANUAL**

Syntax:

```
MANUAL
    {/OUTPUT = file_name}
    {/NO_ALPHABET}
```

Description: The MANUAL command allows the user to request a "manual" of all commands, options, and the associated help messages (see DEFHELP).

The MANUAL command is only valid in the UC file (not in the IPS script).

Note that the manual will not list the names of commands which are simply aliases for other definitions.

Note also that MANUAL is one of the few commands which takes an actual file name, and *\*not\** an equivalence name, as an argument.

The options associated with a given command are always reported in the order in which they are defined in the IPS.

**OUTPUT:** If specified, the "manual" will be written to the named file. If the file does not already exist, it will be created. If it already exists, the manual will overwrite contents of the file. If OUTPUT is not specified, the "manual" will be written to STDOUT.

**NO\_ALPHABET:** By default, the commands list reported when the MANUAL command is given is alphabetized. If /NO\_ALPHABET is specified, then the commands list will be reported in the order in which the commands are defined in the IPS files.

*Example:*

```
MANUAL /OUTPUT = manual.dat
```

would write all the available help information and definitions to file manual.dat. The command listing would be alphabetized.

# Commands Recognized only in the IPS File

## ALIAS

Syntax:

```
ALIAS alias_name {command_name}
                {/COMMAND (D) | OPTION}
```

Description: The ALIAS command can be used to define aliases for command (or option) names which have already been defined. If an alias is defined, the specified command can henceforth be evoked by using either the original name, or the defined alias\_name.

**alias\_name:** The name of the alias name to be attached to the specified command or option. Valid command names must consist only of alphanumeric characters and underscore "\_" characters. The special star character "\*" can be used to define acceptable abbreviations. Characters before the "\*" are required. Characters after the "\*" are optional, but must match exactly if provided. If no "\*" is specified, the alias name cannot be abbreviated. E.g.

```
COMM*AND  would match
COMM
COMMAN
COMMAND
```

but not

```
COMMANT
```

The required part of alias\_name must not be redundant with the required parts of names of other commands already defined. If it is, an error will be reported.

For additional details on name matching see section XV of Chapter 3: Features and Considerations.

**command\_name:** Can be used to specify which command the alias\_name applies to. If command\_name is not specified, the ALIAS command *must* appear in a COMMAND definition block, and the alias will apply to the command being defined. Valid abbreviations can be used in specifying the command\_name field. If a command\_name is specified which does not correspond to a previously-defined command, a warning will be issued and the program will continue without setting up any alias.

**COMMAND:** Indicates that the alias is being performed for a command name. This is the default.

A command alias will remain in effect for the duration of the program.

**OPTION:** Indicates that the alias is being performed for an option name. In this case, "command\_name" gives the name of the option whose alias is being defined. The ALIAS command must appear within a COMMAND definition block, and the option whose alias is being defined must be an option for the command being defined.

An option alias remains in effect only while the corresponding main level command is being executed. When a new command is started, option alias definitions will be cleared.

All alias names for options **MUST** be defined before the first OPTION block for the parent command appears, or an error will result.

*Examples:*

```
ALIAS new*comm oldcomm
```

would define command "newcomm" as an alias of "oldcomm".

```
ALIAS new*option oldoption /OPTION
```

would define "newoption" as an alias of option "oldoption" while executing the current command.

**ARGUMENTS**

Syntax:

```

ARGUMENTS (argument_list)
  { /EQUIVALENCE = equivalence_symbol }
  { /STRICT }
  { /NAMES = associated_namelist }
  { /FIELDS = variable_name }
  { /MISSING = STOP (D) | CONTINUE }
  { /MISS_TEXT = "string" }
  { /SUBSTITUTE = template | uc (D) }

```

Description: The ARGUMENTS command appears in either a COMMAND or OPTION block, and defines what arguments--how many, what data types, and in what order--will be specified with the respective command or option. The ARGUMENTS command should not be used outside of a COMMAND/OPTION block.

**argument\_list:** Specifies the order, number, and types of arguments to be read. This specifier is identical to the portion of a free-format specification which appears to the left of the colon (:). (see the FORMAT SPECIFICATION section). Briefly, such a specification takes a form such as

$$(nA\{m\}, nI, nF, nI, nS, \dots)$$

where n is an integer constant, and

nA is the number of character variables to be read;

nI is the number of integer variables to be read;

nF is the number of real variables to be read;

nS is the number of character fields to be "skipped" (read but not stored)

nA, nI, and nF can appear in any order, and as many times as necessary to fully specify the data transfer list. If n is omitted, a value of n=1 is used. If one wishes to use an variable expression for n, enclose it in <> brackets.

The outer parentheses "()" are required.

If an integer is specified following A, this is the number of characters in the character field to be transferred. If  $m > \text{length\_of\_actual\_variable}$ , the field is padded with spaces on the right.

Parentheses can also be used in argument\_list specifications, e.g.

$$(4I, 3(2F, 2A)) \text{ is equivalent to } (4I, 2F, 2A, 2F, 2A, 2F, 2A)$$

See the FORMAT SPECIFICATION section for more details.

What is expected at the UC file level:

For a command or option which is defined to take arguments, use:

```
command equivalence_symbol (argument1 , argument2 , argument3 , ...)
      or
command / option equivalence_symbol (argument1 , argument2 , ...)
```

Each field specified in the argument\_list is read from the appropriate file in free format. Numerical fields can contain either a constant or expression. Character fields can contain quoted or unquoted character strings (the string must be quoted if it contains delimiter characters such as commas or slashes). *For an argument list, only commas (not spaces) delimit fields.*

Parentheses surrounding the list of arguments are optional, but *must* surround the list if any non-quoted argument contains the character which delimits options (usually "/").

Recaping how fields in the UC file are treated:

integer	the expression is evaluated before storage
real	the expression is evaluated before storage
character	the expression is stored as-is (surrounding quotes, if any, will be removed). Note that character intrinsic functions will not be evaluated. If one wishes to use e.g. the character subrange specifier ("string"(i:j)), surround the expression with <>'s, so that it is evaluated before storage.

**EQUIVALENCE** = equivalence\_symbol: Changes the equivalence\_symbol. By default, the equivalence\_symbol which separates a command or option from the associated arguments list is the value assigned for EQVU using a DEFINE command (which is an equals sign "=") if not otherwise specified). The EQUIVALENCE options allows this symbol to be changed. equivalence\_symbol must be a single non-blank character (do not surround in quotes). To specify a space, set EQUIVALENCE = ^.

Setting the equivalence\_symbol to a space effectively results in the following useful syntax:

```
command (argument1 , argument2 , argument3 , ...)
```

In the special case where /MISSING=CONTINUE has been specified, the equivalence character can be omitted when specifying the actual command if (and only if) the argument list is also omitted. This will result in all arguments being set to 0, 0.0, or ''.

**STRICT:** Forces strict value type specification. By default, if an argument is specified in the argument\_list to be real, and an integer is provided in the associated command

argument list, the floating representation of the value (FLOAT(value)) is used. Likewise, if the value is specified to be an integer, and a real is provided, the integer portion (INT(value)) is used.

By specifying /STRICT, the program will stop if the specification type does not match the type of the value provided in the associated command argument list.

**NAMES** = associated\_namelist: Allows association of user-defined names with the values read. By default, the values which are read because of an ARGUMENTS command can be later referenced only by their IVS specifier, i.e.

command#call#option#line#argument.

By using the /NAMES qualifier, you can assign names of your choosing to the data read because of the ARGUMENTS command.

The syntax of the **associated\_namelist** is

name1 , name2 , name3 , ...

where name1, name2, etc. are the names to be assigned, sequentially, to the values read. The associated\_namelist can optionally be enclosed in a set of parentheses "()". If fewer names are specified than values are to be read, the remaining variables will only be accessible by their IVS specifiers. If more names are specified than variables are to be read, the remaining names are ignored. If no name appears between a pair of commas, no name is assigned to the corresponding variable. No name should be specified for a field which is skipped ("S" format item in argument\_list).

Names in the associated\_namelist must follow the standard rules for variables: They must start with a alphabetic character, and must contain only alphanumeric and "\_" characters. Name assignments are by default made at the IPS file level (i.e. equivalent to ASSIGN). If you wish the assigned name and value to be accessible both in the UC file and the IPS script (the equivalent of GASSIGN), precede the name in the associated\_namelist by a "&" character. If you wish the assigned name and value to be accessible only in the UC file (the equivalent of CASSIGN), precede the name in the associated\_namelist by a "%" character. The "&" or "%", if any, is stripped before the assignment is made. (For more on the difference between local, global, and complementary level assignments, see commands ASSIGN, GASSIGN, and CASSIGN).

Names assignments made using the /NAMES qualifier are by default "volatile" to memory clears. To make a "non-volatile" name assignment, precede the variable name by an exclamation point "!". (For more on the difference between volatile and non-volatile assignments, see commands NVASSIGN and CLEARMEM). The exclamation point can be combined with the & or % characters to force various types of assignments:

specifier

assignment made

&name	gassign name	=	value
%name	cassign name	=	value
!&name	nvgassign name	=	value
!%name	nvcassign name	=	value

Note that any specific names assignment can always be carried out following the ARGUMENTS statement with an ASSIGN statement, e.g.

```
ARGUMENTS (3I,2F)
ASSIGN name = ##$##3
```

would associate "name" with the third value read.

**FIELDS** = variable\_name: If specified, then upon return variable\_name will be set to the number of fields actually specified with the command or option. This can be different from the number given in the argument\_list if /MISSING=CONTINUE has been specified. For example, one could specify

```
ARGUMENTS (30A)
/MISSING=CONTINUE /FIELDS=HOWMANY
```

This would read up to 30 character fields following the appropriate command. The actual number found would be stored in the variable HOWMANY. If you wish the variable\_name/value assignment to be made at both the UC and IPS levels, precede "variable\_name" by an ampersand ("&"). If you wish the variable\_name/value assignment to be made only at the UC level, precede variable\_name by a percent sign ("%"). If you wish the variable\_name/value assignment to be made "non-volatile", precede the variable\_name by an exclamation point ("!").

**MISSING** = STOP (D) | CONTINUE:

Determines what the program will do if fewer values are specified with the appropriate command than are indicated in the argument\_list. By default, the program will stop with an error.

If /MISSING=CONTINUE is specified, then missing arguments will be set to 0 (integers), 0.0 (reals) or " " (characters) and the program will continue

If MISSING=CONTINUE is specified, it is legal to specify the corresponding command\_name without any arguments or equivalence character. Then all expected arguments will be set to 0, 0.0, or " ".

**MISS\_TEXT** = "string":

By default, if /MISSING=STOP is specified or implied, the default program reporting functions are used to report an error when fewer values are given with the command than were specified in the argument\_list. If MISS\_TEXT = "string" is specified, the supplied character "string", and only this string, will be reported to the user.

**SUBSTITUTE** = TEMPLATE | UC (D):

When a variable name occurs in a field being parsed as an argument, the name will, by default, be resolved against known variable name assignments previously made at the user-command (or global) level. This corresponds to `/SUBSTITUTE = UC`. By specifying `/SUBSTITUTE=TEMPLATE`, variable names will instead be resolved against known variable name assignments previously made at the template (or global) level. (See `ASSIGN` and `CASSIGN` for more details).

Examples:

1)

```
COMMAND = commname
  ARGUMENTS (A,I,F,2A)/EQUIVALENCE = ^/NAMES =(A1,I1,F1,A2,A3)
END COMMAND
```

would take the following command in the UC file:

```
COMMNAME text, 3*4, cos(1.), "text, number two", text three
```

and make the following assignments:

```
A1 = text
I2 = 12
F1 = 0.0
A2 = text, number two
A3 = text three
```

(Note the use of quotes to protect the comma in the second text string).

2)

```
COMMAND = commname
  OPTION = opt1
  ARGUMENTS (F,3A) /MISSING=CONTINUE/FIELDS=FNUM \
    /NAMES = (ARG1,ARG2,ARG,ARG4)
  END OPTION
END COMMAND
```

would take the following command in the UC file:

```
COMMNAME /OPT1 = (1.0/3.0, one, two)
```

and make the assignments:

```
ARG1 = 0.3333333
ARG2 = one
ARG3 = two
FNUM = 3
```

Note that we surround the arguments list by parentheses to "protect" the first field, where the delimiter "/" character is used as a divide operator. Also note that FNUM=3, reflecting that we read only three fields (although four were specified in the argument\_list).

**BOUNDS**

Syntax:

```

BOUNDS {/OUTFILE = file_equivalence_name}

    Statement A: (logical expression)
    Statement B:
    B1) RESET (variable_name = expression) {/TEXT = "string"}
    B2) CRESET (variable_name = expression) {/TEXT = "string"}
    B3) GRESET (variable_name = expression) {/TEXT = "string"}
(Choose 1 "B" type) B4) STOP {/TEXT = "string"}
    B5) WARNING {/TEXT = "string"}
                ...
    (continue Statement A+B pairs until done)

END BOUNDS

```

**Description:** The BOUNDS block provides a terse and convenient means to do a variety of variable checks, to reset variables, to echo warnings to the user, or to stop, based on the results of arithmetic comparisons. The BOUNDS block is initiated by the BOUNDS statement. This is followed by a series of statement pairs (Statements A&B above). As many statement pairs can be specified as desired, followed by an "END BOUNDS" statement to complete the BOUNDS block. Blank lines and comment lines within a BOUNDS block will be skipped.

Each pair of statements in the bounds block is evaluated sequentially, and the appropriate action (if any) is taken.

**Statement A:** This is any valid logical expression. The surrounding parentheses are required. The expression is evaluated, and if it is true, the associated Statement B is executed. If it is false, the associated Statement B is skipped.

**Statement B:** There are five options for Statement B. Only one can be specified.

**RESET** (variable\_name = expression): Sets the given variable\_name to the given value. variable\_name must adhere to the standard naming rules for variables (start with an alphabetic character, contain no special parsing characters or spaces). "expression" can be any arithmetic or character constant or expression. Assignment is done at the "local" IPS file level (same as ASSIGN). The parentheses surrounding the assignment are required.

**CRESET** (variable\_name = expression): Same as RESET, except the assignment is done at the "complementary" UC level (same as CASSIGN), and the assignment thus only affects the value of the variable at the UC level. The difference between RESET and CRESET is the same as that between ASSIGN and CASSIGN.

**GRESET** (variable\_name = expression): Same as RESET, except the assignment is done at the "global" IPS + UC level (same as GASSIGN), and the assignment thus affects the

value of the variable at both the UC and IPS levels. The difference between RESET and GRESET is the same as that between ASSIGN and GASSIGN.

**STOP:** Stop program execution immediately.

**WARNING:** Echo the associated text string, if any, and continue.

**OUTFILE** = file\_equivalence\_name: By default, any text output as a result of the BOUNDS block is output to the standard output file (STDOUT, usually unit 6). The OUTFILE option allows this output to be redirected to the file with the specified file\_equivalence\_name. This must correspond to the equivalence name of a file previously opened using the OPEN command, and must be a formatted file.

**TEXT** = "string": If, for any statement pair, the logical expression in Statement A is true, the text string specified by /TEXT="string", if any, will be echoed. If the text contains any special non-alphanumeric characters, it should be enclosed in quotes.

Text is output starting in column 1, ending in column 80, and is broken into multiple lines at appropriate break points if too long to fit on one line. If file\_alias\_name refers to a file opened as "BUFFERED", the text will be output starting at the "current" line, any may overwrite contents of the file regardless of whether or not the "/OVERWRITE" flag was specified when opening the file.

*Examples:*

```

DEFINE X = 2
BOUNDS
    (X.LT.3)
    RESET (X = X+4)/TEXT = "resetting X"
    (X.LT.5)
    WARNING /TEXT = "X is less than 5"
    (X.GT.5)
    WARNING /TEXT = "X is greater than 5"
    (X.GT.5)
    STOP /TEXT = have to stop
END BOUNDS

```

would produce

```

resetting X
X is greater than 5
have to stop

```

**CALLED**

## Syntax:

```

CALLED
    (...any series of directives...)
END CALLED

```

Description: The CALLED block is used to signify a block of directives which will be executed only when the command or option within whose definition the CALLED block appears is specified in the UC file. This is the default: unless indicated by a NOT\_CALLED block, all commands are implicitly in a CALLED block. The CALLED ... END CALLED block construct is only provided for programming clarity and to complement the NOT\_CALLED ... END NOT\_CALLED construct. But it is not necessary to use the explicit CALLED block construct.

A CALLED block should only appear within a COMMAND or OPTION definition block. As many CALLED blocks can be defined as are desired, and they can appear anywhere within the COMMAND/OPTION definition, with the following exceptions: They cannot be used in a DO or IF construct; They cannot be used in a DEFER block; and they cannot be nested within other CALLED or NOT\_CALLED blocks.

An explicit CALLED block cannot contain the following command: OPTION. Any IF or DO structures initiated in an explicit CALLED block must also be terminated in that block. Any GOTO within a CALLED block must refer to a label within the CALLED block.

*Example:*

```

COMMAND = commname
    CALLED
        ECHO "Routine commname called"
    END CALLED
END COMMAND

```

would result in the following when command "commname" was issued in the UC file:

```

Routine commname called

```

**CLEARMEM**

Syntax:

```

CLEARMEM
  { /MARKMEM }
  { /NONVOLATILE }
  { /NOPREVIOUS }
  { /FIRSTMARK }
  { /NOCOMMANDS }
  { /NONAMES }

```

Description: The CLEARMEM can be used to clear all integer, real and character variable registers. If CLEARMEM is called without any qualifier, all variables (both IVS and user-defined) become undefined. The only exceptions are:

- 1) Index variables for DO-loops, which never become undefined once used.
- 2) Any user-defined named variables which were assigned as “non-volatile” (unless the /NONVOLATILE flag is given here). All variables assigned at the UC level are by default non-volatile. Variables assigned in the IPS script using NVASSIGN commands (or a !name construct in a /NAMES qualifier) are non-volatile.
- 3) Any named variables for which the name was originally defined before the portion of memory being cleared, even if the value of the variable changed during during the span of memory being cleared (unless the /NONVOLATILE and/or /NOPREVIOUS flags are given).
- 4) Statement labels (unless the /NONVOLATILE flag is given).

CLEARMEM does not affect any files which are opened, commands which are currently defined, or control parameters which were assigned using the DEFINE command.

By default, when the CLEARMEM command is issued, all command history information, with the exception of the current command, is reset. The current command is now considered the "first" command issued. This will have ramifications for any structure which is dependent on the number of times a specified command has been issued, such as a DEFER block, or a MAXCALL or EXCLUSIVE statement. It will also affect any IVS specifier where the command name field is explicitly specified.

CLEARMEM can be useful to help affect a "fresh start" of the program, or to clear up memory when a considerable amount of storage has been used, and is no longer needed.

When a long series of commands is to be performed, or when one is carrying out a large number of iterations of a do loop containing commands requiring significant

storage, it is often important to conserve storage memory. This can frequently be done by either issuing a CLEARMEM command at the appropriate time, or by judicious use of the MARKMEM ... CLEARMEM/MARKMEM block.

**MARKMEM:** If this qualifier is specified, memory will be cleared back to a "mark" set by the user with the MARKMEM command. Variable storage used since the mark was set will become available for reuse when CLEARMEM/MARKMEM is specified. By default, 1) any variable names *initially* defined between the MARK and the CLEARMEM/MARKMEM become undefined; 2) an IVS cannot subsequently refer to any variable stored after the mark (and before the CLEARMEM/MARKMEM was issued). In addition, by default the command history pointers are reset to indicate that the command being executed when the mark was issued is the "current" command (and all subsequent history between the mark and the clear is lost).

One can set more than one memory MARK. In this case, CLEARMEM/MARKMEM will clear memory back to the most recent MARK encountered. This will clear the mark, so that it effectively no longer exists. The next time CLEARMEM/MARKMEM is encountered, it will again clear memory back to the mark most recently set, and then clear the mark. And so on. If a CLEARMEM/MARK is issued and there is no corresponding mark left to clear, the result will depend on whether a mark has *ever* been issued during this program execution:

- A) If a mark has ever been set, memory will be cleared back to the last mark which was set (even though it was subsequently cleared).
- B) If no mark has ever been set, all real and character memory (but not variable names) will be cleared.

Note that a CLEARMEM command by default completely unsets all marks, and if CLEARMEM/MARK is issued after a CLEARMEM command, but before any additional mark is set, action (B) above will result.

**NONVOLATILE:** By default, any memory clear effected by a CLEARMEM command will not affect the values of any variables stored as "non-volatile." (e.g. using a NVASSIGN command or a !name specifier in a /NAMES qualifier). If /NONVOLATILE is specified, non-volatile variables initially defined during the span of memory being cleared will also become undefined.

Specifying NONVOLATILE also gives the behavior for NOPREVIOUS (see below).

**NOPREVIOUS:** By default, any named variable which was originally defined before the section of memory being cleared will retain its proper value after the memory clear, even if the value assigned to the variable changed during the span of memory being cleared. If /NOPREVIOUS is specified, the variable\_name/value assignments for such variables will not necessarily be maintained.

Such values are never maintained if /NONVOLATILE has been specified.

**FIRSTMARK:** If specified, memory is cleared back to the first mark set, regardless of how many marks have subsequently been set. All marks except for the first one are cleared. Otherwise, the behavior of the FIRSTMARK qualifier is the same as described for MARKMEM above.

If only one or no marks have been set, CLEARMEM/FIRSTMARK is completely equivalent to specifying CLEARMEM/MARK (see description above).

FIRSTMARK is useful to allow the assignments made at the beginning of a program to remain unchanged while clearing all other data. Once the series of assignments which one wishes to remain unchanged is made, the first mark (MARKMEM) is set. Then, when a CLEARMEM/FIRSTMARK command is subsequently issued, these initial assignments will not be changed, but other stored data will be cleared. This is one way to reinitialize all variables in a program except those you wish to remain inviolate. A second way to assign a set of inviolate assignments is to use “non volatile” assignment statements. Any such assigned variables will not change with any CLEARMEM command. (See NVASSIGN).

**NOCOMMANDS:** If specified, then the user command call history is not changed when a CLEARMEM/MARKMEM command is processed. This allows an IVS specifier to be used to reference a value defined since the MARK *if* the reference is made before additional variables are stored. Once additional values are stored, the IVS/value association will become unreliable. If you wish to use the /NOCOMMANDS qualifier, you should also use the /NONVOLATILE qualifier.

NOCOMMANDS has no affect if MARKMEM or FIRSTMARK is not specified. The effect of the NOCOMMANDS qualifier will be unreliable if NONVOLATILE is not specified.

*The NOCOMMANDS qualifier should generally be avoided, where possible, by judicious use of non-volatile named-variable assignments.*

**NONAMES:** If specified, then any variable names *initially* defined between the MARK and the CLEARMEM/MARKMEM command remain defined after the clear is performed. Variable names referring to integer values will behave as if no memory clear was performed. Names initially defined between the MARK and the CLEAR/MARK which refer to real or character values can only be reliably referenced after the CLEARMEM statement *until* additional variables are stored. Once additional values are stored, the value/variable association for these values will become unreliable.

Do not use NONAME without specifying NONVOLATILE as well.

NONAMES has no affect if MARKMEM or FIRSTMARK is not specified.

**Note** that if a named variable is used *immediately* after a CLEARMEM/MARKMEM/NONAMES command, it will still contain the correct value until additional variables are stored. This is true regardless of the type of data associated with the name.

*Use of the NONAME qualifier should generally be avoided in favor of judicious use of non-volatile assignments.*

*The MARKMEM and CLEARMEM/MARKMEM commands are provided so that the skillful user can optimize memory usage when this is a consideration.*

The MARKMEM...CLEARMEM/MARKMEM construct should be used as part of definitions of frequently-issued commands for which permanent variable storage is unnecessary. See Example III.

In general, be careful when a MARKMEM ... CLEARMEM/MARKMEM sequence spans more than one command definition. Remember that IVS specifiers will become unreliable once the clear is made, as will any reference to character data stored during the block. Also recall that you must explicitly specify /NONAMES or all variable names initially defined in the block will become undefined.

*Examples:*

```
I)
    ASSIGN I = 3
    CLEARMEM
    ECHO <I>
```

would result in an error, as "I" would be undefined when the ECHO command was issued.

II)

```

ASSIGN BISTRING = " "
MARKMEM
DO I = 1,5
    ASSIGN BIGSTRING = BIGSTRING + "STUFF"
END DO
CLEARMEM/MARKMEM
    
```

The above block illustrates a common use of the MARKMEM and CLEARMEM/MARKMEM commands. The DO loop is used to create the concatenated string

STUFFSTUFFSTUFFSTUFFSTUFF

Now, because of the write-once scheme used by INTERFACE to store character data, prior to the CLEARMEM command, separate character memory has been used to store the strings "STUFF", "STUFFSTUFF", "STUFFSTUFFSTUFF", "STUFFSTUFFSTUFFSTUFF", and "STUFFSTUFFSTUFFSTUFFSTUFF". Obviously, we only need to store the final result. By issuing the CLEARMEM/MARKMEM command, we clear all the character storage back to the point before the DO LOOP. Recall that by default, when we do a CLEARMEM/MARKMEM, any named variable that was first assigned before the mark was set will retain its proper value when after the clear is carried out. Thus, since BIGSTRING was first assigned before the mark was set, the final value of this variable will remain.

The net result is that we have reclaimed all the character memory except that actually required by the final value of BIGSTRING.

III)

```

MARKMEM
ASSIGN I = "This is I"
NVASSIGN J = "This is J"
CLEARMEM/MARKMEM
    
```

In this example, variable I is assigned with a default assignment statement, while variable J is assigned with a "non-volatile" assignment statement. This means that after the CLEARMEM/MARKMEM statement is issued, variable I will become undefined, while variable J will retain its assigned value of the character string "This is J" (assuming neither I nor J was previously defined before the MARKMEM was issued)..

IV)

```

COMMAND = ECHOIT
MARKMEM
ARGUMENTS (3A) / NAMES = (ECH1,ECH2,ECH3)
DO ILOOP = 1,3
    ECHO "Argument <I> = <ECH<I>>"
END DO
CLEARMEM/MARKMEM
    
```

END COMMAND

In this example, we define a command, "ECHOIT", which takes three character arguments and echoes them to the user, one at a time. By default, each time this command is executed, storage would be allocated for A) The three arguments provided with the command; B) Variable names "ECH1", "ECH2", "ECH3". If this command is executed many times, this can require a significant amount of unnecessary storage, if the argument names are not used again. So we reclaim the storage required for the arguments after we are done executing the command. Placing the MARKMEM before any commands which result in variable storage, and placing the CLEARMEM/MARKMEM command at the end of the definition is a common method for reclaiming all the space used by a code block.

Note that if any of the names "ECH1", "ECH2" or "ECH3" had been initially defined before the command was issued, the final value assigned to each of these names would remain defined after this block was executed.

V)

```
ASSIGN NAME = "harry"
ASSIGN DATE = "JAN 20 1992"
MARKMEM
    (...any series of commands...)
MARKMEM
    (...any series of commands...)
CLEARMEM/FIRSTMARK
```

In this example, two variables, NAME and DATE are assigned at the beginning of an IPS script. Regardless of what else we do, we don't want these values to ever be cleared, so we set the first MARK immediately after these assignments. Then, no matter how many MARK's we subsequently set, we can clear memory using the CLEARMEM /FIRSTMARK command. The assignments that preceded the first MARK will remain safe. Note that if we issued a CLEARMEM command without any qualifiers, *all* memory would be cleared, including variables assigned before the first mark.

VI)

```
NVASSIGN NAME = "harry"
NVASSIGN NAME = "JAN 20 1992"
    (...any series of commands...)
CLEARMEM
```

In this example, we assign two variables, NAME and DATE, in the "non-volatile" mode (NVASSIGN). We can then carry out any series of commands and then issue a CLEARMEM command. This will clear all volatile memory storage, but the NAME and DATE assignments will never be affected unless the /NONVOLATILE qualifier is also specified. NVASSIGN commands can appear anywhere in the IPS script. All such assignments are maintained through all CLEARMEM commands.

**CLOSE**

Syntax:

```
CLOSE file_equivalence_name  
      { /DELETE }
```

Description: The CLOSE command closes a file previously opened using an OPEN command. When the file is closed, the equivalence\_name becomes undefined. To reaccess the file after it has been closed, it must be explicitly OPENed again.

*file\_equivalence\_name*: The equivalence name attached to the file when it was opened using the OPEN command.

**DELETE**: If specified, the file will be deleted upon closure. By default, the file will not be deleted when closed.

**COMMAND**

Syntax:

```
COMMAND = command_name { /REPLACE } { /LOCKOUT }
          (...any series of commands...)
END COMMAND
```

**Description:** The COMMAND block defines commands which will be recognized in the UC file. When a recognized command is issued in the UC file, the commands in the corresponding COMMAND block are executed. Commands within a COMMAND block are executed in the order they are written. The equals sign is required between COMMAND and command\_name.

**command\_name:** The name of the command name to be defined. Valid command names must consist only of alphanumeric characters and underscore "\_" characters. The special star character "\*" can be used to define acceptable abbreviations. Characters before the "\*" are required. Characters after the "\*" are optional, but must match exactly if provided. If no "\*" is specified, the command name cannot be abbreviated. E.g.

```
COMM*AND    would match
COMM
COMMAN
COMMAND
```

but not

```
COMMANT
COMMANDS
```

The required part of command\_name must not match the name (or allowed abbreviation) of another previously defined command. If it does, an error will be reported. (See section XV of Chapter 3: Features and Considerations for more details on command name matching).

**REPLACE:** By default, if the required (pre "\*") part of the command\_name is redundant with a command (or allowed abbreviation) which has already been defined, an error will be reported. If /REPLACE is specified with the new command definition (and if /LOCKOUT has not been specified for the first command; see below), the new COMMAND block definition will replace that of the old one.

**LOCKOUT:** If specified, then the current COMMAND block definition cannot be replaced by any subsequent COMMAND block definitions. If a subsequent COMMAND block attempts to define a command name which is redundant with the current name, a warning message will be reported, and the second COMMAND block will be ignored.

REPLACE and LOCKOUT can be useful when several template files are being read. For example, there may be a single fixed "master" IPS script. In this case, users could customize the script by reading in their own IPS files (using a GETTEMPLATE command in the UC file). Using /LOCKOUT qualifiers, the administrator could ensure that important commands in the master IPS script didn't get accidentally modified.

*Example:*

We could define the following command:

```
COMMAND = dost*uff / LOCKOUT
        ECHO "This is command dostuff"
END COMMAND
```

Now, if subsequently a second command definition is read (due to a GETTEMPLATE command in the UC, for example):

```
COMMAND = dost*uff /REPLACE
        ECHO "This is the NEW command dostuff"
        ECHO "It replaces the old one"
END COMMAND
```

This command definition would replace the first one. Note that if either /LOCKOUT had been specified with the first definition, or /REPLACE had not been specified with the second, no replacement would have occurred.

**COPY**

Syntax:

```

COPY {lines}
    /INFILE=file_alias_name
    * /OUTFILE=file_alias_name
    { /IN_FORM = format}
    * { /OUT_FORM = format}
    { /UNTIL}
    /ISTART = M}
    * { /STORE} { /NOSTORE (D)}
    /QUOTE = quote_character}
    * { /REPLACE {= (M1,M2)} }
    { /STRICT}
    { /DELIMIT = delimit_character}
    { /FUNCTIONS}
    { /SUBSTITUTE (= template | uc (D))}
    { /MISSING = stop | continue | next}
    { /MISS_TEXT = "string"}
    { /END = stop | continue}
    { /END_TEXT = "string"}
    { /NAMES = (associated_namelist)}
    { /LINES = variable_name}
    { /FIELDS = variable_name}

```

Description: COPY is used to copy a specified number of lines of data from one file to another. The user can specify the format of the data for either file, or accept defaults. The user can also specify that the data which is being copied be stored in memory for future use.

The COPY command is very similar to the READ command. The only differences involve the options marked with an asterisk (\*) above. These are only options described here. The remainder are described under the READ command description.

**OUTFILE** = file\_alias\_name: Specifies the alias name of the file to which the COPY command will write. file\_alias\_name must correspond to a file which has previously been opened using the OPEN command.

**OUT\_FORM** = format: Specifies the format to be used in writing the data from each read. The format can be either a standard FORTRAN format specifier, or a machine-dependent list-directed free-format specifier (\*). ("(\*)" can only be used with sequential access file; do not use this specifier with direct access or "buffered" files). See the section on FORMAT SPECIFICATION for more information on format descriptors.

If /OUT\_FORM is not specified:

If a standard FORTRAN-type format was specified for IN\_FORM, this format statement will be used for output.

If IN\_FORM was specified to be free-format type, then the machine-dependent free format (\*) specifier will be used. (This will result in an error if writing to a formatted file opened as "buffered" or direct-access; in such a case you *must* specify a FORTRAN-type format for /OUT\_FORM, if IN\_FORM=(\*)).

If IN\_FORM is not specified, then the data will be output as character strings 80 characters long (A80), just as it was read.

If a line-number is specified in the /OUT\_FORM format specifier, and multiple lines are being written, the line number in the format refers to the first line written. Subsequent lines will be written sequentially from this point.

**STORE:** If specified, then any data transferred by the COPY command will also be stored in memory, and can be accessed by subsequent commands.

**NOSTORE:** If specified, then data transferred by the COPY command will not be stored in memory. Since memory is fixed and may be limited, it is recommended that data not be stored if it will not be required after the COPY command is carried out. This is the default.

Note that STORE and NOSTORE have the same affect as they do for READ, but the default (NOSTORE) is different here.

**REPLACE** {=(M1,M2)}: By default, when a write to a pre-existing line is made to a file which was opened with the options /BUFFERED and /OVERWRITE=MERGED, non-blank characters in the new line of data to be written are merged into the pre-existing line, while any columns of the new line which are blank remain as they were in the old line. If /REPLACE = (M1,M2) is specified for such a buffered file, then the characters between columns M1 and M2 (inclusive) of the old line are completely replaced by the data of the new line in these columns, regardless of the flag specified for /OVERWRITE.

The range specifier "= (M1,M2)" is optional. If /REPLACE is specified with no arguments, the entire old line will be replaced by the entire new line.

If M1 <= 0, M1 = 1 will be used. If M2 <=0, M2=record\_length will be used. REPLACE applies to every *physical* line which is written, not every COPY/REFORM *command*.

/REPLACE {=(M1,M2)} has no affect for files not opened as "buffered", or for files opened with /OVERWRITE=REPLACE.

See the READ command for discussion of the other options.

*Examples:*

1)

```
COPY 3 /INFILE = file1 /OUTFILE = file2 \
      /IN_FORM = (2S,3I):2(ff) /OUT_FORM = (3I5)
```

would read 3 lines, starting with line 2, from the file which was opened with equivalence name "file1". For each line, the first two fields would be skipped, and the following 3 integer values would be read in free format. These would then be written to the "current" line in the file opened with equivalence name "file2" in format (3I5).

2)

```
COPY/INFILE = file1 /OUTFILE = file2 /REPLACE=(4:10)
```

Assume the line to be read from file1 is

```
Line          one.
```

and file2 was opened with the /BUFFERED/OVERWRITE=MERGED qualifiers, and the line to be (re)written in file2 is currently

```
AAAAAAAAAAAAAAAA
```

Then after the copy, this line in file 2 will be

```
Line          AAAone.
```

**DEFER**

Syntax:

```

DEFER {command_name}
      { /AFTER (D) } { /BEFORE }
      { /END = NEVER (D) | ALWAYS | FIRST | LAST }
      { /NOT }
      { /TIMES = M }
      { /EACH }
      { /UNTIL = command_name2 }
      { /CATCHUP (D) | NOCATCHUP }

      ( . . . )

END DEFER

```

Description: The DEFER block construct can be used to define a series of commands which are not to be executed until/unless the specified command has been issued the specified number of times. A DEFER block can also be used to specify a series of commands to be executed at the end of an INTERFACE run.

All commands within the DEFER and END DEFER statements are part of the command block. Any valid command (including another DEFER block) can appear within a defer block, with the following exceptions: ARGUMENTS; MAXCALL; OPTION; CALLED; NOT\_CALLED. GOTO statements must not transfer into a DEFER block.

DO and IF blocks initiated within a defer block must also be terminated within the DEFER block. Any GOTO statement in a DEFER block must refer to a label within the DEFER block.

When conditions are appropriate for the commands within the DEFER block to be executed, the block is "released", and these commands are executed sequentially. The current value of "command" when a defer block is "released" is the current (or last) command parsed when the "release" was triggered (which will not necessarily be the same as the COMMAND block within which the DEFER block occurred). This will affect both the IVS associated with any variable stored during a DEFER block, and the default value used for the "command" field in any IVS specifier used in a DEFER block. For this reason, it is often valuable to assign explicit names to values read during a DEFER block, and to use IVS specifiers with the "command" field explicitly specified.

If the invocation of a command results in more than one DEFER block being released at the same time, the blocks will be executed in the order in which they were originally encountered.

If a DEFER block is still active (has not been executed, or was specified with the /EACH qualifier), reading the same DEFER block again will have no affect. When a

DEFER block has become inactive (has been executed and /EACH was not specified), reading the same definition again will reactivate the DEFER block.

**command\_name:** Name of the command whose invocation is required to "release" the DEFER block. This command must correspond to a command which is defined by a COMMAND block in the current IPS file, or which has already been defined in a previously read IPS (template) file. Valid abbreviations and aliases can be used. `command_name` refers to commands specified in UC file, and must refer to a command defined in the IPS file (not an intrinsic command).

The `command_name` field can be left blank. In this case the DEFER block will only be released if the `"/END ="` option is specified.

**AFTER:** When the appropriate invocation of the specified command occurs, the command definition for `command_name` is executed, and then DEFER block is released. This is the default.

If more than one DEFER block hinges on the invocation of the same command, the blocks will be executed in the order in which they were encountered.

**BEFORE:** When the appropriate invocation of the specified command occurs, the DEFER block is released immediately, before the command definition for `command_name` is executed.

If more than one DEFER block hinges on the invocation of the same command, the blocks will be executed in the order in which they were encountered.

**END = NEVER (D) | ALWAYS | FIRST | LAST:**

**END = NEVER:** If the specified `command_name` is not invoked the required number of times, the DEFER block is never released. This is the default.

**END = ALWAYS:** The DEFER block will always be executed after all command in the UC file have been executed, if it is not released sooner. DEFER blocks released at the end of UC file parsing because of `END=ALWAYS` qualifiers will be executed in the order in which they were originally encountered. To change this default, use `END=FIRST` or `END=LAST`.

**END = FIRST:** Same as `END=ALWAYS`, except that in this case if the DEFER block is not released until the end of the program, it will be the first DEFER block released at that time.

**END = LAST:** Same as `END=ALWAYS`, except that in this case if the DEFER block is not released until the end of the program, it will be the last DEFER block released at that time.

If more than one DEFER block with `"/END=FIRST` remains to be released after all commands in the UC file have been parsed, these remaining DEFER blocks will be

executed in the reverse of the order in which they were encountered during program execution (so the last remaining DEFER/END=FIRST block is the first block executed).

If more than one DEFER block with /END=LAST remains to be released after all commands in the UC file have been parsed, these remaining DEFER blocks will be executed in the order in which they were encountered during program execution (so the last remaining DEFER/END=LAST block is the last block executed).

In summary, after all UC file commands have been parsed, the following are released, in the order given:

- 1) DEFER blocks not yet executed, for which /END=FIRST was specified  
(last encountered, first executed)
- 2) DEFER blocks not yet executed, for which /END=ALWAYS was specified  
(first encountered, first executed)
- 3) DEFER blocks not yet executed, for which /END=LAST was specified  
(last encountered, last executed)

Note that DEFER blocks are released *after* any commands within NOT\_CALLED blocks (if any) have been executed.

**NOT:** If specified, then the DEFER block will be released when the first defined command which is not the same as the specified command\_name is issued. The block will be released either before or after this non-matching command, depending on whether the BEFORE or AFTER option has been specified.

Only commands defined in the IPS (template file) can trigger the DEFER block. The DEFER block with /NOT specified will not be executed when an intrinsic command (e.g. DO, IF, etc.) is issued.

/EACH can be specified with /NOT. This will result in the contents of the DEFER block being executed each time any command other than the specified command\_name is issued.

command\_name can be omitted with the /NOT option. In this case, the DEFER block will be released when the next IPS defined command is issued, regardless of what it is. Specifying /EACH and omitting command\_name will result in a DEFER block which is executed before/after every IPS-defined command is issued.

**TIMES = M:** Specifies the number of times command\_name must be invoked in the UC file before the DEFER block will be released. M can be any numerical constant or expression. By default, a DEFER block is released the first time command\_name is invoked (M = 1).

**EACH:** By default, the DEFER block will only be executed one time. Once it is executed, the DEFER is cleared. If /EACH is specified, then the DEFER block will be executed every time command\_name is given. (If TIMES = M has been specified, then the DEFER block will be executed every time command\_name is given after the Mth invocation).

If /EACH is not specified, then once a DEFER block is cleared, the block can be set again if it is again encountered.

A DEFER block which has been executed at least one time will not be re-executed after all UC file commands have been parsed, whether or not /EACH is specified.

**UNTIL = command\_name2:** By default, a DEFER block remains in effect for the entire Interface session, unless it is released and cleared by the appropriate conditions, as specified above. If UNTIL is specified, the DEFER block will be cleared (but not executed) when/if command command\_name2 is issued. If command\_name2 is issued before the conditions specified above for release have occurred, the DEFER block will not be executed. If the DEFER block is executed and cleared before command\_name2 is issued, the UNTIL condition will have no additional effect.

**CATCHUP:** When a DEFER block is encountered, if the specified command\_name has already been invoked the requisite number of times (1 or set by /TIMES=), the DEFER block will be "released" immediately. This is the default.

**NOATCHUP:** When a DEFER block is encountered, it will not be released until at least the next invocation of command\_name, regardless of how many times command\_name has already been issued.

*Examples:*

1)

The block:

```
DEFER /END = ALWAYS
    ECHO "That's all folks!"
END DEFER
```

would result in the line

```
That's all folks!
```

being echoed to the user after (and not until) the UC file had been entirely parsed.

The IPS construct

```
DEFER COMNAME /TIMES=3
    ECHO "I =" <I>
END DEFER
COMMAND = COMN*AME
```

```
        ECHO "comname called"  
    END COMMAND
```

combined with the UC file construct:

```
DO I = 1,5  
    GASSIGN I = I  
    COMNAME  
END DO
```

would result in

```
comname called  
comname called  
comname called  
I = 3  
comname called  
comname called
```

2)

The block:

```
DEFER    /NOT/EVERY/AFTER/UNTIL=stoppit
        ECHO "IPS-defined command just executed"
END DEFER
```

would result in the line "IPS-defined command just executed" being echoed to the user after each and every invocation of an IPS-defined command from the UC file until the command "stoppit" was issued. After the command "stoppit" was issued, this defer block would be cleared.

**DEFHELP**

Syntax:

```
DEFHELP "string"
  { /START = M}
  { /END = M}
  { /INDENT = M}
  { /BREAKCHAR = character}
  { /NOBREAKCHAR}
  { /FORCEBREAK = character}
  { /NOFORCEBREAK}
  { /LINE = M}
  { /NOADVANCE}
  { /DEFAULT}
```

Description: DEFHELP defines text which will be echoed to the user if, and only if, an appropriate HELP command is issued (see "HELP"). Unless a HELP or MANUAL command is issued from the UC file, DEFHELP commands are skipped.

When the user specifies the HELP command with a particular command\_name, all DEFHELP commands within the corresponding COMMAND block (but not within a nested OPTION block) will be executed.

When the user specifies the HELP command with a particular command\_name and option\_name (or a "wildcard" option name), all DEFHELP commands nested within the corresponding OPTION block will be executed.

When a HELP command is issued, DEFHELP commands within the appropriate COMMAND or OPTION block will all be executed, regardless of other control structures (such as IF, DO, NOT\_CALLED or DEFERED). A DEFHELP command can appear anywhere within any COMMAND or OPTION block. You can specify as many DEFHELP commands as desired.

The syntax of the DEFHELP command is identical to that for the ECHO command, and in fact the arguments of the DEFHELP command are passed to the same routines to output the text "string". There is one difference: Do not specify the "/OUTFILE=" qualifier with the DEFHELP command. If you wish to send help output to a specific file, use the MANUAL command from the user interface.

**"string"**: The text string to be output. If the text string contains any option delimiter characters (i.e. embedded slash characters "/"), it must be surrounded by quotes. Surrounding quotes are also required to maintain any runs of multiple spaces. Otherwise, runs of multiple spaces will be reduced to a single space. If you wish to use any character function operators, the text string must be enclosed in <>'s, e.g. <"yes"(1:1)>, not "yes"(1:1).

The "string" field can be omitted, in which case a blank line will be output.

By default, the text is output to STDOUT (usually unit 6).

**START** = M: Specifies in what column the output will start. The initial default is column 1. The value specified for START must be  $\leq$  the specified (or default) value for END.

**END** = M: Specifies the last column to be used for output. If "string" exceeds the space available between the allowed first and last columns, the string will be broken into as many lines as are required to keep the string within the specified boundaries on each line.

For non-buffered, non direct access files, the initial default value of END is 79.

For buffered or direct access files, the initial default value of END is the recordlength specified when the file was opened. If END is specified to be larger than the recordlength for a buffered/direct access file, it will be reset to the recordlength.

If a different specific default value for END is specified with the /DEFAULT option, the defaults described above can subsequently be restored by specifying a value of END<0 together with the /DEFAULT qualifier.

**INDENT** = M: Specifies a relative offset for the first column to be used for all lines after the first (if any). If INDENT is set, all lines after the first will start in column START+INDENT (1+INDENT if START was not specified). If INDENT is not specified, the initial default is that all lines start in the column specified or implied for START. If START+INDENT < 1, all lines after the first will start in column one. START+INDENT must not be larger than the value being used for END. For DEFHELP, a value of 5 is used by default for indent.

**BREAKCHAR** = character: Used to specify a character at which text may be divided when "string" will not fit entirely on one line. By initial default, a space break\_character is used. Text is broken after the specified break character(s).

The user may specify up to five break\_characters, by specifying the /BREAKCHAR = character option multiple times. If any /BREAKCHAR is specified, all break characters, including a space (if desired) must be specified. Only one character may be specified after each /BREAKCHAR= directive, and it should not be enclosed in quotes.

If no suitable break character is found to break the line within the required borders, the line will be arbitrarily broken at the right border (in the column specified/implied for END).

To specify a space, you must use /BREAKCHAR = ^  
To specify a forward slash, you must use /BREAKCHAR = ~

**NOBREAKCHAR:** Specifies that no special break characters will be used (not even a space). In this case, all lines will be broken arbitrarily at the right border column (the column specified/implied for END).

**FORCEBREAK = character:** Allows specification of a special "forced break" character. If FORCEBREAK is specified, the given character will cause a line break every time it is encountered in "string". The FORCEBREAK character itself is not echoed. If no FORCEBREAK character is encountered before the defined column limits are reached, the regular BREAKCHAR rules for line breaks will apply. Only one FORCEBREAK character may be defined. By initial default, no forcebreak character is used.

To specify a space, you must use /FORCEBREAK = ^

To specify a forward slash, you must use /FORCEBREAK = ~

**NOFORCEBREAK:** Specifies that no "forced break" character will be used. This option can be used to override a default previously set with an DEFHELP/DEFAULT or ECHO/DEFAULT command.

**LINE = M:** The line number at which output is to start. If not specified, output will start on the "current" line (1+the last line written to the file). The special relative line specifiers "\$+M" and "\$-M" may be used, if desired, to specify a line relative to the end of the file (first unwritten record). E.g. \$-1 is the last written line of the file. It is unlikely that you would use this option with DEFHELP, though it is included for consistency with ECHO.

**NOADVANCE:** If specified, the "\$" descriptor will be included in the output format statement used by DEFHELP. If the line is being written to STDOUT (default), this will have the effect of leaving the cursor at the end of the echoed string. It is unlikely that you would use this option with DEFHELP, though it is included for consistency with ECHO.

**DEFAULT:** If specified, then the values given for /START, /END, /INDENT, /BREAKCHAR, /NOBREAKCHAR, /FORCEBREAK and/or /NOFORCEBREAK will become the defaults on all future ECHO (DEFHELP) commands, unless/until reset by another DEFHELP/DEFAULT or ECHO/DEFAULT command. Only the options specified with the particular DEFHELP/DEFAULT command will have their defaults reset.

When /DEFAULT is specified, "string", if any, is ignored. There is no output from a DEFHELP/DEFAULT command.

Note that any defaults reset by a DEFHELP command also become the defaults of all ECHO commands (and vice-versa). Typically, DEFAULT will be specified with the ECHO command rather than the DEFHELP command, though either is legal.

*Example:*

Supposed you defined a command:

```
COMMAND = COMNAME
  (...)
  DEFHELP "COMNAME is a sample command for example. "/START=2
  (...)
  OPTION = OPTNAME
  (...)
    DEFHELP "OPTNAME is an option of COMNAME. "/START=2
    (...)
  END OPTION
END COMMAND
```

Then if the user specified

```
HELP COMNAME
```

they would see

```
Command = COMNAME
COMNAME is a sample command for example
```

If the user specified

```
HELP COMNAME/OPTNAME
```

they would see

```
Option = OPTNAME
OPTNAME is an option of COMNAME.
```

**DEFINE**

Syntax:

```
DEFINE variable_specifier = value
```

Description: The DEFINE command can be used to define the values of certain variable specifiers. The "variable\_specifier" can be any of a number of constants which are recognized, and which will affect various aspects of how INTERFACE behaves.

Depending on the variable\_specifier, "value" is either a numerical or character value. Variable expressions can be used. Character string values *must* be specified within quote characters

The variable\_specifier name cannot be abbreviated.

It is legal to use variable names which are the same as the names recognized as variable\_specifiers. The two descriptors are entirely independent, and changing the value of one does not affect the value of the other.

Note that the examples and discussion in the documentation generally assume the default values for OSP, EQV and QUO. If you change these, you need to take this into account.

A special note about DEFINE commands: When an IPS file is parsed, it is first checked for any DEFINE statements at the beginning of the file. If any DEFINE statements appear before *any* other statement types (even comments and blank lines), these will be executed before parsing the remainder of the IPS file. Thus, for example, to change the comment character to be used for the IPS file itself, the "DEFINE CMT =" statement *must* appear before any other statement (except, perhaps, another DEFINE statement).

<u>variable specifier</u>	<u>type*</u>	<u>description</u>
ICASE	I	=0: Case insensitive (default) =1: Case sensitive
SP1,SP2,SP3	A1	SP1, SP2 and SP3 are characters which separate multiple arguments in a free-format read. By default, SP1 = "," and SP2 = SP3 = " " (space). Do not affect reads due to an ARGUMENTS command, where a comma (only) is always used.
OSP	A1	Character which separates options from one another in a command (default is "/"), except for IPS-defined commands which are issued in the UC file.

OSPU	A1	Character which separates options from one another in an IPS-defined command which is issued in the UC file. Default is “/”.
EQV	A1	Character which indicates equivalence between two arguments (default is "=").in all syntaxes except for IPS-defined commands which are issued in the UC-file.
EQVU	A1	Character which separates commands and options from their associated arguments for IPS-defined commands which are issued in the UC file. Default is “=”. Can also be modified by the EQUIVALENCE option of the ARGUMENTS command.
QUO	A1	Character which begins and ends a "quoted" text string. Default is double quote (")
INOMT	I	=0: Attempt to use a variable name which has not been previously defined will result in the value RNOF  =1: Attempt to use a variable name (either IVS or user-defined) which has not been previously defined will result in an error (default).
RNOF	R	If INOMT=0, RNOF is the value used in place of any previously undefined variable name. Default = 0.0.
IVERIF	I	=0: Commands are not echoed as they are parsed (default). =1: Commands read from UC files are echoed as parsed. =2: Commands read from IPS files are echoed as parsed. =3: Commands read from both UC and IPS files are echoed as parsed.
CMT	A1	Character which starts a comment line (default = "!").  Note that CMT is used in translating the IPS file to the “template” file, but once the template file is created, subsequent changes to CMT will have no affect on parsing commands therein (CMT will always affect parsing of the UC file).
BUFFIL	A6	Six character name of temporary files opened for "buffered" files. The name must contain the sequence "%%", which will be replaced at run-time by the temporary unit number. Default = "XBUF%%"

By default, a file with a unique system-dependant name will be used for any temporary file which is opened. For most applications, this default is *strongly* recommended.

CONT	A1	Continuation character. Any line which ends with this character will be continued on the next line. Default = "\".
		Note that CONT is used in translating the IPS file to the "template" file, but once the template file is created, subsequent changes to CONT will have no affect on parsing commands therein (CONT will always affect parsing of the UC file).
XCS	I	CHAR(XCS) gives the character used internally to separate stored character strings. This character should be a non-printing character. Default = 12.
IRECLT	I	Record length for template file records. Default = 80.
SMALL	R	Small real number used internally in tolerance test comparisons with 0. Values smaller than SMALL are assumed to be 0.0 Default = 1.0D-7.
INFOO	I	=0: Do not echo informational messages =1: Echo informational messages (default) =2: Abbreviated format informational messages are echoed.
IWRNO	I	=0: Do not echo warning messages =1: Echo warning messages (default) =2: Abbreviated format warning messages are echoed.
IWRNS	I	Program will be stopped if $\geq$ IWRNS warning messages occur and IWRNS $>$ 0. Default is IWRNS = 0.
FORMEC	A13	Format to be used when performing <expression> variable substitution, and the resultant is of real type. Any standard FORTRAN-type format specifier is acceptable. The format specifier must be $\leq$ 13 characters.  Default = G16.5
PROMPT	A15	Specifies a "prompt" string to be echoed, without line-

advance, to STDOUT before each read at the UC level when UC commands are being read from STDIN. PROMPT has no affect if UC commands are being read from a file. If PROMPT is set to a blank string (default), no prompt string is issued.

IHLALP	I	=0: When the HELP command is issued without any qualifiers, the resulting list of known commands is alphabetized.  =1 When the HELP command is issued without any qualifiers, known commands are listed in the order they were defined.
DEFDIR	A80	String to use in place of the string DEFDIR: when this string appears as part of the file specification in a REDIRECT command. See REDIRECT.
SEED	I	Seed to use in the random number generator function RAN(X1,X2). The random number series is generated from the seed using a "multiplicative congruential" algorithm. The same seed is used by default every time INTERFACE is run, leading to the same sequence of random numbers. You must specify a different seed to generate a different sequence. Default = 71277
IFOCNT	I	=0: If a command is issued in the UC file which is not recognized, the program will stop (default).  =1: If a command is issued in the UC file which is not recognized, the program will continue with a warning.
IIRST	I	do not use.
ISUPER	I	Not currently used.
IWIDE	I	Not currently used.

\*type: I indicates an integer value is expected. R indicates a real value is expected.  
Am indicates a quoted character value of length m (or less) is expected.

*Examples:*

```
DEFINE IVERIF = 1
```

would cause all commands in the UC file to be echoed to STDOUT as they were being parsed.

```
DEFINE FORMEC = "F10.0"
```

would case all <expression> constructs which evaluated to reals to be replaced by floating representation F10.0. Note that if <expression> construct evaluates to a value which does not fit in this format, e.g. 1.0D+12, an error will occur.

```
DEFINE CMT = "#"
```

would result in any line starting with the pound sign being treated as a comment.

**ECHO**

Syntax:

```

ECHO "string"
    { /OUTFILE = file_alias_name }
    { /START = M }
    { /END = M }
    { /INDENT = M }
    { /BREAKCHAR = character }
    { /NOBREAKCHAR }
    { /FORCEBREAK = character }
    { /LINE = M }
    { /NOADVANCE }

```

Description: The ECHO allows a specified string of text to be output to a chosen file (or STDOUT). ECHO also functions as a simple text formatter. The starting column and ending columns for text can be specified. If the text string exceeds the length of a resulting line, it will be broken up into as many pieces as required to fit within the chosen bounds on multiple lines.

**"string"**: The text string to be output. If the text string contains any option delimiter characters (i.e. embedded slash characters "/"), it must be surrounded by quotes. Surrounding quotes are also required to maintain any runs of multiple spaces. Otherwise, runs of multiple spaces will be reduced to a single space. If you wish to use any character function operators, the text string must be enclosed in <>'s, e.g. <"yes"(1:1)>, not "yes"(1:1).

The "string" field can be omitted, in which case a blank line will be output.

By default, the text is output to STDOUT (usually unit 6).

**OUTFILE** = file\_alias\_name: If given, specifies the alias name of the file to which the string is to be echoed. This must correspond to an alias name assigned in a previous OPEN statement.

Note that if ECHO is used to write to a file opened as "/BUFFERED", it will overwrite the contents (if any) of the line(s) to which the ECHO write occurs. This will be the case whether or not the "/OVERWRITE" flag was specified when the file was opened.

**START** = M: Specifies in what column the output will start. The initial default is column 1. The value specified for START must be <= the specified (or default) value for END.

**END** = M: Specifies the last column to be used for output. If "string" exceeds the space available between the allowed first and last columns, the string will be broken into as many lines as are required to keep the string within the specified boundaries on each line.

For non-buffered, non direct access files, the initial default value of END is 79.

For buffered or direct access files, the initial default value of END is the recordlength specified when the file was opened. If END is specified to be larger than the recordlength for a buffered/direct access file, it will be reset to the recordlength.

If a different specific default value for END is specified with the /DEFAULT option, the defaults described above can subsequently be reset by specifying END<0 with /DEFAULT.

**INDENT = M:** Specifies a relative offset for the first column to be used for all lines after the first (if any). If INDENT is set, all lines after the first will start in column START+INDENT (1+INDENT if START was not specified). If INDENT is not specified, the initial default is that all lines start in the column specified or implied for START. If START+INDENT < 1, all lines after the first will start in column one. START+INDENT must not be larger than the value being used for END.

**BREAKCHAR = character:** Used to specify a character at which text may be divided when "string" will not fit entirely on one line. By initial default, a space break\_character is used. Text is broken after the specified break character(s).

The user may specify up to five break\_characters, by specifying the /BREAKCHAR = character option multiple times. If any /BREAKCHAR is specified, all break characters, including a space (if desired) must be specified. Only one character may be specified after each /BREAKCHAR= directive, and it should not be enclosed in quotes.

If no suitable break character is found to break the line within the required borders, the line will be arbitrarily broken at the right border (in the column specified/implied for END).

To specify a space, you must use /BREAKCHAR = ^

To specify a forward slash, you must use /BREAKCHAR = ~

**NOBREAKCHAR:** Specifies that no special break characters will be used (not even a space). In this case, all lines will be broken arbitrarily at the right border column (the column specified/implied for END).

**FORCEBREAK = character:** Allows specification of a special "forced break" character. If FORCEBREAK is specified, the given character will cause a line break every time it is encountered in "string". The FORCEBREAK character itself is not echoed. If no FORCEBREAK character is encountered before the defined column limits are reached, the regular BREAKCHAR rules for line breaks will apply. Only one FORCEBREAK character may be defined. By initial default, no forcebreak character is used.

To specify a space, you must use /FORCEBREAK = ^

To specify a forward slash, you must use /FORCEBREAK = ~

**NOFORCEBREAK:** Specifies that no "forced break" character will be used. This option can be used to override a default previously set with an ECHO/DEFAULT command.

**LINE = M:** The line number at which output is to start. If not specified, output will start on the "current" line (1+the last line written to the file). The special line relative specifiers "\$+M" and "\$-M" may be used, if desired, to specify a line relative to the end of the file (first unwritten record). E.g. \$-1 is the last written line of the file.

**NOADVANCE:** If specified, the "\$" descriptor will be included in the output format statement used by ECHO. If the line is being written to STDOUT (default), this will have the effect of leaving the cursor at the end of the echoed string.

**DEFAULT:** If specified, then the values given for /START, /END, /INDENT, /BREAKCHAR, /NOBREAKCHAR, /FORCEBREAK and/or /NOFORCEBREAK will become the defaults on all future ECHO (DEFHELP) commands, unless/until reset by another ECHO/DEFAULT (or DEFHELP/DEFAULT) command. Only the options specified with a particular ECHO/DEFAULT will have their default values changed.

Most text messages output by INTERFACE (except error messages) are processed ECHO, and resetting any defaults will affect any such output.

When /DEFAULT is specified, "string", if any, is ignored. There is no output from an ECHO/DEFAULT command.

Note that any defaults reset by a ECHO command also become the defaults of all DEFHELP commands (and vice-versa).

*Examples:*

```
ECHO "This is a test of the echo-command." /START=1 \
    /END=11 /BREAKCHAR = ^ /BREAKCHAR=-
```

would give

```
This is a
test of
the echo-
command.
```

```
ECHO "Example of a forced#break in
action." /FORCEBREAK=#
```

would give

```
Example of a forced
break in action.
```

**EXCLUSIVE**

Syntax:

```

EXCLUSIVE { /OUTFILE = file_equivalence_name}

      command/option_name
      { /TEXT = "string" }
      { /STOP (D) } {/CONTINUE} {SKIP = COMMAND |
OPTION}
      {/COMMAND (D)} { /OPTION}
      { /TIMES = M}
      (... )
      (repeat as many exclusivity definitions as
desired)

      END EXCLUSIVE
    
```

Description: The EXCLUSIVE block construct allows one method for interdependencies between calls to various commands and options to be established. The EXCLUSIVE block contains a series of command or option names. If any of these has been called the specified number of times when the EXCLUSIVE block is encountered, the specified action is taken. Possible actions include stopping, skipping the remainder of the current command, or simply issuing a message to the user. If a specified command/subcommand has not been called the required number of times, no action is taken.

Exclusivity directives in an EXCLUSIVE block are read sequentially until either a STOP or SKIP qualifier is found attached to a command/option which has been called the required number of times, or until the END EXCLUSIVE statement is read.

Blank lines and comment lines with an EXCLUSIVE block are skipped.

The EXCLUSIVE block will typically appear within a COMMAND/OPTION definition block.

**OUTFILE = file\_equivalence\_name:** Specifies the file to which TEXT strings will be written if the /TEXT qualifier is attached to any command which has been called the required number of times. By default, the text is output to STDOUT (usually unit 6).

**command/option\_name:** The name of the command or option to be checked. If the command/option has been called the required number of times in the UC file, the specified action(s) will be taken. This field will be a command name unless /OPTION is specified, in which case this field must contain the name of a valid option for the COMMAND block in which this EXCLUSIVE block appears. Valid abbreviations are acceptable. Only commands defined in the IPS file (not intrinsic commands) can be checked.

**TEXT** = "string": If the command/option has been called the required number of times, the specified text string will be output. By default, no text is output.

**STOP**: If the command/option has been called the required number of times, the program will stop. This is the default.

**CONTINUE**: If the command/option has been called the required number of times, only output the text string provided, if any, and continue.

**SKIP** = command | option: If the command/option has been called the required number of times, skip all remaining commands in the current command or option definition, depending on the argument given for SKIP. Do not specify SKIP=option unless the EXCLUSIVE block appears within an option definition block. If a SKIP directive is executed, any subsequent exclusivity checks in the EXCLUSIVE block will be ignored.

STOP, CONTINUE and SKIP are mutually exclusive. Specify at most one.

**COMMAND**: The name provided corresponds to a command. This is the default.

**OPTION**: The name provided corresponds to an option defined within the COMMAND block containing this EXCLUSIVE block.

COMMAND and OPTION are mutually exclusive. Specify at most one.

**TIMES** = M: Specifies the number of times the specified command must have been called when the EXCLUSIVE block was encountered. If the command has been specified this number of times, the action(s) specified by the the associated option flags will be performed. By default, TIMES = 1.

Note that options are only called at most once during any command definition. Thus TIMES is ignored if /OPTION was specified.

*Examples:*

Assume we encounter the following EXCLUSIVE block within the definition for command MAINCOM

```

COMMAND = MAINCOM
  EXCLUSIVE
    command1 /TEXT = "command1 was executed"
    command2 /TIMES=3/STOP
    command3 /TEXT = "command3 was executed"
\
    /SKIP = command
    command4 /STOP
  END EXCLUSIVE
  (...)
END COMMAND

```

If we have already called command1, command2 and command4 once each, and have not called command3 at all, then the following would be echoed to the user:

```
command1 was executed  
command3 was executed
```

at which point control would transfer from out of the MAINCOM command definition block without executing any additional commands. Notice that we did not stop, despite the fact that command4 had been executed the required number of times, because the previous check transferred control out of the MAINCOM definition.

**EXTERNAL**

Syntax:

```
EXTERNAL "string"  
        { /WAIT (D)} { /NOWAIT}
```

Description: The EXTERNAL command sends the provided "string" to the host operating system for parsing and execution.

**"string"**: The command to be executed by operating system of the host computer. Syntactical rules for valid commands will depend on the specific operating system. *You must enclose the string in double quotes if the string contains option delimiter characters (usually "/") or runs of consecutive spaces which should not be compressed.*

**WAIT**: Wait for the command "string" to complete before continuing. This is the default.

**NOWAIT**: Continue reading/parsing IPS and UC file commands after the command "string" is sent to the resident operating system.

Examples:

(unix):      EXTERNAL "date > date.file" /WAIT

would put the current time and date into a file called date.file. This file could subsequently be accessed by INTERFACE.

(VMS):      EXTERNAL "submit/queue=queue\_1 job.com" /WAIT  
            EXTERNAL "@job.com" /NOWAIT

The first command would submit the command file job.com to the batch queue and then continue. The second would start the command file job.com running as a subprocess, and then continue.

**FORMAT SPECIFICATION**

Syntax:

(argument\_list) : {line\_no} (format\_description)

Description: The ARGUMENTS, COPY, READ, REFORM, and WRITE commands allow or require that format specifier(s) be provided. The general format of an INTERFACE format specifier is given above. Depending on the requirements of the command, information to the left or right of the colon (:) may be omitted, i.e.:

**ARGUMENTS:** Only the argument\_list is specified. The colon, and everything to the right of it, is omitted. A free-format read is always performed for ARGUMENTS.

**COPY/READ/REFORM:**

IN\_FORM is specified using the complete format specifier.

OUT\_FORM is specified using only the portion of the specifier to the right of the colon (omit argument\_list and colon).

**WRITE:** OUT\_FORM is specified using only the portion of the specifier to the right of the colon (omit argument\_list and colon).

**argument\_list:** Specifies the numbers, types and order of arguments to be read. Argument list specifiers (listed below) are used to describe the expected input. Argument list specifiers can appear in any order, and can be repeated as often as desired. The surrounding parentheses "()" are required.

Valid argument list specifiers:

nA{m} : specifies that n character fields are to be read.

For free-formatted reads

If A is followed by an integer constant m, each field will be stored as a character variable exactly m characters long. In this case, the value which is actually read will be truncated or filled on the right with spaces to effect the specified length. By default, character fields are stored as variables with the same length as the value that was read.

For explicit format or unformatted reads:

m *must* be specified for each character field. The length m gives the length of the character variable which will be allocated for the field, and must be consistent with the length specified in the specified FORTRAN format statement.

nI: specifies that n integer fields are to be read.

nF: specifies that n real fields are to be read.

nS: specifies that n fields are to "skipped". A "skipped" field is treated as a character field, and is read but not stored in memory.

nS should only be used in free format read specifiers.

Tm: Tab indicator. Specifies that subsequent fields are to read starting at position m of the line.

Tm should only be used in free format read specifiers. Do not use Tm in the argument\_list specifier for the ARGUMENTS command. For free format input that is read from more than one line, the tab level is reset to 1 (i.e. an implicit T1) each time an additional line is read.

For any specifier, n can be omitted, resulting in a default value of n=1. n and m must be integer constants. If you wish to use variable expressions in these fields, surround them with "<>" brackets.

Left-right parentheses pairs "(" can be used to group argument list specifiers, and can be preceded by an integer constant (or variable expression in "<>") which indicates how many times the construct in the "(" is to be repeated. E.g.

( 3 ( 2A , I ) , F ) is equivalent to ( 2A , I , 2A , I , 2A , I , F )

Left-right parentheses must not be nested, E.g.

( 2 ( F , 2 ( A , F ) ) ) would result in an error..

**line\_no:** An optional line number, which indicates the line at which the data transfer for which the format statement is being used is to start. If line\_no is omitted, transfer starts at the "current" line, which is the first line after the last line read/written in the relevant file.

line\_no can be a numerical constant or numerical expression. line\_no can also be specified using the optional constructs "\$+M", "\$-M", ".+M" and "-M", where M is the numerical part. In these expressions, "\$" represents the end of the file, and "." represents the "current" line. So, for example, "\$-1" is the last existing line in the file, and "-4" is the fourth line before the "current" line.

**format\_description:** There are three valid possibilities for this field. Each must be enclosed in a set of parentheses "()".

1) (ff):

Indicates that a free-format read is to be performed. Data will be read from the appropriate file (or as arguments to the appropriate command), using the

information specified in the argument list. "ff" must be the only non-blank characters between the parentheses.

The value in each numerical field may be evaluated, if it is a function or expression rather than a constant. This is the default for data read in an ARGUMENTS list, and an option for READ/COPY/REFORM commands.

2) (uf):

Indicates that an unformatted read is to be performed. This `format_description` *must* be used for a file if, and only if, it was specified as "/UNFORMATTED" when it was opened. "uf" must be the only non-blank characters between the parentheses.

3) (standard FORTRAN format statement):

If (ff) or (uf) is not specified, the `format_description` is assumed to be a valid FORTRAN format specifier. In this case, the user must ensure that the number and types of values in the data transfer list match those in the format statement. If they do not, a fatal error may be generated by the host computer.

One special formatted specifier is recognized: "(\*)". This specifier can only be used for *write* procedures, and cannot be used for direct access or "buffered" files. If this format is specified, the "\*" must be the only non-blank character between the parentheses. This specifier will result in the data being output using the free-format list-directed formatting procedures of the host computer.

See a standard FORTRAN manual for syntactical rules of FORTRAN format statements.

*Examples:*

1)

```
( 3I , 2A8 , F ) : 12 ( 3I5 , 2A8 , F10 . 3 )
```

would read three integer variables, 2 character variables of length eight, and one real variable from line 12 in the specified FORTRAN format.

2)

```
( 3I , 2A , F ) : 12 ( ff )
```

would read three integer variables, two character variables, and one real variable from line 12. Arguments would be parsed using free format rules. The character variables would be made as large as required to hold the actual character arguments which were read.

3)

```
( 3I , 2A8 , F ) : 12 ( uf )
```

would read three integer variables, two character variables of length eight, and one real variable from line 12 of a file which was opened as /UNFORMATTED.

4)

```
ASSIGN i = 2  
      (...)  
(<i>(A,F)):(ff)
```

would read the following list of variables: character, real, character, real. A free format read would occur, with character variables made as large as required to hold the actual character arguments which were read.

## **MARKMEM**

Syntax:

MARKMEM

Description: The MARKMEM command is used to "mark" the current state of character and real variable usage. When used in conjunction with the CLEAR/MARKMEM command, the skillful user can reclaim character or real variable storage used since the mark. See CLEARMEM/MARKMEM for more details.

MARKMEM is typically required in two cases:

- 1) Where memory storage is an issue. I.e. scripts where a large number of storage and/or commands are performed. This includes scripts where a number of I/O operations are performed repeatedly by a DO loop.
- 2) When it is desired to effect a "fresh start" of the program, back to some point defined by the MARKMEM. For example, it might be desirable to undefine the values of all variables issued since the MARK was set, when a set of commands is being executed numerous times.

One can set more than one memory MARK. In this case, CLEARMEM/MARKMEM will clear memory back to the most recent MARK encountered. The appropriate MARK will then be cleared.

**MAXCALL**

Syntax:

```
MAXCALL ncalls
      { /SKIP }
      { /ERRTEXT = "string" }
```

Description: MAXCALL can be used to limit the number of times a command (or part of a command) can be executed. When the MAXCALL command is encountered, a check is performed to see if the command currently being executed has been called the specified number of times. If it has, the program will either stop, or, optionally, the remaining directives in the command definition will be skipped, and program execution will continue.

MAXCALL is only valid within a COMMAND block definition. It can appear anywhere within such a block, except within the following constructs: DEFER and NOT\_CALLED block. MAXCALL can appear within a nested OPTION block, but will still refer to the parent command.

**ncalls:** The maximum number of times the command defined by the COMMAND block containing this MAXCALL can be called. If ncalls is exceeded, the appropriate action is taken (stop or skip). ncalls can be any numerical constant or expression.

**SKIP:** By default, if ncalls is exceeded, the program will stop. If /SKIP is specified, then if ncalls is exceeded, any remaining directives in the current COMMAND block will be skipped, and program execution will continue. A MAXCALL command with /SKIP specified is frequently the first directive in a COMMAND block. This ensures that if ncalls is exceeded, no other directives in the command definition will be executed.

**ERRTEXT = "string":** If specified, gives a text string which will be echoed to STDOUT (usually unit 6) if ncalls is exceeded.

*Examples:*

```
COMMAND = COMNAME
      MAXCALL 1 / SKIP \
      /ERRTEXT= "you cannot call COMNAME more than
once"
      (...)
END COMMAND
```

This construct would ensure that the COMNAME command definition would only be executed once. Any subsequent calls to COMNAME would result in the specified warning message being echoed to the user, with the command effectively ignored otherwise.

```
COMMAND = COMNAME  
  ASSIGN I = I + 3  
  MAXCALL 1 /SKIP  
  ASSIGN J = J + 2  
END COMMAND
```

This construct would update the value of I every time COMNAME was called. But the value of J would only be updated on the first call to COMNAME.

**NOT CALLED**

Syntax:

```
NOT_CALLED
    ( . . . )
END NOT_CALLED
```

Description: The NOT\_CALLED block is used to signify a block of directives which will be executed only if the command or option within whose definition the NOT\_CALLED block appears is not specified in the UC file.

After the UC file has been completely executed, any commands which were defined but not executed will be searched for NOT\_CALLED blocks. The directives in these blocks will be executed, in the order the commands were defined in the IPS file(s). If more than one NOT\_CALLED block occurs in a single COMMAND block definition, these will be executed in the order they were defined. NOT\_CALLED blocks are executed before DEFER blocks at the end of the program.

A NOT\_CALLED block within an OPTION definition for a COMMAND will only be executed when the parent command is issued in the UC file, but the parent option is not. When a user-command is issued in the UC file, the corresponding COMMAND block in the IPS file is executed in sequential order. As each OPTION definition is found, the user-command is checked to determine if the relevant option was specified. If it was, the option block is executed. If it was not, the option block is skipped, with the exception of any commands appearing within nested NOT\_CALLED blocks. The directives in these blocks will be executed.

If the parent command is not issued, directives in a NOT\_CALLED block *within an OPTION definition* will never be executed.

A NOT\_CALLED block should only appear within a COMMAND or OPTION definition block. As many NOT\_CALLED blocks can be defined as are desired, and they can appear anywhere within the COMMAND/OPTION definition, with the following exceptions: They cannot be used in a DO or IF construct; They cannot be used in a DEFER block; and they cannot be nested within other CALLED or NOT\_CALLED blocks.

A NOT\_CALLED block cannot contain the following commands: OPTION; MAXCALL; ARGUMENTS. Any IF or DO structures initiated in the NOT\_CALLED block must also be terminated in that block. Any GOTO within a NOT\_CALLED block must refer to a label within the NOT\_CALLED block.

The complement of the NOT\_CALLED block is the CALLED block. By default, any commands not within a NOT\_CALLED block are implicitly in a CALLED block.

*Example:*

Suppose we define command "commname" and option "optname" as follows:

```
COMMAND = commname
  NOT_CALLED
    ECHO "Routine commname not called"
  END NOT_CALLED
  OPTION = optname
    ECHO "commname called, optname too"
    NOT_CALLED
      ECHO "commname called, but optname not specified"
    END NOT_CALLED
  END OPTION
END COMMAND
```

Then if the command "commname" was issued with no options, the following would be echoed:

```
commname called, but optname not specified.
```

If the command "commname/optname" was issued, the following would be echoed:

```
commname called, optname too
```

While if the command commname was not issued at all in the UC file, at the end of the run the following would be echoed at the end of the run:

```
Routine commname not called.
```

**OPEN**

Syntax:

```

OPEN filename file_equivalence_name
  { /FORMATTED (D) } { /UNFORMATTED }
  { /OLD } { /NEW } { /UNKNOWN (D) }
  { /DIRECT }
  { /BUFFERED }
  { /OVERWRITE = replace | merge (D) | stop }
  { /SCRATCH }
  { /FIXED = M }
  { /LINE = M }
  { /ATTACH = target_file_equivalence_name }

```

Description: The OPEN command is used to open files, or to define additional file\_equivalence\_names which will refer to an already-opened file. No read or write can be directed to a file until that file has been opened using an OPEN statement (exceptions are the REDIRECT and MANUAL commands).

In the OPEN command, the name of the file is specified, as well as a mnemonic file\_equivalence\_name. The latter is the name used in all subsequent commands to refer to this file.

Several types of files are supported. These include:

1) Sequential access, either formatted or unformatted.

In this mode, each line of output is written to or read from the line following the most recently written line (unless a specific line number is given in the FORMAT SPECIFICATION for a particular read or write). Writing to a line other than the last line in such a file will typically destroy the file contents below that line. Access to a specified line number (other than the "current" line) can be slow.

This is the mode most commonly used by applications written in standard programming languages.

2) Direct access, either formatted or unformatted.

In this mode, a line number is typically attached to each input/output statement, and the input/output is read from/written to that line. If no line number is given, the line immediately following the last line accessed is used. Any line can be rewritten without affecting the lines around it. Access to a specified line number is relatively fast.

3) "Buffered" access, formatted files.

This is a special access mode unique to INTERFACE. Like direct access, a line number is typically attached to each input/output statement, and the input/output is read from/ written to that line. Line access is relatively fast, and a specific line can be rewritten without affect surrounding lines.

In addition, a "buffered" file also supports these valuable features:

A) Any formatted file can be opened as "buffered"; a pre-existing file does not have to have been created as direct access. When a buffered file is closed, it will by default revert to the file type (sequential access or direct access) it had before the run started. A resulting sequential access file can in most cases be used directly as input to a chosen application. (In contrast, this is not typically true of a direct access file).

B) A "buffered" access file allows optional OVERWRITE protection and a merged text protocol. Before any data is written to such a file, a check is made to determine if the new data would change any non-blank characters already in the file. If specified by the user, such an overwrite can be prohibited. The user can also request that non-blank characters of the new line be merged with the remaining contents of the old line. The effect is to allow, for example, blank fields in one part of a specified line to be filled without changing any other part of the line.

Using the /REPLACE option on the WRITE command the user can specify that only a portion of the line be subject to overwrite protection or merged operation.

Note that accessing specific lines in sequential access files can be slow. If you plan to be doing a lot of reading or writing to specific records, it is highly recommended that you open the file as "buffered." The only limitations/drawbacks to opening a file as "buffered" are

A) This type of access is only allowed for formatted files;

B) Free-formatted "list" type *output* (format = "(\*)") cannot be used for "buffered files"; and

C) You must specify the maximum line length for the file when it is opened, and you cannot write a line longer than this.

If you open a file as "buffered", be sure you review the /OVERWRITE flag options to determine which mode you desire.

**filename:** Name of the file to be opened. "filename" must be a valid file name for the host computer. It may be as long as required, and may contain a path, but must not contain any embedded blanks. It is not necessary (but is acceptable) to surround the filename with quotes, even if it contains forward slashes (/).

**file\_equivalence\_name:** The mnemonic name to be attached to the file. This is the name that will be used in subsequent read and write commands to specify a file for input/output. The file\_equivalence\_name can contain any alphanumeric string, but should not contain any special parsing characters (e.g. "/").

**FORMATTED:** The file to be opened is of formatted type. This is the default. Do not attempt to open a pre-existing unformatted file as formatted.

**UNFORMATTED:** The file to be opened is of unformatted type. Do not attempt to open a pre-existing formatted file as unformatted.

FORMATTED and UNFORMATTED are mutually exclusive. Specify at most one.

**OLD:** The file to be opened already exists.

**NEW:** The file to be opened does not already exist. Depending on the host operating system, specifying "/NEW" with a "filename" corresponding to a pre-existing file will either create a new version of "filename" (e.g. on VAX/VMS) or result in an error (e.g. on UNIX).

**UNKNOWN:** If the specified "filename" corresponds to a pre-existing file, open it as "OLD". If it does not correspond to a pre-existing file, open it as "NEW". This is the default.

OLD, NEW, and UNKNOWN are mutually exclusive. Specify at most one.

**DIRECT:** The file is to be opened for direct access. If the file is pre-existing, it must previously have been created as a direct access file to successfully use this option.

If /DIRECT is specified, /FIXED=M must also be given.

If /DIRECT is not specified, the file is assumed to be of sequential access type.

**BUFFERED:** The file is to be opened for "buffered" input and output. Only formatted files may be opened as "buffered". When INTERFACE terminates, a file specified as buffered will retain the access characteristic (sequential or DIRECT) specified when it was opened.

If /BUFFERED is specified, /FIXED=M must also be given.

NOTE: The "current" line pointer for a file opened as "buffered" will be the end of the file. Therefore, if you wish to read/write to the first line of the file, you should include an explicit line number in the first IO operation to the file.

**OVERWRITE** = replace | merge | stop: Specifies what type of data overwrite procedure and protection will be used when writing to a "buffered" file. These only apply when writing a "new" string to a line that has previously been written, and only for files opened as "buffered". If the write is to a new line, the text string is written as-is.

replace: For every write, the new string completely replaces the old string.

**merge** : Any non-blank characters of the "new" line(s) to be written will replace the corresponding characters in the previous contents of the line(s). Any position where the new line is blank will not be changed.

**stop**: If any non-blank character of the new line is different than the corresponding non-blank character in the old line, INTERFACE will stop. Otherwise, non-blank characters in the new line which correspond to blank characters in the old line will be merged into the line.

**SCRATCH** Specifies that the file is to be opened with status="scratch". This means the file will be deleted when program execution is terminated. BUFFERED and OVERWRITE flags are not allowed for a file opened as SCRATCH. The "filename" is ignored for a file opened SCRATCH, but a dummy placeholder "filename" argument must be provided.

**FIXED = M**: Specifies the recordlength for a direct access file. If one is opening a pre-existing direct access file, the value specified for FIXED must correspond to the length of records in that file established when the file was created. If one is opening a sequential access or pre-existing "buffered" file, FIXED specifies the longest record that will be allowed in that file. If the longest record in a pre-existing sequential access "buffered" file is greater than the value specified for FIXED, the larger value will be used.

FIXED =M must be specified if /DIRECT or /BUFFERED has been specified.

**LINE = M**: For sequential access files, LINE=M gives the line to which the "current" line will point when the file is opened. By default, M = 1 (which corresponds to a "rewind"). LINE can optionally be specified as "\$+M" or "\$-M", which are relative displacements from the end of the file. E.g. \$-1 would be the last written record in the file. M can be any arithmetic expression.

LINE has no effect for files opened as BUFFERED or DIRECT.

**ATTACH = target\_file\_equivalence\_name**: ATTACH allows specification of additional file\_equivalence\_names which will refer to a file which has already been opened. The target\_file\_equivalence\_name given with the ATTACH option is the equivalence name of the file which has already been opened. The file\_equivalence\_name provided as an argument to OPEN is a second mnemonic name which is to henceforth refer to the opened file currently referred to by target\_file\_equivalence\_name. After an OPEN/ ATTACH command, both file\_equivalence\_name and target\_file\_equivalence\_name will refer to the same file (the file previously opened with equivalence name given as an argument to ATTACH).

Note that file\_equivalence\_name alias can correspond to a unique name, or to an equivalence which has been previously defined. In the latter case, the prior association will be lost when the ATTACH is performed, and one will have to use another OPEN statement to reaccess the file formerly associated with the equivalence\_name.

When /ATTACH is specified, the "filename" and all other qualifiers (other than /ATTACH) are parsed but ignored.

For example, if we wanted to reassign the meaning of STDOUT, we could use:

```
OPEN capture.dat CAPTUREFILE / NEW/FORMATTED
OPEN dummyname STDOUT /ATTACH = CAPTUREFILE
```

The first command would open a new file named capture.dat, with equivalence name CAPTUREFILE. The second would associate the equivalence name STDOUT with the file opened with equivalence\_name CAPTUREFILE (i.e. capture.dat). The dummy placeholder name in the second command is required. Henceforth, all default writes made by the program to file\_equivalence\_name STDOUT would go to capture.dat.

*Examples:*

1) Buffered output:

Suppose we opened file oldfile.dat as

```
OPEN oldfile.dat data /FORMATTED/OLD/BUFFERED \
/OVERWRITE=merge/FIXED=80
```

If line 3 of oldfile.dat contained

```
This is the season of our discontent
```

and we attempted to write a string of /'s (//////////...) to line 3, we would get

```
This/is/the/season/of/our/discontent//////////.
```

A more useful example might be the following. Assume line 5 contains

```
(1) The percentage increase in sales, , compares .
```

and we attempt to write

```
(2) 300%
```

we will get

```
(3) The percentage increase in sales, 300%, compares
```

Note that if we attempted to write line (2) on top of line (1) in a "normal" file, we would simply get a line identical to line (2). This would also be the result if we had specified `OVERWRITE=replace`.

## 2) Setting additional file\_equivalence\_names:

All errors reported by INTERFACE are written to file\_equivalence\_name `ERROUT`, which is, by default, unit 6. To reassign error output to a desired file, one could use:

```
OPEN error.dat ERRFILE /NEW/FORMATTED
OPEN dummy ERROUT/ATTACH=ERRFILE
```

If it was subsequently desired that error information once more be echoed to unit 6, another reassignment could be performed:

```
OPEN dummy ERROUT/ATTACH = &SIX
```

Note that we use the special file\_equivalence\_name `&SIX`, which is always set up to point to unit six (standard output on most computers). `&FIVE` serves the same purpose for standard input. An analogous set of commands could be used to reassign `STDOUT` (default output unit for many writes) and `STDIN`.

**OPTION**

Syntax:

```
OPTION = option_name
      (... )
      END OPTION
```

Description: The OPTION block defines options which will be recognized when specified with the "parent" command. An OPTION block is only valid nested within a COMMAND block, and the corresponding COMMAND block defines the "parent" command. The equals sign is required.

When the parent command is given, all directives within the COMMAND block are executed in the order defined. When an OPTION block is encountered, it will be executed if the appropriate option was issued with the command. Note that if multiple options are specified, the corresponding OPTION blocks will be executed in the order they are defined in the IPS file, *not* in the order that they were specified with the command issued in the UC file. Any OPTION block that defines an option which was not specified with the parent command will be skipped.

(There is one exception: Commands inside a NOT\_CALLED block, which itself lies within the definition of an option which was not specified, *will* be executed. See NOT\_CALLED for more details).

Commands within an OPTION block are executed in the order they are written.

**option\_name:** The name of the option name to be attached to this OPTION block. Valid option names can consist of any characters which do not have special meanings for parsing, although limiting names to alphanumeric and the underscore "\_" character is encouraged for safety. The special star character "\*" can be used to define acceptable abbreviations. Characters before the "\*" are required. Characters after the "\*" are optional, but must match exactly if provided. If no "\*" is specified, the option name cannot be abbreviated. E.g.

```
OPT*ION    would match
OPT
OPTIO
OPTION
```

but not

```
OPTIOG .
```

The required part of option\_name must not be redundant with the names of other options already defined for the same parent command. If it is, an error will be reported. You can use the same option name for different parent commands. (See “Rules for Determining Command Name Matches” in Chapter 3: Features and Considerations for additional discussion).

*Example:*

If we define a command "comname" in the IPS file as:

```
COMMAND = comname
  OPTION = opt1
    ECHO "option1 specified"
  END OPTION
  OPTION = opt2
    ECHO "option2 specified"
  END OPTION
END COMMAND
```

then if we encountered the following command in the UC file:

```
comname/opt2/opt1
```

we would get the output:

```
option1 specified
option2 specified
```

Note that the order in which the options are executed depends on the order in which they are defined in the IPS file, not on the order in which they are specified with the command.

**POINT**

Syntax:

```
POINT file_equivalence_name
      { /LINE = M }
```

Description: POINT allows the "current line pointer" to be repositioned for a sequential access file. Note that this has the same result as specifying a line number in a format statement, but can be useful in certain constructs.

POINT should only be used for sequential access files. It will simply result in a warning if applied to direct access or "buffered" files (see OPEN).

**file\_equivalence\_name:** Equivalence name assigned to the file when it was opened. Point will simply return a warning if file\_equivalence\_name does not correspond to a currently opened file. file\_equivalence\_name must correspond to a sequential access file.

**LINE = M** The "current line pointer" will be positioned at the beginning of line M. LINE can be a numerical constant or numerical expression. LINE can also be specified using the optional constructs "\$+M", "\$-M", ".+M" and ".-M", where M is the numerical part. In these expressions, "\$" represents the end of the file, and "." represents the "current" line. So, for example, "\$-1" is the last existing line in the file, and ".-4" is the fourth line before the "current" line.

If LINE is specified as a value  $\leq 1$ , the current line pointer will be placed at the first line of the file. If LINE is specified as a value  $>$  the last line of the file, the current line pointer will be placed after the last line of the file.

If LINE is not specified, the current line will be reset to the first line in the file (an effective "REWIND").

*Examples:*

```
POINT file1 / LINE = $-1
```

would place the current line pointer for the file opened with equivalence name "file1" at the beginning of the last line in the file.

```
POINT file1
```

would "rewind" the file opened with equivalence name "file1", placing the current line pointer at the beginning of the first line.

**READ**

Syntax:

```

READ {lines}
    /INFILE=file_alias_name
    { /IN_FORM = format}
    { /UNTIL}
    { /ISTART = M}
    { /STORE} { /NOSTORE (D)}
    { /QUOTE = quote_character}
    { /STRICT}
    { /DELIMIT = delimit_character}
    { /FUNCTIONS}
    { /SUBSTITUTE (= template | uc (D))}
    { /MISSING = stop (D) | continue | next}
    { /MISS_TEXT = "string"}
    { /END = stop (D) | continue}
    { /END_TEXT = "string"}
    { /NAMES = (associated_namelist)}
    { /LINES = variable_name}
    { /FIELDS = variable_name}

```

Description: READ is used to read a specified number of lines of data from a specified file. The user can either specify the format of the data to be read, or accept a default. The user can also specify that the data which is being read not be stored in memory, if reads are being performed simply to effect a desired file pointer placement.

Note that the COPY and REFORM commands are very similar to READ, with the addition of a few commands, and differing defaults for /STORE | NOSTORE. In the following, any reference to the READ command also applies to COPY and REFORM.

**lines:** The number of read operations to be performed. If each read corresponds to reading one physical line, then “lines” will be the number of lines to read. "lines" may be omitted, in which case it defaults to 1.

**INFILE = file\_alias\_name:** The alias name of the file from which data will be read. file\_alias\_name must correspond to a file which has been previously opened using the OPEN command.

The name "COMINQQ" is a special file\_alias\_name. When this name is given, the current UC file (adjusted for any redirection) is used for the READ. If such a read takes place, the line following the line(s) read by the READ command will contain the next UC file command.

Note that using COMINQQ for a READ statement appearing in a NOT\_CALLED or DEFER block which is not executed until the end of the program will result in an error.

**IN\_FORM** = format: Specifies the format to be used in parsing the line being read. The format specifier can describe free-format, standard FORTRAN format, or unformatted input. See the section on FORMAT SPECIFICATION for more information on format descriptors.

If /IN\_FORM is not specified, lines will be read and stored as character strings 80 characters long (A80).

If a line-number is specified in the format, and multiple lines are being read (lines > 1), the line number in the format refers to the first line read. Subsequent lines will be read sequentially from this point.

**UNTIL**: If the /UNTIL option is specified, the READ command is followed by a CONDITIONAL block:

```

READ lines /UNTIL/...
    (logical expression)
    (logical expression)
    (...)
    (supply as many logical expressions as desired)

END CONDITIONAL

```

Each line of the CONDITIONAL block consists of a single logical expression surrounded by left-right parentheses "()". (A logical expression is any expression that evaluates to a value of true or false; see the section on Logical Expressions in the Features and Considerations Chapter). As many logical expressions may be defined as are desired, one per line. The CONDITIONAL block is terminated by an END CONDITIONAL command.

After each read operation is carried out, the logical expressions are evaluated in sequence. If any expression evaluates to a value of true, the read is terminated, and control is transferred to the first command past the END CONDITIONAL line. If no expression evaluates to a value of true, the command continues, as usual. If the read is not terminated early because of a logical expression in the CONDITIONAL block, "lines" reads will be performed, as usual.

If the read is terminated because a true (logical expression) was found, the "current line pointer" for the file being read will be set to the beginning of the last *physical* line which was read. If the last READ operation corresponded to reading one physical line of data, then the "current line pointer" will be at the beginning of this line. However, if the last READ operation corresponded to reading multiple physical lines of data, then the "current line pointer" will be at the beginning of the *last* line of this group. (There is one exception: If a READ/UNTIL operation is being performed on the "standard input" unit (STDIN) rather than a file, the current line pointer at the end of the read will be the line *following* the last line read).

Blank lines and comment lines within the **CONDITIONAL** block are skipped.

Any variables assigned during the most recent read command can be used in the (logical expression)'s. If the **/NAMES=** qualifier was used, these can be referenced by specified names; otherwise they are referenced by their **IVS**, e.g. **##\$##N** would refer to the Nth variable stored in the last **READ**. Note that if **/NOSTORE** is implied or specified, logical expressions in the **CONDITIONAL** block can still reference variables assigned by the most recent read operation (but they cannot reference variables read by earlier read operations due to this same **READ** statement).

**ISTART = M**: Specifies the first line to be read from a sequential access file. If multiple physical lines are being read (either due to "lines" > 1, or because each **READ** operation corresponds to more than one physical line), **ISTART** only refers to the first physical line read. Subsequent lines are read sequentially.

The value specified by **ISTART** is only used when reading from sequential access files. It is ignored for direct access or "buffered" files. For these types of files, the starting line number for reads can be specified in the **FORMAT SPECIFIER**. If both **ISTART** and a line number in the **FORMAT SPECIFIER** are given for a sequential access file, the value in the **FORMAT SPECIFIER** will be used.

**ISTART** can be a numerical constant or numerical expression. **ISTART** can also be specified using the optional constructs "\$+M", "\$-M", ".+M" and "-M", where M is the numerical part. In these expressions, "\$" represents the end of the file, and "." represents the "current" line. So, for example, "\$-1" is the last existing line in the file, and ".-4" is the fourth line before the "current" line.

**STORE**: If specified, then any data transferred by the **READ** command will also be stored in memory, and can be accessed by subsequent commands. This is the default for **READ**.

**NOSTORE**: If specified, then data transferred by the **READ** command will not be stored in memory. Since memory is fixed and may be limited, it is recommended that data not be stored if it will not be required after the **READ** command is carried out, e.g. when reads are being carried out only to effect a desired file pointer placement. When **/NOSTORE** is specified, memory is reclaimed after each read operation, but not until the **CONDITIONAL** block (if any) is evaluated. This allows **/NOSTORE** to be specified in conjunction with a **CONDITIONAL** block which refers to variables from the most recent read operation.

**QUOTE = quote\_character**: For free-format reads, **/QUOTE** can be used to specify a single character which delimits quoted text strings in the data being read by this command. Text within a quoted text string is not searched for field delimiter characters. E.g. if the comma "," is a delimiter, you must define and use a quote string to keep the character field "an example, this is" from being interpreted as two separate fields. Any quoted text string will be considered a character type argument.

To specify the double quote character " as quote\_character, use /QUOTE = "" (a pair of double quotes). To specify any other character, use only a single unquoted character. Do not specify a quote\_character which is the same as a delimiter character.

By default, no quote character is defined or used. QUOTE is only used for free-format reads.

**STRICT:** Forces strict value type specification. By default, if an argument is specified in the argument\_list part of a free-format specifier to be real, and an integer is read in the associated field of the line being read, the floating representation of the value (FLOAT(value)) is used. Likewise, if the value is specified to be an integer, and a real is read, the integer portion (INT(value)) is used.

By specifying /STRICT, the program will stop if the specification type does not match the type of the value read in the corresponding field.

STRICT is only used for free-format reads.

**DELIMIT** = delimit\_character: When performing a free-format read, fields are separated by delimiter characters. If DELIMIT is not given, the delimiter characters used are those contained in registers SP1, SP2, SP3. These fields can be set by the DEFINE command, and are by default "," and " " (comma and space). /DELIMIT can be used to specify different delimiter characters for a particular read command. If any /DELIMIT = delimit\_character option is specified here, then SP1, SP2, and SP3 will not be used.

Up to five DELIMIT characters may be specified. Each requires separate specification of a /DELIMIT = delimit\_character option. delimit\_character must be a single unquoted character.

To specify a space delimiter, use: /DELIMIT = ^

To specify a forward slash (/) delimiter, use: /DELIMIT = ~.

When a space " " is defined to be a delimiter, any number of contiguous spaces will constitute a single delimiter. To protect delimiter characters in text strings in the input data, define and use a quote character (see /QUOTE). To protect delimiter characters in function references (e.g. commas), use the /FUNCTIONS option.

The DELIMIT option is used only for free-format reads.

**FUNCTIONS:** By default, parentheses are not considered in parsing data which is read during a free-format read. If /FUNCTIONS is specified, then delimiter characters appearing between balanced left-right parentheses "()" will not function as field delimiters. So, for example, MIN(3.,4.) would be interpreted as a single field, even if the comma was a field delimiter, if /FUNCTIONS was specified. /FUNCTIONS has no effect for character fields.

The FUNCTIONS option is only used for free-format reads.

**SUBSTITUTE** {= template | uc (D)}: By default, in-line variable substitution of <...> constructs is not performed for data read by a READ/COPY/REFORM command. If /SUBSTITUTE is specified *and* a free format read is being carried out, any <...> constructs will be resolved before the line is parsed.

The /SUBSTITUTE qualifier can optionally take an argument. By default, if any <> variable substitution is done, any variable name references will be resolved by searching the list of variables defined at the user-command (or global) level. This corresponds to the default "uc" option. If /SUBSTITUTE = template is specified, then variable name references will be resolved by searching the list of variables defined at the template (or global) level. (See the ASSIGN,CASSIGN, and GASSIGN commands for more discussion of local/global/complementary variable definitions).

/SUBSTITUTE has no effect on formatted reads.

**MISSING** = stop (D) | continue | next: For free format input, MISSING determines what the program will do if fewer values are found on a single line of the input file than are indicated in the argument\_list portion of the format. By default, the program will stop with an error.

If /MISSING=CONTINUE is specified, then missing arguments will be set to 0 (integers), 0.0 (reals) or " " (characters) and the program will continue.

If /MISSING=NEXT is specified, then an additional line (or lines, if necessary) will be read to obtain the remaining required arguments. No arguments are read from a blank line.

Note that if additional lines are read because /MISSING=NEXT was specified, these do not count towards the "lines" specifier given with the READ command. "lines" refers to the number of READ operations, not the number of actual lines read.

The MISSING option is only used for free-format reads. For formatted and unformatted reads, read behavior is dictated by the standard rules of FORTRAN.

**MISS\_TEXT** = "string": By default, if /MISSING=STOP is specified or implied, the default program reporting functions are used to report an error when fewer values are read on the given line than were specified in the argument\_list. If MISS\_TEXT = "string" is specified, the supplied character "string", and only this string, will be reported to file\_equivalence\_name ERROUT.

The MISS\_TEXT option is only used for free-format reads.

**END** = stop (D) | continue: Specifies the program behavior if fewer than "lines" READ commands can be executed before reaching the end-of-file. By default, the program will stop. If /END=continue is specified, the READ command will terminate upon reaching the end-of-file, and the program will continue.

**END\_TEXT** = "string": By default, if /END=STOP is specified or implied, the default program reporting functions are used to report an error when the end-of-file is encountered before "lines" READ commands could be performed. If END\_TEXT="string" is specified, the supplied character "string", and only this string, will be reported to file\_equivalence\_name ERRROUT.

**NAMES** = associated\_namelist: Allows association of user-defined names with the values read. By default, the values which are read because of an READ command can be later referenced only by their IVS specifier, i.e.

command#call#option#line#argument.

By using the /NAMES qualifier, you can assign names of your choosing to the data read because of the READ command.

The syntax of the associated\_namelist is

name1 , name2 , name3 , ...

where name1, name2, etc. are the names to be assigned, sequentially, to the values read. The associated\_namelist can optionally be enclosed in a set of parentheses "()". If fewer names are specified than values are to be read, the remaining variables will only be accessible by their IVS specifiers. If more names are specified than variables are to be read, the remaining names are ignored. If no name appears between a pair of commas, no name is assigned to the corresponding variable. No name should be specified for a field which is skipped ("S" format item in argument\_list).

Names in the associated\_namelist must follow the standard rules for variables: They must start with a alphabetic character, and must contain *only* alphanumeric and "\_" characters. Name assignments are by default made at the IPS file level (i.e. equivalent to ASSIGN). If you wish the assigned name and value to be accessible both in the UC file and the IPS script (the equivalent of GASSIGN), precede the name in the associated\_namelist by a "&" character. If you wish the assigned name and value to be accessible only in the UC file (the equivalent of CASSIGN), precede the name in the associated\_namelist by a "%" character. The "&" or "%", if any, is stripped before the assignment is made. (For more on the difference between local, global, and complementary level assignments, see commands ASSIGN, GASSIGN, and CASSIGN).

Names assignments made using the /NAMES qualifier are by default "volatile" to memory clears. To make a "non-volatile" name assignment, precede the variable name by an exclamation point "!". (For more on the difference between volatile and non-volatile variable assignments, see NVASSIGN and CLEARMEM). The exclamation point can be combined with the & or % characters to force various types of assignments:

specifier

assignment made

&name	gassign name	= value
%name	cassign name	= value
!&name	nvgassign name	= value
!%name	nvcassign name	= value

Note that using the /NAMES option with a READ command where "lines" is specified to be > 1 will result in multiple reassignments of the associated\_namelist, so that when the command finishes, the names in the associated\_namelist will contain the values for the last READ operation only.

Note also that any specific names assignment can also be carried out following the READ statement with an ASSIGN statement, e.g.

```
READ 3 /INFILE = filein/IN_FORM=(2A,F):(ff)
ASSIGN name = ##$##$-2#3
```

would associate "name" with the third value on the first line read by the READ command.

If /NOSTORE is specified (or implied by default for COPY and REFORM), the values associated with the associated\_namelist will become undefined or incorrect when the READ command terminates. They can still be used safely in the CONDITIONAL block, if any (see above).

**LINES**= variable\_name: If specified, upon return variable\_name will be set to the number of READ commands successfully completed. If no lines were read (end of file condition was encountered immediately), and /END=continue has been specified, variable\_name will be set to 0.

If an end-of-file condition was encountered after the first line, and /end=CONTINUE was also specified, variable\_name will be returned with the negative of the number of lines read.

Note that "LINES" corresponds to the number of READ operations successfully completed. If multiple lines are read by each operation (either because of the format statement, or because MISSING=next was specified with free-format input), these will *not* be reflected in the reported number of lines.

**FIELDS** = variable\_name: If specified, upon return variable\_name will be set to the number of fields actually read (on the last READ operation, if "lines" > 1). This can be different from the number given in the argument\_list if /MISSING=CONTINUE has been specified. For example, one could specify

```
READ /INFILE = filein /IN_FORM=(30A):(ff) \
/ MISSING=CONTINUE /FIELDS=HOWMANY
```

This would read up to 30 character fields from the "current" line in the file with equivalence\_name filein. The actual number found would be stored in the variable HOWMANY.

If you wish the variable name and associated value to be subsequently accessible by commands in the UC file, as well as the IPS file, precede "variable\_name" by an ampersand ("&"). If you wish the variable name/value assignment to be made only at the UC level, precede "variable\_name" by a percent sign ("%"). If you wish the variable name/value assignment to be made "non volatile", precede "variable\_name" by an exclamation point ("!").

*Example:*

Suppose we have a file, already opened with file\_equivalence name "file1", which contains a list of names, room numbers, and other information about a group of people. We are interested in information about a particular person, JONES, who is in either room S947 or room S940 (we can't remember which). The following would find the appropriate JONES entry, and then read the desired information in the required format. Note that we use the /NOSTORE option so that we don't unnecessarily waste memory for data we will not be using later.

```

READ 99999 /INFILE = file1 /IN_FORM = (S,2A):2(ff) \
        /UNTIL /NOSTORE /NAMES = (NAME, ROOM)

        (NAME.EQ."JONES" .AND. ROOM.EQ."S947")
        (NAME.EQ."JONES" .AND. ROOM.EQ."S940")

END CONDITIONAL

READ /INFILE = file1/INFORM = \
        (I,A8,A4,A8,F,A8):(I5,T12,A8,T25,A4,T30,A8,2X,F12.2,1X,A8) \
        /NAMES = (GROUPID, NAME, ROOM, PHONE, SALARY, BIRTHDATE)

ECHO GROUPID = <GROUPID>, NAME      = <NAME>, ROOM = <ROOM>, \
        PHONE   = <PHONE>, SALARY = <SALARY>, BIRTHDATE = <BIRTHDATE>

```

The first command does a free format read of the second and third fields, parsing them as character data, and assigning them to names NAME and ROOM (note that we don't provide a name for the first field, since it was "skipped"). The search starts on line 2. When one of the logical expressions in the CONDITIONAL block is satisfied, the current line pointer is moved to the beginning of the line which was read when the logical expression was satisfied (the line containing JONES and S947 or JONES and S960), and control transfers to the first command after the END CONDITIONAL. This command reads the record containing information about JONES in the specified FORTRAN format, and places the information into variables named GROUPID, NAME, ROOM, etc.

The final ECHO command will output the requested information.

**REFORM**

Syntax:

```

REFORM {lines}
      /INFILE=file_alias_name
      * /OUTFILE=file_alias_name
      * /OUT_FORM = format
      { /IN_FORM = format}
      { /UNTIL}
      { /ISTART = M}
      * { /STORE} { /NOSTORE (D)}
      /QUOTE = quote_character}
      * { /REPLACE {= (M1,M2)} }
      { /STRICT}
      { /DELIMIT = delimit_character}
      { /FUNCTIONS}
      { /SUBSTITUTE (= template | uc (D))}
      { /MISSING = stop | continue | next}
      { /MISS_TEXT = "string"}
      { /END = stop | continue}
      { /END_TEXT = "string"}
      { /NAMES = (associated_namelist)}
      { /LINES = variable_name}
      { /FIELDS = variable_name}

```

Description: REFORM is used to copy a specified number of lines of data from one file to another, optionally changing the format of the data in the process. The user can specify the format of the data to be read, or accept a default. The output format must be specified for a REFORM command. The user can also specify that the data which is being copied be stored in memory for future use.

The REFORM command is very similar to the READ command. The only differences involve the options marked with an asterisk (\*) above. These are only options described here. The remainder are described under the READ command.

The REFORM command is nearly identical to the COPY command, except that the output format must be specified. There is no default.

**OUTFILE** = file\_alias\_name: Specifies the alias name of the file to which the REFORM command will write. file\_alias\_name must correspond to a file which has previously been opened using the OPEN command.

**OUT\_FORM** = format: Specifies the format to be used in writing the data from each read. The format can be either a standard FORTRAN format specifier, or a machine-dependent list-directed free-format specifier (\*). ("(\*)" can only be used with sequential access file; do not use this specifier with direct access or "buffered" files). See the section on FORMAT SPECIFICATION for more information on format descriptors.

If a line-number is specified in the format, and multiple lines are being written, the line number in the format refers to the first line written. Subsequent lines will be written sequentially from this point.

**STORE:** If specified, then any data transferred by the REFORM command will also be stored in memory, and can be accessed by subsequent commands.

**NOSTORE:** If specified, then data transferred by the COPY command will not be stored in memory. Since memory is fixed and may be limited, it is recommended that data not be stored if it will not be required after the REFORM command is carried out.

This is the default.

Note that STORE and NOSTORE have the same affect as they do for READ, but the default (NOSTORE) is different here.

**REPLACE** {=(M1,M2)}: By default, when a write to a pre-existing line is performed on a file which was opened with the options /BUFFERED and /OVERWRITE=MERGED, non-blank characters in the new line of data to be written are merged into the pre-existing line, while any columns of the new line which are blank remain as they were in the old line. If /REPLACE = (M1,M2) is specified for such a buffered file, then the characters between columns M1 and M2 (inclusive) of the old line are completely replaced by the data of the new line for these columns, regardless of the flag specified for /OVERWRITE.

The range specifier "= (M1,M2)" is optional. If /REPLACE is specified with no arguments, the entire old line will be replaced by the entire new line.

If  $M1 \leq 0$ ,  $M1 = 1$  will be used. If  $M2 \leq 0$ ,  $M2 = \text{record\_length}$  will be used. REPLACE applies to every *physical* line which is written, not every COPY/REFORM *command*.

/REPLACE {=(M1,M2)} has no affect for files not opened as "buffered", or for files opened with /OVERWRITE=REPLACE.

See the READ command for discussion of the other options.

*Example:*

```
REFORM 3 /INFILE = file1 /OUTFILE = file2  \
      /IN_FORM = (2S,3I):2(ff) /OUT_FORM = (3I5)
```

would read 3 lines, starting with line 2, from the file which was opened with equivalence name "file1". For each line, the first two fields would be skipped, and the following 3 integer values would be read in free format. These would then be written to the "current" line in the file opened with equivalence name "file2" in format (3I5).

**WRITE**

Syntax:

```
WRITE (argument_list)
      /OUTFILE = file_equivalence_name
      { /OUT_FORMAT = format }
      { /REPLACE {=(M1,M2)} }
```

Description: The WRITE command allows a specified list of data items to be written to the chosen file, in the chosen (or default) format.

**argument\_list:** The list of data to be output. The argument\_list must be surrounded by left-right parentheses "()". Within the argument list, each argument is separated from the next by a comma ",". Any literal character string argument *must* be surrounded by double quotes.

Elements of the argument list can be numerical or character constants, variables, variable expressions, or IVS pointers. In addition, IVS pointers which refer to a range of variables can be used in the argument\_list. These will be expanded into a list of variables before executing the WRITE command. See the discussion of "Internal Variable Specifiers" in the Features and Considerations chapter.

Elements of the argument list are associated, in sequence, with elements of the output format statement. You can specify as many arguments as desired in the argument\_list.

**OUTFILE = file\_equivalence\_name:** Specifies the alias name of the file to which the REFORM command will write. file\_equivalence\_name must correspond to a file which has previously been opened using the OPEN command.

**OUT\_FORMAT = format:** Specifies the format to be used in writing the data from each read. The format can be either a standard FORTRAN format specifier, or a machine-dependent list-directed free-format specifier (\*). ("(\*)" can only be used with a sequential access file; Do not use this specifier with direct access or "buffered" files). See the section on FORMAT SPECIFICATION for more information on format descriptors.

**REPLACE {=(M1,M2)}:** By default, when a write to a pre-existing line is performed on a file which was opened with the options /BUFFERED and /OVERWRITE=MERGED, non-blank characters in the new line of data to be written are merged into the pre-existing line, while any columns of the new line which are blank remain as they were in the old line. If /REPLACE = (M1,M2) is specified for such a buffered file, then the characters between columns M1 and M2 (inclusive) of the old line are completely replaced by the data of the new line for these columns, regardless of the flag specified for /OVERWRITE.

The range specifier " $= (M1,M2)$ " is optional. If /REPLACE is specified with no arguments, the entire old line will be replaced by the entire new line.

If  $M1 \leq 0$ ,  $M1 = 1$  will be used. If  $M2 \leq 0$ ,  $M2 = \text{record\_length}$  will be used. REPLACE applies to every *physical* line which is written, not every WRITE *command*.

/REPLACE  $\{=(M1,M2)\}$  has no affect for files not opened as "buffered", or for files opened with /OVERWRITE=REPLACE.

See documentation on the OPEN command for more details on BUFFERED files.

*Examples:*

The following WRITE command would write the values associated with names charac1, real1 and integ1 to the file opened with file\_equivalence\_name dataout. The data would be written using the FORTRAN-type format specifier (A12,6X,F10.3,2X,I5):

```
WRITE ( CHARAC1 , REAL1 , INTEG1 ) /OUTFILE=DATAOUT  \
      /OUT_FORM=( A12 , 6X , F10 . 3 , 2X , I5 )
```

If the command GETTHEM results in one line of data being read, the following statement would write all the values read by this command to the standard output device using list-directed formatting (see section on IVS specifiers for more details on Internal Variable Specifiers like the one used in this example):

```
WRITE( [GETTHEM##$##1#1-$] ) / OUTFILE = STDOUT  \
      / OUT_FORM = ( * )
```

---

## Chapter 5

# Design and Debugging Tips

The Interface Programming Script (IPS) language allows for the straightforward generation of complex interface environments. However, as with any programming language, attention must be paid to design when generating the script. Otherwise, an improperly functioning interface may result. The purpose of this section is to provide some helpful suggestions to help the IPS programmer avoid common pitfalls, and to make debugging of such a script more efficient.

It is first important to be aware that the IPS is an *interpreted* language. This differs from such common languages as FORTRAN, C, etc., which are *compiled* languages. In an interpreted language, each command in the “program” is parsed and processed as it is encountered. Relatively little reorganization of the “program” code is performed to improve performance (execution speed). By contrast, the commands in a compiled language are parsed and converted to the low-level language which is the basic instruction set of the host computer. At the same time this conversion is being performed, changes are made in the code which provide the same results, but which can result in greater execution efficiencies.

There are a number of advantages of interpreted languages. Two of the most important are 1) Since no direct translation into machine code is required, complex and useful high-level functionalities may be introduced into the language with much less work. This means that such languages often offer many more intrinsic functionalities than compiled languages; and 2) Since such languages processors are themselves written in well-established languages (Interface is written in FORTRAN), they can be ported to almost any computer with little effort. For a compiled language, a separate version needs to be written for every machine architecture, and the development of the code for any specific machine can be difficult and extremely time-consuming.

There are also two disadvantages to an interpreted language. The first is that because language processor does not convert commands to machine-code, interpreted languages are usually *much* slower than compiled languages for the same task. For this reason, one does not typically use an interpreted language when the script provided to the language defines a series of compute-intensive instructions which could easily be programmed in a standard compiled language. The second disadvantage to interpreted languages is that most instructions in the language are not typically parsed until they are encountered in the script. This means that syntactical and other errors in the language script will not be flagged until the offending portion of the script is actually encountered in operation. Thus, one needs to be more diligent in testing the correctness of the script than one might need to be with a compiled language. On the plus side, any changes that turn out to be required can be made quickly, and the code re-run immediately, without the need for any (sometimes time-consuming) “re-compilation”.

## *Design and Debugging Tips*

Below, we discuss how the characteristics the Interface Programming Script language should affect design and testing of the IPS. We also present a number of common programming errors one should be attuned to when designing an IPS program.

### **Common IPS Programming Mistakes:**

Below are listed several mistakes commonly made, especially by the novice IPS programmer. Obviously, this list is not complete. The numerous error messages provided by Interface should lead you relatively quickly to the source of your mistake in many cases.

#### **1) Special characters in a text string or character argument must be protected.**

Several characters have special meaning to the Interface parsers. In particular, the forward slash (/) character delimits command options from each other (unless the delimit character has been changed by a DEFINE OSPU or DEFINE OSP command), and the diamond brackets (<>) enclose strings to be evaluated at run-time. If you wish to use these characters as part of a text string or argument, you need to “protect” them from the parser. For the / character, this can be done by surrounding the string containing the character by double quotes. A common mistake would be to specify a filename on a Unix computer without the protecting quotes, e.g:

```
REDIRECT = /usr/tmp/define.def
```

would result in an error, since Interface would think that /usr, /tmp and /define.tmp were all options of the REDIRECT command. The correct way to specify the above line is

```
REDIRECT = “/usr/tmp/define.tmp”
```

The surrounding double quotes protect the embedded forward slashes.

Similarly, when providing arguments to a command issued at the UC level, it is necessary to protect the option delimiter character. You have two options at the UC level: Either surround the argument with double quotes (only if it is a character argument) or else surround the entire argument list with left-right parentheses (). Remember that if you provide an arithmetic argument containing a forward slash representing the “divide-by” operation, you must protect this argument with left-right parentheses. For example, the following argument list specification at the UC level would result in an error

```
COMDO = This is a sample/example of specifying an argument.
```

The character string argument, which contains an embedded /, can be correctly specified as either

```
COMDO = “This is a sample/example of specifying an argument”
```

or

COMDO = (This is a sample/example of specifying an argument)

For an arithmetic argument containing a divide operation, only the surrounding brackets syntax is available:

COMDO2 = (3./4.)

would be equivalent to specifying COMDO2 = 0.75.

If the delimiter character has been redefined (e.g. if OSPU is set to a hyphen (-) using an appropriate DEFINE command), analagous restrictions to the above would apply to character string arguments containing a hyphen, and to arithmetic arguments containing a minus sign.

The expression within a pair of diamond brackets is always evaluated, regardless of whether or not the brackets appear between double quotes. The only way to protect diamond brackets from being so interpreted is to preface them with backslash characters. That is, \< and \> will be translated into < and > when a string is parsed, but the enclosed expression will not be evaluated.

**2) For a command continued onto one or more lines, the last non-blank character of each line except the last one must be the continuation character.**

This error frequently occurs when one is defining long text strings, such as those in an ECHO or DEFHELP command. It will typically result in an error message regarding an unrecognized command in the IPS, since Interface attempts to parse part of the character string as a command. It is valuable to create a “manual” for an IPS script as a quick check on missed continuation marks (see below).

**3) By default, attempting to access an undefined variable will result in an error.**

Interface does not, by default, provide a value for variables which have not previously been set. Any attempt to access an uninitialized variable will, by default, result in an error. This is often a valuable feature for debugging, since it allows you to quickly find variables being use but inadvertently uninitialized. However, you can override the default, effectively initializing all variables to a chosen real value, by setting the INOMT and RNOF options using the DEFINE command. If you will be using a variable whose assignment status is unknown, you can determine if it has previously been assigned by using the IASVAL() intrinsic function.

**4) Intrinsic commands and options can typically be abbreviated to no fewer to four characters.**

Every intrinsic command longer than four characters which is recognized in the UC and IPS scripts can be abbreviated. However, most require at least four characters (unless the

command/option is shorter than four characters). A very small number of options require more than four characters to differentiate them from other options available for the same command (e.g. MISSI and MISS\_ are the minimum acceptable abbreviations for MISSING and MISS\_TEXT with the READ/COPY/REFORM commands).

**5) MARKMEM and CLEARMEM/MARKMEM commands must be correctly balanced.**

For some scripts, it is necessary to perform memory management, either to optimize utilization of the memory, or to reinitialize memory (and defaults) before re-executing a sequence of code. In either case, one uses the markmem...clearmem/markmem commands to reclaim/reset memory. It is important that one ensure these commands are correctly balanced. It is not typically a good idea to spread markmem...clearmem/markmem command blocks over several command definitions, since this opens the possibility that the commands would become unbalanced if the user specified one command more than the other. (Some sophisticated scripts require such constructs; the point here is that one should attempt to keep things as simple and localized as is reasonably possible).

**6) To redefine the comment and/or continuation characters to be used in an IPS file, the appropriate DEFINE commands must appear before ANY other commands**

It is sometimes desirable to change either the default comment character (!) or default continuation character (\). If you wish to redefine these characters and want the new definitions to be in effect for the IPS file in which the redefinitions appear, the DEFINE commands which carry out the redefinitions must appear as the first commands in the IPS file. They must appear before ANY other commands, blank lines, or comments. Otherwise, the redefinitions will not go into effect until the IPS file in which they appear has been parsed.

**7) The value of a DO-loop iteration variable upon loop completion is the value the variable had on the last iteration of the loop.**

In standard FORTRAN, the do loop iteration variable will be set to the value of the iteration variable on the last loop iteration +1 (if the iteration variable increased with each iteration) or -1 (if the iteration variable decreased with each iteration). In INTERFACE, the value of the iteration variable is identical to its value on the last iteration. One must keep this in mind when trying to translate FORTRAN code to INTERFACE code.

## **Debugging Tips**

As noted above, because the IPS is an interpreted language, syntactical errors are often not flagged until the offending piece of code is actually executed. (Some syntactical errors will be flagged when the IPS script is translated into the direct access template file actually used by Interface). It is therefore suggested that you test the various options of your IPS script as thoroughly as possible as you develop it. A few procedures can help expedite the process:

- 1) Once you have developed a script which can be read without error by Interface, create a “manual”. You can do this by specifying “STDIN” as the input file for interface, and then issuing the command “MANUAL/OUTPUT=filename”. This will create a file named “filename” containing all the help information defined by DEFHELP commands in the IPS script. In the process, a variety of constructs in the IPS script will be examined, including COMMAND...END COMMAND blocks, OPTION...END OPTION blocks, and some (but not all) IF...THEN and DO...END DO constructs. Frequently, for a long script several errors will be flagged when the MANUAL command is issued. Thus, this provides a quick way of locating some errors. The MANUAL command entry provides more information on the MANUAL command.
- 2) You can set the IFVARI flag using the DEFINE command, e.g.

```
DEFINE IFVARI = 3
```

This will result in an echoing (trace) of UC and IPS commands being parsed as they are encountered. This can often make tracking down functional bugs in the IPS script easier. IFVARI can also be defined as 1 or 2, to echo only UC commands or only IPS commands.

- 3) It is frequently a good idea, when in the debugging phase of code development, to comment out any IFNOMT definition commands (see DEFINE IFNOMT). This will result in all variables being undefined unless/until they are explicitly assigned during program execution. Any attempt to access such an undefined variable (except in an IASVAL intrinsic function) will result in an error.

## **Error Report Format**

Every error reported by Interface is presented in a standard informative format. For example

```
-ERROR - from routine: EVALAR
  Unrecognized character "&" in string
  Current TEMPLATE-file line (from /usr/tmp/def/calc.dat):
    echo "7&&&" = <7&&&> /start = 5
  Last UC-file line (from input.dat):
    calculate 3&4
```

The following information is provided by the error message:

- 1) The name of the routine which flagged the error. In this case, routine EVALAR;
- 2) The error message. Here, an unrecognized character was found in the string being evaluated;
- 3) The last template (IPS) line parsed, and the file from which the line was read. If the error occurred while executing an IPS file line, the IPS line will be listed as “Current

TEMPLATE-file line”. If the error occurred while executing a UC file line, the template line will be listed as “Last TEMPLATE-file line”;

- 4) The last UC line parsed, and the file from which the line was read. If the error occurred while executing an IPS file line, the UC line will be listed as “Last UC-file line”. If the error occurred while executing a UC file line, the UC line will be listed as “Current UC-file line”.

### **Speed Considerations**

As noted above, the fact that the IPS is an interpreted language necessarily means that an IPS script will more slowly than the comparable code written in a compiled language. Still, there are some constructs that should be used sparingly in the IPS to avoid unnecessarily impacting performance. Specifically:

- 1) Accessing specific line numbers in sequential access files. It is possible to specify a line number in any read or write command, regardless of the access type of the file to be read/written. However, sequential access files do not directly support access by line number. INTERFACE compensates for this by determining the current line pointer, and then reading lines of the file until the appropriate line is encountered. This can be relatively slow for a long file, particularly if carried out numerous times.

A file opened for direct or buffered access allows quick direct specification of line numbers. It is suggested that if you plan to do many reads/writes to a file in which you specify a line number, that you open the file as either buffered or direct (see OPEN).

- 2) The time to access variables in Interface increases as the number of variables stored increases. Thus, you can improve performance by not storing large numbers of unnecessary variables, or by clearing memory when large contiguous blocks of variables are no long needed.
- 3) Free format reads are slower than standard formatted reads. If a formatted read can serve your purposes equally well (for example, when reading a file only to accomplish pointer placement), you should use it.

The following appendices contain examples of Interface scripts. Note that in the standard distribution, these scripts (and others) are contained in the directory `../interf/examples`, in files `*.def`.

## Appendices

### APPENDIX I) A calculator program.

The intrinsic functionalities of INTERFACE make it very easy to write a calculator program. By comparison, writing the same program with a standard language like FORTRAN or C could take thousands of lines of code. Below, we give the entire IPS required to generate a calculator. Note that there are only 15 lines of IPS code required.

```

!
! This is a calculator program
! It will accept both expressions (e.g. COS(45.*3.14/180.)) and variable
!   name assignments (e.g. A = 3*12.). Any variable name assigned can
!   subsequently be used in an expression

! Set the default value for undefined variables to 0...

    DEFINE INOMT = 0
    DEFINE RNOF = 0.0
!
    do i = 1,99999

! Note use of protecting backslash to output the ">" bracket.
!.redit is a statement label:

    redit: echo "\> "/outfile=stdout/noadv
!
! We use the special file_equivalence_name STDIN, which reads from standard
!   input (usually unit 5). We do not specify an input format, so the
!   default (A80) is used. If the end-of-file is encountered, stop with
!   no message. The string read is associated with the name "arg":

        read/infil=STDIN/name=arg/end_text = " "

! If the string is blank, skip to the next one

        if (len(blstr(arg)).eq.1 .and. arg(1:1).eq." ") goto redit

! See if the user entered "stop". If so, exit gracefully. Note use of
!   intrinsic function UCASE, which converts the string to all upper
!   case, and makes our name comparison easier.

        if (UCASE(arg(1:4)).eq."STOP") stop "y'all come back now"

! See if the string entered contains an "=" sign. If so, assume this is an
!   assignment statement. If not, just evaluate the expression:

        if (index(arg,"=").gt.0) then

! echo the variable name (the name before the = sign), with the evaluated
!   value of the righthand expression. Offset the output to start in
!   column 5. Note use of the BLSTR intrinsic function to remove any
!   leading or trailing blanks from the expression to the left of
!   the = sign.

            echo <blstr(arg(1:index(arg,"=")-1))> = \

```

```
        <<arg(index(arg,"")+1:len(arg))>>/start=5
! Note use of <>'s. The assign statement gets expanded to
!   assign variable = expression, where "variable = expression" is what
!   the user entered:
        assign <arg>
    else
! If no = sign found, just echo the input expression, an "=" sign, and it's
!   evaluated value.
        echo "<BLSTR(arg)>" = <<BLSTR(arg)>> /start = 5
    end if
end do
```

**APPENDIX II) Providing additional "intrinsic" commands in the UC file.**

It is sometimes desirable to allow some of the command capabilities which are by-default only available in the IPS file at the UC level. For example, you might wish to allow the user to issue an ECHO command from the UC file. This can be done by defining a command in the IPS file as follows:

```

COMMAND = ECHO
      ARGUMENTS (A) / EQUIVALENCE = ^/NAMES =
ECHO_ARG
      ECHO <ECHO_ARG>
END COMMAND

```

Then if the user specified the following command in the UC file

```
ECHO ("Hi There, Mom" /START=2)
```

the intrinsic echo command would output

```
Hi there, Mom
```

starting in line two. Note the use of quotes to protect the comma from being interpreted as a field delimiter, and the use of the outer brackets to protect the slash from being interpreted as an option delimiter by the UC command processor. Alternatively, we could have specified

```
ECHO "Hi There, Mom /START=2"
```

in the UC file with the same result.

### **APPENDIX III) A simple DELETE script: Interfacing to the host operating system.**

In more sophisticated scripts, you will often wish to pass commands to the host operating system. You may also wish to define commands which can take variable numbers of arguments. The following simple script, which defines a DELETE command at the UC level, illustrates both functionalities

```
! delete.def
! Define a delete command at the user level:

    command = dele*te
        markmem
        defhelp DELETE file1 {,file2, file3, ...}.
        defhelp
        defhelp Deletes the given set of files.\
            Up to 20 filenames, separated by commas, can be \
            specified. Enclose any filenames which contain\
            forward slashes "/" by double quotes, or enclose \
            the entire list by left-right parentheses ().

            arguments (20a) / missing=continue / fields = numdel /equiv = ^

! The following EXTERNAL command is appropriate for a Unix(r) operating
! system. Other operating systems would require a different string.
! For example, on a VMS/VAX, you would specify
!
!         external "delete/noconfirm <####<i>>"
!
! Note the use of the IVS to very simply designate each argument read in
! sequential order

        do i = 1,numdel
            external "\rm <####<i>>"
        end do

        clearmem/mark
    end command
```

**APPENDIX IV) Allowing the interactive user to change the prompt string**

If you are using INTERFACE to create an interactive environment, it is often desirable to create a prompt string that reminds the user what environment has been invoked. In addition, you might wish to allow the user to change the prompt string to something of their own preference. The following IPS script defines a UC command, `changeprompt`, which would allow this:

```
! changeprompt.def
! This IPS script creates a UC command
!
!     changeprompt = new_prompt_string
!
! This command allows the user to change the prompt string used when UC
! input is being read from STDIN (INTERFACE is being run interactively).
! Note we use the DEFINE command to change the prompt. Note also that
! if the user wishes to specify a prompt with a "<" or ">" character in it,
! the bracket character must be preceded by TWO backslashes, e.g. \\>.

    command = changeprompt
        markmem
        arguments (a) /name=promptstring
        if (len(promptstring).gt.15)\
            echo "Warning: prompt cannot exceed 15 characters"
        define prompt = "<promptstring>"
        clearmem/mark
    end command
```

## **APPENDIX V) A command to return the names of files in a directory to the UC level.**

Sometimes you will want to perform some function on all files in a directory which match a given specification. For example, you might have a “graph” command defined for the UC file, and wish to graph all files matching a certain name specification. This can easily be done using the UC construct

```

makdir directory_specification /nfile=numfiles
do i = 1,numfiles
  getdir /file=tempfile /number=i
  graph/file=tempfile
  delete tempfile
end do

```

(where the delete command is defined as in Appendix III).

The MAKDIR and GETDIR commands, which make it simple to obtain a list of all files that match a directory specification and use that list in a UC file, are defined in the example below.

```

! This is the IPS definition of commands MAKDIR and GETDIR for *Unix*
! systems. It assumes that the "ls" command results in a list of files,
! output as the first argument on a line, one per line.:
!
!     MAKDIR   file_specification   {/nfile = variable_name}
!             creates a file with the list of files corresponding
!             to a specified directory listing. If /nfile is specified
!             variable_name will be set to the total number of files
!             that matched the file specification.
!
!     GETDIR  /file = variable_name /number = N
!            gets the Nth filename in the list created by a MAKDIR
!            command. Variable variable_name is set equal to the
!            name of the Nth file.
!
! Example:
! At the UC level, the following script would echo the names
! of all the files *.f to the user, one per line (assuming the
! command "echo" has been defined at the UC level).
!
!     makdir *.f /nfile=nread
!     do i = 1,nread
!       getdir /file=flnam/number=i
!       echo <flnam>
!     end do
!
! Author: David A. Pearlman

```

```

command = makdir/replace
defhelp MAKDIR "" "directory specification" {/nfile=variable_name}"
defhelp
defhelp Creates a scratch file with a list of the files matching the\
passed directory specification. The "/nfile" option can be used\
to set the specified variable_name to the total number of\
entities which matched the directory specification.

```

```

markmem
arguments (a)/name=specif/equivalence=^

if (iasval("imkdir").ne.0) close dirlist
nvassign imkdir = 1
open dummy dirlist /scratch/direct/fixed=80

external "\ls <specif> \> tmpdrx0"
open tmpdrx0 dirlist2/old
copy 9999999/end=continue/lines=nlinn \
      /infile=dirlist2 \
      /outfile=dirlist
assign nlinn = abs(nlinn)
close dirlist2/delete

option = nfil*e
defhelp "NFILE = variable_name"
defhelp If specified, the total number of files which matched the\
variable specification will be placed in variable_name.

arguments (a)/name=nmdir0
nvcassign <nmdir0> = nlinn
end option

clear/mark
end command

command = getdir/replace
if (iasval("imkdir").eq.0) stop "You must specify MAKDIR before GETDIR"
defhelp "GETDIR /FILE = filename / NUMBER=I"
defhelp
defhelp Is used to read the list of filenames matching the\
specification given in a MAKDIR command (which must have\
preceded the GETDIR command). "/FILE" is used to specify the\
variable_name which will contain the name of the chosen file in\
the list. "/NUMBER" specifies that the Ith filename which\
matched the specification given in the MAKDIR command should be\
placed in "filename".

markmem

option = file
defhelp "FILE = filename"
defhelp Specifies the variable_name to be set to the Ith file\
matching the specification given in MAKDIR.

arguments (a)/name = drflnm
not_called
stop "must specify file= with getdir"
end not_called
end option

option = number
defhelp "NUMBER = I"
defhelp Specifies that the Ith file matching the specification given \
with the MAKDIR command will be placed in the variable name\
given with the FILE option.

arguments (i)/name=drnmbr
not_called
stop "must specify name= with getdir"
end not_called

```

## Appendices

## V) Returning directory list to UC level

```
end option

read \
  /infile=dirlist      \
  /in_form=(a):<drnbr>(ff) \
  /names = tmpnam
nvcassign <drflnm> = tmpnam

clear/mark
end command
```

**APPENDIX VI) Creating implied do-lists /multiple dimensional arrays.**

INTERFACE does not require or support variable declarations, and so does not support explicit arrays. In the description of in-line variable substitution (<>) it was shown how using such brackets one could easily generate the functionality of a one-dimensional array.

Generating the functionality of a multidimensional array is a bit more involved, but not too difficult, once one understands the IPS language. Everything hinges on the ability to surround a character variable by triangular brackets <variable> to yield the value of the variable.

The following example shows an IPS script which would define a command MAK2D to implement the functionality of an implied do-list for a 2-dimensional array at the UC level. This command would take six arguments: A variable name, the lower and upper indices for the 2 "dimensions", and whether the first or second dimension should vary more quickly. E.g.

```
MAK2D (array, 1,2 , 4,3 , 0)
MAK2D (array, 1,2 , 4,3 , 1)
```

would assign the variable\_name array to the following lists, respectively:

```
array_1_4, array_1_3, array_2_4, array_2_3
array_1_4, array_2_4, array_1_3, array_2_3
```

and the construct <array> could be used, e.g. in either the /NAMES list of a READ command or the argument\_list of a WRITE command to effect the functionality of a 2-dimensional implied DO-loop. So, for example, at the UC level we could use

```
MAK2D (array, 1,5 , 1,5 , 0)
READIT ARRAY
MAK2D (array, 1,5 , 1,5 , 1)
WRITIT ARRAY
```

where READIT and WRITIT are commands defined as standard formatted reads and writes, respectively, of the passed argument list. This would read a matrix of numbers in, then write the transpose of the matrix out. Understand that the MAK2D command does not specify or change the value of any named variable. It only specifies the order in which they will be expanded when the construct <array> is used.

To implement the 2D functionality at the IPS level, one could simply remove the "COMMAND =", "END COMMAND" and "ARGUMENTS" lines from the MAK2D script below, change CASSIGN to ASSIGN, and then use the lines

```

ASSIGN NAME = array_name
ASSIGN I1 = 1 (or any integer)
ASSIGN I2 = 5 (or any integer)
ASSIGN J1 = 1 (or any integer)
ASSIGN J2 = 5 (or any integer)
ASSIGN IORDER = 0 (or 1)
REDIRECT = mak2d.script

```

where mak2d.script is the file containing the MAK2D script.

*The MAK2D script used for the above:*

```

!
! Command which will redefine a variable name to be an implied do-list
! any time it is used in a namelist (or output values list) in the
! context <name>. This command is issued at the UC level as
!   MAK2D (variable_name , i1,i2, j1,j2 , iorder)
! which creates a list of names whose indices run from j1 to j2 and
! then from i1 to i2, each of the form "name_i_j". iorder determines
! whether i or j varies more quickly. This command can be used in
! reads and writes as an implied do-list, similar to the FORTRAN list
!
!   ((variable_name(i,j),i=i1,i2),j=j1,j2) (iorder = 0)
! or ((variable_name(i,j),j=j1,j2),i=i1,i2) (iorder = 1)
!
!   COMMAND = MAK2D
!           ARGUMENTS (A,5I)/NAMES = (NAME,I1,I2,J1,J2,IORDER) /EQUIV = ^ \
!           /MISS_TEXT = "Missing arguments for MAK2D 2-D array command"
!
! Mark the memory at the start of the concatenation process, so we can
! reclaim most of it.
!
!   MARKMEM
!
! Define the direction of each do-loop
!
!   ASSIGN INC1 = 1
!   IF (I2.LT.I1) ASSIGN INC1=-1
!   ASSIGN INC2 = 1
!   IF (J2.LT.J1) ASSIGN INC2=-1
!
! Do the concatenation. Note the CLEAR/MARKMEM command. This command
! reclaims the memory required by the previous character string assignment
! to NAMBIG. We assign variable NAMBIG with a NVASSIGN statement so that
! its current value will be maintained following the CLEAR. The
! MARKMEM...CLEAR/MARKMEM sequence is issued to reclaim memory on each
! iteration:
!
!   ASSIGN NAMBIG = " "
!   IF (IORDER.EQ.0) THEN
!     DO I = I1,I2,INC1
!       DO J = J1,J2,INC2
!         MARKMEM
!         ASSIGN NAMBIG = NAMBIG + NAME + "_<I>_<J>,"
!         CLEAR/MARKMEM
!       END DO
!     END DO
!   ELSE
!     DO J = J1,J2,INC2
!       DO I = I1,I2,INC1

```

```
MARKMEM
ASSIGN NAMBIG = NAMBIG + NAME + "_<I>_<J>,"
CLEAR/MARKMEM
      END DO
    END DO
  END IF

! Remove the trailing comma

      ASSIGN NAMBIG = NAMBIG(1:LEN(NAMBIG)-1)

! Clear all character memory used here. Then immediately assign the passed
! variable_name to the concatenated string (before NAMBIG becomes
! undefined). Use a CASSIGN command so the variable is accessible in the
! UC file, but will not affect any variables in the IPS.

      CLEAR/MARKMEM
      CASSIGN <NAME> = NAMBIG
    END COMMAND
```