? !

Hook-Bang-Crunch

(Programming the CLI)

by Greg Shalless

ISBN 0 7316 6442 6

? !

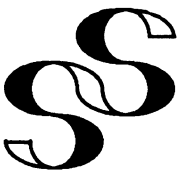
Hook-Bang-Crunch

(Programming the CLI)

by Greg Shalless

ISBN 0 7316 6442 6

Published by



P.O. Box 469,
Malvern,
Victoria 3144,
Australia

NOTICE

?!#
Hook-Bang-Crunch
(Programming the CLI)

Revision History: Effective with:

Revision 1 Apr 1989 AOS/VS Rev. 7.57, SORT/MERGE Rev. 3.20

(Revision 2 of this Manual will not appear until after the authors have had considerable exposure to AOS/VS II. If changes are, as we hope, minimal, Revision 2 will be made available to existing owners of the Manual, in the form of a low-cost replacement pages Update).

INTRODUCTION

This Manual is designed to teach its readers how to use the **CLI** (Command Line Interpreter) provided with Data General's AOS and AOS/VS Operating Systems, as a Programming Language. **Hook-Bang-Crunch** is its name because this is **DG-ese** for the three characters **?**, **!** and **#**, which occur commonly in CLI macros, and after all **? ! #** is what complex CLI Macros are to most people. Hopefully this manual will help remove some of the mystery for you. It is designed primarily to fill the huge gap left by Chapter 5 (CLI Macros) of Data General's own "Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)" (093-000122). This chapter merely whets the Macro writing appetite, but seasoned programmers, system managers and operators need to, and with the aid of this Manual will be able to, get much more out of the CLI. Most people usually only learn to write CLI Macros from looking at other people's macros, and with all due respect to Data General, this usually means Data General's macros, which on many occasions leave a lot to be desired. Many of the so-called tricks of the trade only come with a wealth of CLI Macro writing experience. The author has had some eight years experience with Data General's AOS and AOS/VS Operating Systems, firstly as a Training Instructor at Data General Australia and subsequently as a Consultant in a variety of roles including both Application and Systems Programming. He brings much of his depth of experience and skill as a CLI Macro writer to you in this manual. The Manual is liberally spiced with meaningful and useful Macros, all of which can be separately purchased on Tape.

Special Features include techniques for writing your own Pseudo-Macros, correct use of PUSH, POP and PROMPT POP, correct use of /1 and /2 switches and what they really mean, techniques for writing macros without the need for more than one STRING variable or for that matter for a longer one, and how to avoid "CLI out of Memory" errors. In addition there are special appendices on DG's two other hidden programming languages, namely the text editor **SPEED** and the **SORT/MERGE** utility.

It is our firm belief that this manual will become a necessary tool of trade for all users of Data General's AOS and AOS/VS Command Line Interpreter, with a need to write CLI Macros.

CONVENTIONS

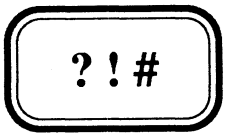
All Shalless Software Macros listed in this document begin with the valid AOS and AOS/VS filename character **$** (dollar), to minimize the likelihood of a name clash with any of your own macros. Therefore if you adopt the convention of never starting any of your Macros with a **$** then all Shalless Software Macros may be implemented without change on your own System. To distinguish Macros from User-Pseudo-Macros (UPMs see Chapter 19) all Shalless Software UPMs begin with **$$** (a pair of dollars). All of course have a filename extension of **.CLI**. All Shalless Software Macros and UPMs use full spelling of all CLI commands and Pseudo-Macros, except for a few commonly accepted abbreviations such as **X** (for XEQ or EXECUTE), **DIR** (for DIRECTORY), **!USUB** (for !USUBTRACT) and **!FILENAME** (for !FILENAMES when only one filename is being expanded).

All Shalless Software SORT/MERGE Command Files also start with a **$**, but have an extension of **.SMC**, while Shalless Software SPEED Command Files start with a **$** and have an extension of **.SPD**.

TABLE OF CONTENTS

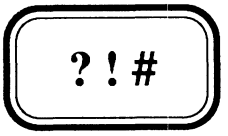
APPENDICES

SECTIONAL CONTENTS



CHAPTER 1

The CLI, Macros and Programming



CHAPTER 1

The CLI, Macros and Programming

1.1 The CLI

The Command Line Interpreter, or CLI (cee-el-eye) as it will be referred to in future is the simplest interface Data General provides to its AOS and AOS/VS Operating Systems. For the remainder of this document only AOS/VS will be referred to, but in most cases it is equally valid for AOS.

Many CLI commands translate directly into a single System Call, which is really nothing more than a request for AOS/VS to do something for you. Although the full range of System Calls is made available to Programmers, through most of DG's (as Data General will be subsequently referred to) Programming Languages, a quick glance at the AOS/VS System Call Dictionary (093-000241) will make you realise that to make even the simplest Systems Calls from any of these Languages is significantly more difficult than it is to issue the corresponding CLI command.

Provided that the appropriate Commands are available, you simply enter them in response to CLI's "I'm waiting for a Command" Prompt and Bob's your uncle. It is not the intention of this Manual to teach you all the CLI commands, which are of course fully documented in the CLI User's Manual (093-000222) and are summarized quite well in the AOS and AOS/VS User's Handbook (093-000150). It is our intention however, to look in detail at some of those commands, which we believe require further explanation so that their implications can be more readily appreciated.

In fact this manual assumes that you have a basic knowledge of CLI and its commands, at about the level you would require in order to Operate a PC. In particular this means you know basically what a file is, how to move around the disk directories to locate files, how to delete, create, edit, rename and print them, and how to run Programs.

1.2 Macros

In its simplest form a CLI Macro is merely a file of CLI Commands, whose name can be entered as if it were itself a CLI Command and all the Commands in the Macro will be carried out in turn. You often create such Macros, or Command Files for often repeated sequences of Commands to reduce typing. The filename of a macro should contain the extension **.CLI**. The macro is then invoked by entering the Macro's name without the extension. Thus the Macro held in the file **MYMACRO.CLI**, would be invoked by typing in **MYMACRO**.

If this was all there was to CLI Macros, then they wouldn't be all that useful, and certainly wouldn't be worth writing a Manual such as this about. The next level of complexity comes with the ability to pass arguments to your Macros in exactly the same way that you give arguments to CLI commands. This enables you to make your macros more like your very own CLI commands and therefore more useful in the long term.

For example:

If you are prone to accidently delete files (isn't everybody?) you might wish to turn the **Permanence** attribute on certain important files such as your Macros to prevent this. However this sometimes becomes a nuisance when you wish to edit the file, because you have to remember to turn if off before editing and on again afterwards. Let's say that in addition you dislike the **.ED** files **SED** (DG's easiest to use text editor) leaves around cluttering up your disk. So write a macro.

File $PSED.CLI

```
PERMANENCE %1% OFF
X SED/NO_ED %1%
PERMANENCE %1% ON
```

You would then use the **$PSED** Macro whenever you wanted to edit one of your permanent files thus:

```
) $PSED MYMACRO.CLI
```

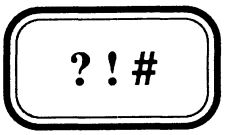
The expression **%1%**, is interpreted by CLI as this Macro's first argument. This feature will be explained in more detail in Chapter 3. Chapter 4 then goes on to explain how you can make your Macros even more like CLI commands by giving them switches, which are usually used to slightly modify the functionality of your Macro, as they are in CLI to slightly modify a command's functionality.

1.3 Programming

Programs are written to manipulate Data stored on Disks attached to Computers. Basically a program consists of a series of commands telling the computer how to manipulate the Data. They are usually written in a Programming Language (such as COBOL, FORTRAN, C etc.), which are designed to simplify the task of writing Programs. In one form or another they usually provide the Programmer with mechanisms or constructs for:

a) Reading in Data
b) Writing out Data
c) Manipulating Data
d) Doing Arithmetic on Numeric Data
e) Performing a Task a specified number of times
f) Performing a Task until some Condition is satisfied
g) Communicating with the Operating System
h) Reporting the failure of a Task to perform as expected
i) Receiving input from the User of the Program
j) Displaying output to the User

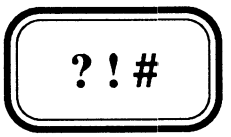
There are no doubt others. Since the CLI through its macro facility provides constructs for doing all of these things, even though at times not as easily as we might like, then it too is really a **Programming Language**. The ability to use it as a programming language is unfortunately (for you - fortunately for us) not as obvious as perhaps it should be. Hence this Manual. We hope it meets your expectations.



CHAPTER 2

HELP and how to get it

2.1 Help



CHAPTER 2

HELP and how to get it

2.1 Help

2.1 Help

The most complete sources of Help on the CLI are the various Manuals DG provide, and of course this Manual. These include:

Manual	DG - Model no.
Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)	093-000122
AOS and AOS/VS User's Handbook	093-000150

However it is sometimes difficult to lay your hands on a manual and for this reason the CLI has its own extensive on-line help facility, through the CLI **HELP** command. By simply typing HELP you will get a list of topics upon which HELP is available. By entering:

```
) HELP *topic
```

you will get the Help for that topic displayed on the screen. One of the topics is **COMMANDS**, so HELP *COMMANDS will tell you all the CLI commands. By entering:

```
) HELP command
```

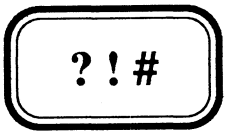
you will get the Help for that command. This usually only tells you about the command's arguments and its valid switches. For a more detailed explanation including the command's purpose, the meaning of all its arguments and switches, the calling sequence and sometimes some examples of its use you type **HELP/V command**. However as this is generally more than a screen's worth of output and to save you having to hit ^S (CTRL-S) to suspend output to catch what you want to read and ^Q (CTRL-Q) to resume output, it is better to use the **HELPV** Macro which automatically suspends output after each screen full. To see the next screen you then hit ^Q (CTRL-Q). Thus:

```
) HELPV FILESTATUS
```

would give you the detailed Help available on CLI's FILESTATUS command, a screen at a time.

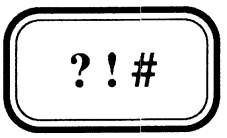
It is possible to use the CLI's HELP mechanism, for your own Help messages by creating your own Help Text Files in the :HELP directory. Details of how to do this are given in Chapter 12.

THIS PAGE INTENTIONALLY LEFT BLANK



CHAPTER 3

Macro Arguments



CHAPTER 3

Macro Arguments

3.1 Macro Invocation

People have been known to invoke CLI Macros in different ways, for example some include the **.CLI** extension, while others do not. Some include the Macro call in square brackets, while others do not. What is the correct way? Well sometimes this depends on what you are doing, but if you understand how CLI processes your Macro, it is most likely that you will get it right.

CLI arguments are separated from each other by a Space, Tab or a Comma. These separators are equivalent except that two Commas in a row indicates a null argument, while multiple white space characters (Spaces and Tabs) do not constitute additional arguments. Furthermore there may be additional white space characters surrounding a Comma but this still only delimits one argument. A CLI command or CLI macro is separated from its arguments by a standard argument separator, and this is why the Macro name is often referred to as Argument Zero. If we invoke a CLI macro in the most common way, let us now examine how CLI will process it. For example:

```
) $PSED MYMACRO.CLI
```

CLI will:

a) Isolate the First Word of the Command Line, ie. Argument Zero (Arg0). In this case it's **$PSED**.

b) Is it a known CLI command or an abbreviation thereof? If so, invoke the command unless the abbreviation is not minimally unique, in which case an error is reported. In this case it is not a CLI command.

c) Search in the User's current **Working Directory** (ie. the Directory you are currently "in"), for the file Arg0.CLI which in this case means to search for **$PSED.CLI**. If found read in the contents of that file and go to step i).

d) If not found in the Working Directory, then search the Directories specified in the User's **Searchlist**, for the file Arg0.CLI. Each process on the System has a Searchlist associated with it, which specifies a list of directories to be searched whenever a reference to a file cannot be satisfied by the Working Directory. If found read in the contents of that file and go to step i).

e) If not found search the Working Directory for the file Arg0 which in this case would mean to search for **$PSED**. If found read in the contents of that file and go to step i).

f) If not found search for Arg0 (**$PSED**) in the Directories on the Searchlist. If found read in the contents of that file and go to step i).

h) If not found report the error "Error: Not a command or macro"

i) Get the command line and make appropriate argument and switch substitutions, before beginning interpretation of the macro.

It should be clear that our macro would run correctly, whether we invoked it by calling it with or without the .CLI extension. However if we invoked it with the .CLI extension we would waste CLI's time making it search for $PSED.CLI.CLI first in both the Working Directory and the Directories on the Searchlist. Similarly it is not even necessary for our macro to have a .CLI extension, since it will still be found, however there will first be a fruitless search for $PSED.CLI. What's more the search for $PSED.CLI may not be fruitless and we may run the wrong version of the macro.

The issue of Macro Calls in square brackets will be dealt with in more detail in Chapter 9 (Understanding Brackets and Templates) but for the time being we will state simply that when a macro call is enclosed in square brackets (eg. **[$PSED MYMACRO.CLI]**), CLI assumes that Arg0 is in fact a Macro and will therefore skip Step b) above, and not bother checking to see if it's a command. You could use this technique to call a macro that had the same name as one of CLI's commands, but this is not recommended.

In conclusion then, it should be clear to you that you should always give your Macros a .CLI extension, but that you should never specify the extension when invoking the Macro. The most efficient way of calling our macro however is to enclose the call in square brackets:

```
) [$PSED MYMACRO.CLI]
```

Efficiency is not always the only consideration however, and other implications of calling macros by enclosing the call in square brackets will be dicussed in Chapter 9.

3.2 Single Arguments

Within the body of a Macro the expression **%n%** refers to argument n. Substitution of these expressions with the actual arguments takes place immediately after CLI has read in the macro. Argument n is the string of text that appears between the nth and the n+1th argument delimiters, in the Command Line. Thus the expression %n% must also include all the switches, if any on argument n. References to non-existent arguments are replaced by the null string. **%0%** is a special case, referring to argument zero which as previously explained refers to the macroname. Strictly speaking though it is the string of text that appears before the first argument delimiter (or the end of the command if there are no delimiters). For example if a macro were invoked using the Full Pathname to the macro. eg:

```
) :UDD:MYUSERNAME:MYMACRO
```

then any reference to the expression **%0%** inside MYMACRO.CLI would be replaced by the character string :UDD:MYUSERNAME:MYMACRO.

Summarizing then we have:

%n% Argument n and all its switches

%0% Argument Zero (ie the macroname) and its switches

3.3 Multiple Arguments

Ranges of arguments can be referenced from within the body of a Macro using the string expression **%m-n%**, which is replaced by Arguments m through n (n > m), from the command line. The values **m** and/or **n** may be dropped from the range expression with CLI assuming a default value for **m** of 1, and a default value of 32768 for **n** which in effect, since references to non-existent arguments are replaced with the null string, means up to however many arguments there happen to be. Summarizing then we have:

%m-n% Arguments m through n

%m-% Arguments m onwards

%-n% Arguments 1 through n

%-% Arguments 1 onwards (ie all this macro's arguments)

%0-% Arguments 0 onwards (ie the entire command line)

%0-n% Arguments 0 through n

In all cases of range references the above expressions always include all the included arguments' switches. It is possible to isolate a single argument from its switches with the expression **%n\%**, but there is no equivalent expression for isolating the arguments in a range from their switches, other than by referring to each argument separately, thus: **%0\% %1\% %2\% %3\%** etc. up to the maximum number of arguments you expect. The author cannot, however find a great deal of use for such an expression. Thus then we have

%n\% Argument n without its switches

There is actually a more general form of the Range expression of which all the preceding expressions are special cases. This is **%m-n,i%**, where **i** is the increment parameter which has a default value of one. Thus the expression **%-,2%** would expand to every second argument starting at argument one. This variant of the Range expression is rarely used, and the author can recall using it meaningfully only once, and this use of it was so obscure, he cannot now recall it. Finally then we have

%m-n,i% Arguments m, m+i, m+2i, m+3i etc. till m+?i not > n

3.4 Examples

It is difficult to give you particularly meaningful example Macros, to demonstrate the various argument expressions, without using concepts that we have yet to cover. Thus we have decided to use as an example a macro that does nothing more than use the CLI **WRITE** command to display information on the screen about the arguments it has received. First a word about the WRITE command. It writes out the arguments it receives (Note: commands have arguments just like macros) separating each argument with a space. Thus you will notice the use of multiple commas in the macros **$ARGTEST** and **$ARGTEST.1** to give the WRITE command some null arguments, to force the generation of multiple spaces in its output, in order to align the dashes.

File $ARGTEST.CLI

```
WRITE THIS IS THE MACRO - $ARGTEST
WRITE THE ARGUMENTS ARE - %-%
WRITE ARGUMENTS 2-4 ARE - %2-4%
WRITE CONCATENATED 2-4,,- %2%%3%%4%
WRITE NO SWITCHES 1-3,,,- %1\% %2\% %3\%
WRITE COMMAND LINE WAS,,- %0-%
WRITE
WRITE NOW I'M CALLING $ARGTEST.1 DROPPING ARGUMENT 1
WRITE
$ARGTEST.1 %2-%
WRITE
WRITE CALLING %0\%.1 DROPPING EVERY SECOND ARGUMENT
WRITE
%0\%.1 %1-,2%
```

File $ARGTEST.1.CLI

```
WRITE This is the macro - "%0\%"
WRITE First three args,,- "%-3%"
WRITE Other args are,,,,- "%4-%"
WRITE Reverse&join 1-5,,- "%5%%4%%3%%2%%1%"
```

You will notice that $ARGTEST calls $ARGTEST.1 explicitly once and then a second time implicitly by use of the expression **%0\%.1**. However if we tried to invoke $ARGTEST by including the .CLI extension thus:

```
) $ARGTEST.CLI ONE TWO THREE
```

the implicit call would expand to **$ARGTEST.CLI.1** giving rise to the error:

```
Error: Not a command or macro, $ARGTEST.CLI.1,ONE,THREE
```

First let's try a simple invocation of $ARGTEST using easily identifiable arguments:

```
) $argtest one two three four five six 7 8 9 10

THIS IS THE MACRO - $ARGTEST
THE ARGUMENTS ARE - one two three four five six 7 8 9 10
ARGUMENTS 2-4 ARE - two three four
CONCATENATED 2-4  - twothreefour
NO SWITCHES 1-3   - one two three
COMMAND LINE WAS  - $argtest one two three four five six 7 8 9 10

NOW I'M CALLING $ARGTEST.1 DROPPING ARGUMENT 1

This is the macro - "$ARGTEST.1"
First three args  - "two three four"
Other args are    - "five six 7 8 9 10"
Reverse&join 1-5  - "sixfivefourthreetwo"

CALLING $argtest.1 DROPPING EVERY SECOND ARGUMENT

This is the macro - "$argtest.1"
First three args  - "one three five"
Other args are    - "7 9"
Reverse&join 1-5  - "97fivethreeone"
```

Most of the above results are self-explanatory and expected. One point of interest for the **UNIX** buffs is that AOS/VS is case insensitive with respect to filenames, and thus the same macro is invoked whether it is invoked as $ARGTEST.1 or $argtest.1. This to the author is eminently more sensible than the UNIX convention.

Now a more complex call:

```
) $argtest/switch one/withswitch,,arg2 was null .six.

THIS IS THE MACRO - $ARGTEST
THE ARGUMENTS ARE - one/withswitch  arg2 was null .six.
ARGUMENTS 2-4 ARE - arg2 was
CONCATENATED 2-4  - arg2was
NO SWITCHES 1-3   - one arg2
COMMAND LINE WAS  - $argtest/switch one/withswitch  arg2 was null .six.

NOW I'M CALLING $ARGTEST.1 DROPPING ARGUMENT 1

This is the macro - "$ARGTEST.1"
First three args  - "arg2 was null"
Other args are    - ".six."
Reverse&join 1-5  - ".six.nullwasarg2"

CALLING $argtest.1 DROPPING EVERY SECOND ARGUMENT

This is the macro - "$argtest.1"
First three args  - "one/withswitch arg2 null"
Other args are    - ""
Reverse&join 1-5  - "nullarg2one/withswitch"
```

Some of the output of this example requires further explanation. The two spaces between **one/withswitch** and **arg2** (which is really argument 3) is a result of the fact that argument 2 was null and CLI's WRITE command writing out a space between arguments. Thus it wrote **one/withswitch**<space><null><space>**arg2**. Note by <null> we do not mean the ASCII null character but simply the null string ie. a string containing no characters.

Using the same line of reasoning one might have expected the fifth line of $ARGTEST's output to read:

NO SWITCHES 1-3 - one arg2

with two spaces because argument 2 was null, but there is only one space. Although white space characters (Spaces and Tabs) may be used to separate arguments, before processing any command or macro CLI always removes all the white space separating each argument from the next with a single comma. However we were not so careful, note the line of our macro:

WRITE NO SWITCHES 1-3,,,- %1\% %2\% %3\%

because argument 2 is null this becomes

WRITE NO SWITCHES 1-3,,,- one arg2

as expected, but before being processed by CLI becomes

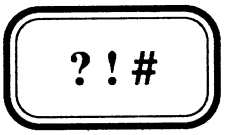
WRITE,NO,SWITCHES,1-3,,,-,one,arg2

resulting in only one space between **one** and **arg2**. We would have got the expected output if we had coded our macro thus:

WRITE NO SWITCHES 1-3,,,- %1\%,%2\%,%3\%

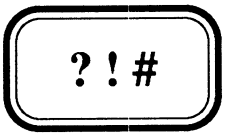
This may seem a trivial distinction to make, however when you have macros expecting certain types of arguments in particular argument numbers the distinction and correct coding of your macro can be critical. What you thought was argument 3 may well become argument 2 or even argument 1 in a subsidiary macro if arguments 2 and/or 1 are null arguments.

Hopefully you also observed that argument 1 lost its switch in this line because of the expression **%1\%** rather than **%1%**.



CHAPTER 4

Macro Switches



CHAPTER 4

Macro Switches

4.1 Single Switches

Switches are really just another form of parameter, for passing information about what you want done to a command or macro. The term presumably is derived from our concept of the electrical switch which can be either on or off. In the case of CLI macros and commands a switch is either on or off, by virtue of it being either present or not present. Most commonly when it is on or present you want the macro to do something extra, or to perform its function in a slightly different way. Because CLI commands normally only have switches attached to the command, we tend to think this way about the macros we write. However switches can actually be attached to any argument to perhaps modify the way in which that argument is treated.

A single switch in CLI is recognized as the string of characters beginning with the slash character "**/**" and terminating at the character before either an argument delimiter (Space Tab or Comma), a command delimiter (Newline, Carriage-Return, Erase-Page or Semi-colon) or another slash indicating the beginning of another switch. The expression **%n/switch%** inside a macro will be replaced by the complete string of characters forming the **/switch** switch on argument **n**, if of course such a switch is present on that argument. If the switch is not present on argument n then the expression is replaced with the null string.

The most common form of this to pick up switches on the macro itself, that is argument zero, is expressed thus **%0/switch%**. This is commonly abbreviated to drop the argument number altogether thus **%/switch%**, where CLI assumes a default of argument zero.

Unlike electrical switches CLI macro switches may be in any number of states when present, in that they may have a value. A switch with a value is expressed **/switch=value** when placed on an argument to be passed to a command or macro. Inside the macro body the expression **%n/switch%** always includes the string of characters making up the switch and its value. To isolate the switch's value the expression **%n/switch=%** is used.

The following table shows 3 switch expressions appearing in the body of the macro **TEST**, and several different ways in which the macro **TEST** might be called. The corresponding entry in the table shows the resulting expansion of the expressions in the macro, where an empty entry indicates the null string:

Call \ Expression	**%/D%**	**%0/L=%**	**%1/DEF%**
TEST/D/L=FRED XYZ/DEF=1	/D	FRED	/DEF=1
TEST/l=fred xyz		fred	
TEST/L/d /dEF xyz/DEF	/d		/dEF
test a/L=fred/def=hi/D			/def=hi
test/d=fred/L==fred.ls	/d=fred	=fred.ls	

Thus for single switches we have the following expressions:

%n/switch% /switch if /switch present on argument n, otherwise <null>. Note the expansion includes the switch's value if it has one.

%/switch%
%0/switch% /switch if /switch present on argument zero, that is the macro, including any value the switch may have, otherwise <null>.

%n/switch=% expands to the value of the /switch switch on argument n, if it is present and has a value, otherwise <null>.

4.2 Multiple Switches

We discussed in Chapter 3 how the expression **%n\%** would isolate an argument from its switches, well it is also possible to isolate all of the switches from their argument with the expression **%n/%**. This is particularly useful if you wish to pass all of a macro's switches on to some command, program or subordinate macro. To expand particular switches only from an argument you simply specify these switches in the order you want them expanded thus **%n/sw1/sw2%**. This expression will expand to **/sw1/sw2** if both switches are present and regardless of their order of appearance on argument n. If only one of the switches is present only that switch is expanded, while the null string results from neither being present. This concept can be extended to any number of switches.

In a situation where we have a large number of potential switches and we wish to pass all except a particular one onto some other command, program or macro, it can become tiresome to use the above technique and explicitly mention each switch we want passed. Fortunately CLI also recognizes the expression **%n\switch%**, as meaning all the switches on argument n except the **/switch** switch. Furthermore you can make multiple exceptions with the expression **%n\sw1\sw2%**, which expands to all the switches on argument n except the **/sw1** and **/sw2** switches.

As with single switches any of these expressions may have the value zero for n to refer to the switches of the macro itself (which as previously explained is viewed as argument zero). The reference to n may also be omitted where zero is assumed. Thus we have:

%n/% All the switches on argument n

%n/sw1/sw2%

<null>	if neither /sw1 nor /sw2 present
/sw1	if /sw1 present but not /sw2
/sw2	if /sw2 present but not /sw1
/sw1/sw2	if both switches present on argument n regardless of their order of appearance on argument n.

%n\switch% All the switches on argument n except /switch

%n\sw1\sw2% All the switches on arguemnt n except /sw1 and /sw2

4.3 Examples

As we did in Chapter 3 on macro arguments we will use demonstration macros consisting purely of CLI WRITE commands to demonstrate the use and resultant expansion of the various switch expressions. Armed with the tools we will have built up by the end of the next chapter we will be able to give you much more meaningful examples.

File $SWTEST.CLI

```
WRITE THIS IS THE MACRO - $SWTEST
WRITE THE SWITCHES ARE,,- %0/%
WRITE MACRO /D SWITCH,,,- %/D%
WRITE LISTFILE FROM /L,,- %0/L=%
WRITE ARG 1 /DEF SWITCH - %1/DEF%
WRITE COMMAND LINE WAS,,- %0-%
WRITE
WRITE NOW I'M CALLING $SWTEST.1 PASSING /D AND /L SWITCHES
WRITE
$SWTEST.1%/D/L% %-%
WRITE
WRITE CALLING %0\%.1 PASSING ALL BUT /L SWITCH LISTFILE IS ARG 1
WRITE
%0\%.1%0\L% %0/L=% %-%
WRITE
WRITE SAME AGAIN BUT ALLOWING FOR NON-EXISTANT /L SWITCH
WRITE
%0\%.1%0\L%,%0/L=%,%-%
```

File $SWTEST.1.CLI

```
WRITE This is the macro - "%0\%"
WRITE My switches are,,,- "%/%"
WRITE Listfile is Arg1,,- "%1%"
WRITE /L & /D switches,,- "%0/L/d%"
WRITE other switches,,,,- "%0\D\L%"
```

We will call this macro using switches similar to those used in the table on Page 4-1 to verify that we actually get those results:

```
) $SWTEST/D/L=FRED/EXTRA XYZ/DEF=1

THIS IS THE MACRO - $SWTEST
THE SWITCHES ARE  - /D/L=FRED/EXTRA
MACRO /D SWITCH   - /D
LISTFILE FROM /L  - FRED
ARG 1 /DEF SWITCH - /DEF=1
COMMAND LINE WAS  - $SWTEST/D/L=FRED/EXTRA XYZ/DEF=1

NOW I'M CALLING $SWTEST.1 PASSING /D AND /L SWITCHES

This is the macro - "$SWTEST.1"
My switches are   - "/D/L=FRED"
Listfile is Arg1  - "XYZ/DEF=1"
/L & /D switches  - "/L=FRED/D"
other switches    - ""

CALLING $SWTEST.1 PASSING ALL BUT /L SWITCH LISTFILE IS ARG 1

This is the macro - "$SWTEST.1"
My switches are   - "/D/EXTRA"
Listfile is Arg1  - "FRED"
/L & /D switches  - "/D"
other switches    - "/EXTRA"

SAME AGAIN BUT ALLOWING FOR NON-EXISTANT /L SWITCH

This is the macro - "$SWTEST.1"
My switches are   - "/D/EXTRA"
Listfile is Arg1  - "FRED"
/L & /D switches  - "/D"
other switches    - "/EXTRA"
```

These results are fairly straightforward and expected. Furthermore they are consistent with the similar example in the Table on Page 4-1. Notice also that despite the reference on Line 4 of **$SWTEST.1**, to the **/D** switch using a lower-case d, **%0/L/d%** the expansion retains the upper-case /D from the command line. Thus CLI is case insensitive with respect to switch names.

This is again an example based on the table on Page 4-1, but unlike in that example we have added an extra switch between the /L and /d switches:

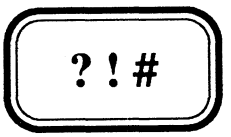
```
) $SWTEST/L/extra/d /dEF xyz/DEF

THIS IS THE MACRO - $SWTEST
THE SWITCHES ARE  - /L/extra/d
MACRO /D SWITCH   - /d
LISTFILE FROM /L  -
ARG 1 /DEF SWITCH - /dEF
COMMAND LINE WAS  - $SWTEST/L/extra/d /dEF xyz/DEF

NOW I'M CALLING $SWTEST.1 PASSING /D AND /L SWITCHES

This is the macro - "$SWTEST.1"
My switches are   - "/d/L"
Listfile is Arg1  - "/dEF"
/L & /D switches  - "/L/d"
other switches    - ""

CALLING $SWTEST.1 PASSING ALL BUT /L SWITCH LISTFILE IS ARG 1

This is the macro - "$SWTEST.1"
My switches are   - "/extra/d"
Listfile is Arg1  - "/dEF"
/L & /D switches  - "/d"
other switches    - "/extra"

SAME AGAIN BUT ALLOWING FOR NON-EXISTANT /L SWITCH

This is the macro - "$SWTEST.1"
My switches are   - "/extra/d"
Listfile is Arg1  - ""
/L & /D switches  - "/d"
other switches    - "/extra"
```

The point to note about this example apart from producing results consistent with the table, is that even though a /L switch was present it did not have a value, resulting in a null argument in the last two calls on $SWTEST.1. However because of the way CLI removes white space characters only the second call on it picked up the fact that this was a null argument. In the first case what was meant to be argument two became argument one.

Finally

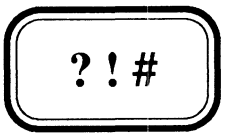
```
) $swtest/switch/L==fred.ls/d_end one/DE/DEFACL

THIS IS THE MACRO - $SWTEST
THE SWITCHES ARE  - /switch/L==fred.ls/d_end
MACRO /D SWITCH   -
LISTFILE FROM /L  - =fred.ls
ARG 1 /DEF SWITCH -
COMMAND LINE WAS  - $swtest/switch/L==fred.ls/d_end one/DE/DEFACL

NOW I'M CALLING $SWTEST.1 PASSING /D AND /L SWITCHES

This is the macro - "$SWTEST.1"
My switches are   - "/L==fred.ls"
Listfile is Arg1  - "one/DE/DEFACL"
/L & /D switches  - "/L==fred.ls"
other switches    - ""

CALLING $swtest.1 PASSING ALL BUT /L SWITCH LISTFILE IS ARG 1

This is the macro - "$swtest.1"
My switches are   - "/switch/d_end"
Listfile is Arg1  - "=fred.ls"
/L & /D switches  - ""
other switches    - "/switch/d_end"

SAME AGAIN BUT ALLOWING FOR NON-EXISTANT /L SWITCH

This is the macro - "$swtest.1"
My switches are   - "/switch/d_end"
Listfile is Arg1  - "=fred.ls"
/L & /D switches  - ""
other switches    - "/switch/d_end"
```

The results shown here are fairly obvious, but we would like to point out if you hadn't already realised, that switches cannot be abbreviated in macros in the same way as they can in many CLI commands. Thus neither **/DE** nor **/DEFACL** are recognized as the **/DEF** switch of argument one, and **/d_end** is not the **/D** switch.



CHAPTER 5

Switch Evaluation and Conditional Logic



CHAPTER 5

Switch Evaluation and Conditional Logic

5.1 Switch Evaluation

It is possible to write fairly useful macros without the need for conditional logic, using arguments, but when we write macros using switches, whose purpose is usually to slightly modify a macro's functionality, we must have some means of evaluating switches in order to modify what we do. We have seen in the previous chapter the various expressions for isolating individual switches and their values, but without a method of testing whether or not these switches are present or what their values are, we can't do a great deal with them. Fortunately CLI does provide us with such a mechanism through its conditional logic Pseudo Macros.

5.2 Conditional Logic Pseudo Macros

Q. **What is a Pseudo Macro?**

A. A Macro that pretends to be a macro, perhaps ?? Well you can put them in macros, and they can have switches and/or arguments like macros, but they aren't macros so we call them Pseudo Macros.

Q. **So what are they really?**

A. Well they can be different things in different circumstances depending on what type of Pseudo Macro they are!

Q. **OK! Cut the BULL. What is it in the context of conditional logic?**

A. Well that's a bit easier to explain. It is a syntactical expression or group of such expressions recognized by CLI as a directive to tell it to conditionally interpret certain commands, macros and/or expressions.

Forgetting about the name Pseudo Macro and why it is thus called let's just look at the syntactic definition of a Pseudo Macro, ie. what CLI needs in order to recognize one of its Pseudo Macros:

The expression: **[!Pseudomacro**{/switches}{,arguments}**]**

is what CLI recognizes as a Pseudo Macro, provided that **Pseudomacro** is one of its recognized Pseudo Macros. Note we have used and will continue to do so, the curly brackets **{** and **}** to indicate that what is enclosed by them is optional. You can get a complete alphabetical list of all Pseudo Macros with the command:

```
) HELP *PSEUDO
```

while

```
) HELPV !Pseudomacro
```

will give you complete details of a particular Pseudo Macro.

For example the expression **[!END]** is the !END (Read as Bang-END) Pseudo Macro, while the expression **[!EQUAL,string1,string2]** is the !EQUAL Pseudo Macro. The !END Pseudo Macro has no switches or arguments, while the !EQUAL Pseudo Macro has two (and precisely two) arguments. If there are any more or any less arguments you will receive the error message: **Error: Pseudomacro has wrong number of arguments.**

These two Pseudo Macros, (**[!EQUAL,string1,string2]** and **[!END]**) can be used to embrace another expression such that that expression is only interpretted by CLI when **string1** and **string2** are equal. We have used the term "expression" here to be as general as we can, because they can embrace anything from a single character on the one line, to a whole series of commands and macro calls over many lines.

Clearly if the two character strings being compared are fixed values they will either always be equal or always be not equal, in which case there would seem little point to the conditional Pseudo Macro. However DG often distribute demo versions of certain macros, where the entire macro body is embraced in such an expression, to render the macro inoperable until modified to suit your particular environment. Furthermore back in pre-COMMENT command AOS days, some of us used the expression:

[!EQUAL,,COMMENTS]

> As a sneaky method of including some comments right here at the beginning of our macro to explain its purpose and any tricky coding etc.
>
> When we had finished with our comments we would enter a Bang-END and nothing embraced by these two pseudo macros would ever be interpreted by CLI making them effectively comments.

[!END]

Notice in this example the use of a pair of commas to indicate a null argument. Since the string containing no characters can never be equal to the string **COMMENTS**, all lines up to the [!END] are ignored by CLI.

Now we know how to do certain things in a macro when a certain condition is true but what if we want to do something when the condition is **NOT** true? For this CLI has the Bang-NEQUAL Pseudo Macro thus **[!NEQUAL,string1,string2]** which must, like all CLI's conditional Pseudo Macros, be matched by a corresponding **[!END]** Pseudo Macro. But what if we want to do something when it's true and something else when it's not? Well for this CLI provides a Bang-ELSE Pseudo Macro thus **[!ELSE]** which must appear somewhere between the conditional Pseudo Macro and the corresponding [!END].

As an example let's look again at our $PSED macro, which requires as an argument the name of a file to edit. We can test for the absence of that argument and therefore we might modify it thus:

File $PSED.CLI

```
[!EQUAL,,%1%]
     WRITE
     WRITE ERROR $PSED requires the file to edit as an argument
     WRITE
[!ELSE]
     PERMANENCE %1% OFF
     X SED/NO_ED %1%
     PERMANENCE %1% ON
[!END]
```

CLI also provides a set of numeric comparison Pseudo Macros where the relational operator is assumed to appear between the first and second arguments thus:

[!UEQ,dec1,dec2]	which reads as	**if dec1 = dec2**
[!UNE,dec1,dec2]	which reads as	**if dec1 NOT = dec2**
[!UGT,dec1,dec2]	which reads as	**if dec1 > dec2**
[!UGE,dec1,dec2]	which reads as	**if dec1 > or = dec2**
[!ULT,dec1,dec2]	which reads as	**if dec1 < dec2**
[!ULE,dec1,dec2]	which reads as	**if dec1 < or = dec2**

where

dec1 and **dec2** must be (if you take CLI's error message as a guide) decimal numbers. Strictly speaking they must be non-negative integers, otherwise you will get the error message:

Error: Illegal decimal number, dec1

This can in fact lead to the somewhat obscure error message Error: Illegal decimal number, -1.

These conditional logic Pseudo Macros are often used in conjunction with CLI's numeric variables, whose values can be set using the **VARn** command (n - 0 to 9) and retrieved using the **[!VARn]** Pseudo Macro. These numeric variables can also only have non-negative integer values, and attempts to set them to negative values will result in similar error messages. At least CLI is consistent.

There is an interesting difference between the evaluation by CLI of the following conditional Pseudo Macros:

[!EQUAL,01,%1%]

and

[!UEQ,01,%1%]

when argument 1 is the string of characters **01** they both evaluate as **TRUE**, but if argument 1 were just the single character **1**, the string comparison of !EQUAL would evaluate to **FALSE**, while the numeric value comparison of !UEQ would evaluate to **TRUE**.

You will usually see Conditional Logic Pseudo Macros and their corresponding [!ELSE]'s and [!END]'s on lines by themselves in CLI macros with the conditional commands indented between them but this is not necessary and is usually done for clarity and legibility. For example our $PSED macro could have been equivalently, but much less sensibly coded as:

```
[!EQUAL,,%1%]WRITE
WRITE ERROR $PSED requires the file to edit as an argument
WRITE[!ELSE]PERMANENCE %1% OFF
X SED/NO_ED %1%
PERMANENCE %1% ON[!END]
```

Unfortunately CLI does not have **[!AND]**, **[!OR]** or **[!NOT]** pseudo macros for some of the more complex compound logic expressions that from time to time we would like to be able to use. But in Chapter 13 Complex Logic, we will discuss methods of overcoming this deficiency.

5.3 How to evaluate Switches

When a switch is not present on an argument the expression **%n/switch%** expands to the null string, thus to detect its absence we simply compare the expression against the null string for equality, thus:

```
[!EQUAL,,%0/V%]
     WRITE THE /V SWITCH WAS ABSENT
[!END]
```

To test for its presence thus:

```
[!EQUAL,/V,%0/V%]
     WRITE THE /V SWITCH WAS PRESENT
[!END]
```

Or which is generally the author's preference

```
[!NEQUAL,,%0/V%]
     WRITE THE /V SWITCH WAS PRESENT
[!END]
```

The reason for the preference apart from the reduced typing for long switches, is that this latter test evaluates true whether or not the switch has a value, whereas the former test would only evaluate true when the switch does not have a value. For example /V might be a verbosity switch which can have a value to indicate the level of verbosity. If the macro were called with the switch **/V=2**, the former test would compare the two strings /V and /V=2 and determine that they were not equal, and therefore **not** execute the WRITE command, while the latter would compare the null string with /V=2 and determine that they were not equal, and therefore **execute** the WRITE command.

We might have a macro that supports a **/COPIES=n** switch with a default of one copy when the switch is omitted. Let's say we wish to hold the number of copies in numeric variable 0. Thus we might write:

```
[!EQUAL,,%0/COPIES%]
     VAR0 1
[!ELSE]
     VAR0 %0/COPIES=%
[!END]
```

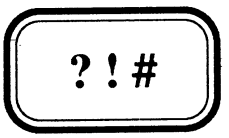
Or equivalently and more efficiently:

```
VAR0 [!EQUAL,,%0/COPIES%]1[!ELSE]%0/COPIES=%[!END]
```

This could result in an unexpected result if the user called our macro with the **/COPIES** switch without a value. It would result in the command VAR0 without an argument being executed which would display the value of variable 0. We could guard against this with the following subtle change:

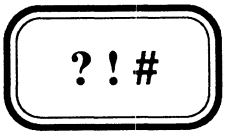
```
VAR0 [!EQUAL,,%0/COPIES=%]1[!ELSE]%0/COPIES=%[!END]
```

We do not intend to give any further examples of switch evaluation at this point, as the technique will be used over and over again, throughout the manual and will eventually become second nature to you.



CHAPTER 6

Loops and Recursion



CHAPTER 6

Loops and Recursion

6.1 Iteration

All programming languages have a means of iteration. Assembler programmers from the early days of computing, along with some pre-F77 Fortran programmers, and unfortunately even some present day COBOL programmers will be familiar with the method of using a **GO TO** statement to jump out of a loop when some condition is satisfied, and another **GO TO** statement at the bottom of the loop to jump back to the top. Concepts like structured programming have largely seen such structures disappear from most programmers' code, or in some cases even from the programming language. CLI does not have a **GO TO** command, or even a facility to **Label** a point in the code (which of course would only be necessary if we had a **GO TO** statement), and is thus such a language.

Wise though this may be, somewhat less wisely CLI does not provide any natural means of iteration at all. There is no **DO** statement, **FOR** loop, **REPEAT .. UNTIL** or **PERFORM** statement.

One simple way of performing a task several times is to place the code you want performed several times in a subordinate macro, and simply invoke that macro the required number of times. This would get rather tedious if we wanted to perform the macro 100 times, or if we wanted to perform it until some condition became true, in which case we would also have to repeat the code to test the condition once for each time we wished to perform the code.

For a limited set of circumstances this mechanism can often be enough. For example if you wanted to allow a typically one argument macro to take multiple arguments, you could use CLI's round bracket syntax to generate one invocation of a subordinate macro for each argument in the list. For example a multiple argument version of our $PSED macro might look like this:

File $PSED.CLI

```
[!EQUAL,,%1%]
     WRITE
     WRITE ERROR $PSED requires the file to edit as an argument
     WRITE
[!ELSE]
     $PSED.1 (%-%)
[!END]
```

File $PSED.1.CLI

```
PERMANENCE %1% OFF
X SED/NO_ED %1%
PERMANENCE %1% ON
```

CLI'S round bracket syntax generates one command for each argument enclosed in the brackets, and thus gives us a primitive means of iteration. This concept will be discussed more fully in Chapter 9 Understanding Brackets and Templates.

Fortunately there is a better way and it relies on the fact that CLI macros are recursive. That is CLI macros can call themselves.

6.2 Recursion

The only other useful mechanism available to us in CLI macros, for repetition is the recursive call, that is a macro calling itself. The most frequent way recursion is used is to process a list of arguments dropping off the most recently processed one until there are none left thus:

```
[!EQUAL,,%1%]
     WRITE Done!
[!ELSE]
     WRITE Processing %1%
          .
          .
     %0%,%2-%
[!END]
```

We could use this technique to re-write our $PSED macro:

```
[!NEQUAL,,%1%]
     PERMANENCE %1% OFF
     X SED/NO_ED %1%
     PERMANENCE %1% ON
     %0%,%2-%
[!END]
```

Hopefully the expression **%0%,%2-%** is obvious and requires no further explanation but just in case you're confused if we called the Macro thus **$PSED FRED.CLI MYMACRO.CLI ABC** the following sequence would take place:

a) FRED.CLI would be edited then **$PSED,MYMACRO.CLI,ABC**
b) MYMACRO.CLI would be edited then **$PSED,ABC**
c) ABC would be edited then **$PSED,**
d) Argument 1 is null so macro stops

The problem with this version of $PSED is that we have now lost the ability to tell the difference between when the macro was called without any arguments and the case when we have simply hit the end of our argument list. One could overcome this limitation by placing a switch on the recursive call so that the two situations can be distinguished, giving us the following:

```
[!EQUAL,,%1%%0/RECURSIVE%]
     WRITE
     WRITE ERROR $PSED requires the file to edit as an argument
     WRITE
[!ELSE]
     [!NEQUAL,,%1%]
          PERMANENCE %1% OFF
          X SED/NO_ED %1%
          PERMANENCE %1% ON
          %0\%%0\RECURSIVE%/RECURSIVE,%2-%
     [!END]
[!END]
```

This is getting unnecessarily obscure (although at times when writing CLI macros we become necessarily obscure), especially when there is a perfectly good iterative solution using the round bracket syntax.

Readers should not get so carried away with recursion that they forget the simplest means of iteration. This is true in any programming language. Obscure though this macro is (and the author certainly prefers the former version using round brackets rather than recursion), it does serve to illustrate a couple of interesting points.

Firstly the conditional Pseudo Macro

[!EQUAL,,%1%%0/RECURSIVE%]

will evaluate to TRUE whenever argument one is null **AND** the /RECURSIVE switch is missing. By concatenating the two expressions **%1%** and **%0/RECURSIVE%** we have effectively built a complex logic expression using **AND**, a feature which we have previously stated CLI does not support. More of this in Chapter 13 but we should point out that the technique only works because we were testing for them both to be a particular value, in this case null.

Secondly the expression

%0\%%0\RECURSIVE%/RECURSIVE

may need some explanation. Effectively it is

$PSED/anyswitches$PSEDhadexceptRECURSIVE/RECURSIVE

The somewhat simpler expression **%0%/RECURSIVE** would result in the /RECURSIVE switch being added once for each recursive call, so that after three calls it would become **$PSED/RECURSIVE/RECURSIVE/RECURSIVE.** Unfortunately when a switch appears multiple times the expression %0/RECURSIVE% expands to include all occurrences, and the conditional Pseudo Macro to test for the switch [!EQUAL,/RECURSIVE,%0/RECURSIVE%] would evaluate FALSE, since /RECURSIVE/RECURSIVE/RECURSIVE is not the same character string as /RECURSIVE. The conditional Pseudo Macro [!NEQUAL,,%0/RECURSIVE%] would however still evaluate as desired and adds further weight to the author's preference for this method of evaluating switches. Because our particular example is impervious to this difference the expression %0%/RECURSIVE would have sufficed in this case. The expression **%0\%/RECURSIVE** could also have been used with equal effect but only because $PSED does not support any switches of its own. If $PSED did support its own switches then these would have to be passed on in the recursive call, and this expression explicitly drops them.

One final point of interest about recursive macros that drop off one argument at a time until there are none, is that invariably we drop the first (or at least some fixed argument until it disappears). This is because we do not know how many there are and CLI does not provide any syntactic way of referring to the last or second last argument.

Now let's look at how we might use recursion to perform a task a specified number of times. We have already explained that CLI has a set of numeric comparison Pseudo Macros, and that we have up to 10 numeric variables available to us to hold such values, so all we need is some means of adding or subtracting from a numeric variable in order to use it as a loop counter. CLI provides a set of arithmetic Pseudo Macros for such a purpose.

6.3 Arithmetic Pseudo Macros and Loop Counting

The complete set of arithmetic Pseudo Macros is:

[!UADD,dec1,dec2]	giving dec1 + dec2
[!USUBTRACT,dec1,dec2]	giving dec1 - dec2
[!UDIVIDE,dec1,dec2]	giving dec1 / dec2
[!UMULTIPLY,dec1,dec2]	giving dec1 X dec2
[!UMODULO,dec1,dec2]	giving dec1 mod dec2 ie remainder on division of dec1 by dec2

In each case **dec1** and **dec2** are as before non negative integers. Strictly speaking they are integers in the range 0 - 4294967295, which for you mathematically inclined ones is 2 ^ 32 - 1 (since CLI's numeric variables are held as two's complement integers in a 32 Bit Word). The result (provided you supplied valid arguments and the result was computable eg. you didn't try to divide by zero) is expanded in place of the Pseudo Macro itself. In other words the Pseudo Macro itself represents the result.

Integer arithmetic is performed for example **[!UDIVIDE 7 2]** will return the result 3. If you wanted the remainder you could use [!UMODULO 7 2]. Although !UDIVIDE and !UMODULO check for attempts to divide by zero and give an error on such cases, !USUBTRACT does not check for negative results. The point here is that CLI will still do the appropriate two's complement arithmetic but since it doesn't recognize negative integers it will return a garbage result. Well for those mathematically inclined it's not really garbage it's what you would expect eg.

```
) WRITE [!USUBTRACT 0 1]
4294967295
```

In two's complement arithmetic -1 is represented by the word with all bits set to 1, but since CLI ignores the sign bit it displays the result as the positive integer represented by 31 bits set to 1, namely 2 ^ 32 - 1. For you less mathematically inclined readers who are, if not bored by such trivia, at least now completely confused (and are wondering if we'll ever get back to talking about performing a loop a specified number of times), there is a point to the diversion even for you. The point is **[!USUBTRACT,dec1,dec2]** where **dec2 is greater than dec1** will return a **positive** result undoubtedly very much greater than either dec1 or dec2. This could result in unexpected results when subsequently using a numeric comparison Pseudo Macro.

Now getting back to the main stream, we can force a task to be performed a specified number of times, by setting a numeric variable to that number, then use !USUBTRACT to count it down to zero thus:

File $COUNTDOWN.CLI

```
[!UEQ,0,[!VAR0]]
     WRITE BLASTOFF!
[!ELSE]
     VAR0
     PAUSE 1
     VAR0 [!USUBTRACT [!VAR0] 1]
     %0%
[!END]
```

Thus if we issued the following commands:

```
) VAR0 3
) $COUNTDOWN
```

the output would be:

```
3
2
1
BLASTOFF!
```

Our countdown macro would be nicer if we could simply pass it the number we wanted to count down from as an argument. Furthermore the numeric variable is redundant and can simply be eliminated giving us:

File $COUNTDOWN.CLI

```
[!UEQ,0,%1%]
     WRITE BLASTOFF!
[!ELSE]
     WRITE %1%
     PAUSE 1
     %0% [!USUBTRACT %1% 1]
[!END]
```

For which the call **$COUNTDOWN 3** would give the same result as above. To emphasize a point made previously let's say we wanted $COUNTDOWN to count down by two, we could simply change the recursive call line to **%0% [!USUBTRACT %1% 2]**. You would probably also change the PAUSE command to PAUSE 2 seconds assuming this was the purpose of the macro. The call $COUNTDOWN 6 would give the output:

```
6
4
2
BLASTOFF!
```

But the call **$COUNTDOWN 3** would give the output:

```
3
1
4294967295
4294967293
4294967291

etc.
```

until we hit 1 again and then around we go again in an infinite loop. We can of course fix it, and even generalize it a little to allow an optional switch to specify what we want to count down by. Let's call it the **/BY=n** switch. The important point to note here about recursive macros is to be certain your escape clause is absolutely watertight, to prevent infinite loops.

File $COUNTDOWN.CLI

```
[!EQUAL,,%0/BY=%]
     %0%/BY=1 %1%
[!ELSE]
     [!ULT,%1%,%0/BY=%]
          [!UNE,0,%1%]
               WRITE %1%
               PAUSE %1%
          [!END]
          WRITE BLASTOFF!
     [!ELSE]
          WRITE %1%
          PAUSE %0/BY=%
          %0% [!USUBTRACT %1% %0/BY=%]
     [!END]
[!END]
```

Thus

```
     ) $COUNTDOWN/BY=2 3
     3
     1
     BLASTOFF!

     ) $COUNTDOWN/BY=6 15
     15
     9
     3
     BLASTOFF!
```

You will note that we had to write some special code to handle the situation to WRITE out the last number less than what we were counting down by, and to PAUSE for that number of seconds, before BLASTOFF. Effectively we had to repeat the entire body of the macro. The WRITE and PAUSE commands are really the main body, the rest is just telling CLI how and when to perform it. If the main body of the code were many lines this would be somewhat cumbersome and it would be better if we could write it in such a way that the main body only appeared once. We can. We will leave it as an exercise for you to work out how and why this macro does what it does.

File $COUNTDOWN.CLI

```
[!EQUAL,,%0/BY=%]
     %0%/BY=1 %1%
[!ELSE]
     [!UEQ,0,%1%]
          WRITE BLASTOFF!
     [!ELSE]
          [!ULT,%1%,%0/BY=%]
               %0\%/BY=%1% %1%
          [!ELSE]
               WRITE %1%
               PAUSE %0/BY=%
               %0% [!USUBTRACT %1% %0/BY=%]
          [!END]
     [!END]
[!END]
```

6.4 Recursion and CLI out of memory errors

Recursive macros have been known from time to time to give unexpected "**Not enough memory, restarting CLI**" errors. When CLI runs out of memory it starts a new CLI using your current Searchlist and Directory. If you are in the habit of using the CLI command **DIR/I** (DIRECTORY/I) to return to your original Working Directory and you were not in that Directory when the "CLI out of memory error" occurred, you'll have a problem because your new CLI process has a different Initial Working Directory. More serious than this though is that macros written for Production Use which through the Initial IPC Macro (carefully written by your Systems Programmer) have been specially designed to prevent the User getting CLI access, will all of a sudden give CLI access to the User because CLI was invoked without the Initial IPC macro.

Consider the following useless macro, which does nothing more than count the number of recursive calls it makes, until it has made 10000 such calls in which case it writes out THAT'LL DO. To give you some idea of its progress (ie what call it is up to), for every call up to 10, then every tenth call up to 100, then every 100th call up to 1000 and finally every 1000th call until we get to 10000 (in which case it stops), it displays the current call number.

File $OUTOFMEMORY1.CLI

```
[!EQUAL,,%1%]
     %0% 1
[!ELSE]
     [!UGT,%1%,9999]
          WRITE THAT'LL DO
     [!ELSE]
          [!UGT,%1%,999]
               [!UEQ,0,[!UMODULO %1% 1000]]
                    WRITE %1%
               [!END]
          [!ELSE]
               [!UGT,%1%,99]
                    [!UEQ,0,[!UMODULO %1% 100]]
                         WRITE %1%
                    [!END]
               [!ELSE]
                    [!UGT,%1%,9]
                         [!UEQ,0,[!UMODULO %1% 10]]
                              WRITE %1%
                         [!END]
                    [!ELSE]
                         WRITE %1%
                    [!END]
               [!END]
          [!END]
          %0% [!UADD %1% 1]
          WRITE
     [!END]
[!END]
```

As you might have expected from the name of the Macro it never actually gets to 10000 calls but crashes out with a "Not enough memory, restarting CLI" message. It's output is shown on the following page:

```
) $OUTOFMEMORY1
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
1000
```

And somewhere between the 1000th and 2000th recursive call it crashes with the "**Not enough memory, restarting CLI**" error. Why? Well recursion is only possible because of things called STACKS, in which for each recursive call information about the previous environment is stacked and popped back on return. In addition any code that needs to be executed on return from the recursive call must also be stacked (for the programmers this would not be necessary with pure re-entrant code but we are talking about INTERPRETED code, which cannot by its nature be re-entrant), and this STACK is built in memory which is a limited resource. When the STACK gets full CLI runs out of memory and BOOM!

What can we do about it? Well one thing we can do is minimize the amount of information that needs to be saved on the stack. One way of doing this is to make sure that there is no executable code placed after a recursive call. Take for example the innocuous looking WRITE command after the recursive call in $OUTOFMEMORY1. This command never actually got executed because we never returned, but if we had it would have written out 10000 blank lines. Consider a second macro called **$OUTOFMEMORY2**, identical in every way to $OUTOFMEMORY1 except that this WRITE command is removed. When this was tested it didn't run out of memory until somewhere between the 3000th and 4000th call.

Extending this concept of removing Code after a recursive call, to include the removal of non-executable code, ie. the unnecessary white space (which of course serves the useful purpose of clearly showing the structure of our macro) we present the following equivalent to **$OUTOFMEMORY2.**

File $NOTOUTOFMEMORY.CLI

```
[!EQUAL,,%1%]
     %0% 1
[!ELSE]
     [!UGT,%1%,9999]
          WRITE THAT'LL DO
     [!ELSE]
          [!UGT,%1%,999]
               [!UEQ,0,[!UMODULO %1% 1000]]
                    WRITE %1%
               [!END]
          [!ELSE]
               [!UGT,%1%,99]
                    [!UEQ,0,[!UMODULO %1% 100]]
                         WRITE %1%
                    [!END]
               [!ELSE]
                    [!UGT,%1%,9]
                         [!UEQ,0,[!UMODULO %1% 10]]
                              WRITE %1%
                         [!END]
                    [!ELSE]
                         WRITE %1%
                    [!END]
               [!END]
          [!END]
%0% [!UADD %1% 1][!END][!END]
```

As the name implies this macro did manage to complete satisfactorily, carrying out 10000 recursive calls before writing out the message THAT'LL DO. Notice that we did not modify the code before the recursive call in any way.

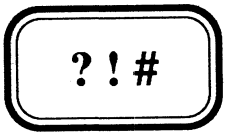
As we are not privy to AOS/VS sources we cannot be precisely sure what CLI puts on the STACK for a recursive call or how much memory it takes, but we are confident that with the following simple precautions:

a) **Never place executable code after a recursive call in a CLI macro.**

b) **Never place a CLI PUSH command (and its corresponding POP) inside a recursive Macro.**

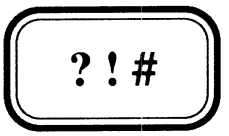
you will almost certainly eliminate such errors. We will return to this subject when we discuss the PUSH and POP commands in the next Chapter. There are however other causes of CLI running out of memory and they will be discussed in the appropriate Chapters.

THIS PAGE INTENTIONALLY LEFT BLANK



CHAPTER 7

PUSH, POP and Environmental Parameters



CHAPTER 7

PUSH, POP and Environmental Parameters

7.1 Environmental Parameters

The CLI has a number of what we call environmental parameters. They are so called because they affect the environment in which CLI works, and therefore the way CLI reacts to your various commands. You can get a complete list of CLI's environmental parameters and their **current** values with the CLI command **CURRENT**. These are:

LEVEL
an integer which is initially zero and increased by one every time you issue the CLI command PUSH, and decreased by one every time you issue the CLI commnad POP. (More about this later)

SUPERUSER, SUPERPROCESS, SCREENEDIT and **SQUEEZE**
which are either ON or OFF.

CLASS1 and **CLASS2**
each of which is either IGNORE, WARNING, ERROR or ABORT

TRACE
which is either not set or any combination of /COMMAND, /MACRO and/or /PSEUDO

The numeric **VARIABLES**, the **STRING**, **PROMPT** and terminal **CHARACTERISTICS**
and their current values

LISTFILE, DATAFILE and **LOGFILE**
and their current settings

and perhaps most importantly as far as the degree to which they affect the environment is concerned:

DEFACL
Your Default Access Control List, being the Access Control List assigned to any file created by you or any job you run during the current session. Access Control Lists (ACLs) are fully explained in Chapter 17, but for now the ACL of a file determines who has access to the file, and what level of access they have.

DIRECTORY
Your current Working Directory, generally referred to as the directory you are "in". It is the directory in which CLI searches for files you wish to access, or where it creates files, if you reference the file by its filename only. This could be circumvented by specifying a specific PATHNAME to the file.

SEARCHLIST
This is a list of up to 8 directories in which you would like CLI to search for files you wish to access, if CLI cannot find the file in the Working Directory. You can prevent CLI using the searchlist by specifying an explicit PATHNAME to the file.

CLI supports commands, usually with the same name as the parameter to display or change the current value of each of these parameters. In some cases CLI also supports Pseudo Macros to expand to the current value of the relevant parameters. These Pseudo Macros are [!LEVEL], [!LISTFILE], [!DATAFILE], [!DIRECTORY], [!SEARCHLIST], [!DEFACL], [!STRING] and the numeric variable Pseudo Macros [!VARn].

7.2 PUSH and POP

PUSH Pushes all the CURRENT environmental parameters onto the stack, and increments the current LEVEL number. That is it saves them on the stack.

POP Pops all the PREVIOUS environmental parameters off the top of the stack and makes them CURRENT. Logically it would also decrement the current LEVEL number, but we assume it does not bother, since the LEVEL number is itself an environmental parameter and the act of POPping the stack will automatically reset it to its previous value.

Once you have issued a PUSH command you can change any of the environmental parameters (or all of them if you like), knowing that you can always restore your previous environment with a POP. If you have PUSHed then the CLI command **PREVIOUS** will display all the environmental parameters of the previous environment. Some of the CLI commands for displaying and changing the environmental parameters, and all of the Pseudo Macros, support a **/P** switch to refer to the value of the previous environment rather than that of the current environment. You cannot however change the value of the previous environment's parameters.

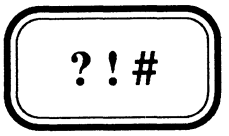
Although the ability to restore the previous environment with a single command is a nice feature, it is really only a convenience, because you could always issue a counter set of commands to restore them to their original values, albeit with a little more difficulty. If this were the only reason for PUSH and POP we doubt very much whether they would have bothered with going to the trouble of implementing such commands.

One of the most common uses of the CLI PUSH command is purely to make available a second CLI STRING parameter. In our view this is really a **misuse** of the PUSH command, and the reason for our view should become clear when you read Chapter 14 Nesting and STRING-less Macros.

The main reason for **PUSH** and **POP**, is to enable you to write general purpose macros, which require certain changes to the environmental parameters in order to function correctly (such as adding a special directory to the SEARCHLIST), but once having completed their task can restore the user's original environment.

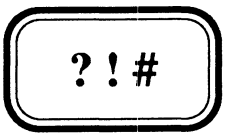
In case you think we're splitting straws here, and you can't see what's so special about general purpose macros, we'll put it another way. The point is that with a general purpose macro, that is one that can be used by anybody at any time, **you don't know what the original values for the environmental parameters were.** But with PUSH and POP it doesn't matter what they were. You simply PUSH first, do your stuff, then POP when you've finished. We will discuss the techniques for doing this further in Chapter 22 Preserving the Environment (PROMPT POP).

PUSH and POP are slow commands and impinge heavily on CLI's internal stack, because so much information has to be saved. **For this reason you should NEVER code them inside a recursive macro.** The correct way to write an environment preserving recursive macro is to PUSH first, then call a subordinate macro in which you've placed your recursive code (and which doesn't itself have a PUSH and POP), and then POP.



CHAPTER 8

Pseudo Macros



CHAPTER 8

Pseudo Macros

8.1 Overview

We first introduced the concept of Pseudo Macros in Chapter 5 with the following dialogue:

Q. **What is a Pseudo Macro?**

A. A Macro that pretends to be a macro, perhaps ?? Well you can put them in macros, and they can have switches and/or arguments like macros, but they aren't macros so we call them Pseudo Macros.

Q. **So what are they really?**

A. Well they can be different things in different circumstances depending on what type of Pseudo Macro they are!

Then we went on to define them in terms of their syntactical definition thus:

Forgetting about the name Pseudo Macro and why it is thus called let's just look at the syntactic definition of a Pseudo Macro, ie. what CLI needs in order to recognize one of its Pseudo Macros:

The expression: **[!Pseudomacro**{/switches}{,arguments}**]**

is what CLI recognizes as a Pseudo Macro, provided that **Pseudomacro** is one of its recognized Pseudo Macros. Note we have used and will continue to do so, the curly brackets **{** and **}** to indicate that what is enclosed by them is optional. You can get a complete alphabetical list of all Pseudo Macros with the command:

```
) HELP *PSEUDO
```

while

```
) HELPV !Pseudomacro
```

will give you complete details of a particular Pseudo Macro.

For example the expression **[!END]** is the !END (Read as Bang-END) Pseudo Macro, while the expression **[!EQUAL,string1,string2]** is the !EQUAL Pseudo Macro. The !END Pseudo Macro has no switches or arguments, while the !EQUAL Pseudo Macro has two (and precisely two) arguments. If there are any more or any less arguments you will receive the error message:

Error: Pseudomacro has wrong number of arguments.

It is our intention in the remainder of this Chapter to touch briefly on most of CLI's Pseudo Macros, and refer you to other sections of the manual if they are treated more thoroughly elsewhere. Where relevant we will group these Pseudo Macros together and for completeness we will begin with those we have already discussed, and with which you are by now familiar.

8.2 Conditional Logic Pseudo Macros

The conditional logic Pseudo Macros have been fully discussed in Chapter 5, they are:

[!EQUAL,string1,string2] meaning **if string1 = string2**
[!NEQUAL,string1,string2] meaning **if string1 NOT = string2**

where

string1 and **string2** are any character strings, including the null string, not including an argument delimiter.

and

[!UEQ,dec1,dec2]	which reads as	**if dec1 = dec2**
[!UNE,dec1,dec2]	which reads as	**if dec1 NOT = dec2**
[!UGT,dec1,dec2]	which reads as	**if dec1 > dec2**
[!UGE,dec1,dec2]	which reads as	**if dec1 > or = dec2**
[!ULT,dec1,dec2]	which reads as	**if dec1 < dec2**
[!ULE,dec1,dec2]	which reads as	**if dec1 < or = dec2**

where

dec1 and **dec2** must be (if you take CLI's error message as a guide) decimal numbers. Strictly speaking they must be non-negative integers, otherwise you will get the error message:

Error: Illegal decimal number ..

and finally:

[!END] one of which is required to delimit the END of the code under control of one of the above Pseudo Macros.

[!ELSE] one of which may occur between a Conditonal Pseudo Macro and its corresponding !END to delimit true and false branches in the code.

8.3 Environmental Parameter Pseudo Macros

These Pseudo Macros were discussed in Chapter 7. They return and therefore represent the following:

[!LEVEL]	The current environmental Level PUSHed to
[!LISTFILE]	The current Listfile setting
[!DATAFILE]	The current Datafile setting
[!DIRECTORY]	The current Working Directory
[!SEARCHLIST]	The current Searchlist
[!DEFACL]	The current Default Access Control List
[!STRING]	The current contents of the (upto 128 char) string
[!VARn]	The current value of numeric variable n (n=0-9)

All these Pseudo Macros take a **/P** switch to return the corresponding value in the previous environment, while **[!DIRECTORY]** also takes a **/I** switch to return your process's Initial Working Directory, which is usually **:UDD:username**.

8.4 Arithmetic Pseudo Macros

The arithmetic Pseudo Macros were discussed in Chapter 6 they are:

[!UADD,dec1,dec2]	giving dec1 + dec2
[!USUBTRACT,dec1,dec2]	giving dec1 - dec2
[!UDIVIDE,dec1,dec2]	giving dec1 / dec2
[!UMULTIPLY,dec1,dec2]	giving dec1 X dec2
[!UMODULO,dec1,dec2]	giving dec1 mod dec2 ie remainder on division of dec1 by dec2

In each case **dec1** and **dec2** are non negative integers. Strictly speaking they are integers in the range 0 - 4294967295 (2 ^ 32 - 1). The result (provided you supplied valid arguments and the result was computable eg. you didn't try to divide by zero) is expanded in place of the Pseudo Macro itself. In other words the Pseudo Macro itself represents the result.

8.5 File Information Pseudo Macros

The following Pseudo Macros return and therefore represent certain information about or relating to files:

[!ACL pathname]	returns the Access Control List of the specified file
[!SIZE pathname]	returns the Size being the number of bytes occupied by the specified file
[!FILENAMES template{s}]	returns a list of files matching the specified template or templates
[!PATHNAME pathname]	returns the full Pathname to the specified file

ACL's are discussed fully in Chapter 17, and Chapter 18 is devoted to a thorough explanation of the !FILENAMES and !PATHNAME Pseudo Macros.

8.6 System Information Pseudo Macros

The following Pseudo Macros return and therefore represent certain information about your System:

[!DATE] returns the current System Date in the format **DD-MON-YY**, where MON is the first 3 characters of the month name.

[!TIME] returns the current System Time in the format **HH:MM:SS**.

[!LOGON] returns the string **BATCH** if you are running under an EXEC Batch stream, or the string **CONSOLE** if you are running at a console logged via EXEC, or the null string if the process is not running under EXEC at all.

[!OPERATOR] returns either **ON** or **OFF** depending on whether or not the Operator has given the appropriate EXEC command, to tell the System that he is on or off duty. Can be used to determine whether or not to issue a tape MOUNT request.

[!HID{ hostname}] returns your system's **Host-ID** or if a hostname in your Network is supplied its Host-ID.

[!HOST{ hostid}] returns your system's **Hostname** or if a Host-ID in your Network is supplied its Hostname.

[!PID] returns the **Process-ID** which uniquely identifies this process in your System (often used to make temporary filenames unique).

[!PIDS{ hid-or-host}] returns a list of all **Process-IDs** on your system, or the specified system if a Host-ID or Hostname in your Network is specified.

[!SONS{ procid}] returns a list of Process-IDs (PIDs) of those processes which are direct offspring of the specified process. The list does not include grandchildren. Without an argument it means this process's sons. Procid can be anything that uniquely identifies the process, thus PID, username:processname (which is unique) or processname (where Username OP is assumed), or even hostname:PID.

[!CONSOLE] returns the processname, which is usually the string **CONnn** (but may be BATCH_INPUT_n), of this process, or if this process is not a son of EXEC but a descendent of it, then the processname of the ancestral immediate son of EXEC (a mouthful but more accurate than HELPV)

[!SYSTEM] returns **AOS/VS** or **AOS/DVS** depending on system.

[!USERNAME] returns this process's username.

8.7 String Manipulation Pseudo Macros

The following macros manipulate the character strings supplied as arguments to return a portion of or a variant form of the arguments.

[!EXPLODE arg{s}] returns the characters forming the supplied argument string{s}, as single character arguments, that is, each character in the supplied string is separated from the next by a comma in the returned string.

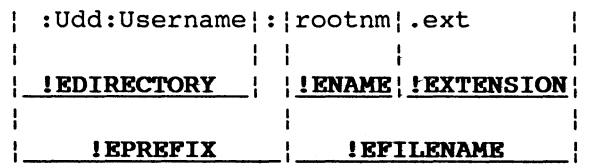
for example assuming it is 8 pm on the 13th December 1988

[!EXPLODE [!DATE]] would return **1,3,-,D,E,C,-,8,8** while

[!EXPLODE [!DATE] [!TIME]] **1,3,-,D,E,C,-,8,8,,,2,0,:,0,0,:,0,0**

Notice that the Comma that CLI always uses to separate arguments is itself surrounded by commas in the expansion of the second example. The above examples are often used to enable a subordinate macro to put a Date or Date-Time stamp into a filename, by recombining the arguments and dropping the non-filename characters. For example in the second case we could use the expression **%8%%9%%4%%5%%6%%1%%2%_%12%%13%%15%%16%** which would return **88DEC13_2000**, in a subordinate macro.

The remaining group of string manipulation Pseudo Macros are designed to return portions of a supplied pathname, and are best described with the following diagram. Assuming they have each been passed the argument **:Udd:Username:rootnm.ext**,



for example

[!ENAME :Udd:Username:rootnm.ext] returns **rootnm**

[!ENAME rootnm.ext1.ext2] returns **rootnm.ext1**

and if I was logged on at an EXEC controlled terminal and on the System Disk the file UDD was a linkfile to :DISK1:UDD,

[!EDIRECTORY [!directory/I]] would return **:DISK1:UDD**

It is important to note that despite the fact that these Pseudo Macros expect a pathname (or filename) as an argument they are purely string manipulators and what they return is dependent only on the presence or absence of Colons (:) or Periods (.) in the string supplied (although other prefixes such as **@** and = are recognized). They will all accept any garbage string of characters and try to return the appropriate result. They will accept multiple arguments and will return exactly the same number of arguments after applying the appropriate string manipulation to each one.

8.8 Data Conversion Pseudo Macros

These Pseudo Macros will convert valid arguments to an alternative form:

[!DECIMAL octno] returns the Decimal equivalent of the non-negative octal integer **octno.**

[!OCTAL decno] returns the Octal equivalent of the non-negative decimal integer **decno.**

[!ASCII octno{s}] returns the character{s} whose Ascii value{s} is{are} represented by the octal number{s} supplied as the argument{s}. This is often used to send certain control characters to the terminal and is discussed further in Chapter 11.

8.9 Input Pseudo Macro

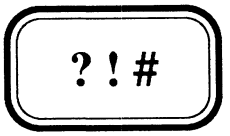
This is the special Pseudo Macro that allows the macro writer to get input from the User. It will be discussed fully in the Chapter 10.

[!READ prompt text] displays the supplied **prompt text** on the terminal, and immediately after the last character thereof waits for the user to respond. Whatever the user types in replaces the entire !READ Pseudo Macro in the body of the Macro.

for example:

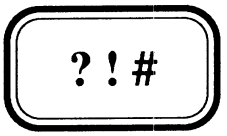
```
) [!READ gimme a rhyme]
gimme a rhymeHumpty Dumpty sat on the wall
Error: Not a command or macro, Humpty,dumpty,sat,on,the,wall
```

would most likely result because CLI tried to interpret the first word of the response as a Command. Most often (although we'll show you in Chapter 14 how it can be much less often) the **!READ** Pseudo Macro is preceded with the CLI **STRING** command in order that the response may be stored in the CLI STRING for later recall with the **[!STRING]** Pseudo Macro.



CHAPTER 9

Understanding Brackets and Templates



CHAPTER 9

Understanding Brackets and Templates

9.1 Angle Brackets

Angle brackets, or more correctly the two characters < and > give **In-line** expansion of expressions to save you the trouble of retyping repeated character strings in a command. This is best explained with examples. Let's say we wanted to use $PSED to edit the three files, OURFILE_1 OURFILE_2 and OURFILE_3. We could do this with the command:

) **$PSED OURFILE_1 OURFILE_2 OURFILE_3**

or to save typing in the repeated character string **OURFILE_** we could use the angle bracket syntax to generate the same command:

) **$PSED OURFILE_<1 2 3>**

These brackets can appear anywhere in a character string depending on where the repeated portion of that string is. For example it is often necessary to CREATE Link files that point to files of the same name in another Directory. This would be necessary after a User's :UDD:username directory were relocated on a different disk to help balance the load across our System. Let's say we have a user on our system whose username is **DISKHOG** and we have just moved and subsequently deleted his :UDD:username directory and everything in it to a subdirectory of some other disk, say **:BIG_DISK:FOR_HOGS**, and that our current Working Directory is **:UDD**. In order to actually have the access privileges to do this we are probably an Operator or System's Programmer with the SUPERUSER privilege, which is almost certainly turned on. Thus to create the appropriate Link file we could:

*) **CREATE/LINK DISKHOG :BIG_DISK:FOR_HOGS:DISKHOG**

or using the angle bracket syntax to save repeating **DISKHOG**

*) **CREATE/LINK <,:BIG_DISK:FOR_HOGS:>DISKHOG**

Notice how we used a Comma to generate a null argument which is first applied to **DISKHOG**, leaving of course **DISKHOG**, and then the second argument inside the angle brackets is applied to **DISKHOG** giving us **:BIG_DISK:FOR_HOGS:DISKHOG.**

If we wanted to achieve the same task while positioned in another directory (other than :UDD), without moving from that directory we could do it like so:

*) **CREATE/LINK :<UDD BIG_DISK:FOR_HOGS>:DISKHOG**

Notice here how the brackets are now in the middle with repeated strings both before (:) and after (**:DISKHOG**). The equivalent command longhand is:

*) **CREATE/LINK :UDD:DISKHOG :BIG_DISK:FOR_HOGS:DISKHOG**

By the way we sincerely hope that at least some of you (preferably most of you) have never seen the CLI prompt you with the character string ***)**. If so then for your interest CLI always prefixes the standard prompt with an asterisk, whenever **SUPERUSER** is turned **ON.**

Multiple pairs of Angle Brackets in the one command line operate **independantly** from argument to argument, but combine **distributively** when part of the one argument. This is best shown with an example:

<1,2,3>*,*<x,y> yields **1*,2*,3*,*x,*y**

but

<1,2,3><x,y>** yields **1**x,1**y,2**x,2**y,3**x,3**y**

Angle brackets may be nested for example:

1<,a<,:+.<CLI PR>,->,*>,b,c<1<,0<,0<,0>>> 2,>

yields

1,1a,1a:+.CLI,1a:+.PR,1a-,1*,b,c1,c10,c100,c1000,c2,c

9.2 Round Brackets

Round brackets, or more correctly the two characters **(** and **)** give **Multiple Command** expansion. Multiple pairs of Round Brackets in the one command line operate **associatively** by corresponding position, whether or not they are part of the one argument, thus repeating the two examples used to demonstrate Angle Brackets:

(1,2,3)*,*(x,y)

yields

1*,*x
2*,*y
3*,*

while

(1,2,3)(x,y)**

yields

1x**
2y**
3**

With multiple sets of brackets the number of commands generated is as many as the number of arguments in the set with the most arguments. Null expressions are substituted for the missing arguments in the sets with less than that number of arguments. To get distributive expansion into multiple commands you can use both kinds of brackets thus:

(<1,2,3><x,y>)**

yields

1x**
1y**
2x**
2y**
3x**
3y**

There are plenty of useful ways to combine sets of Angle and Round brackets to reduce typing, but be warned, sometimes it takes longer to work out the correct syntax to generate what you want, than it would have taken to type it in longhand anyway.

One particularly useful technique comes in handy when you want to rename a whole batch of files with a particular extension. For example let's say we wanted to rename all **.BU** files in our current working directory so that they have a **.BU.SAVE** extension. At first thought one might write:

```
RENAME ([!FILENAMES +.BU])<,.SAVE>
```

Let's say we had three such files **TOM.BU, DICK.BU** and **HARRY.BU**. This would then expand to:

```
RENAME (=TOM.BU,=DICK.BU,=HARRY.BU)<,.SAVE>
```

thus

```
RENAME =TOM.BU<,.SAVE>
RENAME =DICK.BU<,.SAVE>
RENAME =HARRY.BU<,.SAVE>
```

and finally

```
RENAME =TOM.BU,=TOM.BU.SAVE
RENAME =DICK.BU,=DICK.BU.SAVE
RENAME =HARRY.BU,=HARRY.BU.SAVE
```

to some of you this may look alright, but the character = is a special filename prefix meaning the current working directory, thus **=HARRY.BU** is equivalent to **[!DIRECTORY]:HARRY.BU** and is therefore a full pathname to a file. The rename command does not support a pathname as its second argument. Its syntax is **RENAME pathname filename**, and so the above three commands would crash. **!FILENAMES** always expands to lists of pathnames, which it guarantees when supplied a simple template implying the Working Directory, by prefixing each name with =. Now the three commands above would be okay if the = prefix wasn't there so all we have to do is remove it. You might recall from the previous chapter that there is a Pseudo Macro specifically designed to remove prefixes. Thus:

```
RENAME ([!EFILENAME [!FILENAMES +.BU]])<,.SAVE>
```

will do the job. What one is more likely to want to do, however is replace the **.BU** extensions with **.SAVE** extensions, and this we can do by simply using a different Pseudo Macro so that the .BU extension is not included inside the list expanded in the round brackets, thus:

```
RENAME ([!ENAME [!FILENAMES +.BU]]).<BU,SAVE>
```

becomes RENAME (TOM,DICK,HARRY).<BU,SAVE> which expands to:

```
RENAME TOM.BU,TOM.SAVE
RENAME DICK.BU,DICK.SAVE
RENAME HARRY.BU,HARRY.SAVE
```

Round brackets are probably most often used to enable a command or macro which typically only takes one argument to process several as demonstrated in Chapter 6. However one should not lose sight of the fact that the syntax works equally well when you want a number of commands to process the one argument. For example at the top of some processing macro we may well want to open a new listfile which is used to log the events of the job, replacing the previous log if it existed, thus we might:

```
DELETE/2=IGNORE THISJOB.LOG
CREATE THISJOB.LOG
LISTFILE THISJOB.LOG
```

equally well we could write

```
(DELETE/2=IGNORE CREATE LISTFILE) THISJOB.LOG
```

Round brackets also have another completely different purpose. This is that when we want to tell CLI that the expression delimited by the brackets (and including them) is to be treated as a single argument. This technique is most often used when we wish to test a string of characters against the null string. For example let's say that at the end of a macro we want to take some special action if one or more files with the extension **.ACTION** exists. We might write:

```
[!NEQUAL,,[!FILENAMES +.ACTION]]

    WRITE Processing action files
        .
        .

[!END]
```

This would be satisfactory if there were no **.ACTION** files or just one of them, but as soon as there were more than one our macro would fall over with the **Pseudo Macro has wrong number of arguments** error. However we can use round brackets to group the entire list expanded by !FILENAMES and tell CLI to treat it as one argument. Thus:

```
[!NEQUAL,,([!FILENAMES +.ACTION])]
```

This particular test will however always evaluate true even when no files exist, because the single argument formed by the brackets includes the brackets, and the string of characters **()** is not equal to the null string, thus our test must become:

```
[!NEQUAL,(),([!FILENAMES +.ACTION])]
```

If however our test was purely for the existence of the file **MORE.ACTION**, then the test

```
[!NEQUAL,,[!FILENAME MORE.ACTION]]
```

is perfectly satisfactory. Notice also our preference here of dropping the **S** from !FILENAMES when we know no more than one filename can be expanded.

Now there is a potential problem here. How does CLI differentiate between round brackets used to generate multiple commands and round brackets used to group a list of arguments as a single argument?

For example what does:

MYMACRO one,two,(a group of args),four,five

mean?

Well CLI's rule is that Round Brackets that occur inside some other round or square bracket (including those that delimit Pseudo Macros) pair, are argument grouping brackets, otherwise they are multiple command expansion brackets. Thus the above example generates four macro calls with a different third argument. If however we wanted them to be treated as a single grouped third argument in a single invocation of our macro MYMACRO, we would do this:

[MYMACRO one,two,(a group of args),four,five]

You may remember when we talked about Angle Brackets we mentioned that they could be nested and gave an example. When Round Brackets are nested the ones inside have a different meaning to the ones outside. For example if we return to our $PSED macro, which in its original form which was

```
PERMANENCE %1% OFF
X SED/NO_ED %1%
PERMANENCE %1% ON
```

we could, though somewhat obscurely, use nested round brackets to code it in one line thus:

(PERMANENCE (X SED/NO_ED) PERMANENCE) %1% (OFF,,ON)

9.3 Square Brackets

Square brackets also have two uses and both have already been fully discussed. Distinction between the two is obvious:

[!...] Marks off a CLI Pseudo Macro and its arguments.

[name] This is occasionally described in some DG manuals as meaning the contents of the file **name**. However as explained in Chapter 3 this is only true provided that **name.CLI** does not exist in the Working Directory or one of the directories on the Searchlist, for it is actually the most efficient means of invoking a Macro. Note however as demonstrated above that **macro,(arguments...)** and **[macro,(arguments)]** are not equivalent. Invoking macros this way is an essential tool for invoking and writing User Pseudo Macros, which are discussed in Chapter 19.

Before looking at Templates a final word about the other sort of brackets, the curly ones { and }. They have absolutely no meaning whatsoever to CLI, which is why we use them to delimit optionals in describing a macro or command's syntax.

9.4 Templates

AOS/VS recognizes some special non-filename characters as special match characters or wild cards in strings. These strings usually represent filenames but can be used to represent other things, for example usernames in the ACL of a file. Strictly speaking these too are filenames in the sense that all usernames must consist of valid filename characters, since the profiles are held in **:UPD** in a file the same name as the username. The template or wild card characters therefore only match valid filename characters which are **A-Z** (upper or lower case equivalent), **0-9** and the characters **?**, **.**, and **$**.

These special match characters are:

+ represents any string of filename characters including none.

- represents any string of filename characters including none, but not including a period, ie . (full stop).

***** represents any single filename character (but not none), but not including a period.

For example:

+.*- all files with an extension (other than . since strictly speaking a file ending in a full stop has an extension of ., according to !EEXTENSION).

In addition to these special AOS/VS template characters CLI also recognizes two other special characters in returning lists of filenames to you, whether via the FILESTATUS command or the !FILENAMES Pseudo Macro. These are:

can be used in place of an entire filename or filename template to represent a directory (either the Working Directory or that specified by the pathname prefix preceding the #) and the entire directory tree beneath it. This includes all files in the directory and in any subdirectories it has, and in subdirectories they have etc.

**** Introduces a second filename template which specifies that any filenames matching the first template, are not to be expanded if they also match this second template. Thus it specifies an exception to the original template. Multiple exceptions can be made by specifying additional \ characters.

For example:

+\TEST:# All files in the Working Directory except the file TEST, and all subdirectory trees beneath them.

+\+.PR\+.CLI All files in the Working Directory except Programs and Macros.

#\?+ All files in the directory tree except any that begin with ?.

Care must be taken when mixing templates and brackets, that what you get, is what you really want, especially when using the **DELETE** command. For example let's say we wanted to delete all files in the current working directory except the following **PROG<1,2><,.PR>**. One might be tempted to use the following:

```
) DELETE +\PROG<1,2><,.PR>
```

The actual effect of this command would be to delete all files, including **all** those you supposedly excluded. In addition you would receive a series of **no files match template** Warnings. This is because the expression **+\PROG** is the common expression repeated for each argument generated by the Angle Brackets. The command is equivalent to:

```
) DELETE +\PROG1,+\PROG1.PR,+\PROG2,+\PROG2.PR
```

Thus in processing the first template CLI would delete every file in the Working Directory except **PROG1**. In processing the second template **PROG1** is comprehensively deleted (being the only file left in the directory it is thus the only one matching the template +\PROG1.PR). The remaining templates give rise to no files match template warnings because there aren't any files left. Basically you cannot use the Angle Bracket syntax to expand exceptions because by their very nature they expand to multiple arguments and you need all your exceptions to be applied to the first argument. The safest way to do it is to explicitly specify each exception thus:

```
) DELETE +\PROG1\PROG1.PR\PROG2\PROG2.PR
```

Or if you can find a template that matches all your exceptions then you could use that or something like this:

```
) DELETE +\PROG*\PROG*.PR
```

The **crunch** template, **#**, is often used in conjunction with INFOS files because one INFOS file consists of two Control Point Directories (Directories which are limited in the amount of disk space the files within them can occupy), usually **infosname** and **infosname.DB** where these files represent the **index** and **database** respectively, and contained in those directories are files usually with names like **VOL01 VOL02**, and possibly **DVL01 DVL02** etc. holding the actual data. To get the ACL of all the AOS/VS files making up the INFOS file **FRED**, you could use the command:

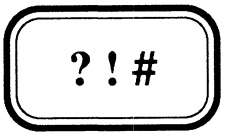
```
) ACL FRED<,.DB>:#
```

There is a subtle difference between the templates **+:#** and **#**. The latter (#) includes the current Working Directory, whereas the former (+:#) does not. This difference is usually not important, but it can be. For example let's say that to speed up ACL checking for access to files in a software directory, you decided to use the universal template + to allow everybody **RE** (Read and Execute) access. You could still restrict access to this software by having a more explicit ACL on the directory itself. Assuming you are in that directory you might issue the command:

```
) ACL # +,RE
```

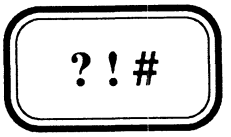
This would, however, change the ACL of the software Directory itself, meaning that not only have you now granted universal access to the software, but more importantly no-one including yourself has **O**wnership, **W**rite or **A**ppend access to the directory any more. The correct command to have entered is of course:

```
) ACL +:# +,RE
```



CHAPTER 10

Input - the !READ Pseudo Macro



CHAPTER 10

Input - the !READ Pseudo Macro

10.1 Input

No programming language can be complete without a means of interacting with Users and accepting responses to questions and other user entered information from them. CLI's input mechanism is probably the weakest link in its ability to function as a programming language (which by the way it was never intended to be, it is but a command line interpreter. It just happens to be a particularly powerful one. So much so that we can make it seem like a programming language).

There is just one free format input mechanism. Unlike most programming languages there is no built in mechanism for validating input, no formatting or data conversion facilities. It is virtually impossible, once CLI's input mechanism is used, to prevent the user responding with character sequences that will cause your macro to crash, no matter how careful you are. Various techniques for minimizing the probability of the User knocking your macro out have been tried, but with enough determination the smartie-pants will always find you out. For example, because the CLI recognizes **;** (semi-colon) as an in-line command delimiter, and because CLI is an interpreter, the user can always respond by entering **;X :CLI;**. The semi-colon will terminate the present command, and CLI will then interpret the next one which is the user entered command **X :CLI** which then invokes a son CLI, giving the user CLI access, which you are unlikely to want him to have. There are various tricks to minimize the likelihood of this happening, such as imbedding the entire command in a pair of Angle Brackets, so that when the user prematurely terminates your command with the semi-colon, he will get a **mis-matched brackets** error, but once he works out that you've done this he simply has to type **>;X :CLI;<** and he's done you in. There is only one failsafe means by which you can prevent the User breaking into CLI, which we will discuss at the end of this Chapter.

The input mechanism is the **!READ** Pseudo Macro, which was introduced in Chapter 8.

10.2 The !READ Pseudo Macro

Repeating the definition of the !READ pseudo macro from Chapter 8 we have:

[!READ prompt text] displays the supplied **prompt text** on the terminal, and immediately after the last character thereof waits for the user to respond. Whatever the user types in replaces the entire !READ Pseudo Macro in the body of the Macro.

This is normally used (although we don't think it should be - see Chapter 14) in conjuction with the CLI **STRING** command, to store the user's response (for later recall with **[!STRING]**) in CLI's 128 character string buffer, thus:

```
) STRING [!READ Answer the Question ]
```

The arguments supplied to the **!READ** Pseudo Macro form the text with which the User is prompted. This text is displayed from the current cursor position and is displayed in the same way as CLI's WRITE command displays text, but unlike WRITE it does not display a line terminating Newline character, but leaves the cursor positioned immediately after the last character displayed. The last character before the closing] of the !READ is usually a space so that the prompt text is not displayed hard up against the first character entered by the User. However, be aware that the reason this forces a space to be displayed is that it delimits a null argument for which CLI (like it does with the WRITE command) displays a space. Multiple spaces before the final closing], will still only generate one space in the display, because multiple spaces only delimit one argument. If we want multiple spaces displayed we use the same trick previously described for the WRITE command, which is to have multiple commas, which delimit multiple null arguments, therefore forcing CLI to display extra spaces.

Unfortunately the prompt text, as is text displayed by the CLI WRITE command, is interpreted by CLI. This severely restricts the kind of output you can display. For example brackets must match, semi-colons terminate the command, square brackets will cause unwanted or more likely non-existent macros to be invoked, angle brackets will generate unexpected combinations or disappear and commas cannot be displayed. We will discuss several techniques for overcoming these limitations in the next Chapter on Output. One advantage !READ has over WRITE though, is that because the text is enclosed inside !READ's delimiting square brackets, round brackets can be displayed because they are treated as argument grouping brackets, unlike WRITE which will cause multiple WRITE commands to be expanded. For example:

```
) STRING [!READ OK to proceed ? (Y N) ]
OK to proceed ? (Y N) Maybe

) WRITE OK to proceed ? (Y N)
OK to proceed ? Y
OK to proceed ? N
```

10.3 Response Evaluation

User responses to simple questions like the preceding example, are often evaluated in the folowing manner:

```
STRING [!READ OK to proceed ? (OK NO) {NO} ]
[!EQUAL,OK,[!STRING]]
     WRITE
     WRITE Continuing..
     WRITE
       .
       .
[!END]
```

The test here is for the non-default of the two acceptable answers, namely **OK**. The default being the option displayed in the curly brackets. We note that the DG standard for defaults is to display them in square brackets, but this would force the probably non-existant macro **NO** to be invoked, so we will use curly brackets until we show how, in the next Chapter, to generate square ones.

There are several problems with this particular code. Notwithstanding the fact that the User who really wants to, can break into CLI with the previously mentioned technique, consider the genuine User, it may well be you running your own macro for your own purposes, who has no desire to break the software, but accidently hits the Space Bar between the **O** and the **K**. This will cause the macro to crash with a **Pseudo macro has wrong number of arguments** error. This is because the !EQUAL Pseudo macro expands as **[!EQUAL,OK,O K]** giving it three arguments. The solution is of course to use the argument grouping brackets. Another problem with it is that it treats any response other than OK as implying NO. This may well be satisfactory but if not then each valid response must be separately tested, including the null response for the default. We will discuss this again later in this Chapter, so for now however we will assume it is acceptable as is.

Touching briefly on a subject we will return to in Chapter 14, you should be able to see that the use of the CLI **STRING**, is completely unnecessary in this example:

```
[!EQUAL,(OK),([!READ OK to proceed ? (OK NO) {NO} ])]
     WRITE
     WRITE Continuing..
     WRITE
       .
       .
[!END]
```

Numeric input is often evaluated as in the following example, but if a non-numeric response is entered, the macro will crash on the VAR0 command with an **Illegal decimal number** error.

```
VAR0 [!READ How many copies ? {1} ]
[!UEQ,0,[!VAR0]]
     WRITE OK %1% will not be printed
[!ELSE]
     QPRINT[!UNE,1,[!VAR0]]/COPIES=[!VAR0][!END] %1%
[!END]
```

10.4 Response Validation

Alphanumeric responses are relatively straightforward to validate, even if a bit longwinded, by simply testing for each valid response and trying again if no valid response has been entered. For example the macro **$YN** will return either Y or N in the STRING.

Its syntax is:

Format: **$YN{/DEF={Y/N}{ prompt text}**

where the Default response is taken as **N**, unless overridden by a **/DEF=Y** switch, and the Default prompt text is **Do you wish to continue.**

File $YN.CLI

```
[!EQUAL,(),(%-%)]
     %0% Do you wish to continue
[!ELSE]
     STRING [!READ/S %-% ? (Y N) {[!EQUAL,Y,%0/DEF=%]Y[!ELSE]N[!END]} ]
     [!EQUAL,(),([!STRING])]
          STRING [!EQUAL,Y,%0/DEF=%]Y[!ELSE]N[!END]
     [!ELSE]
          [!NEQUAL,(Y),([!STRING])]
               [!NEQUAL,(N),([!STRING])]
                    WRITE
                    WRITE ERROR: Invalid response please enter Y or N
                    WRITE
                    %0-%
               [!END]
          [!END]
     [!END]
[!END]
```

The principle used here could be extended to validate any number of possible responses. Validating numeric input however is a different kettle of fish (what on earth is a kettle of fish?), as already suggested in the previous section. The **/S** switch on !READ is explained at the end of the Chapter.

10.5 Validating Numeric Input

The basic trick to validating numeric input, requires you to invalidate one (and only one) of the valid responses. We may as well use the one least likely to be entered which of course is the largest. Then by setting a numeric variable to that value and then setting that variable to the user's response, but also ignoring any errors, then if the numeric variable still has its original value then either the input was invalid or the user actually entered that very large number. Thus:

```
VAR0 4294967295
VAR0/2=IGNORE [!READ Enter a number: ]
[!UEQ,4294967295,[!VAR0]]
     WRITE ERROR Input must be an integer
     %0%
[!ELSE]
     .
     .
[!END]
```

Often with numeric input as well as validating that the response is numeric we also want to allow only those values within a certain range. This can be done using the technique demonstrated in the macro **$NUM**, whose syntax is as follows:

Format: **$NUM{/MIN=m}{/MAX=n}{/DEF=d}{ prompt text}**

Where **/MIN=m** can be used to specify a minimum value, **/MAX=n** to specify a maximum and **/DEF=d** to specify a default value. Any or all of these switches may be omitted. The macro does not try to validate that the programmer using this macro supplied sensible switches but merely to validate the user's response. For example **$NUM/MIN=10/MAX=1** would send it into an infinite loop. The macro only returns after entry of a valid value. On return that value is held in **VAR0**, and the CLI **STRING** is always cleared. To allow a non-numeric response, which a null response (the default) is, means we must first check that possibility before doing the non-numeric check described above.

File $NUM.CLI

```
[!EQUAL,(),(%-%)]
     %0% Enter a number
[!ELSE]
     VAR0 4294967295
     STRING [!READ/S %-% &
([!EQUAL,,%0/MIN=%]0[!ELSE]%0/MIN=%[!END] -&
 [!NEQUAL,,%0/MAX=%]%0/MAX=%[!END])&
 [!NEQUAL,,%0/DEF=%]{%0/DEF=%} [!END]]
     [!EQUAL,(),([!STRING])][!NEQUAL,,%0/DEF=%]
          STRING %0/DEF=%
     [!END][!END]
     VAR0/2=IGNORE [!STRING]
     [!UEQ,[!VAR0],4294967295]
          WRITE
          WRITE ERROR: Non-numeric input please enter a number
          WRITE
          %0-%
     [!ELSE]
      STRING OUT_OF_RANGE
      [!UGE,[!VAR0],[!NEQUAL,,%0/MIN=%]%0/MIN=%[!ELSE]0[!END]]
      [!ULE,[!VAR0],[!NEQUAL,,%0/MAX=%]%0/MAX=%[!ELSE]4294967294[!END]]
          STRING/K
      [!END]
      [!END]
      [!EQUAL,OUT_OF_RANGE,[!STRING]]
          WRITE
          WRITE ERROR: Please enter a number within range
          WRITE
          %0-%
      [!END]
     [!END]
[!END]
```

In Chapter 13 we will be discussing Complex Logic using **ANDs**, **ORs** and **NOTs**, which CLI doesn't support in an obvious way, but it is perhaps worth pointing out some of the complex logic used in **$NUM**. For example provided you don't need an ELSE branch an **AND** can be achieved simply by nesting, which we did when we tested whether a /DEF=d switch had been supplied **AND** the user chose the default by giving a null response. In doing the range check we have effected an **OR**, by using the STRING to enable us to take the **NOT** of an **AND**.

10.6 Failsafe Input

The only method we have been able to discover to prevent a determined user from getting to CLI from a !READ, requires another process to do the input. Such a process could of course be written in a real programming language, or we could set up a son CLI process to do it, as follows:

File $READ.CLI

```
X/S :CLI $READIT %-%
```

File $READIT.CLI

```
PROMPT BYE
BYE/L=@NULL [!READ %-%]
```

NOTE:

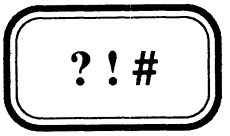
BYE message	Sends the termination **message** to the father process
X/S	Returns the process's termination message in the CLI STRING
/L=@NULL	Discards the normal **AOS/VS CLI Terminating** message displayed on the screen.

The response will be returned in the CLI STRING. The reason this trick works is because of the location of the !READ Pseudo Macro. If the response were temporarily stored in the STRING, the User could still use the **;X :CLI;** trick. But no matter what the User types in **$READIT**, the **BYE** command is going to be interpreted first and once the process has died, it cannot interpret any further commands the user may have entered.

The extra process is a heavy price to pay, and fortunately DG saw fit to remedy the situation by adding the **/S** switch on !READ. Presumably **/S** in this case represents **S**emi-colon, for its effect is to force !READ to treat Semi-colon as an input delimiter, and so any commands typed after a Semi-colon are ignored thereby safely preventing the user from getting to CLI. This switch seems like a good idea and was probably included in response to some STR's (Software Trouble Reports), but unfortunately the determined user can still break it, using the following technique:

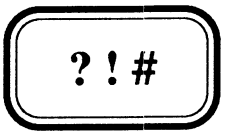
Instead of typing **;X :CLI;** the user types **[!ASCII 73]X :CLI[!ASCII 73].** Input is not terminated by a semi-colon because there is no semi-colon in the input, but on expanding the !ASCII Pseudo Macro, CLI substitutes a semi-colon for it, terminates the current command and eXecutes a son CLI. We have used the **/S** switch in our macros (as shown in $NUM), because we believe it is better than not using it, but the technique described above using **$READ** (or something similar), remains the only failsafe method of accepting user input to a CLI macro.

THIS PAGE INTENTIONALLY LEFT BLANK



CHAPTER 11

Output - WRITE and TYPE



CHAPTER 11

Output - WRITE and TYPE

11.1 WRITE and TYPE commands

The two main ways of creating screen output from a CLI macro are with the CLI commands WRITE and TYPE. We have already used the WRITE command in several of our examples. The TYPE command simply types out a text file. Comparing the two we have:

WRITE command

- One line of output per command.
- Variable output using Arguments, Switches and Pseudo Macro expansions.
- Output is slow.
- What you see is not always what you get. Mutiple commas must be used to generate multiple spaces and round brackets generate multiple WRITE commands instead of being displayed.
- Have to resort to special tricks (described in next Section) to generate ordinary characters like commas, round, square and angle brackets.
- Text output forms part of the macro, (one file to maintain).

TYPE filename

- Many lines of output per command.
- Fixed non-varying output.
- Output is fast.
- What you see is what you get.
- No special tricks required to display certain characters.
- Text output is held in a separate file (perhaps several).

In general we would recommend that you use TYPE whenever you can, and only use WRITE for one-liners, or where Pseudo Macro, Argument or Switch expansions are necessary. Unfortunately this turns out to be more often than not but you should bear in mind that if the first line of a block of text to be displayed requires such expansions but the rest is static, then you can use WRITE for the first line and TYPE for the rest.

There is a third method of creating screen output, which allows many lines of output per command, supports variable output through argument and switch expansions, is fast, what you see is what you get, requires no special tricks and remains part of the macro. It in fact gives us the best of both worlds. It is relatively obvious technique and it is surprising to us that it isn't used more frequently. We will discuss it at the end of this Chapter.

11.2 The !ASCII Pseudo Macro

[!ASCII oct1{,oct2...}]

> The **!ASCII** Pseudo Macro takes **oct**al integers as arguments and for each such argument returns the single ASCII character represented by that value.

We can therefore use this Pseudo Macro to generate any character we like including non-printing control characters. They can be used as arguments to the WRITE command to enable us to display the characters we can't otherwise display. For example the character represented by the Ascii value 7, CTRL-G will when sent to DG terminals cause the Bell to sound. Therefore we might use:

> **WRITE** ERROR: Ringing bell to draw attention to it**[!ASCII 7]**

However this would cause the following output to be displayed

> ERROR: Ringing bell to draw attention to it^G

without sounding the bell. The problem is that CLI interprets the characters that form arguments to the WRITE command, and decides that since this is a non-printing Control character, it will replace it with a two-character sequence to show which control character it was. So the [!ASCII 7] never actually got sent to the terminal.

Most of you will be aware the ASCII characters, 0 - 127 (177 octal) are a seven bit code, and therefore the top bit (Bit 0) of the 8-bit byte in which a character is stored is never used. CLI uses it, however to tell it not to interpret the control character represented by this byte and send it to the terminal as is. To set the bit all we need to do is add 128 (200 octal) to it, and thus we have:

> **WRITE** ERROR: Ringing the bell to draw attention to it**[!ASCII 207]**

The terminal itself ignores Bit 0 and thus recognizes the character as the one whose Ascii code is 7, and therefore rings the Bell. The following is a list of the !ASCII sequences most often used in conjunction with the CLI WRITE command (and !READ Pseudo Macro) and the effect they produce:

[!ASCII 207]	**Rings the Bell**
[!ASCII 211]	**Tab**
[!ASCII 212]	**Newline**
[!ASCII 213]	**Erase to EOL (Blank from cursor to EOL)**
[!ASCII 214]	**Form-feed (Blank screen)**
[!ASCII 215]	**Carriage Return**
[!ASCII 216]text[!ASCII 217]	**Blink text**
[!ASCII 220,2cc,2rr]	**Position Cursor at row rr column cc**
[!ASCII 224]text[!ASCII 225]	**text**
[!ASCII 234]text[!ASCII 235]	text **(ie Dim text)**
[!ASCII 250]text[!ASCII 251]	**(text)**
[!ASCII 254]	**Comma**
[!ASCII 274]text[!ASCII 276]	**<text>**
[!ASCII 333]text[!ASCII 335]	**[text]**

You could use this technique to generate other useful sequences.

One of the most useful of the above sequences is perhaps the ability to Position the Cursor, anywhere you like on the screen. You must remember that **cc** and **rr** are octal numbers (since that's what !ASCII requires) and that line 1 column 1 is represented by cc=00 and rr=00. To prompt the user for a response in column 79 (117 octal) of line 23 (27 octal) you could do as follows:

```
STRING [!READ [!ASCII 220 200 226]Continue?[!ASCII 220 316 226]]
```

There are a couple of pitfalls with cursor positioning of which you should be aware. Firstly never prompt the User for input on Line 24, if you do not wish the screen to scroll. Secondly beware of Line 10 and/or Column 10. Why? Well line 10 would be generated with rr=11 and column 10 with cc=11 and 11 just happens to be the ASCII code for the TAB character. Now it seems DG terminals are unable to distinguish an 11 code that is part of a cursor positioning command from a TAB, and your output gets screwed up. We suggest that to position on line 10, you position on line 9 and use the down cursor command, and for column 10 position to column 9 and use the right cursor command thus:

```
WRITE [!ASCII 220 210 210 232 230]This starts in Line 10 Col 10
```

The other solution is to change the characteristics of your terminal before issuing your WRITE command, to turn off the Simulate Tabs characteristic thus:

```
CHARACTERISTICS/OFF/ST
WRITE [!ASCII 220 211 211]This too starts in Line 10 Col 10
```

As you can no doubt see, to the uninitiated CLI macro code with lots of !ASCII sequences can easily become unintelligible and often impossible to maintain. In Chapter 19 we will look at ways of making such code both intelligible and maintainable.

One final word of warning about the !ASCII Pseudo Macro. The character generated when you add 200 octal turns Bit-0 of the byte on. This forms part of the character and although when sent to the terminal it may look like the character you want, it is not that character in other contexts. Consider the following macro:

File $TEST A.CLI

```
DELETE/2=IGNORE A_CHAR<,.FAKE>
WRITE/L=A_CHAR(,.FAKE) (A,[!ASCII 301])
STRING [A_CHAR]
PUSH
STRING [A_CHAR.FAKE]
WRITE [!STRING] [!NEQUAL,[!STRING],[!STRING/P]]Not[!END]= [!STRING/P]
POP
```

```
) $TEST_A
A Not= A
```

This difference is unimportant as long as output is merely displayed but if it is written to a file and used subsequently it can be critical. A Comma written to a file using [!ASCII 254] for example will not subsequently be recognized as an argument delimiter.

11.3 The Best of Both Worlds

The best way we have discovered to create screen (or equally well Batch Output File) output, is with the following trick:

File $OUTDEMO.CLI

```
CREATE/M @OUTPUT

     Wow!       Free format output, Yes that's a Comma
                Indentation with Tabs
                Text in brackets [square] (round) and <angle>

     What's more they don't have to match eg. 1 < 10

     We can even expand Arguments, eg. Arg1 is "%1%"
     and switches                  Switches are "%0/%"

     But ALAS! No Pseudo Expansion eg. [!DIRECTORY]

     What you see is what you get!
)
```

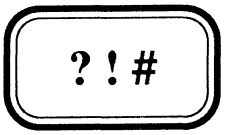
And when we run it for example:

```
) $OUTDEMO/switch one

     Wow!       Free format output, Yes that's a Comma
                Indentation with Tabs
                Text in brackets [square] (round) and <angle>

     What's more they don't have to match eg. 1 < 10

     We can even expand Arguments, eg. Arg1 is "one"
     and switches                  Switches are "/switch"

     But ALAS! No Pseudo Expansion eg. [!DIRECTORY]

     What you see is what you get!
```

CREATE/M @OUTPUT opens a file to the current generic **output** file (See Chapter 15 for details), which for processes logged on at consoles will be the terminal or the Batch Output File for Batch Jobs. The **/M** switch tells CLI that the contents of this file is to be read from the body of the macro, and will be terminated by a line beginning with a single closing round bracket **)**. Because arguments and switches are substituted with their actual values throughout the macro, before CLI begins interpreting it, they can be used to produce variable output. Pseudo Macros are processed while CLI is interpreting, and so once CLI interprets the **CREATE/M** it simply sends to the output file everything it encounters without interpreting it, until it hits the closing round bracket at the beginning of a line. Thus Pseudo Macros become just another string of characters to be output.

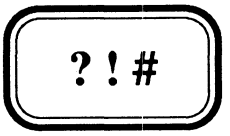
With this technique we have the added advantage of the text output remaining part of the macro itself.

Need we say more? We think not!



CHAPTER 12

Documenting Macros



CHAPTER 12

Documenting Macros

12.1 Documentation

If it is to be maintainable all Software needs good documentation. CLI Macros are no different. In the early days DG made this somewhat more difficult than it should have been by not providing a COMMENT facility, but CLI does now support such a command. There are basically two sorts of documentation for software, namely, that directed at Users of it explaining what it does, and that directed at Maintenance Programmers explaining how it does it. We will look at each of these separately.

12.2 User Documentation

User documentation for a CLI macro should be clear, concise and complete. It should cover all of the following:

Purpose: A clear statement of what the macro does including the input required and the output produced. Any changes to the CLI Environmental Parameters (STRING DIRECTORY etc.) should be stated. (We actually prefer Macros not to change the Environment and Chapter 22 explains how to preserve it.)

Format: The calling sequence of the macro, preferrably distinguishing optional from required switches or arguments.

Switches: A list of all the macro's switches and a brief description of each. If not already implied by the Format you should state whether or not a switch is optional and if so the default when it's omitted.

Arguments: A list of all the macro's arguments and a brief description of each. If not already implied by the Format you should state whether or not an argument is optional and if so the default when it's omitted.

If such documentation forms part of a manual then one can afford to be more descriptive, but when it forms part of an on-line Help facility, where possible it is best to keep this to no more than a screen of information. This saves the User having to use CTRL-S/CTRL-Q or the HOLD key to freeze and unfreeze the screen to look at the information. We recommend no more than 22 lines so that, if the CLI User has a PROMPT set that displays the TIME or their current working DIRECTORY (a common practice), your documentation still fits neatly on a screen.

We have adopted a convention that all our macros should provide their own Help facility by supporting a **/H** switch, which whenever present forces the macro to display its own Help text. There are two main ways we recommend this be done:

```
[!NEQUAL,,%0/H%]
     TYPE %0\%.HELP
[!ELSE]
          ..
     body of macro
          ..
[!END]
```

Or

```
[!NEQUAL,,%0/H%]
     CREATE/M @OUTPUT

Free format Help Text is inserted here

)
[!ELSE]
          ..
     body of macro
          ..
[!END]
```

Both techniques have their merits. Those who prefer the latter argue that it is better because the documentation is kept alongside the macro and there is only one file to be maintained. Those who prefer the former argue that it is better because the documentation is kept separate from the macro and can therefore form part of a general Help facility independent of the macro itself. For example if you named your Help file **CLI.TPC.macroname** instead of our recommended **macroname.HELP**, you could move it into the **:HELP** directory, and it would subsequently appear in the list of Help topics listed when you use the CLI command **HELP** and **HELP *macroname** would type out your Help file.

For reasons already explained including placing too many restrictions on the format of your help text and from a maintenance programming point of view they are generally a pain in the butt, we are not at all fond of using the CLI WRITE commands for providing Help.

As an example of the sort of documentation we recomend here is the help for the **$NUM** macro described in Chapter 10:

File $NUM.HELP

```
$NUM              This macro accepts and validates numeric input. The
                  macro loops until a valid response is entered and
                  returns this response in [!VAR0]. The CLI STRING is
                  always cleared in the process. Control over the
                  range of valid responses is provided with switches
                  but the macro makes no attempt to check that these
                  switches are consistent with each other.

Format:           $NUM{/MIN=m}{/MAX=n}{/DEF=d}{/H}{ prompt text}

Switches:

   /MIN=m         Sets minimum valid response to m (Default: 0)
   /MAX=n         Sets maximum valid response to n (Default: 2^31-2)
   /DEF=d         Sets default response to d       (No default)
   /H             Displays this Help Text

Arguments:

   prompt text    Optional prompt text (Default: Enter a number)
```

Blank lines and the /H switch can be dropped when tight for room and the general format **$NUM{/switches}{ args}** used if there are too many to fit in a single line. If it is absolutely impossible to reduce your Help text to no more than a screen full without destroying its effectiveness, we recommend you automatically turn the Page Mode characteristic on, in the same way as **HELPV** does, to guarantee the User sees all the information and which therefore only requires him to hit CTRL-Q when he is ready to see the next screen full. For example:

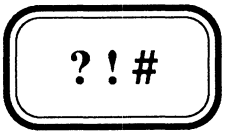
```
[!NEQUAL,,%0/H%]
     CHARACTERISTICS/ON/PM
     TYPE %0\%.HELP
     CHARACTERISTICS/OFF/PM
[!ELSE]
          ..
     body of macro
          ..
[!END]
```

The last line of your first screen will be line 23 of your Help file and the last line of each subsequent screen every 24 lines after that (ie. 47 71 95 etc). The bottom of each screen should tell the user to hit CTRL-Q to see the next screen. We recommend for such Help files that the last screen always be full (even if it's mostly blank lines), because of timing problems with PMGR sending the last buffer to your screen and the subsequent commands being executed. For an example of such a file type out one of the **:HELP:CLI.CMD.command** files. You will notice the clever use of highlighting in these files to make the Help more readable. If you would like to use highlighting you may recall from the previous Chapter, that the character with the ASCII value 34 octal will turn dim mode on, and 35 to turn it off. These characters can be inserted directly into a Help file using **SED** by typing the characters **CTRL-** (Dim on) and **CTRL-]** (Dim off). Note for alignment purposes, in SED these characters will display as ^\ and ^], appearing to occupy two character positions, whereas when typed they will not. For this reason we recommend that if you want highlighting in your Help files to add the highlighting after everything else is right. Note also that we don't have any problem here with respect to the top bit (adding 200 octal) as we do with the WRITE command because CLI does not interpret the contents of the file, it merely outputs it to the terminal. This is why accidently typing out .PR files does crazy things to your screen.

12.3 Programmer Documentation

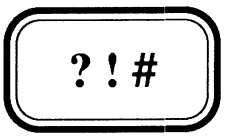
Programmer documentation is that documentation over and above the User documentation, that is needed by the maintenance programmer to help him understand how the macro achieves its aims. It is generally provided with a smattering of CLI **COMMENT** commands throughout the macro and this is usually sufficient. Note that the COMMENT command is useless as far as User documentation is concerned, because generally the User does not get to see the contents of the macro. Unfortunately the COMMENT command is still parsed by CLI and therefore comes with similar restrictions on the format of the comment as does the CLI WRITE command.

In general we prefer macros to be sufficiently obvious that the User documentation, the macro itself and the odd COMMENT line are enough, but as you will no doubt see with some of our more complex examples later in the manual this is not always possible. Where more detailed explanation is required, rather than clutter the macro with it we recommend that it be held elsewhere with an appropriate COMMENT line directing the programmer to consult whatever or wherever 'elsewhere' happens to be.



CHAPTER 13

Complex Logic



CHAPTER 13

Complex Logic

13.1 AND

CLI does not provide us with a mechanism for complex Boolean expressions, using **AND**s and **OR**s. In Chapter 10 with our **$NUM** macro we constructed an **AND** expression simply by nesting the two conditions:

```
[!EQUAL,(),([!STRING])][!NEQUAL,,%0/DEF=%]
     STRING %0/DEF=%
[!END][!END]
```

or in plain English

```
If the Default was chosen AND a default was supplied
     Set the STRING to that supplied default
```

We cannot however provide an **ELSE** branch, because it will only match one of the conditions, depending on the location of the **[!ELSE]**. We could of course provide an ELSE for each condition and duplicate the code but this is inappropriate if the required code is more than a few lines.

However let's say that when no default is supplied **AND** the default is chosen, we wish to choose the minimum value (instead of rejecting the response as $NUM does in its present form), otherwise process as before. We could do this as follows:

```
[!EQUAL,(),([!STRING]%0/DEF=%)]
     STRING [!EQUAL,,%0/MIN=%]0[!ELSE]%0/MIN=%[!END]
[!ELSE]
     ....
[!END]
```

Here we have achieved an **AND** simply by concatenating the strings we are testing, and since there is only one conditional Pseudo Macro involved we can provide an **ELSE** branch. This technique can be used whenever you are testing for two strings to have null values. In some cases they can be non-null and different with **almost** equal effect. For example:

```
[!EQUAL,%1%%2%,value1value2]
```

tests if Argument 1 is "value1" **AND** Argument 2 is "value2". However if it were possible for Argument 1 to have the value "value1value2" and Argument 2 to be null this would satisfy the above test but is not the **AND** of our two conditions. However the following expression is:

```
[!EQUAL,**,[!EQUAL,%1%,value1]*[!END][!EQUAL,%2%,value2]*[!END]]
```

Two asterisks can only be expanded on the right-hand side if both our test conditions are true, so the entire expression can only evaluate true when they are both true. This technique can be expanded to **AND** together as many conditional expressions as you wish, and perhaps more importantly the conditional expressions can be mixed (ie one can be an **EQUAL**, another a **NEQUAL**, and yet another a **UGT**). However if you wished to test for the existence of the INFOS index file (Argument 1) and its associated Data Base file, the following would suffice:

```
[!EQUAL,=%1%=%1%.DB,[!FILENAME %1%][!FILENAME %1%.DB]]
```

13.2 OR (inclusive)

In our **$NUM** macro of Chapter 10 we effected an **OR** by taking advantage of the fact that in Boolean algebra the expression **a OR b** is equivalent to the expression **NOT (NOT a AND NOT b)**. In English and referring to our particular example we want to test:

If Variable 0 is less than the minimum **OR** greater than the maximum

This is equivalent to testing:

If Variable 0 is **NOT** (greater or = minimum **AND** less or = maximum)

Resulting in:

```
STRING OUT_OF_RANGE
[!UGE,[!VAR0],[!NEQUAL,,%0/MIN=%]%0/MIN=%[!ELSE]0[!END]]
[!ULE,[!VAR0],[!NEQUAL,,%0/MAX=%]%0/MAX=%[!ELSE]4294967294[!END]]
     STRING/K
[!END]
[!END]
[!EQUAL,OUT_OF_RANGE,[!STRING]]
```

However this is a fairly cumbersome way of expressing:

If Var0 < min **OR** Var0 > max

But using a similar technique to that used in our generalized **AND** expression we can generate a generalized **OR**:

```
[!NEQUAL,,[!EQUAL,%1%,value1]*[!END][!EQUAL,%2%,value2]*[!END]]
```

Tests if Argument 1 is "value1" **OR** Argument 2 is "value2". The right-hand side expands an asterisk if one (and one only) of our conditions is true, two asterisks if they are both true and null if neither is true. So by testing for a non-null result we are generating an **inclusive OR**. Using this technique our range check would become:

```
[!NEQUAL,,&
[!ULT,[!VAR0],[!NEQUAL,,%0/MIN=%]%0/MIN=%[!ELSE]0[!END]]*[!END]&
[!UGT,[!VAR0],[!NEQUAL,,%0/MAX=%]%0/MAX=%[!ELSE]42949etc[!END]]*[!END]]
```

This looks more complex than it should because of the imbedded code to set the minimum to the supplied value or the default if none was. In a similar fashion to certain types of **AND** conditions, we can sometimes come up with an extremely simple expression for **OR** using string concatenation. This situation arises whenever we want to test if any one (or more) of a number of expressions is non-null. The classic example is when you want to test if any one of a number of switches is present, or more importantly when a switch or any one of its valid abbreviations is present. For example let's say that instead of supporting a **/H** switch, we supported a **/HELP** switch of which any abbreviation including **/H** was acceptable. We could write:

```
[!NEQUAL,,%0/H/HE/HEL/HELP%]
     TYPE %0\%.HELP
[!ELSE]
```

13.3 OR (exclusive)

The only way of generating an **exclusive OR**, ie one which is only true when one and only one of the various conditions is true is with a slight variation of our generalized **inclusive OR** thus:

[!EQUAL,*,[!EQUAL,%1%,value1]*[!END][!EQUAL,%2%,value2]*[!END]]

Tests if Argument 1 is "value1" **OR** Argument 2 is "value2" but where both conditions are not true. The right-hand side expands an asterisk if one (and one only) of our conditions is true, two asterisks if they are both true and null if neither is true. So by testing for a single asterisk we are generating an **exclusive OR.**

13.4 Summary

To sum up we have the following three generalized complex logic expressions:

AND

[!EQUAL,**,[!conditionalPSM1]*[!END][!conditionalPSM2]*[!END]]

inclusive OR

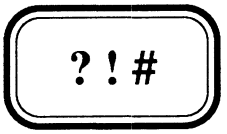
[!NEQUAL,,[!conditionalPSM1]*[!END][!conditionalPSM2]*[!END]]

exclusive OR

[!EQUAL,*,[!conditionalPSM1]*[!END][!conditionalPSM2]*[!END]]

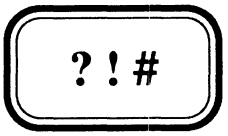
where of course **conditionalPSMn** is any conditional Pseudo Macro including other complex logic ones. These conditions are the actual conditions we are **AND**ing or **OR**ing, and not their **NOT** (just in case you got confused along the way).

THIS PAGE INTENTIONALLY LEFT BLANK



CHAPTER 14

Nesting and STRING-less Macros



CHAPTER 14

Nesting and STRING-less Macros

14.1 The CLI STRING

The CLI STRING is a CLI environmental parameter, which is a buffer for the temporary storage of a string of up to 128 characters. Proper programming languages provide for any number of alphanumeric variables, generally of arbitrary name, and of whatever length you choose (up to some extremely large maximum). In addition they provide several mechanisms for manipulating and processing them. CLI provides one such variable with a fixed name and of a relatively short maximum length, and few tools for manipulating them. Using the CLI PUSH command one can take advantage of CLI's **/P** switch on **[!STRING]** to make a second such variable available. This is often the only reason some macro writers ever use the command PUSH.

In this chapter we will show how you can effectively make any number of alphanumeric variables available to CLI, where each one can be of an arbitrarily large length.

Typically the STRING is used in combination with the !READ Pseudo Macro for storing User responses to prompts, as shown in Chapter 10. However as we pointed out in that Chapter sometimes the STRING is used unnecessarily, in that often the !READ Pseudo macro can be expanded directly into the !EQUAL Pseudo Macro testing the user response, instead of temporarily storing it in the STRING.

14.2 Multiple Responses

The typical code seen in CLI macros for soliciting a second response from a user is:

```
STRING [!READ first prompt ]
PUSH
STRING [!READ second prompt ]
```

However using **[!STRING]** and **[!STRING/P]**, you only have access to at most 2 responses at the one time, which are arbitrarily restricted to no more than 128 characters.

A much more flexible method is through **NESTING** subordinate macro calls thus:

File $NEST.CLI
```
$NEST.1 [!READ first prompt ]
```

File $NEST.1.CLI
```
$NEST.1.1, %1%, [!READ second prompt ]
```

File $NEST.1.1.CLI
```
$NEST.1.1.1, %1-2%, [!READ third prompt ]
```

File $NEST.1.1.1.CLI
```
WRITE
WRITE Responses to prompts were:
WRITE
WRITE (first second) (%1%,%2%)
WRITE third %3-%
WRITE
```

Note that the **commas** are significant in $NEST.1 and $NEST.1.1, to allow for the possibility of a null response to each prompt.

$NEST might be invoked thus:

```
) $NEST
first prompt A RESPONSE
second prompt TWO
third prompt THREE

Responses to prompts were:

first A
second TWO
third THREE
```

Note however the difference between this and the result of running $NEST2.

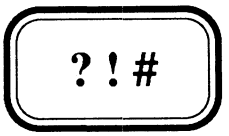
File $NEST2.CLI

```
$NEST.1.1.1, [!READ prompt1 ], [!READ prompt2 ], [!READ prompt3 ]

) $NEST2
prompt1 A RESPONSE
prompt2 TWO
prompt3 THREE

Responses to prompts were:

first A
second RESPONSE
third TWO THREE
```

$NEST2 is clearly not satisfactory. In order to support a multi-word response to a prompt (notice we discarded the 2nd and subsequent arguments entered in the first and second prompts of $NEST) we have to arrange for it to be the last prompt so that we can reference it with **%n-%**. For example:

```
) $NEST
first prompt A RESPONSE
second prompt TWO
third prompt THIS TIME MULTI WORD IS OK

Responses to prompts were:

first A
second TWO
third THIS TIME MULTI WORD IS OK
```

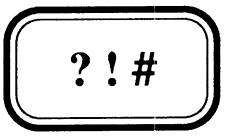
With this technique we can reference as many different responses as we like, with no arbitrary limit to their length, without having to use the CLI STRING or to PUSH. This also means that we have preserved the CLI environment (discussed further in Chapter 22). In response to those of you who argue that this is inefficient, because each of the Macros $NEST, $NEST.1, $NEST.1.1 and $NEST.1.1.1 have to be read in, our argument is that if you really wanted efficiency you wouldn't write it with CLI macros in the first place.



CHAPTER 15

Generic Files

15.1 Generic Files



CHAPTER 15

Generic Files

15.1 Generic Files

15.1 Generic Files

AOS/VS maintains six generic files for each process. They are **@INPUT**, **@OUTPUT**, **@CONSOLE**, **@DATA**, **@LIST** and **@NULL**. They offer a generic way of referring to a certain file, where the specific file actually referenced usually varies. For example **@CONSOLE** is a generic way of referring to the particular console at which you are working, without having to know what your actual console line number is.

For those of you not familiar with the **@** prefix, other than knowing that Magnetic Tapes, Consoles and Printers are prefixed with it, we will explain. You may logically from your experience think of it as meaning a Device rather than a file, and in a sense you are right. More correctly however, **@** is merely an abbreviation for **:PER:**. In other words **@LPT** actually refers to the file LPT in the directory :PER, known as the Peripheral Directory. This Directory is rebuilt by AOS/VS every time the Operating System is re-booted, based on the definitions of the attached Peripheral Devices in the particular tailored version of the Operating System that defines your environment.

Therefore there is a file in the directory :PER for each of the six generic files. Each process will have its own actual file representing each generic file. Basically the Generic Files have the following meanings:

@INPUT The default input file from which a process reads data. In the case of CLI it is the file from which it receives commands, which is usually your console.

@OUTPUT The default output file to which a process sends data. In the case of CLI it is the file to which the output of its commands is sent, which is also usually your console.

@CONSOLE The console device attached to the process, which is always your console. However for some processes, in particular Batch Processes, @CONSOLE is not defined. This is why some programs will not run in Batch. (Specifically DG **COBOL** Programs which use **DISPLAY SCREEN-SECTION-ITEM** will not run in Batch, whereas Programs that use **DISPLAY WORKING-STORAGE-ITEM** will, because COBOL uses @CONSOLE for Screen Section Displays and @OUTPUT for Working Storage Section Displays.)

@DATA A secondary (input) file to/from which a process may do I/O.

@LIST A secondary (output) file to/from which a process may do I/O.

@NULL A black hole, used mainly for discarding output. Output directed at @NULL will not appear anywhere, whereas an attempt to Read from @NULL will generate an End-Of-File condition.

@INPUT, **@OUTPUT** , **@CONSOLE**, **@DATA** and **@LIST** are defined (or not defined as the case may be, eg. @CONSOLE for Batch Jobs) at process creation time and cannot be changed for the life of the process. Note also that for CLI processes created by EXEC, **@DATA** and **@LIST** are not defined. If you are not convinced of this try typing in the CLI command:

```
) WRITE [@LIST]
```

The name **@DATA** implies input and is usually used for this, while the name **@LIST** implies output and is generally used this way, however there is nothing to stop a process writing to **@DATA** or reading from **@LIST.**

@DATA and **@LIST** should not be confused with the CLI Environmental Parameters **DATAFILE** and **LISTFILE**, which like all Environmental Parameters can be changed at any time during the process's life. However the default values for **DATAFILE** and **LISTFILE** just happen to be **@DATA** and **@LIST.**

CLI supports a **/L** switch on all its commands, which when used tells CLI to direct the output (that which would normally appear on the screen) of this command to the current **LISTFILE.** There is an important difference between **/L** and **/L=filename.** Once **LISTFILE** has been set with the LISTFILE command the file is opened, and it is not closed until the next LISTFILE command or the termination of the process. The use of the switch **/L=filename** however will open, output to, and close the file. Thus:

```
) WRITE/L=filename NOTHING
) TYPE filename
NOTHING
)
```

whereas:

```
) LISTFILE filename
) WRITE/L NOTHING
) TYPE filename
)
```

This is because CLI only does a physical write to the LISTFILE when the buffer is full or the LISTFILE is closed, and so in the second case the output **NOTHING** is still sitting in the buffer.

However note that:

```
WRITE/L=filename .....
WRITE/L=filename .....
WRITE/L=filename .....
     ...
     ...
```

is considerably slower than:

```
LISTFILE filename
WRITE/L .....
WRITE/L .....
WRITE/L .....
     ...
     ...
LISTFILE/G
```

because the former opens and closes the file for each WRITE command. **LISTFILE/G** resets the **LISTFILE** back to its original generic setting, ie **@LIST**, whereas **LISTFILE/K** clears it. However both cause the current LISTFILE to be closed.

The CLI **EXECUTE** (**XEQ** or **X**) command passes on the generic files, **@INPUT**, **@OUTPUT** and **@CONSOLE**, to the program being run and sets the generic files, **@DATA** and **@LIST** to the current CLI Environment Parameters **[!DATAFILE]** and **[!LISTFILE]**. This often leads to the misconception that they are one and the same. To reference these two files in the program, one simply has to reference the file by its generic name in the program (whatever the language). Wherever you normally specify the filename in whatever language you are using, usually by **"filename"**, you simply specify **"@DATA"** or **"@LIST"**. Then by setting the LIST and/or DATA file, before executing the program you can make it read from or write to a different file on different runs.

For example in a COBOL program I might have:

```
ASSIGN MY-INPUT-FILE "@DATA" .....
```

and

```
ASSIGN MY-OUTPUT-FILE "@LIST" .....
```

Then the commands:

```
) DATAFILE @CONSOLE
) LISTFILE MY_OUTPUT
) X MYPROG
```

would cause MYPROG to read from the console but output to MY_OUTPUT. Note to generate an End-Of-File condition for the console input file you would have to hit CTRL-D (^D).

On a subsequent execution I might:

```
) DATAFILE MY_INPUT
) LISTFILE @CONSOLE
) X MYPROG
```

which would cause MYPROG to read from MY_INPUT, but output to the console.

@NULL can also be quite useful. In Chapter 10, when discussing a failsafe method of doing input to a CLI macro while preventing the user from breaking into CLI you may remember the BYE command:

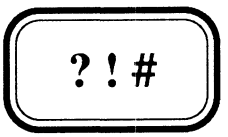
```
BYE/L=@NULL [!READ %-%]
```

This tells CLI to re-direct the output of the BYE command to the black hole **@NULL**, which causes its normal display on termination to be discarded. It is also extremely useful when you wish to force a pause in a macro (say to ensure the user reads the error message on the screen) but are not really interested in what the user enters in acknowledging that he has seen the message. Thus:

```
@NULL [!READ Hit NEWLINE to continue ]
```

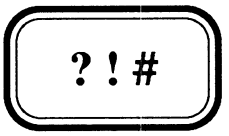
This effectively generates a null command, and saves you using the CLI STRING (and possibly a PUSH which may be necessary to preserve the previous contents of the STRING), simply to wait while the User hits NEWLINE.

THIS PAGE INTENTIONALLY LEFT BLANK



CHAPTER 16

Standard CLI Switches - /1 /2 & /L



CHAPTER 16

Standard CLI Switches - /1 /2 & /L

16.1 The /1= and /2= Switches

There are four switches that all CLI commands support. They are the **/1=severity**, **/2=severity**, **/Q** and **/L** switches. Personally we wish the **/Q** switch did not exist, because then we could for example on the QPRINT command abbreviate the /QUEUE switch to /Q instead of /QU. Furthermore we have yet to find a good use for it (/Q that is), and therefore we will not even bother to discuss it.

The **/1=severity** and **/2=severity** switches may be used to override for the command on which they are supplied, the default setting for the **CLASS1** and **CLASS2** error conditions respectively. **CLASS1** and **CLASS2** are CLI environmental parameters which can be displayed and set with CLI commands of the same name. The values that **severity** can take including when used in conjunction with the CLASSn commands are:

IGNORE ignore the occurrence of such an error and continue on regardless doing the best you can.

WARNING ignore the occurrence of such an error as with IGNORE but give me a **WARNING** message to let me know it has occurred.

ERROR give an **ERROR** message and discard the rest of the command line or macro currently being interpreted. This will usually result in the User receiving CLI's command prompt.

ABORT give an **ERROR** message and abort this process, which will usually result in the User being logged off.

An error condition is raised whenever CLI is for some reason unable to carry out the command issued. AOS/VS returns an error message number and the text associated with that number is looked up in the **:ERMES** file. The **CLASS** setting determines how that error is to be treated. **CLASS1** errors which are generally considered more serious, occur whenever a command that alters the CLI Environment (See Chapter 7) such as DIRECTORY or SEARCHLIST, fails, while other commands such as DELETE give rise to a **CLASS2** error. The **/1=severity** or **/2=severity** switches enable us for a particular command to override the way CLI handles that error. For example, one of the most common uses of these switches is:

```
) DELETE/2=IGNORE filename
```

This is used because since the most likely reason a delete command will fail is that the file you wished to delete does not exist. Since what we are trying to achieve has already been done, it makes sense to ignore the error. **However be warned**, this is not the only reason that the DELETE command might fail. For example if you don't have Ownership Access to the File or Write Access to the Directory in which it resides (See Chapter 17 Understanding ACLs) then the DELETE command will fail but the file will still exist. A safer way to DELETE ignoring the fact that a file may not exist in the first place is:

```
[!NEQUAL,,[!FILENAME filename]]
     DELETE filename
[!END]
```

In this way if the command fails for any other reason other than non-existence, your macro will not proceed. Other reasons that DELETE can fail include the fact that the file may have the PERMANENCE attribute turned on, or the file may be a directory containing directories (which gives rise to a DIRECTORY DELETE ERROR). If you are likely to use this "safe delete" frequently it might be a good idea to put the code in a macro, possibly also making it a delete that ignores PERMANENCE. It would need to support all of DELETE's switches. eg:

File $DELETE.CLI

```
[!NEQUAL,,[!FILENAME %1%]]
     PERMANENCE %1% OFF
     DELETE%0/% %1%
[!END]
```

We recommend therefore that **/1=severity** and **/2=severity** switches be used very sparingly, and only in very exceptional circumstances would you set **CLASS1** and **CLASS2** to anything less than their default settings of **ERROR** and **WARNING** respectively. In most cases for production users (ie those without direct access to CLI) the settings of **ABORT** and **ERROR** are probably more appropriate.

16.2 The /L switch

The **/L** switch has already been fairly thoroughly discussed in Chapter 15 and we do not intend to repeat that detail here, but in summary:

/L

Direct output to the Current Listfile. Note however that a physical write is not performed until the buffer is full or the Listfile is closed (by another **LISTFILE** command or process termination). For BATCH jobs output is sent to a temporary file, which on termination of the Batch Job will be Queued to the Spooler Queue **BATCH_LIST**. If you didn't have any commands with **/L** switches in the Batch Job the message "List file empty, will not be Printed" appears at the end of the **BATCH_OUTPUT** listing (which is the Batch Job equivalent of @OUTPUT).

/L=pathname

Direct output to the file whose pathname is specified. If no such file exists it is created. The file is Opened, the output appended to it, after which it is closed.

It would be sensible, in order to maintain consistency if your own macros (predominantly those whose main purpose is to do output to the console), also supported this switch in both of the above forms. The simplest way to do this is to include **%0/L%** on every command within the macro (particularly WRITE commands). However for long macros with lots of such commands it is more efficient, to include a **/L** switch on all commands and do some preliminary examination of the form of the switch supplied. This is because if the user uses the **/L=pathname** form of the switch it will run considerably slower because **pathname** is opened and closed on every command. Your macro might look something like this:

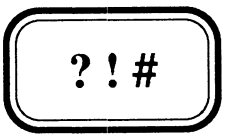
```
COMMENT User may have LISTFILE set but supplies /L=pathname
COMMENT Thus PUSH to preserve environment. See Chapter 22 for details

PUSH; PROMPT POP

COMMENT Cater for no /L

[!EQUAL,,%0/L%]
     LISTFILE @OUTPUT
[!ELSE]

     COMMENT Now Cater for /L=pathname

     [!NEQUAL,,%0/L=%]
          LISTFILE %0/L=%
     [!END]
[!END]

COMMENT Note otherwise /L so just use current Listfile

COMMENT The body of the macro now appears with /L on every command

WRITE/L This is a lot of trouble to go to if the macro only has a few
WRITE/L commands. Such as this one but there are times when it's worth
WRITE/L the effort.

COMMENT Close the Listfile - unless User's current Listfile

[!NEQUAL,/L,%0/L%]
     LISTFILE/G
[!END]

COMMENT Restore User's environment

POP
```

Note that the reason we set the LISTFILE when no /L switch is supplied is to simplify the commands in the body of the macro. If we hadn't done this we would have to include a test for the absence of the switch on each command thus:

```
     WRITE[!NEQUAL,,%0/L%]/L[!END] whatever
```

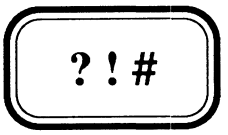
Note also that in such a situation we set it to **@OUTPUT** not **@CONSOLE**, so that our macro will also work when used by a Batch Job.

THIS PAGE INTENTIONALLY LEFT BLANK



CHAPTER 17

Understanding ACLs



CHAPTER 17

Understanding ACLs

17.1 What is an ACL?

An **ACL** is an **A**ccess **C**ontrol **L**ist, which is a list associated with a file that determines who has access to the file and what kind of access they have. Each element of an ACL consists of two parts:

1. A **Username** or Group of Usernames specified by a **Template** (ie a filename template, see Chapter 9 if you've forgotten).

2. A List of up to 5 characters specifying what access rights that User (or Group of Users) has to the file. Each such character is the first letter of the word used to describe the attribute, where these are:

   Ownership, **W**rite, **A**ppend, **R**ead and **E**xecute

   Each of these attributes is independent of the others (ie they are mutually exclusive). For example if you had **W**rite access only to a file, you would not be able to type it out (since you don't have **R**ead access), but you could write something to it. You would not however, be able to edit it with SED since SED needs to read it first.

Typically an ACL consists of a series of such Username/Attribute pairs, where it is obviously possible (and in fact very common), for a Username to match more than one of the Username Templates in the list. For example:

OP,OWARE OP+,WARE +,RE

In this example the Username OP matches all three of templates, while the username OPS.USER matches the last two, and FREDDY only matches the last one. It is fairly clear what is intended here: OP has all access rights, any other user starting with the characters OP has all rights except ownership, while everyone else is to have Read and Execute access. However consider the ACL:

OP,OWARE +.USER,RE OP+,WARE +,RE CEO_MGR,OWARE

What access does OPS.USER have, RE or WARE? To avoid any confusion you need to know what AOS/VS's rule is. This is that the first Username or Template in the list that the Username under consideration matches, is the one whose attributes are taken. Thus OPS.USER has RE access (ie Read and Execute). Any template specified after the Template **+** (The Universal Template) is rendered useless by this rule. In our example above then, CEO_MGR would also only have RE access. The general rule in constructing an ACL that grants access to a file in the way it was intended, is to have specific usernames first, followed by templates increasing in their generality. The universal template when used must always come last.

Every file including directories, has an ACL associated with it. In order to get access to a file, you must also have access to each of the directories in the Pathname to the file. For example if you have Read access to the file **:DISK1:UDD:FREDDY:FREDDYS_FILE** you must also have access to the LDU (Logical Disk Unit) **DISK1**, and the directories **UDD** (on DISK1) and **FREDDY** within that, before you can actually read it. What kind of access you need will be discussed in the next section.

17.2 The ACL attributes

We will discuss each of the five mutually exclusive attributes of **ACLs** under two headings, the general meaning, including its interpretation for non-directory **files**, and likewise for **directory** type files (which includes LDUs - Logical Disk Units, DIRs - Directories, and CPDs - Control Point Directories).

To help you understand the difference between the interpretation of the attribute when applied to an ordinary file and a directory, consider the following. A directory is a file. Files have data in them which can be **R**ead and **W**ritten. The data in a directory file is basically the list of files in it, and other details associated with those files other than their data, such as their location on the disk and their ACL. If you are allowed to Read that data, then you can find out what it is. If you are allowed to Write that data, then you can alter it.

Ownership

Files

Ownership access to a file enables you to DELETE it, RENAME it, and change its ACL. It does not enable you to look at its contents or edit it, but since you can change its ACL you can give your self the appropriate access attributes to enable you to do so.

Directories

Ownership access to a directory is the same as for any other file, with the exception of LDUs which cannot be DELETEd other than with the RELEASE command, and can only be renamed by DFMTR (The Disk Formatter).

Write

Files

Write access to a file enables you to Write to it. You could do this with say a WRITE/L command, or with any /L directed at the file for that matter. You can also do this from a program that opens the file for OUTPUT. You will not be able to Edit it, however, unless you also have Read access, because the Editor will need to be able to read it. For this reason Write access is rarely, if ever, granted without also granting Read access.

Directories

Write access to a directory enables you to Modify its data. This means you can change the data. Since its data is primarily the list of files that reside in it, you can change this list. That is you can add names to it (by CREATing files in the directory), change names in it (by RENAMing files in it), and remove names from it (by DELETing files in it). Since the ACL of each file in a directory is also part of its data, you can also change the ACL of the files in the directory. This makes you an effective Owner of every file in the directory, regardless of the ACL of the individual files. It is pointless to try to limit access to files if users have Write access to the directory.

Read

Files

Read access to a file enables you to look at the file. In terms of CLI commands, this includes to be able to TYPE, PRINT and COPY it (a fact which is often overlooked). You also need Read access to a CLI macro in order to be able to execute it, since CLI needs to be able to Read the macro file in order to interpret it. The separate attribute Execute only refers to Program (.PR) files.

Directories

Read access to a directory enables you to look at its data. This means you can find out what files are in the directory, ie you can issue a FILESTATUS command on the directory (or the !FILENAMES Pseudo Macro). You can also discover all the other information recorded in the directory entries for the files, such as their size, Elementsize and ACLs.

Execute

Files

Execute access is meaningless for all non-directory files except executable program images ie .PR files. For .PR files you must have Execute access to be able to run the program. That is to be able XEQ, EXECUTE or PROC it.

Directories

Execute access to a directory is required in order to be able to use that directory in a pathname. Thus any users who did not have execute access to :UDD would be unable to log on to the system, because they would be unable to access their initial Working Directory :UDD:[!USERNAME]. Since all pathnames begin at the system root directory (:), then +,E is a minimum requirement when setting the ACL of a system disk using DFMTR.

Append

Files

Append access has no meaning for non-directory files.

Directories

Append access to a directory enables a user to append to it. That is to be able to add filenames to it, which therefore means to be able to CREATE files in it. However once you have added the file to the directory, what you can do to it, is determined by the other attributes. In other words it enables you to add files to the directory without making you an effective Owner of the files in the Directory (which Write access to a directory does).

17.3 The DEFACL

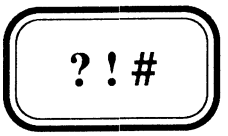
The DEFACL is your Default Access Control List which is the ACL AOS/VS associates with a file, when creating it for you. It would not make much sense for users to have a DEFACL which didn't allow them Ownership to the files they were creating. The initial DEFACL given to Users when AOS/VS initializes a process for them is [!USERNAME],OWARE, but this is usually overridden by a DEFACL command in the User's initial IPC file (startup Macro).

17.4 Superusers

When a User with the SUPERUSER privilege has it turned on, the Access Control system is effectively disabled for that user, giving him all access to all files on all disks on the system. This is very dangerous and really should not be given to any User except OP.

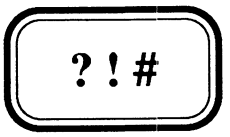
17.5 The Universal Template

Using the Universal template + as the only template in an ACL can speed up AOS/VS's ACL check. For example you could make the ACL of all the files in a software directory **+,RE**. If you needed to limit access to this software you could do so with a more complex ACL on the directory itself.



CHAPTER 18

!FILENAMES and !PATHNAME



CHAPTER 18

!FILENAMES and !PATHNAME

18.1 The Differences

The **!FILENAMES** and **!PATHNAME** Pseudo Macros are perhaps the most commonly used Pseudo Macros. They are most often used in Conditional Pseudo Macros to test for the existence or absence of a file (or group of files) eg:

```
[!EQUAL,,[!PATHNAME FRED]]
     WRITE FRED doesn't exist
[!ELSE]
     WRITE FRED exists in [!EDIRECTORY [!PATHNAME FRED]]
[!END]
```

They are sometimes used interchangeably, but they are in fact quite different. If you fully understand these differences, you will have little difficulty deciding which one to use. These are summarized in the table below:

Attribute	!FILENAMES	!PATHNAME
Number of arguments	Takes one or more	Takes one only
Templates in arguments	Allowed	Not allowed
Links in directory portion of arguments	Not resolved	Resolved
Searchlist	Ignored	Traced
Expansion of filename	Prefixed by =	Full pathname

18.2 The Implications

The main implications of these differences are:

1) **[!FILENAME file]** will expand to **=file** if and only if file exists in the Current Working Directory, whereas **[!PATHNAME file]** will expand to the full pathname to file, if file exists in the Current Working Directory or in any of the directories on the searchlist. (Note our use of !FILENAME rather than !FILENAMES whenever only one file can possibly be expanded). CLI in fact makes a point of this difference by returning = in the expansion. (Where = is a special AOS/VS prefix character to stand for the Working Directory). If you wish to expand the full pathname to a file but only if it exists in the working directory (ie you want to stop !PATHNAME tracing the searchlist) you can with **[!PATHNAME =file]**.

2) If **dir** is a Link file **[!FILENAMES dir:template]** will always return null regardless of how many files in dir match the template. This presents us with a problem since **!PATHNAME** doesn't take a template argument, but there is a solution. To expand the list of files one simply resolves the link first using !PATHNAME as follows: **[!FILENAMES [!PATHNAME dir]:template]**.

18.3 More about !FILENAMES

You should have noticed our preference for the use of the negated test for absence to test if a file exists, thus:

```
[!NEQUAL,,[!FILENAME file]]
```

rather then the positive test for presence:

```
[!EQUAL,=file,[!FILENAME file]]
```

Note that the = in the Left-Hand-Side expression is essential or the condition could never be satisfied. However if you wish to use a single condition (because you need an ELSE branch) to test for the existence of more than one file, you must use the positive expression eg:

```
[!EQUAL,(=file1 =file2),([!FILENAMES file1 file2])]
```

It is important here that you list every file explicitly as arguments to !FILENAMES, rather than a template such as **file***, to guarantee the order of expansion. Otherwise you will be left to the whimsy of AOS/VS's hashing algorithm which may expand the files in a different order to the order you listed them on the Left-Hand-Side, which would of course lead to the two strings not matching. Notice the use of brackets to tell CLI to treat the lists as a single argument. An alternative form of such an expression that doesn't require the brackets (which we used as an example in Section 13.1) is:

```
[!EQUAL,=file1=file2,[!FILENAME file1][!FILENAME file2]]
```

While on the subject of order of expansion of the filenames returned by **!FILENAMES**, it is worthwhile noting that !FILENAMES expands the **#** (crunch) template in a top down fashion, whereas all CLI commands process it in a bottom up fashion. This is not always desirable. For example let's say someone with write access to your directory, moved a whole bunch of files and directories, **HIS_STUFF:#**, into your directory but left his ACLs on them (which included the Universal Template +,RE), so you couldn't change them. You have Write access to your own directory, so you can change the ACLs of any files in it, so you might try:

```
) ACL/D HIS_STUFF:#
```

to set your DEFACL onto all these files. However because the ACL command expands the template # from the bottom up this will fail, because you don't yet have Write access to HIS_STUFF or its subdirectories. However because !FILENAMES expands from the top down, ie HIS_STUFF first, you can do it as follows:

```
) ACL/D ([!FILENAMES HIS_STUFF:#])
```

Note that if you didn't have at least R access to HIS_STUFF and its subdirectories, this technique would not work either, since the !FILENAMES expansion would fail. Programmers for systems with INFOS files may well find this useful.

18.4 !FILENAMES and CLI out of memory errors

CLI out of memory errors can from time to time be caused by !FILENAMES. This happens when the list of files matching the supplied template is simply too big for CLI to fit it in its command processing buffer. The problems associated with such an error were discussed in Chapter 6 on recursion and will not be repeated here. We will endeavour to give you tips on how to avoid these errors.

Let us assume that the expansion of **[!FILENAMES :somedirectory:#]** causes a CLI out of memory error and that we want to do some sort of processing with each file expanded, where that processing is handled by a subordinate macro that takes one argument thus:

```
PROCESSINGMACRO ([!FILENAMES :somedirectory:#])
```

There are a number of ways we might avoid the error.

1) In this expansion **:somedirectory** is expanded along with each filename and the solution can sometimes be as simple as:

```
PUSH; PROMPT POP
DIRECTORY :somedirectory
PROCESSINGMACRO ([!FILENAMES #])
POP
```

2) If the expansion above still causes a CLI out of memory error, then we have to break up the expansion. Once CLI has interpreted a command it no longer occupies memory in the command buffer so we can often avoid the error by breaking up the template eg:

```
PUSH; PROMPT POP
DIRECTORY :somedirectory
PROCESSINGMACRO ([!FILENAMES -:#])
PROCESSINGMACRO ([!FILENAMES +\-:#])
POP
```

We can of course split this up even further if necessary with templates like A+:# B+:# C+:# .... Z+:# +\A+\B+\C+\....\Z+:#. However you would have to make sure that PROCESSINGMACRO catered for the case where it received no arguments, which would usually be done as follows:

```
[!NEQUAL,,%1%]
     body of macro processing %1%
[!END]
```

There may be one problem subdirectory that still causes CLI out of memory errors, in which case you would have to break it up eg:

```
PROCESSINGMACRO ([!FILENAMES +\problemdir:#])
PROCESSINGMACRO ([!FILENAMES problemdir:-:#])
PROCESSINGMACRO ([!FILENAMES problemdir:+\-:#])
```

Both of these methods require prior knowledge of what is to be expanded and which part of the expansion causes the problem. If the template is an argument to our macro we cannot know in advance when or if we are going to have such a problem.

In such situations we generally have a pretty good idea if there is ever likely to be such a problem, and I am afraid there is only one generalized solution to the problem, and it is non trivial but it does work. What's more, the generalized solution will work even with the original template without having to first DIR into :somedirectory.

The basis of the generalized solution is to eliminate the **!FILENAMES** Pseudo Macro altogether and replace it with a **FILESTATUS** command, with output sent to a listfile. The listfile is then edited, using a pre-prepared editing macro. This editing macro places the macro call **PROCESSINGMACRO,** at the beginning of each line. If one then attempted to invoke the listfile as if it were a macro, you would still get a CLI out of memory error because the size of the listfile would be such that CLI would not be able to read it into its buffer, so we appear to be no better off. However the editing macro has to do a bit more work than this. Every so many lines, let's say 500, it inserts a command to invoke another macro (or listfile) in which the next 500 lines are placed. The listfile is then closed after this command and output directed to the new listfile. This process is repeated until there are no more lines. In this way CLI never has more than 500 commands in its command buffer at once.

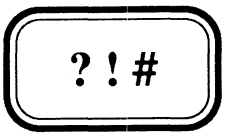
Our macro might look something like this:

```
DELETE/2=IGNORE ?PROCESS+.CLI
FILESTATUS/L=?PROCESS.CLI/CPL=1/NHEADER %1%
X SPEED/I=PROCESS.SPD
?PROCESS_1
DELETE ?PROCESS+.CLI
```

The difficult part is of course coding the editing macro, which in the above example is a SPEED macro. **SPEED** is an obscure, difficult to learn, but extremely powerful editor supplied with AOS/VS, which can be used to write extremely flexible editing programs. Basically the SPEED macro would do the following:

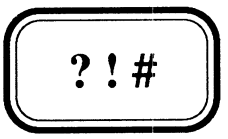
```
1 - Set SPEED's internal buffersize to 500 lines
2 - Open ?PROCESS.CLI for input
3 - Set internal numeric variable zero (V0) to 1
4 - For each buffer (500 lines or less) until there are no more:
     a - Open ?PROCESS_<ascii equivalent of V0>.CLI for output
     b - Read in the next buffer
     c - Insert PROCESSINGMACRO, at the beginning of every line
     d - If buffer has 500 lines
                - increment variable zero
                - append ?PROCESS_<ascii equivalent of V0>
     e - Output contents of buffer
     f - Close output file
     g - Clear buffer
5 - Close input file
6 - Exit.
```

We discuss SPEED Macros briefly in Appendix A, but this won't be in sufficient detail to enable you to code the above SPEED macro. However there is no reason why you couldn't write a little editing program with whatever high-level programming language you're familiar. A lot of trouble we know, but it's the only way to guarantee no CLI out of memory errors in situations like the one described.



CHAPTER 19

User Pseudo Macros



CHAPTER 19

User Pseudo Macros

19.1 What is a User Pseudo Macro

A User Pseudo Macro is a Pseudo Macro you write yourself, of course. Those of you who have not yet discovered the concept of User Pseudo Macros (which we suspect will be most of you), will probably find this Chapter alone will make your purchase of this Manual worthwhile. The ability to write your own Pseudo Macros is such a powerful tool, that once you start writing them, before long you'll be wondering how you ever wrote useful macros without them.

Since we have already given you the definition of a Pseudo Macro twice (Chapters 5 & 8) we will not repeat it here. We will define a User Pseudo Macro as a Macro that does not contain any commands, but simply returns, in place of itself, some information. This is after all what the majority of CLI's Pseudo Macros do.

In order to write a User Pseudo Macro (UPM) there are really only two points you need to bear in mind:

1) A UPM contains no commands, not even one, and therefore must not contain any command terminators. Thus it must contain, no Semi-Colons, Form-Feeds, Carriage-Returns or Newlines. **NEWLINE** characters may appear (and almost always do, after all SED won't let you have a file without at least one) in a User Pseudo Macro, provided that each and every one is immediately preceded by an Ampersand (**&**), to tell CLI that this command (Or non-command as it is in this case), is continued on the next line.

2) Since Pseudo Macros are not commands or macros, and are therefore invoked from within the command line of some actual command or macro, some means is required to tell CLI that the following string of characters identifies a UPM you want it to invoke for you. Fortunately there is a very straightforward way which is simply to enclose the UPM and its arguments in square brackets (which, as discussed in Chapter 3, just happens to be the most efficient way to invoke a macro anyway).

Summarizing then the rules are:

- UPMs must not contain command terminators

- UPMs must be invoked by enclosing them in square brackets

The best way to understand what a UPM is, and how to write one, is to study some examples. The examples presented include a simple one, a suite of simple ones, a complex one, one that calls others, and a recursive one. The one unifying feature of them all is that you should be able to put all of them to immediate use.

As mentioned in the Preface on Conventions all Shalless Software Pseudo Macros begin with $$ to distinguish them from macros.

19.2 A Simple UPM

In Chapter 10 on Input we discussed the problem of multi-word responses to !READ inside a conditional Pseudo macro, and how we overcame the problem by surrounding the compared strings in round brackets. Another method is to write an incredibly simple User Pseudo Macro that returns the first argument from the list of those supplied. We will call this UPM **$$ARG1** for obvious reasons:

File $$ARG1.CLI

```
%1%&
```

Thus using the example from Chapter 10, we could write:

```
[!EQUAL,OK,[$$ARG1 [!READ/S OK to proceed ? (OK NO) {NO} ]]]
```

Or if we want to get really clever:

```
[!EQUAL,Y,[$$ARG1 [!EXPLODE [!READ/S Do you wish to continue? ]]]]
```

Which will accept each of following as positive responses:

```
Y
YES
yeah
yep
You betcha
```

In fact anything that starts with Y will be treated as an affirmative response.

19.3 A Suite of Simple UPMs

In Chapter 11 when discussing the !ASCII pseudo macro, we pointed out how its use could make macros unintelligible, and that we would in this chapter show you how to write such code in an intelligible and more importantly maintainable way. The answer of course is with User Pseudo Macros. Consider the following:

```
$$BELL.CLI              [!ASCII 207]&
$$TAB.CLI               [!ASCII 211]&
$$NL.CLI                [!ASCII 212]&
$$ERASE_EOL.CLI         [!ASCII 213]&
$$ERASE_PAGE.CLI        [!ASCII 214]&
$$CR.CLI                [!ASCII 215]&
$$BLINK.CLI             [!ASCII 216]%-%[!ASCII 217]&
$$UNDERLINE.CLI         [!ASCII 224]%-%[!ASCII 225]&
$$DIM.CLI               [!ASCII 234]%-%[!ASCII 235]&
$$IN_ROUND.CLI          [!ASCII 250]%-%[!ASCII 251]&
$$COMMA                 [!ASCII 254]&
$$IN_ANGLE.CLI          [!ASCII 274]%-%[!ASCII 276]&
$$IN_SQUARE.CLI         [!ASCII 333]%-%[!ASCII 335]&
```

These correspond exactly to the list in 11.2, of common !ASCII sequences (with one exception which we will discuss in the next section) used with the CLI WRITE command.

We could then use them in CLI WRITE commands thus:

```
WRITE [$$DIM You must enclose $$DIM and its arguments in[$$NL]square&
      brackets when you invoke it. eg:] [$$IN_SQUARE $$DIM text]
```

which would write out:

You must enclose $$DIM and its arguments in
square brackets when you invoke it. eg: **[$$DIM text]**

Notice that although $$DIM appears three times in the command only the one marked off with the square brackets actually invoked the Pseudo Macro. Notice also that at the end **$$DIM text**, appears to be in square brackets (since they were written out), but that $$DIM was not invoked, because the displayed square brackets are not real square brackets, and so CLI sees them as just some other characters to WRITE out. You could of course extend the suite of such macros to include anything you found useful.

You should however be aware that the command above will take noticeably longer than the !ASCII equivalent, because of all the Pseudo Macro expansions (each of which requires a file to be read), but we believe the maintainability of the code is more important. Frequently however, it is better to type out a file or use CREATE/M @OUTPUT, as explained in Chapter 11.

19.4 A Complex UPM

We deliberately omitted the Cursor Positioning !ASCII sequence, not because it can't be done, but because it is considerably more difficult. However once we actually go to the trouble of coding it, it probably turns out to be the most useful of the suite.

The User Pseudo Macro **$$CURSOR** will expand to the correct !ASCII sequence to position the cursor at the specified decimal line and column, and will take care of any Line/Col 10 problems itself.

Format: **[$$CURSOR/LINE=r/COL=c]**

File $$CURSOR.CLI

```
[!ASCII 220 &
[!OCTAL [!UADD 127 [!UNE,10,%/COL=%]%/COL=%[!ELSE]9[!END]]] &
[!OCTAL [!UADD 127 [!UNE,10,%/LINE=%]%/LINE=%[!ELSE]9[!END]]]&
[!UEQ,10,%/COL=%] 230[!END][!UEQ,10,%/LINE=%] 232[!END]]&
```

This User Pseudo Macro adds 127 to the value supplied by the Caller because we need to add 200 octal (128 decimal) to force the character to be sent to the terminal, and then subtract one because Column/Line 1 requires a value of zero. If 10 is one of the values then we use 9 instead and move the cursor down or right as appropriate later. This is then converted to octal since !ASCII requires octal arguments. Note that every line ends with an ampersand preceding the NEWLINE. eg:

```
@NULL [!READ/S [$$CURSOR/LINE=23/COL=10][$$DIM Hit&
      NEWLINE to continue] [$$BELL]]
```

19.5 A UPM that calls other UPMs

It is common practice to Date-Time stamp certain kinds of files in their filename, particularly Log files. The most common format we have seen is name_DDMonYY_HHMMSS.LOG eg: MYJOB_07FEB89_070043.LOG, although another common and slightly better format is name_YYMonDD_HHMMSS.LOG. It is better because then FILESTATUS/AS/S will at least list the files in chronological order within each month.

To keep it simple we will assume that we only want to use the Date and create a file **MYJOB_YYMonDD.LOG**. Thus we might write our macro thus:

```
GETDATE [!EXPLODE [!DATE]]
LISTFILE MYJOB_[!STRING].LOG
..
```

where

File GETDATE.CLI

```
STRING %8%%9%%4%%5%%6%%1%%2%
```

however if GETDATE were a Pseudo Macro, it could be coded thus:

```
%8%%9%%4%%5%%6%%1%%2%&
```

and our macro could become:

```
LISTFILE MYJOB_[GETDATE [!EXPLODE [!DATE]]]
```

However if we're going to do it with a Pseudo Macro why not go the whole hog and remove the burden of including the !EXPLODE from the caller (and for that matter !DATE which we'll make the default), and while we're at it let's make the format **YYMMDD**, so that we get filenames listed in full chronological order. Thus:

File $$YYMMDD.CLI

```
[$YYMMDD.1 [!EXPLODE [!EQUAL,,%1%][!DATE][!ELSE]%1%[!END]]]&
```

File $$YYMMDD.1.CLI

```
%8%%9%[$$MM %4%%5%%6%]%1%%2%&
```

File $$MM.CLI

```
[!EQUAL,%1%,JAN]01[!END]&
[!EQUAL,%1%,FEB]02[!END]&
[!EQUAL,%1%,MAR]03[!END]&
[!EQUAL,%1%,APR]04[!END]&
[!EQUAL,%1%,MAY]05[!END]&
[!EQUAL,%1%,JUN]06[!END]&
[!EQUAL,%1%,JUL]07[!END]&
[!EQUAL,%1%,AUG]08[!END]&
[!EQUAL,%1%,SEP]09[!END]&
[!EQUAL,%1%,OCT]10[!END]&
[!EQUAL,%1%,NOV]11[!END]&
[!EQUAL,%1%,DEC]12[!END]&
```

Then our macro could simply be coded as:

```
LISTFILE MYJOB_[$$YYMMDD].LOG
```

Note that subordinate User Pseudo Macro calls must also be enclosed in square brackets and that all lines in all subordinates must have ampersands preceding any NEWLINEs. The extension to include the time in say a Pseudo Macro called **$$DATETIME**, should be fairly obvious, and would also make a good exercise for you. However there are traps for young players in the use of such Pseudo Macros, for example consider the following:

```
(DELETE/2=IGNORE CREATE LISTFILE) MYJOB_[$$YYMMDD].LOG
```

The $$YYMMDD Pseudo Macro is invoked once for each command. Inefficient though this may be, it is not its most serious problem. If for example this command is started a tick before midnight, then any two of the commands could run on different days and would therefore result in different expansions, which is clearly not what is intended. If we were using the $$DATETIME UPM, all three may run at a different time. It would be better to use the STRING, a subordinate macro or:

```
LISTFILE MYJOB_[$YYMMDD].LOG
(DELETE/2=IGNORE CREATE) [!LISTFILE]
```

19.6 A Recursive UPM

User Pseudo Macros can be recursive just like real macros. For example consider the UPM **$$ARGS**, which returns a count of the number of arguments passed to it. It could be coded thus:

File $$ARGS.CLI

```
[!EQUAL,,%1%]0[!ELSE][!UADD 1 [$$ARGS %2-%]][!END]&
```

The UPM returns 0 if there are none, and returns 1 plus however many there are when one of the arguments is dropped, otherwise. Note also that we have used the perhaps invalid assumption that a null argument means there aren't any more. However we have discovered through frequent use of this User Pseudo Macro that this assumption is in fact valid, for every meaningful use to which we have put it.

If the number of arguments is large such code can become horribly inefficient. This is particularly the case if you put it to one of its most practical uses, which is to count the number of files in a directory or matching a particular template. For example say we wanted to count the number of MYJOB_YYMMDD.LOG files:

```
VAR0 [$$ARGS [!FILENAMES MYJOB_******.LOG]]
```

If there were a hundred such files, you might begin to wonder if your machine had crashed. On the other hand though there is really no practical non-recursive solution to the problem, since a Pseudo Macro that simply tested every possible result up to some arbitrary upper limit would be awfully cumbersome. A fairly sensible compromise however between these two extremes is achieved with the following:

File $$ARGS.CLI

```
[!EQUAL,,%1%]0[!ELSE]&
[!EQUAL,,%2%]1[!ELSE]&
[!EQUAL,,%3%]2[!ELSE]&
[!EQUAL,,%4%]3[!ELSE]&
[!EQUAL,,%5%]4[!ELSE]&
[!EQUAL,,%6%]5[!ELSE]&
[!EQUAL,,%7%]6[!ELSE]&
[!EQUAL,,%8%]7[!ELSE]&
[!EQUAL,,%9%]8[!ELSE]&
[!UADD 9 [$$ARGS %10-%]]&
[!END][!END][!END][!END][!END][!END][!END][!END][!END]&
```

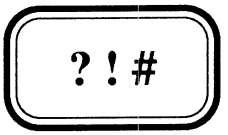
We have also found $$ARGS useful in counting the number of characters in a user response, where we have some restriction on its length. However to simplify it we have simply built another User Pseudo Macro that uses $$ARGS thus:

File $$LENGTH.CLI

```
[$$ARGS [!EXPLODE %1%]]&
```

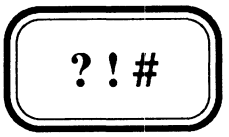
We hope these examples have whetted your appetite for User Pseudo Macros. UPMs can't always be neatly spaced out and great care must be taken with the location of spaces, where unwanted ones can easily become part of the expansion. In particular do not try to indent lines to make conditionals and their !ELSEs & !ENDs line up, as you would in a macro, as this will invariably lead to unwanted spaces expanded in the result.

All these UPMs plus a number of other useful ones we have developed (including $$DOW to return the Day of the Week in words, which is described in Chapter 30) are available, on the Tape you may have purchased with the manual. If you did not and you want to save yourself the trouble of typing them in (perhaps incorrectly), a Tape of all the examples used in this Manual may be separately purchased. Contact your supplier for details. A Tape should be supplied free with every order of 10 or more Manuals.



CHAPTER 20

Understanding Link Files



CHAPTER 20

Understanding Link Files

20.1 What is a Link File?

A link file is a file that links to another file. In other words it is a redirection pointer, stating that attempts to access it, should be directed at some other file. Accessing a link file is just like dialling an extension that has been diverted to someone else's number. In the same way that the number diverted to, may in turn be diverted elsewhere, the "resolution pathname" of the link file may in turn be itself a link file causing further redirection of the attempt to access the original link file. Link files do not have any data associated with them since their "resolution pathname" is held in the directory entry. They do not have their own Access Control List, since there is no data in them to access. A link file can be created with the CLI command:

CREATE/LINK linkfilename resolutionpathname

All CLI commands that attempt to access a link file will be redirected to the resolution pathname, with the exception of **DELETE** and **RENAME**. These commands will delete/rename the link file itself. When specified as or as part of the templates in a **MOVE**, **DUMP** or **LOAD** command any link files, are themselves moved, dumped or loaded. However if specified as the destination argument of such commands the link is "resolved" and its resolution pathname used.

The link files in a directory and their resolution pathnames can be displayed using the **FILESTATUS** command and either the **/ASsortment** or **/LINKname** switch along with the **/TYPE=LNK** switch.

20.2 When and Why they are used

The CLI **MOUNT** command creates a link to the actual physical tape-drive upon which the operator mounted a user's tape. This enables you to write drive independant tape handling macros, where after issuing the command, **MOUNT MYTAPE Operator Message**, a linkfile **MYTAPE** is created in **:UDD:[!USERNAME]**. You then refer to the tape as **:UDD:[!USERNAME]:MYTAPE** (instead of say **@MTDO**).

The most often cited use for a linkfile as a means of abbreviating long frequently used pathnames, is in reality one of its less frequent uses, restricted mainly to the relatively small community of lazy two fingered typist computer programmers.

Most frequently they are used by System Managers to effectively relocate files and directories onto other disks, to help them manage the workload across their System's disks. Linkfiles enable them to do this in a most straightforward manner, because the software still sees the files as existing in their original location. It generally means that a move of a file from one disk to another can be effected without any need to change any of the software that accesses them. In these situations the Linkfile is generally a link to a file of the same name in another directory.

For example an application system may require that the data base of files upon which it operates lives in a directory called **:DATA**. The System Manager may decide that he doesn't want this data base of files on his System Disk, so he makes **:DATA** a link to **:DISK1:DATA** so that this data base of files is located on **DISK1**. The application system is none the wiser.

After some months of operation of this application, the System Manager discovers that the data base has grown in size to such an extent, that before long it is going to blow the capacity of **DISK1**. He buys a new disk. For want of a better name let's call it **DISK2**. One weekend when no users are around he decides he wants to split the workload 50/50 across the two disks DISK1 and DISK2. Let's say he has done his homework properly and worked out that this can best be achieved by moving all the files that start with **F** onto DISK2 and leaving the rest where they are. He makes a Tape Backup from **:DISK1** of **DATA:#** before he starts just in case he stuffs something up. He then DIR's into **:DISK2** and loads from that Backup **DATA:F+:#**. After verifying that all the files starting with **F** were loaded on **DISK2** and that they are all the correct size, he then deletes **F+:#** from **:DISK1:DATA**. The sole remaining task now is to create a set of link files in **:DISK1:DATA** for all the files starting with **F** that are now located in **:DISK2:DATA**. If the number of files is large this can be a tedious task to do manually and there is every likelihood that he will miss one. By clever use of certain Pseudo Macros and brackets this can all be done in one fell swoop, guaranteeing that none of the files is missed:

```
) DIRECTORY :DISK1:DATA
) CREATE/LINK &
<,:DISK2:DATA:>([!EFILENAME [!FILENAMES [!PATHNAME :DISK2:DATA]:+]])
```

or bearing in mind a possible CLI out of memory error:

```
) DIRECTORY :DISK2:DATA
) CREATE/LINK :DISK<1,2>:DATA:([!EFILENAME [!FILENAMES +]])
```

When the Application Software comes back to life the next Monday, once again it is none the wiser, except perhaps that it ought to be running a little faster than it was on Friday (Users of course never notice performance improvements, just performance degradations). As far as it is concerned the entire data base of files still lives in a directory called **:DATA**. We will leave it as an exercise for you to work out how the above command lines achieve the desired aim, and trust that once you have, you will find the technique useful.

20.3 Different Kinds of Link Files

There are really three different kinds of link files distinguished from each other by the form that the resolution pathname takes. We call these three types, Absolute Links, Directory Relative Links, and Linkfile Relative Links.

Absolute Links

These are link files where the resolution pathname is a path to an explicit invariant file. All link files where the resolution pathname begins at the root directory (:) or with the prefix **@** (:PER:) are absolute links. In practice most link files are absolute links. eg:

```
CREATE/LINK EGG :FOOD:HUMPTYDUMPTY
```

or

```
CREATE/LINK EGG @HUMPTYDUMPTY
```

Directory Relative Links

These are link files where the resolution pathname is a path to a file relative to the Current Working Directory. All link files where the resolution pathname begins with the special prefix characters = ([!DIRECTORY]) or ^ ([!EDIRECTORY [!DIRECTORY]]) are directory relative links, thus:

CREATE/LINK EGG =FOOD:HUMPTYDUMPTY

will link to :FOOD:HUMPTYDUMPTY if the working directory is :, or :UDD:WHOEVER_U_R:FOOD:HUMPTYDUMPTY if you are positioned in your initial working directory and your username is WHOEVER_U_R. Note that this remains the case wherever the link file EGG resides.

Linkfile Relative Links

These are link files where the resolution pathname is a path to a file relative to the directory in which the link file exists. All link files where the resolution pathname begins with a simple filename (no special prefix character) are linkfile relative links. thus:

CREATE/LINK EGG FOOD:HUMPTYDUMPTY

will link to :FOOD:HUMPTYDUMPTY if the link file EGG exists in :, or :UDD:WHOEVER_U_R:FOOD:HUMPTYDUMPTY if EGG exists in :UDD:WHOEVER_U_R. Note this remains the case no matter which is your current working directory.

To help you recognize the significance of the different types let's think again about our application that thinks everything resides in **:DATA**. Let's say one of the options in the system operates on a file called **THIS_FILE_GROWS** and that it sometimes does this out of control. Let us also assume, since it doesn't begin with F that it actually exists in **:DISK1:DATA**. Now the System Manager decides to put some control on its growth by putting it in a subdirectory called **CANT_GROW_TOO_BIG**. He could:

```
) DIRECTORY :DISK1:DATA
) CREATE/MAX=10000 CANT_GROW_TOO_BIG
) MOVE/V CANT_GROW_TOO_BIG THIS_FILE_GROWS
  =THIS_FILE_GROWS
) DELETE THIS_FILE_GROWS
) CREATE/LINK <,:DISK1:DATA:CANT_GROW_TOO_BIG:>THIS_FILE_GROWS
```

He has created an absolute link in :DISK1:DATA to where THIS_FILE_GROWS really exists. On the other hand he might decide that since some time he might want to link **:DATA** to some other disk than DISK1, say DISK3, that he should make the link file disk independent by:

```
) CREATE/LINK <,:DATA:CANT_GROW_TOO_BIG:>THIS_FILE_GROWS
```

This is equivalent in the current environment, but is better because he won't have to re-create the link if he ever links :DATA elsewhere. :DATA:THIS_FILE_GROWS is in both cases however an absolute link.

So! What's the point? You might ask. Well let's say at some later time he moves **THIS_FILE_GROWS** and **CANT_GROW_TOO_BIG:#** from **:DISK1:DATA** to **:DISK2:DATA** and creates a link in **:DISK1:DATA** to **THIS_FILE_GROWS** giving us:

```
In :DISK1:DATA (ie :DATA)

THIS_FILE_GROWS      LNK  :DISK2:DATA:THIS_FILE_GROWS

In :DISK2:DATA

THIS_FILE_GROWS      LNK  :DATA:CANT_GROW_TOO_BIG:THIS_FILE_GROWS
CANT_GROW_TOO_BIG    CPD  (containing the real THIS_FILE_GROWS)
```

Unfortunately the application can no longer access THIS_FILE_GROWS because CANT_GROW_TOO_BIG doesn't exist in :DATA any more. We could rectify the problem by creating a link file for CANT_GROW_TOO_BIG in :DATA (ie :DISK1:DATA) or had we made THIS_FILE_GROWS a "linkfile relative" link in the first place we wouldn't have had a problem. That is:

```
) CREATE/LINK <,CANT_GROW_TOO_BIG:>THIS_FILE_GROWS
```

Then after the above moves we would have ended up with:

```
In :DISK1:DATA (ie :DATA)

THIS_FILE_GROWS      LNK  :DISK2:DATA:THIS_FILE_GROWS

In :DISK2:DATA

THIS_FILE_GROWS      LNK  CANT_GROW_TOO_BIG:THIS_FILE_GROWS
CANT_GROW_TOO_BIG    CPD  (containing the real THIS_FILE_GROWS)
```

In which case the application is able to access THIS_FILE_GROWS because the link is relative to the directory in which the link resides, and thus when the link file and its resolution file are moved around together the link does not have to be changed.

20.4 Link Files and ACLs

As stated previously link files do not themselves have ACLs. Any attempt to reference the file results in an evaluation of the ACL of the resolution pathname. When you use the ACL command to find out the ACL of a link file it will therefore tell you the ACL of the resolution pathname. Fair enough. But if you use the ACL command to change the ACL of a link file you actually change the ACL of the resolution file (provided of course you have O access to the resolution pathname or W access to the directory where it lives). You might again say fair enough, but beware. Let's say I've got link files in my own directory to files all over the place, to simplify my access to them. One day I discover some ACL problems with a bunch of files in my directory and for simplicity's sake I fix it with the command:

```
) ACL/D +
```

Unwittingly I have just changed the ACL of every single file, to which I have a link in my directory, wherever it exists. If the System Manager set things up properly I wouldn't have ACL access to enable me to change the ACL of files I shouldn't be able to, and so what I have done may not have caused too much harm.

Maybe, but what if I am the System Manager and some turkey programmer has stuffed up the ACL's on the files in his directory and he has rung me up and asked me to fix them for him. Fair enough:

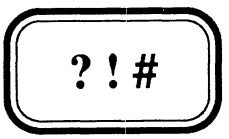
```
) SUPERUSER ON
*) DIRECTORY :UDD:TURKEY
*) ACL + TURKEY,OWARE
```

Fair enough. How was I supposed to know that **TURKEY** had a link file in his directory called **NEWS**, which linked to **:UTIL:LOGON.MESSAGE**. In case you don't know this is the file EXEC displays whenever you log on. Oops now I get phone calls from every Tom, Dick and Harry telling me they can't log on. Our personal belief is that **ACL**, like the other commands afforded owners of files, namely **DELETE** and **RENAME**, should only apply to the link file itself and not the resolution pathname, ie. we believe link files should have their own ACLs. But they don't, so beware.

By the way there is a generalized technique for getting around problems such as the one described, and it involves the use of our **$FLIST** macro described in Appendix A. $FLIST enables you to get a list of non-Link files in User Pseudo Macro format. Thus to change TURKEY's ACLs for him without attempting to change the ACL of any link files, we could:

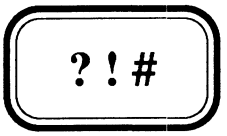
```
) SUPERUSER ON
*) DIRECTORY :UDD:TURKEY
*) $FLIST/L=NON_LINKS/TYPE=\LNK +
*) ACL ([NON_LINKS]) TURKEY,OWARE
*) DELETE NON_LINKS
```

THIS PAGE INTENTIONALLY LEFT BLANK



CHAPTER 21

/M and /I switches



CHAPTER 21

/M and /I switches

21.1 The /M switch

Tells the CLI that for the execution of this command input is to be taken from the body of the **Macro** in which the command appears. The end of input is indicated by a line containing a single close round bracket. Clearly then the **/M** switch has no meaning when used on a command that is not contained in the body of a macro. Thus it has no meaning when issued directly at your terminal. Furthermore it has no meaning (and this is less obvious) inside a macro if that macro is running as a batch job as a result of having been **QSUBMIT**ted. QSUBMIT submits a fixed command file, whose full pathname must be specified (including the .CLI extension if this is a macro), and does not recognize this as a macro. (This also means you cannot expand arguments or switches in a QSUBMITted job). Why anyone would use QSUBMIT in preference to **QBATCH** which does not suffer these restrictions, to submit a batch job is beyond us. On-line Help is available using **HELPV *M_Switch.**

21.2 The /I switch

Tells the CLI that for the execution of this command input is to be taken from the generic input file **@INPUT**. Although you may at first think this redundant since this is where input comes from anyway, you are mistaken because the commands for which it is a valid switch normally don't take input other than from the command line. The end of input is indicated by a line containing a single close round bracket. When you are logged on at a terminal this will usually (but not necessarily - since @INPUT could be a file) mean that input is taken immediately from your terminal. To terminate the input you must therefore enter a single close round bracket in response to the CLI prompt. This prompt is specially modified once you are in this mode to include an extra close round bracket. Until this extra bracket disappears from the prompt, everything you type is input to the command. For a Job running in Batch @INPUT is the temporary (.JOB) command file that CLI creates for you, when you QBATCH the job. (That is the one containing the special DIRECTORY, SEARCHLIST and DEFACL commands QBATCH puts in for you not the macro you QBATCHed.) On-line Help is available using **HELPV *I_Switch.**

21.3 The commands that take them

The CLI command switches **/M** and **/I** which basically redirect input for the command on which they are supplied, are supported by the CLI commands **CREATE**, **QBATCH** and **XEQ** (**EXECUTE** or its common abbreviation **X**). These are described in more detail in the following sections.

21.4 With CREATE

The CLI CREATE command which CREATEs files normally just builds the directory entry for the file. A file thus created has no data in it and has not even been allocated its initial file element, which is only done after the first attempt to write data to the file. However when a /M or /I switch is used, the initial file element is allocated immediately and its contents taken from either the Macro (/M) or @INPUT (/I). Data sensitive reads are performed and an End-Of-File status raised when a record contains a single close round bracket.

/M is most commonly used in Macros to create small temporary files whose contents are taken from one or more of the macro's arguments or switches (eg. SORT/MERGE command files). This works because arguments and switches are substituted with their actual values before CLI begins to interpret the macro. (This is in contrast to Pseudo Macros which are expanded during interpretation.) eg:

File $FIXSORT.CLI

```
(DELETE/2=IGNORE CREATE/M) ?[!PID]SORT.CMD
INPUT FILE IS "%1%", RECORDS ARE %3% CHARACTERS.
OUTPUT FILE IS "%1%.SORTED".
KEY 1/%2%.
SORT.
END.
)
SORT/C=?[!PID]SORT.CMD/S/O
DELETE ?[!PID]SORT.CMD
```

This macro will SORT a file of fixed length records, where the sort key is assumed to start at the beginning of each record.

Format: **$FIXSORT pathname key-length record-length**

where the sorted output file is **pathname.SORTED**.

/I is most commonly used by CLI users directly at their terminals to create short files quickly. eg:

```
) CREATE/I EDGO.CLI
))X SED/NO_ED %1\%.CLI
))%1-%
)))
)
```

where EDGO edits then runs a macro.

21.5 With QBATCH

QBATCH normally submits a Batch Job to run the single command or macro specified in the command line. You can use **/I** or **/M** to enable you to specify a series of commands or macros you want run in this Batch Job. The commands forming the Batch Job (apart from DIRECTORY, SEARCHLIST and DEFACL which CLI always puts in for you), are then taken from @INPUT or the Macro body, instead of from the command line.

21.6 With XEQ

With the EXECUTE (XEQ or X) command CLI builds a temporary file, before beginning execution of the program, containing the input read from @INPUT (**/I**) or the Macro body (**/M**). This temporary file then becomes @INPUT for the new process and is deleted on completion of the program. Looking at our $FIXSORT macro used as the example for CREATE/M we could have written it:

File $FIXSORT2.CLI

```
XEQ/M SORT/C/S/O
INPUT FILE IS "%1%", RECORDS ARE %3% CHARACTERS.
OUTPUT FILE IS "%1%.SORTED".
KEY 1/%2%.
SORT.
END.
)
```

Here we have let CLI build (and later delete) the temporary file for us. Note however that although $FIXSORT and $FIXSORT2 perform the same task when they run there is a slight difference in their appearance. $FIXSORT echoes nothing on the terminal, while $FIXSORT2, which because of the /C (rather than /C=commandfile) switch thinks input is comming from the terminal, echoes the commands there (@OUTPUT).

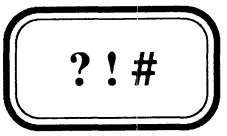
21.7 /M and Indentation

It is common and good practice to indent the commands controlled by conditional Pseudo Macros in a Macro. However beware of indenting the lines for the input for **/M** (and /I for that matter) and never indent the line containing the final close round bracket or the rest of your macro will be gobbled up and you will eventually get an "Insufficient Macro Input for /M" error because it didn't find the End-Of-File indicator. Take for example our $FIXSORT macro enhanced with a couple of validity checks on its arguments. The SORT/MERGE commands can be indented (since the white space is ignored) but not the close bracket.

File $FIXSORT.CLI

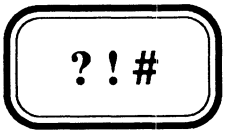
```
[!EQUAL,,[!PATHNAME %1%]]
     WRITE $FIXSORT ERROR - %1% doesn't exist can't sort it
[!ELSE]
     [!UGT,%2%,%3%]
          WRITE $FIXSORT ERROR - keylength exceeds reclength
     [!ELSE]
          (DELETE/2=IGNORE CREATE/M) ?[!PID]SORT.CMD
          INPUT FILE IS "%1%", RECORDS ARE %3% CHARACTERS.
          OUTPUT FILE IS "%1%.SORTED".
          KEY 1/%2%.
          SORT.
          END.
)
          SORT/C=?[!PID]SORT.CMD/S/O
          DELETE ?[!PID]SORT.CMD
     [!END]
[!END]
```

THIS PAGE INTENTIONALLY LEFT BLANK



CHAPTER 22

Preserving the Environment (PROMPT POP)



CHAPTER 22

Preserving the Environment (PROMPT POP)

22.1 Preserving the Environment

When writing general purpose Macros, you should endeavour wherever possible to preserve the Environment for the User on completion of your macro. The Environment is of course the set of CLI Environmental Parameters first described in Chapter 7, the complete set of which can be observed with the CLI command CURRENT. By preservation we mean leaving all these Parameters at the original values they had on entry to your macro.

The environment is most simply preserved by placing a CLI **PUSH** command at the top of the macro and a corresponding **POP** at the end. Some macros of course actually return their result through one of the Environmental Parameters (such as the STRING or a numeric VARiable), in which case complete preservation is not possible. Such macros however, should not disturb other Parameters. For example our **$NUM** macro of Chapter 10 returns its result in **VAR0**, but in addition leaves the CLI **STRING** containing the null string regardless of what it was being used for by the caller. Ideally this macro should leave the CLI STRING undisturbed. We will return to this macro by way of example later in this Chapter.

22.2 PROMPT POP

If a macro specially coded to preserve the environment terminates abnormally (through a CLASS1 or CLASS2 Error), the final POP which restores the original environment will not have been executed. Whenever this happens CLI displays **LEVEL n** to inform you to what depth you had PUSHed when the error occurred. It is perhaps even more important that the environment be restored on abnormal termination than on normal termination. Fortunately CLI maintains a **PROMPT** buffer of up to eight CLI commands that will be executed immediately prior to CLI displaying its command prompt. These commands cannot have any switches or parameters. Operators and Programmers who move around the directory structure quite a bit often place the **DIRECTORY** command in the prompt buffer, so that CLI tells them what directory they are in (their current Working Directory) every time it prompts them for a command. Those of us who don't wear watches often place the CLI command **TIME** in the prompt buffer. The CLI command **PROMPT**, when used without arguments displays the commands currently in the prompt buffer, and when used with arguments, these arguments specify the commands to be placed in the buffer.

If the CLASSn conditions are set to WARNING or IGNORE then such errors will not prevent the final POP from being executed. If they are set to ABORT, the CLI process will die and any discussion re environment preservation is pointless. If however an error occurs where the corresponding **CLASSn** has been set to **ERROR** then all macro input is discarded and the user presented with the CLI Prompt. Thus by placing the command **POP** in the prompt buffer, before presenting the User with its command Prompt, it will **POP** the previous set of environmental parameters from the stack. This also restores the Prompt buffer to its original value because it is also a CLI Environmental Parameter. Thus CLI general purpose macros are often written:

```
PUSH; PROMPT POP
body of macro
POP
```

If you need to PUSH more than once in a macro, (and if you've absorbed everything we've discussed so far this ought to be rare) there is a common misunderstanding about which we would like to make you aware.

Most CLI macro programmers have seen the above trick for preserving the environment in somebody else's macro and adopted it religiously, so that every time they code a **PUSH**, they actually code **PUSH; PROMPT POP**. Actually this is unnecessary because the new environment's initial values are the same as the previous and therefore the PROMPT buffer already has the POP command in it. However if you are not certain what the PROMPT is (in say a subordinate macro called by several others, where you are not sure whether or not the caller has already done this), there can be no harm done by re-setting it.

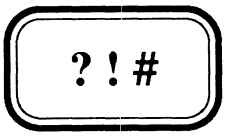
As you will see in Chapter 28 (on Initial IPC Macros) frequently the original environment will have a Prompt buffer containing **BYE** (to prevent Users getting CLI access). It is not necessary to know whether or not the Prompt buffer contains a BYE command as long as you include a PROMPT POP after a PUSH. This is because the POP whether through normal (via POP) or abnormal (via PROMPT POP) termination, will restore the original environment which has a PROMPT of BYE and the BYE is thereby executed before CLI attempts to present the user with its normal round bracket prompt.

22.3 An Example

We will now return to our **$NUM** macro and show you how to code it so as to avoid destroying the User's CLI STRING. If we were to simply put a PUSH at the top and a POP at the end, not only would we be breaking our own rules about PUSH and POP in recursive macros, but we would also discard the user's response in VAR0. The solution is to set VAR0 after we have POPped and in so doing we can leave $NUM unchanged but have an environment preserving version called, say **$NUMBER**. This is done with a recursive call and the special switch **/ANSWER=n** which is an internal switch that would not of course be mentioned in any User documentation:

File $NUMBER.CLI

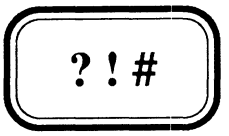
```
[!EQUAL,,%0/ANSWER=%]
      PUSH; PROMPT POP
      $NUM%0/% %1-%
      %0\%/ANSWER=[!VAR0]
[!ELSE]
      POP
      VAR0 %0/ANSWER=%
[!END]
```



CHAPTER 23

Temporary Files and Multi-User Macros

23.1 Multi-User Macros



CHAPTER 23

Temporary Files and Multi-User Macros

23.1 Multi-User Macros

23.1 Multi-User Macros

General purpose CLI Macros should be written in such a way that they can be used simultaneously by many users. Some of these users may well be two different people logged on under the same Username (although from a System security point of view this is a practice of which we take a dim view - in Chapter 28 we discuss a method through a CLI macro of preventing Users logging on twice and it actually serves to highlight a number of the points we make in this Chapter). It is essential that such macros be sufficiently robust that they do not crash when more than one person is using them at the same time.

Temporary files created by macros for their internal use, and deleted at the end of processing, require particular attention in this regard. You must ensure that temporary files are given a unique name, otherwise one user may delete the file straight after another user has created it, but before that user has had a chance to use it. Most temporary files are given an obscure name to prevent clashes with existing file names, and these frequently begin with the special filename character **?** and end with **.TMP**. The sole reason this practice has been adopted is that the disk fixing utility **FIXUP** will delete all such files.

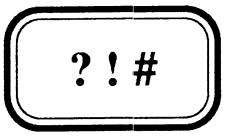
However obscure you make the name of a temporary file, it will not be unique if two users run the macro at the same time. You need some additional qualifier in the temporary file's name. If you can guarantee that no user can be logged on twice (beware of Batch Jobs) then **[!USERNAME]** may be sufficient. Note however that if Username is unique and you can guarantee that the User's Working Directory is the default of **:UDD:[!USERNAME]** then there is no need for temporary files to be unique since they are always being created in a unique directory. Otherwise a Date-Time stamp can often be useful (using say our User Pseudo Macro **$$DATETIME**), but even this doesn't guarantee uniqueness. The most common and easiest to use qualifier is **[!PID]**. PID stands for **P**rocess **ID**entifier, and at any point in time is guaranteed unique. The Username/Simple-Process-Name pair is also a unique identifier of a process but is less accessible in CLI because there is no Pseudo Macro to return the Simple-Process-Name.

Our **$FIXSORT** macro of the Chapter 21 where we created (and deleted at the end of processing) such a temporary file using [!PID] as the qualifier highlights the point we have been making. Recapping

```
(DELETE/2=IGNORE CREATE/M) ?[!PID]SORT.CMD
INPUT FILE IS '%1%", RECORDS ARE %3% CHARACTERS.
OUTPUT FILE IS "%1%.SORTED".
KEY 1/%2%.
SORT.
END.
)
SORT/C=?[!PID]SORT.CMD/S/O
DELETE ?[!PID]SORT.CMD
```

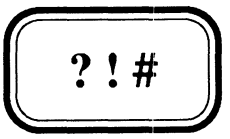
The temporary file is first deleted (in case a previous run crashed leaving the file on disk), then created, used and finally deleted. Two users can safely run $FIXSORT without it crashing when they are running in the same directory. Well almost! If they both decided to use $FIXSORT to sort the same file at the same time, they would clash over their use of the sort output file **%1%.SORTED**.

You will notice in some of our examples in the following chapters that we often specify a temporary filename with a full pathname, such as **:UDD:[!USERNAME]:?[!PID]name.CLI**. This is a worthwhile precaution that guarantees that the temporary file can be created, since almost without exception all users have **W**rite access to their **:UDD:[!USERNAME]** directory. This enables the macro to run (at least with respect to its temporary files) in any Working Directory regardless of the user's access to that directory. It is not uncommon for users to run macros in directories to which they only have **RE** access.



CHAPTER 24

The PROCESS Command



CHAPTER 24

The PROCESS Command

24.1 What is a PROCESS?

A process is basically a Job on the System that uses its resources. Each process runs a particular program which contains instructions telling the computer what it wants done. Each process is given a unique number called a Process ID or **PID** which AOS/VS uses to identify it. The process retains this PID for its entire life but releases it for re-use on completion, (with some exceptions related to Connection Management), whether normally or through an error or termination by a superior process. Each process is also given a **process name** which although not by itself unique, when combined with the Username forms a unique identifier. This enables a process to communicate with another process without having to first know what PID it was allocated. AOS/VS timeshares the processes awaiting use of the Processor (CPU) using a scheduling algorithm in an effort to give all processes the appearance of having continuous service. A single process may have several interrupt driven tasks, although most programs you write will be single task processes. EXEC for example is a multi-task process. One of its tasks sits in a loop "listening" for **CONTROL @EXEC** commands. On receipt of a command to enable a Console for example, it fires up another task to monitor the terminal, awaiting User entry of the Log-on sequence. If a Log-on sequence is correctly entered this task then issues the appropriate System Call to tell AOS/VS to initiate a process for this user. This System Call is called the **?PROC** System Call. From the CLI the EXECUTE (XEQ) command is translated by CLI into the appropriate ?PROC System Call to initiate a Process to run the Program you specified.

24.2 The PROCESS Command

The CLI PROCESS Command is the command that most directly relates to the ?PROC System Call. Its multiplicity of switches enable this command to use all (well almost) the features of the ?PROC System Call. It is not our intention to elaborate on each of these individual switches, which once you understand what the command is about, are adequately described in the on-line help (HELPV PROC) or the CLI User's Manual. Our aim is to help you understand what the command is about, which is basically creating Processes.

We shall look then at what AOS/VS needs to know about a Process in order to run it and how the PROCESS command enables us to provide that information. We would like to use non-sexist terminology and talk about parents and children, but since the product itself uses the sexist terms Father and Son we will also. The father of a Process is its creator. In the case of the CLI it is the father of any process it creates with the PROCESS command. AOS/VS needs to know the following about a process in order to run it:

What Program to Run

> This is specified by the PROCESS command's first argument. Executable programs are usually files ending in **.PR** and CLI saves you the trouble of typing this by appending it to whatever you specified. If this search is unsuccessful it will then look for a file as specified by the first argument.

What arguments (if any) does it have

These are specified as the second and subsequent arguments to the PROCESS command. Note that when the created process sees these arguments it will not see them literally but with all CLI argument delimiters reduced to a single comma.

What are the new process's generic files

These are specified through the /INPUT, /OUTPUT, /CONSOLE, /DATA, /LIST and/or /IOC (for INPUT OUTPUT and CONSOLE) switches. If any of these switches is omitted the corresponding generic file is undefined for the new process. If a switch is supplied without a value then the new process adopts the corresponding generic file of the father. The meaning should be obvious when given an explicit value. Note that the generic files including @DATA and @LIST are lifetime attributes of a process.

Must the father await completion of the son process before continuing

This is specified with the /BLOCK switch (which is always required if you use /IOC or /STRING switches). Most often this switch is used and you may find you do not even have the privilege (The Create-Without-Block privilege) to omit it. Assuming you do have the privilege when omitted after initiating the process CLI displays its PID number and returns control to the father.

What name will the new process have

This is specified with the /NAME=name switch where "name" is the simple process name. No process for the same username must already exist with that name. If omitted AOS/VS will make up a name based on the PID.

What privileges does the new process have

Most of the other switches supported by the PROCESS command relate to an AOS/VS privilege that the new process either has or does not. If none of these switches is supplied the new process has no privileges (and probably couldn't do all that much), while the /DEFAULT switch allows the son to adopt all the privileges of the Father. It is impossible for a Father to give his son a privilege he himself does not have (eg SUPERUSER).

Other important switches

The /STRING switch tells CLI to return the new process's termination message in the CLI string. It is necessary to include the /BLOCK switch with /STRING, otherwise when the new process terminated it would obliterate the STRING which you may well be using for something else by then. The only other way you can observe the new process's termination message is by issuing a CLI CHECKTERMS command after its completion.

Other inportant switches (continued)

The /SONS switch is really a privilege switch but deserves special mention. When omitted the new process may not create any son processes of its own (unless /UNLIMITEDSONS was supplied but you could only do this if you also had this privilege which is unlikely). Without a value it allows the new process to create as many sons as the father less the one the father has just created. Note that sons here really means descendants.

The /DIRECTORY switch is used to specify the new process's initial Working Directory. When omitted it is set to the father's current Working Directory. When included without a value it is made the same as the father's initial Working Directory (note: not necessarily the same as the father's current Working Directory), while when given an explicit value the meaning is obvious.

The CLI command **EXECUTE** (**XEQ** or X) can be thought of as the default **PROCESS** command, where:

X prog

is equivalent to:

PROCESS/BLOCK/DEFAULT/SONS/IOC/DATA=[!DATAFILE]/LIST=[!LISTFILE] prog

While

X/S prog

is equivalent to:

PROCESS/BLOCK/DEFAULT/SONS/IOC/DATA=[!DATA]/LIST=[!LIST]/STRING prog

You will perhaps hear the terminology **PROCcing up a process**, which is really a euphemism for creating a process with the PROCESS command, rather than with the EXECUTE command, which is usually referred to as executing a program. Clearly one is just a special case of the other.

The new process created by the **PROCESS** command will always adopt the **SEARCHLIST** and **DEFACL** (unless the /DACL switch is used) of its father.

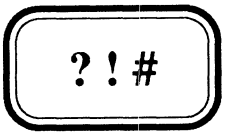
24.3 Why use it?

There are many reasons why you might use the PROCESS command, all of which boil down to enabling some feature not allowed with the EXECUTE command, such as running a process without Blocking yourself, or running a process with a different @INPUT and/or @OUTPUT file to your own. For example we use it run SPEED macros to discard SPEED's "Line too long" and "Lower case input encountered" messages when running tried, proven and tested SPEED editing macros (See Appendix A for more details) eg:

PROCESS/BLOCK/DEFAULT/OUTPUT=@NULL SPEED/I=speedcommandfile filetoedit

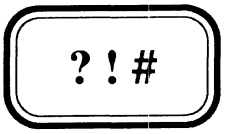
Of course if something went wrong with our SPEED macro we would also throw away any other error messages, which is why we only use it for "tried, proven and tested" SPEED macros.

The System UP macro uses the PROCESS command almost exclusively to create son processes with special privileges, particular simple process names (eg EXEC), and with special explicit priorities and process types.



CHAPTER 25

Error Handling



CHAPTER 25

Error Handling

25.1 What happens when an Error occurs?

CLI will either **IGNORE**, give a **WARNING** message, give an **ERROR** message and discard the remainder of the command line or macro and prompt for more, or give an ERROR message and then **ABORT**, depending on which of these settings the **CLASSn** parameter appropriate for the failed command has been set to. This was first discussed in Chapter 16 where a full explanation of CLASS settings and severity levels was given. Real programming languages provide mechanisms for evaluating Errors and performing some other tasks depending on what the error was. CLI does not provide a straightforward means of either evaluating errors or even retaining control after an error, other than by ignoring it.

In this Chapter we intend to look at various techniques we can employ to help us retain control, evaluate errors, and modify our action accordingly. We will look at some command specific techniques as well as some general purpose ones.

25.2 How to Retain Control after a failed Command

In order to evaluate the success or failure of a CLI command it is imperative to retain control. This can only be done if the CLASS1 (for CLI environment changing commands) or CLASS2 (for others) parameter for this command is set to IGNORE or WARNING. As we stated previously (in Chapter 16) we advise strongly against reducing either of the CLASS settings below the level of ERROR, so this necessitates use of either a **/1=severity** or a **/2=severity** switch on the command in question, where severity is either **IGNORE** or **WARNING**. Note that you never need to use both these switches since any single CLI command can give rise to only one CLASS of error. The problem is determining which CLASS. The quoted general rule that CLI environment changing commands give rise to CLASS1 errors, while other commands give rise to CLASS2 errors has its exceptions. For example the numeric variables are CLI environment parameters, yet the VARn commands give rise to CLASS2 errors. In the absence of a published list (by DG) we suggest you either suck it and see, or use both switches.

25.3 How to Evaluate Success or Failure of a Command

The evaluation of success or failure means testing to see if the command achieved its desired aim. We are not at this stage discussing analysis of the cause of the failure, just simply success or failure. How you determine whether or not its aim is achieved will depend on the particular command, so let's look at some particular commands:

DELETE

```
DELETE/2=IGNORE filename
[!EQUAL,,[!PATHNAME filename]]
     comment Success
[!ELSE]
     comment Failure
[!END]
```

DIRECTORY

```
DIRECTORY/1=IGNORE :dirpath
[!EQUAL,[!DIRECTORY],[!PATHNAME :dirpath]]
     comment Success
[!ELSE]
     comment Failure
[!END]
```

VARn when being set to a User entered value (as used in $NUM)

```
VAR0 4294967295
VAR0/2=IGNORE %1%; comment assuming user entered value in arg 1
[!UEQ,4294967295,[!VAR0]]
     comment Failure
[!ELSE]
     comment Success ie arg1 in range 0-4294967294
[!END]
```

These are of course fairly simple commands and the test for success is fairly simple, but in many cases it is that simple. We have no intention of covering every command, but merely wished to make you aware of the technique.

A more generalized technique that can be used on any command that normally gives no output (eg DIRECTORY, SEARCHLIST or DELETE without a /V switch), is as follows:

```
DELETE/2=IGNORE :UDD:[!USERNAME]:?[!PID].CHK
<command>/n=WARNING/L=:UDD:[!USERNAME]:?[!PID].CHK{ arguments}
[!UEQ,0,[!SIZE :UDD:[!USERNAME]:?[!PID].CHK]]
     comment Success
[!ELSE]
     comment Failure
[!END]
DELETE :UDD:[!USERNAME]:?[!PID].CHK
```

where

<command> is the command in question
n is 1 or 2 as appropriate for the command

In this technique command output (of which there isn't expected to be any) is sent to a temporary file (Note an explicit pathname is given in case we are in a directory to which we do not have Write Access). The **/n=WARNING** not only guarantees that we retain control but also means that if there is an error the error message will be written to the temporary file. Thus if the command was successful the temporary file will be empty, whereas if it failed it will contain the error message. If you wish the error to be displayed to the user you can simply type out the temporary file. Note that with this technique the /n=IGNORE switch cannot be used, since there would be no output on success or failure.

25.4 Discriminating between causes of Failure

This is really getting into the too-hard basket. What is required is to analyze the error message produced by the above generalized technique. These messages contain a minimum of two NEWLINE characters one after the text of the AOS/VS error message and one after the echoing of the command that caused it. It is therefore impossible to expand its contents in a conditional Pseudo Macro without getting a mismatched brackets error, and its contents can't be placed in the CLI STRING since the STRING command would be terminated by the first NEWLINE and then the echoed command would be attempted again. You could discard the second line with a SPEED or SORT/MERGE macro (See Appendices A & B) but very rarely would it be worth the effort. If you're getting this sophisticated you should probably be using a Real Programming Language and doing System Calls. Sometimes however a series of simple tests based on your knowledge of the possible causes, can be used to discriminate between causes of failure, but these are far too command and environment specific to warrant discussion here.

25.5 A Generalized Technique

What we wish to discuss here is a generalized technique for determining whether or not any Job was successful. A Job could be a single command, a macro or series of them. The price we pay for this generalized technique is an extra process and use of the CLI STRING but it is often well worth the trouble.

The technique is as follows:

```
X/S :CLI <job>
[!EQUAL,(OK),([!STRING])]
     comment Success
[!ELSE]
     comment Failure
[!END]
```

where

<job> is a macro and its arguments for running the job

The <job> macro will then look something like this:

File MYJOB.CLI

```
PROMPT BYE
CLASS(1 2) ABORT; Comment or as seen fit but not less than ERROR

Comment now the JOB whatever it is

BYE/L=@NULL OK
```

If the Job gets to complete without errors then and only then will the last command in the <job> macro be executed, which returns a termination message of **OK**, which is returned to our original macro through the STRING via the X/S command. Note that the PROMPT BYE command in the <job> macro prevents the User doing a CTRL-C CTRL-A and then entering his own BYE OK command to fool you.

NOT ANOTHER PAGE INTENTIONALLY LEFT BLANK

OK! Let's fill the page with something. The author while preparing this manual, is, for relief from the mental fatigue as much as anything, building up a collection of light verse which he hopes to publish in a book, called "Tall Tales But True". Only yesterday (1st March 1989) in fact, he sent unsolicited manuscripts to four Publishers. Here is one of the verses, which might help to relieve some of your own mental fatigue, experienced from having got this far in our Manual:

He or She?

by Greg Shalless

Is it he or is it she?
This object of no gender,
It depends on if you see,
The thing as coarse or tender.

What chromosome inside it lurks,
Does it give you a clue?
It could depend on if it works,
Or if it makes you blue.

If of course it has a name,
There can be no aspersions,
'Less of course it is the same
In male and female versions.

Is it he or is it she?
Why bother to debate it?
It depends what sex are thee,
And if you love or hate it,

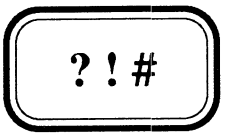
If it's a 'bastard' on a glitch,
You think of it as he,
But if you call the thing a 'bitch',
To you it is a she.

With 'la' and 'le' the French decide
To take a def'nite stance,
But this could make it hard to chide,
Or have with it romance.

A he or she from where you sit,
Or changing at your whim?
Maybe you do not care one bit,
If it's a her or him.

This talk of sex is too obtuse,
So why make such a fuss?
We know it cannot reproduce,
But nor can some of us.

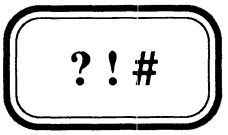
Is it he or is it she?
It should of course be neuter.
Then maybe it is all three.
What sex is your computer?



CHAPTER 26

Debugging Macros

26.1 Working out what went wrong



CHAPTER 26

Debugging Macros

26.1 Working out what went wrong

26.1 Working out what went wrong

CLI macros are interpreted, and are therefore not compiled. This means that unlike most real Programming Languages there is no compiler to get rid of syntax errors. All errors whether they be syntax, typos or logic errors can only be discovered by testing your macro. Never put a macro into production without first testing it, including when you make minor modifications to it. This is particularly important, however much more impractical it is, when the macro concerned is your System's UP and/or DOWN macro.

Apart from all the normal things you do when debugging software, CLI really only provides one debugging aid. However it is a particularly useful one, namely the CLI **TRACE** command. The TRACE command enables you to trace the progress of your macro without having to modify it. You can trace **MACRO** invocations, **COMMAND** executions or **PSEUDO** Macro expansions or any combination of these. What is traced is determined by the combination of switches you use on the TRACE command:

/MACRO

> Tells CLI to trace Macro invocations, which it does by displaying a line on @OUTPUT showing the command line for each macro called, preceded by **###**.

/COMMAND

> Tells CLI to trace Command executions, which it does by displaying a line on @OUTPUT showing the command line for each CLI command interpreted, preceded by ***.

/PSEUDO

> Tells CLI to trace Pseudo Macro expansions, which it does by displaying a line on @OUTPUT for each Pseudo Macro expanded, beginning with **+++**, followed by the Pseudo Macro just expanded and then the rest of the current command line.

When you have discovered the cause of your problem you can turn tracing off with the command TRACE/**KILL**, while **/OFF** and **/ON** switches can be used in conjunction with the three mode switches to establish particular combinations.

If your macro does a lot of screen interaction especially if it uses absolute cursor addressing, the intermingling of the screen output and the trace output can become very distracting. In such situations you may use TRACE/**LOG**, which tells CLI to send trace output to the **LOGFILE** (if one has been opened with the LOGFILE command) rather than to @OUTPUT. The **LOGFILE** command opens a file (which must exist) in which CLI echoes or logs everything that happens on the screen excluding the output of TYPE commands and other Programs.

One other useful debugging technique which isn't immediately obvious, is to place CLI WRITE commands in front of the meaty commands that do the real work. In this way when you run the macro all argument and switch handling is tested but the actual work is not performed, instead what would have been done is echoed for verification. When everything else appears to be right you remove the WRITEs and let 'er rip.

"Let 'er rip" ?? I guess that means I think my computer is female.

One final debugging tip concerns two obscure CLI prompts, namely:

```
!)
```

and

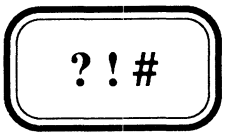
```
\)
```

CLI will present the User with one or other of these prompts whenever it interprets a Conditional Logic Pseudo Macro whose required **[!END]** is missing. You will get the former prompt if CLI is in the **TRUE** branch of the conditional, and the latter if it is in the **FALSE** branch. You can clear this prompt by entering **[!END]**'s until CLI has found one for each conditional (that was missing one). If you've never seen these prompts try the following:

```
) [!EQUAL,A,A]
!) [!ELSE]
\) [!END]
)
```

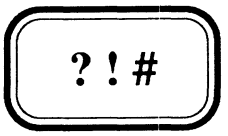
or

```
) [!EQUAL,A,B]
\) [!ELSE]
!) [!END]
)
```



CHAPTER 27

Macros that write Macros



CHAPTER 27

Macros that write Macros

27.1 Macros that Create Macros and other Command Files

The concept of macros that write macros and other Command Files (eg SORT/MERGE Command Files) is nothing new, and has already been covered in Chapters 21 and 23 with our $FIXSORT macro which used the CREATE/M command to create a temporary SORT/MERGE command file, which it subsequently used. These command files or macros can be written with CREATE/M commands, WRITE/L commands or WRITE/L=filename commands. The main reason for writing them is usually to overcome some restrictions, such as the inability to pass arguments to SORT/MERGE command files ($FIXSORT) or to overcome a possible CLI out of memory error as discussed in Chapter 18 on the !FILENAMES pseudo macro. What we want to point out here is some of the traps for young players. We won't repeat our discussion of the differences between WRITE/L and WRITE/L=filename but refer you back to Chapter 15 if you've forgotten.

CREATE/M

Since arguments and switches are substituted before interpretation of the macro by CLI, but Pseudo Macros are expanded when the macro is being run, you can:

Dynamically expand Arguments and Switches

Write **REAL** commas, brackets and therefore Pseudo-macro calls to the macro being created. The brackets (round, square or angle) will not be expanded by your macro, but by the created macro when it is run.

This works because CLI merely copies subsequent lines (following the command line) until it finds one with a single close round bracket. You can even copy a single close round bracket to the file being created (the Macro being written might have its own CREATE/M or X/M command) by a line with two close round brackets.

WRITE/L{=filename}

By contrast everything is expanded dynamically when the WRITE command is being interpreted, therefore you can:

Dynamically expand Arguments and Switches.

Dynamically expand Pseudo Macros including conditionals.

But you cannot:

Write a Pseudo-Macro call (or a UPM) to the created macro, since it is expanded immediately. Fake square-brackets using !ASCII are just that, fakes and will not be recognised as introducing a Pseudo Macro when the created macro is run.

Write Round or Angle brackets to the created macro.

Write Commas (and therefore force a null argument) to the created macro, since the WRITE command writes a single space between arguments. Two spaces in the created macro only constitute a single argument dilimiter.

Often we are only dynamically creating a macro (rather than simply using a pre-prepared one) for the sake of one line. In such cases rather than a whole mass of WRITE statements it might be better to use the CLI **COPY** command thus:

```
DELETE/2=IGNORE ?[!PID]MYMAC.CLI
COPY ?[!PID]MYMAC.CLI MYMAC_PREFIX.CLI
WRITE/L=?[!PID]MYMAC.CLI The one line we create dynamically
COPY/A ?[!PID]MYMAC.CLI MYMAC_POSTFIX.CLI
?[!PID]MYMAC
DELETE ?[!PID]MYMAC.CLI
```

where

MYMAC_PREFIX.CLI contains the common preamble code

and

MYMAC_POSTFIX.CLI contains the common postamble code

Which of these techniques you use will depend on your circumstances.

27.2 Macros that Maintain Parameter Files

Macros can be written that maintain parameter files, ranging from the very simple such as a Mag-Tape cycle counter, to a parameter file that drives a general purpose Printer Co-operative Start-up macro for your System Up macro. Since these are both pretty meaningful examples we will look at them both in detail.

Firstly let's look at a very simple cyclic loop counter, held in a file called CNT, which cycles from 1 to 4 then back to 1 etc. We will call the macro that maintains CNT **$LOOP4**. $LOOP4 will increment the value held in CNT and if this is greater than 4, the value is set back to 1. This new value is returned in VAR0 and written back to CNT. To make sure $LOOP4 returns 1 on its first call we must initially create CNT with the value 4. We can do this as follows:

```
) CREATE/I CNT
))4
)))
)
```

File $LOOP4.CLI

```
VAR0 [%1%]
VAR0 [!UADD [!VAR0] 1]
[!UGT,[!VAR0],4]
     VAR0 1
[!END]
DELETE %1%
WRITE/L=%1% [!VAR0]
```

Where we would call $LOOP4 to maintain CNT thus:

```
) $LOOP4 CNT
```

This macro suffers from inflexibility in that the maximum value (4) and the reset value (1) are fixed. It is relatively simple to generalize by supporting switches say, **/LIMIT=m** and **/RESET=n**, and call it **$LOOP**.

File $LOOP.CLI

```
VAR0 [%1%]
VAR0 [!UADD [!VAR0] 1]
[!UGT,[!VAR0],%0/LIMIT=%]
     VAR0 %0/RESET=%
[!END]
DELETE %1%
WRITE/L=%1% [!VAR0]
```

Where we would call $LOOP to maintain CNT thus:

```
) $LOOP/RESET=1/LIMIT=4 CNT
```

However this is not a very robust macro. It suffers from all of the following limitations:

- It will crash if called without an argument
- It will crash if the counter file doesn't exist or contains junk
- It will crash if /LIMIT, and/or /RESET, are omitted or have junk values
- It can't be used by multiple Users

Furthermore although it is more difficult to code without using a second numeric variable, it seems more logical to us to use the current value then write back to the file the cycled value, for next time. Let's look at a macro or rather a pair of macros that overcomes all these limitations. We will call them **$LOOP_COUNT** and **$LOOP_CREATE**. The reason for separating the maintenance and creation tasks, is that we want to store the RESET value and LIMIT in the Cyclic Loop Counter file along with its current value. The $LOOP_CREATE macro makes sure the file is correctly initialized, so that $LOOP_COUNT doesn't have to worry about validity checks. These macros will highlight the fact that arguments other than zero can meaningfully have switches, as well as techniques for allowing a user to either include or omit a default extension. We wish to enforce a **.CLI** extension on the Cyclic Loop Counter File, to save unnecessary searches for the file. (Remember [FRED] looks for FRED.CLI before it looks for FRED). Finally we plan to use a different technique to any we've previously discussed for making it a multi-user macro, by using **ACL**s.

$LOOP_COUNT

This macro returns in VAR0 the value of the loop counter held in "pathname", and increments it for the next call. The loop counter file holds as well as the current value the limit or maximum value and the reset value (0 or 1). "pathname" should first be created with $LOOP_CREATE. It is cyclic, resetting when the limit is exceeded.

Format: $LOOP_COUNT pathname

Argument:

pathname	Pathname to Loop Counter File (without .CLI extension)

File $LOOP_COUNT.CLI

```
COMMENT Format of loop counter file contents is "value/RESET=n/LIMIT=m"
COMMENT where "value" is reset to "n" whenever value exceeds "m".

[!EQUAL,.CLI,[!EEXTENSION %1%]]
     %0% [!ENAME %1%]
[!ELSE]
     [!EQUAL,,[!PATHNAME %1%.CLI]]
          WRITE
          WRITE ERROR - %1% doesn't exist must create with $LOOP_CREATE
          WRITE,,,,,,,,,Use $LOOP_CREATE/H for details[$$BELL]
          WRITE
     [!ELSE]

COMMENT ACL's are used to control simultaneous access.
COMMENT If the loop counter file's ACL is modified other than through
COMMENT the $LOOP_CREATE and $LOOP_COUNT macros this may result in an
COMMENT infinite loop

          [!NEQUAL,(+,OWARE),([!ACL %1%.CLI])]
               PAUSE 1
               %0-%
          [!ELSE]
               ACL %1%.CLI [!USERNAME],OWARE +,RE
               $LOOP_COUNT.1 %1% [%1%]
               ACL %1%.CLI +,OWARE
          [!END]
     [!END]
[!END]
```

File $LOOP_COUNT.1.CLI

```
VAR0 %2\%
DELETE/2=IGNORE %1%.TMP
WRITE/L=%1%.TMP &
[!UEQ,%2\%,%2/LIMIT=%]%2/RESET=%[!ELSE][!UADD %2\% 1][!END]%2/%
COPY/D %1%.<CLI TMP>
DELETE %1%.TMP
```

Note in **$LOOP_COUNT** we expect the pathname to the file to be specified without the .CLI extension that we insist it must have, but in case the user supplies the full pathname we simply call ourselves again with the extension stripped off. We have used an **ACL** of **+,OWARE** to indicate that that file is not being updated. If we detect the file has any other ACL we loop until the condition clears. When it has, we change its ACL to enable us to update it, but only let everyone else read it. When we've finished updating we reset the ACL. A second level macro to do the dirtywork is necessary because the Loop Counter file no longer simply contains a current count value (as it did in $LOOP which enabled us to expand its contents immediately), but a Reset and Limit. These three values can only be isolated by making them arguments to another macro. Well, that isn't quite true, we could maintain three separate files, or set them up as three separate arguments (rather than using switches) so that the file had the format **value,n,m** rather than the present format of **value/RESET=n/LIMIT=m.**

With this latter technique we could isolate the three values directly in **$LOOP_COUNT** thus:

```
VAR(0 1 2) ([%1%])
```

but then the macro is no longer environment preserving. In order to preserve the environment (apart from VAR0 where we return the result) we would have to PUSH and make a recursive call to return the answer (as we did in $NUMBER which was the environment preserving version of $NUM). We believe our method is better. Ideally of course it would be better if we could make $LOOP_COUNT a User Pseudo Macro, in which case we could also preserve VAR0, but this is impossible because at least one full command is necessary to write the updated value back to the file (and if you remember User Pseudo Macros must not contain any commands).

There is however one major problem with our **$LOOP_COUNT** macro, which is that if the system crashed after the first ACL command and before the second, then any subsequent call (after the system comes back up) will go into an infinite loop, because the ACL is not +,OWARE. You may think this weakness renders it useless, but it is not unusual to have to do recovery processing after a system crash. The recovery processing in this case is simply to set the **ACL** of all Loop Counter files controlled by **$LOOP_COUNT** to **+,OWARE.**

$LOOP_COUNT might be used in a tape cycling backup macro thus:

```
$LOOP_COUNT XYZ_NEXT_TAPE
MOUNT/VOLID=[XYZ_TAPE.[!VAR0]] MT Please Mount .....
```

Now let's look at the **$LOOP_CREATE** macro, which must guarantee to create the Cyclic Loop Counter file in the correct format for $LOOP_COUNT, which is **value/RESET=n/LIMIT=m** where **n** must not exceed **m**, and **value** must be initialized to **n**. (Note, if we increment then use the incremented value as we did in $LOOP, we would initialize **value** to **m**).

LOOP_CREATE

This macro creates a loop counter file for $LOOP_COUNT. The initial value (/RESET=n) and limit or maximum value (/LIMIT=m) may be specified with switches, or you will be prompted for them. $LOOP_COUNT does a cyclic count so when the Counter exceeds its limit, it will be reset to its reset value by $LOOP_COUNT.

Format: $LOOP_CREATE{/switches} pathname

Optional Switches:

/RESET=n	Specifies the initial value of the counter as well as the Reset value when the Limit is exceeded. The value n will usually be 0 or 1 and cannot be greater than m.
/LIMIT=m	Specifies the Limit (Note n cannot be > m)

Argument:

pathname	Pathname to Loop Counter File (.CLI appended if omitted)

File $LOOP CREATE.CLI

```
COMMENT Format of loop counter file contents is "value/RESET=n/LIMIT=m"
COMMENT where "value" is reset to "n" whenever value exceeds "m".

[!EQUAL,,%1%]
     WRITE
     WRITE ERROR - must specify name of Cyclic Loop Counter file
     WRITE [$$BELL]
[!ELSE]
     PUSH; PROMPT POP
     STRING [!EQUAL,.CLI,[!EEXTENSION %1%]][!ENAME %1%][!ELSE]%1%[!END]
     [!NEQUAL,,[!PATHNAME [!STRING].CLI]]
          WRITE
          WRITE ERROR - [!STRING].CLI already exists with [[!STRING]]
          WRITE [$$BELL]
     [!ELSE]
          [!EQUAL,,%0/LIMIT=%]
               WRITE
               $NUMBER/MIN=1 Enter Maximum Value of Cyclic Loop&
                    Counter [!STRING]
          [!ELSE]
               VAR0/2=ERROR %0/LIMIT=%
          [!END]
          VAR1 [!VAR0]
          [!EQUAL,,%0/RESET=%]
               WRITE
               $NUMBER/MIN=0/MAX=[!VAR1]/DEF=1 Enter Reset&
                    [$$IN_ROUND and initial] value
          [!ELSE]
               VAR0/2=ERROR %0/RESET=%
          [!END]
          [!UGT,[!VAR0],[!VAR1]]
               WRITE
               WRITE ERROR - Reset value for Loop Counter exceeds its&
                    limit
               WRITE [$$BELL]
          [!ELSE]
               WRITE/L=[!STRING].CLI &
                    [!VAR0]/RESET=[!VAR0]/LIMIT=[!VAR1]
               ACL [!STRING].CLI +,OWARE
          [!END]
     [!END]
     POP
[!END]
```

Note that although it is most common to use **/n=severity** switches to ignore errors, in this case we have used **/2=ERROR** to force an error if a /LIMIT or /RESET switch is supplied with a garbage value. If the switch is omitted or supplied without a value we use our $NUMBER macro to force entry of a valid value. The same technique of allowing the user to include the .CLI extension even though we expect him not to, is used. However since we have to PUSH (and POP) to preserve the environment (we don't have any value to return), we are free to use as many environmental parameters as we like, and so we use the STRING rather than a recursive call as we did in $LOOP_COUNT.

27.3 Macros that write User Pseudo Macros

In a sense the Cyclic Loop Counter parameter file of the previous section is used as a User Pseudo Macro. Its contents are expanded as one with **[%1%]**. Unfortunately though it contains a NEWLINE character and could therefore never be used in the middle of a command line because the NEWLINE would cause premature termination of the command. With this in mind we always arranged for this expression to be the last thing on a command line, so that the NEWLINE would be followed by another NEWLINE giving rise to a null command line and therefore causing no harm.

The problem of course is the CLI WRITE command which always writes a NEWLINE after its last argument. In order to have a macro create a User Pseudo Macro we need some means of either not writing the Newline, deleting it afterwards, or telling CLI to ignore it. Although it could be deleted with a SPEED macro, since this requires an extra process, it would be better to tell CLI to ignore it. CLI will ignore NEWLINEs immediately preceded by the Ampersand character (**&**). So we have a new problem of how to get the & into the file just before the NEWLINE.

```
WRITE/L=upm.cli data&
```

does not work because this merely serves to continue this command line on the next line. The following will however:

```
WRITE/L=upm.cli data[!ASCII 46]
```

and

```
WRITE/L=upm.cli data<&>
```

We prefer the latter because it makes it more obvious what you are trying to do. A common technique amongst those who were never able to work out how to get rid of (or rather ignore) the last NEWLINE when trying to expand the file's contents but use it within a command was to put it in the STRING and use !STRING subsequently thus:

```
STRING [upm]
WRITE "[!STRING]"
```

but by writing the ampersand we can avoid the use of the STRING thus:

```
WRITE "[upm]"
```

As an example let's consider a macro that maintains a Parameter File (User Pseudo Macro) that holds the parameters for a general purpose **$UP.PRINTER** macro for use in our System UP macro. We will assume that for every **printername.PRM.CLI** file that exists in the directory **:$SYSPARAMS** a spooled printer co-operative running XLPT needs to be initiated. The macro that maintains the **printername.PRM.CLI** User Pseudo Macros will be called **$ED.PRINTER**. In our System Up macro all that is required to bring up all the Print Co-operatives is:

```
$UP.PRINTER ([!ENAME [!ENAME [!FILENAMES :$SYSPARAMS:-.PRM.CLI]]])
```

while to bring up a single printer the command would be:

```
) $UP.PRINTER printername
```

Since only Username OP can give the required **CONTROL @EXEC** commands **$UP.PRINTER** is restricted to use by **OP**. Before writing these macros we need to know what information we need in order to bring up a printer and thus what information we need to store in a **printername.PRM.CLI** file. Since we are writing a macro to maintain this file, its format can be designed to simplify the coding of **$UP.PRINTER**. If the file were to be maintained by a text editor, we may have chosen a different format. However it is generally unwise to allow parameter files that drive macros and require a precise format to be edited in this way (with a text editor) as the format is too easily corrupted and often too cryptic. Furthermore this would prohibit validation.

We will dispense with explanations of why we have chosen the particular format we have for the **printername.PRM.CLI** file and simply present **$UP.PRINTER** for your perusal. However note that value **/NULL** is used whenever we wish to write a null argument to overcome any difficulties we might have writing consecutive commas to the Paramater File. **Note you should be able to use these macros immediately on your System. Their beauty lies in the fact that as printers are added, removed or their specifications changed, your System UP Macro is not touched.**

$UP.PRINTER

This macro initiates the Spooler Co-operative for the specified Printer. The Printer is brought up according to the parameters held in the printer's parameter file maintained by the $ED.PRINTER macro. Only Users with the Username OP can run this macro.

Format: $UP.PRINTER printername

Argument:

printername Arbitrary (though often some queuename) Printer name identifying the parameter file in :$SYSPARAMS. Use $ED.PRINTER/H for further details.

File $UP.PRINTER.CLI

```
[!NEQUAL,OP,[!USERNAME]]
     WRITE
     WRITE ERROR - Only OP can run $UP.PRINTER
     WRITE [$$BELL]
[!ELSE]
     [!EQUAL,,%1%]
          WRITE
          WRITE ERROR - Must specify printername
          WRITE [$$BELL]
     [!ELSE]
          [!EQUAL,,[!PATHNAME :$SYSPARAMS:%1%.PRM.CLI]]
               WRITE
               WRITE ERROR - Parameter file for %1% missing&
                         use $ED.PRINTER to create it
               WRITE [$$BELL]
          [!ELSE]
               $UP.PRINTER.1 %1% [:$SYSPARAMS:%1%.PRM]
          [!END]
     [!END]
[!END]
```

$UP.PRINTER.1

This macro should only ever be called from $UP.PRINTER

Format: $UP.PRINTER.1 printername [:$SYSPARAMS:printername.PRM]

Arguments:

1	printername
2	qname/startswitches (eg LQP/NL)
3	device (eg CON20)
4	UPPER (To convert Lower-To-Upper case) or /NULL
5	lpp (eg 66)
6	cpl (eg 80)
7	headers (0, 1 or 2)
8	even pagination (ON or OFF)
9	cleanupfile (If Laser Printer) or /NULL
10	defaultform (eg QP.DEF.FORM) or /NULL
11-	additional qnames (if any) feeding this device

File $UP.PRINTER.1.CLI

```
DELETE/2=IGNORE :UDD:[!USERNAME]:?[!PID].WHO.TMP
CONTROL/2=IGNORE/L=:UDD:[!USERNAME]:?[!PID].WHO.TMP&
     @EXEC SPOOLSTATUS @%3%
[!ULE,30,[!SIZE :UDD:[!USERNAME]:?[!PID].WHO.TMP]]
     WRITE
     WRITE Co-operative for %1% Printer already running - namely
     WRITE
     TYPE :UDD:[!USERNAME]:?[!PID].WHO.TMP
     DELETE :UDD:[!USERNAME]:?[!PID].WHO.TMP
[!ELSE]
     DELETE :UDD:[!USERNAME]:?[!PID].WHO.TMP
     WRITE
     WRITE Initiating Co-operative for %1% Printer
     WRITE
     CONTROL @EXEC OPEN %2\%
     CONTROL @EXEC START%2/% %2\% @%3% %4\%
     PAUSE 2
     [!NEQUAL,,%11%]
          CONTROL @EXEC START (%11-%) @%3%
     [!END]
     CONTROL @EXEC LPP @%3% %5%
     CONTROL @EXEC CPL @%3% %6%
     PAUSE 2
     CONTROL @EXEC HEADERS @%3% %7%
     CONTROL @EXEC EVEN @%3% %8%
     [!NEQUAL,,%9\%]
          CONTROL @EXEC BINARY @%3% %9\%
     [!END]
     [!NEQUAL,,%10\%]
          CONTROL @EXEC DEFAULTFORMS @%3% %10\%
     [!END]
     PAUSE 2
     CONTROL @EXEC CONTINUE @%3%
     CONTROL @EXEC SILENCE @%3%
[!END]
WRITE
```

We do not wish to dwell on the details of the CONTROL @EXEC commands, however learning the lesson from Chapter 25, see how we used the **CONTROL @EXEC** Command in conjunction with a **/L=filename** switch and a test of that file's resultant **size** to determine if the Print Co-operative is already running. Note that this was originally the command **WHO/... OP:%3%**, which was okay in AOS/VS Rev 7.57, but from Rev 7.62 onwards **XLPT** processes handle multiple devices and are no longer given a process name based on the device. Although not tested under Rev 7.62 or higher we are confident **$UP.PRINTER** will work under these Revs. Arguments 2-11 and onwards of **$UP.PRINTER.1** then form the contents of the **printername.PRM.CLI** User Pseudo Macro that **$ED.PRINTER** has to maintain.

To make **$ED.PRINTER** a robust macro, which is necessary because **$UP.PRINTER** assumes the printer's parameter file has valid contents, is quite difficult as you will see. Partly for this reason, but mainly because it is not intended for simultaneous use, we have not attempted to make it a multi-user macro. The **$ED.PRINTER** macro actually becomes an entire suite of macros, and at times the code is quite complex. Don't worry if we lose you here, but our aim is twofold. Firstly we are nearing the end of the manual and we want to demonstrate how powerful CLI Macro programming can be when you know all the tricks of the trade, by showing you some meaningful meaty macros. **Secondly we want to sell lots of manuals. By providing universally useful macros such as the ones in this Chapter, we think we are more likely to be successful in that endeavour (and if all the macros were too easy to type in, you wouldn't want to buy the companion Tape, would you?).**

It should be clear from the **$UP.PRINTER** macro that all printer parameter files are held in the directory **:$SYSPARAMS.** Neither $UP.PRINTER nor $ED.PRINTER will create this directory, and both will crash if it does not exist. If you intend to use these macros we suggest you create it as follows:

```
) SUPERUSER ON; Comment you'll have to be OP to do it
*) CREATE/MAX=1000 :$SYSPARAMS
*) ACL :$SYSPARAMS OP,OWARE +,RE
*) SUPERUSER OFF
)
```

We will look at the macros comprising the $ED.PRINTER suite in a top down fashion, and when explanation is considered appropriate, it will be included after each macro. So let's begin with $ED.PRINTER.

$ED.PRINTER

This macro maintains parameters held in the printer parameter file :$SYSPARAMS:printername.PRM.CLI . The macro $UP.PRINTER will initiate the Print Co-op based on the parameters in this file.

Format: $ED.PRINTER printername

Argument:

printername Arbitrary (though often some queuename) Printer name identifying the parameter file in :$SYSPARAMS.

File $ED.PRINTER.CLI

```
[!EQUAL,,%1%]
     WRITE
     WRITE ERROR - Must specify printername
     WRITE [$$BELL]
[!ELSE]
     PUSH; PROMPT POP
     DEFACL OP,OWARE +,RE
     [!EQUAL,,[!PATHNAME :$SYSPARAMS:%1%.PRM.CLI]]
          WRITE
          WRITE Parameter file for Printer %1% does not exist -&
                    will create one
          $ED.PRINTER.CREATE %1% [!READ/S What type of Printer is it&
          (L - Laser C - Character) [$$IN_SQUARE lineprinter] [$$BELL]]
          $ED.PRINTER.1 %1% [:$SYSPARAMS:%1%.TMP]
     [!ELSE]
          WRITE
          [!EQUAL,Y,[$$ARG1 [!READ/S Do you want to delete&
          Parameter File for Printer [$$UNDERLINE %1%] (Y N)&
          [$$IN_SQUARE N] [$$BELL]]]]
               $ED.PRINTER.DEL %1% [:$SYSPARAMS:%1%.PRM]
          [!ELSE]
               $ED.PRINTER.1 %1% [:$SYSPARAMS:%1%.PRM]
          [!END]
     [!END]
     POP
[!END]
```

At lower levels especially in parameter validation loops, the macros will be recursive, so a **PUSH** (and corresponding POP) is included at this level so that we only ever need to do it once. The **DEFACL** is set to **OP,OWARE +,RE** to ensure that users other than OP can use **$ED.PRINTER** to display a printer's parameter file. However if a User other than OP enters anything other than **Y** to the **Satisfactory?** question in **$ED.PRINTER.1** the macro will crash. The parameter file has been chosen to have a **.CLI** extension to prevent unnecessary searches for non-existent files. The **$ED.PRINTER.1** macro will display the current values in the Pseudo Macro Parameter file (in a somewhat more meaningful way than merely displaying its contents) and go through an editing loop until the User indicates that the current set of values is satisfactory. If the parameter file already exists we give the user the option of deleting it, through **$ED.PRINTER.DEL**, which must make sure the user is given ample opportunity to correct a mistake. So that all data-entry and validation is controlled by one macro ($ED.PRINTER.1), if no parameter file exists for this printer **$ED.PRINTER.CREATE** is used to create an initial temporary parameter file.

File $ED.PRINTER.CREATE.CLI

```
[!EQUAL,%2%,C]
          WRITE/L=:$SYSPARAMS:%1%.TMP.CLI&
                    %1% ??? /NULL 66 132 1 OFF /NULL /NULL<&>
[!ELSE]
     [!EQUAL,%2%,L]
          WRITE/L=:$SYSPARAMS:%1%.TMP.CLI&
                    %1%/NL ??? /NULL 66 80 0 OFF %1%.CLEANUP /NULL<&>
     [!ELSE]
          WRITE/L=:$SYSPARAMS:%1%.TMP.CLI&
                    %1% ??? UPPER 66 132 1 ON /NULL /NULL<&>
    [!END]
[!END]
```

Note that since we will be expanding the temporary file's contents in the same way as the parameter file, we have given it a **.CLI** extension (rather than the more common .TMP). Note that while we can make sensible guesses at default values for most of the parameters depending on what sort of printer it is, we cannot possibly guess what device this printer will be, so we don't try.

The other important point about this macro is the use of **WRITE/L=file** and **<&>** to make the parameter file a User Pseudo Macro. (In case you've forgotten, this section is actually called "Macros that write User Pseudo Macros".)

File $ED.PRINTER.DEL.CLI

```
$ED.PRINTER.DISP %-%
WRITE Are you sure you wish to Delete[$$BELL]
DELETE/C/V :$SYSPARAMS:%1%.PRM.CLI
[!EQUAL,,[!PATHNAME :$SYSPARAMS:%1%.PRM.CLI]]
     DELETE/2=IGNORE :$SYSPARAMS:%3%.IN_USE.CLI
[!END]
```

This macro is invoked after the user has made the deliberate selection to delete the printer's parameter file, but just in case this was done accidently or the wrong printer was specified we will only delete after displaying the printer's current parameter values and getting express confirmation by using the **/C** switch on the DELETE command. After successful deletion we must also delete the Parameter File that said the **device** associated with this Printer was "in use". (See **$ED.PRINTER.DEV** description for further details.)

$ED.PRINTER.1

This macro should only ever be called from $ED.PRINTER

Format: $ED.PRINTER.1 printername [:$SYSPARAMS:printername.PRM]

Arguments:

1	printername
2	qname/startswitches (eg LQP/NL)
3	device (eg CON20)
4	UPPER (To convert Lower-To-Upper case) or /NULL
5	lpp (eg 66)
6	cpl (eg 80)
7	headers (0, 1 or 2)
8	even pagination (ON or OFF)
9	cleanupfile (If Laser Printer) or /NULL
10	defaultform (eg QP.DEF.FORM) or /NULL
11-	additional qnames (if any) feeding this device

File $ED.PRINTER.1.CLI

```
$ED.PRINTER.DISP %-%
[!NEQUAL,???,%3%]
     [!NEQUAL,Y,[$$ARG1 [!READ/S Satisfactory? (Y N)&
     [$$IN_SQUARE N] [$$BELL]]]]
          WRITE
          $ED.PRINTER.EDIT %-%
          %0% %1% [:$SYSPARAMS:%1%.TMP]
     [!ELSE]
          [!NEQUAL,,[!PATHNAME :$SYSPARAMS:%1%.TMP.CLI]]
               DELETE/2=IGNORE :$SYSPARAMS:%1%.PRM.CLI
               RENAME :$SYSPARAMS:%1%.TMP.CLI %1%.PRM.CLI
          [!END]
          WRITE
     [!END]
[!ELSE]
     $ED.PRINTER.EDIT %-%
     %0% %1% [:$SYSPARAMS:%1%.TMP]
[!END]
```

First we use **$ED.PRINTER.DISP** to display the current values, then ask the User if they are satisfactory (unless these values represent our initial guess at the parameters, ie device=???, in which case we must force entry of at least a valid device). If they are not satisfactory, **$ED.PRINTER.EDIT**, prompts the User for a new value for each parameter (with the default in each case being the current value - provided that it is a valid value). It writes the new values to a temporary version of the parameter file, so after a valid set of parameters has been entered we expand its contents (in User Pseudo Macro format), in a recursive call on ourselves until the User decides the values are satisfactory. Finally when the parameters are satisfactory, we delete the previous parameter file (if one existed), and rename the temporary version to the permanent version, unless there was no temporary version (Note the User may have simply used **$ED.PRINTER** to check the current contents and decided they were okay as is.)

$ED.PRINTER.DISP

This macro should only ever be called from $ED.PRINTER.1
It displays the current Printer Parameters.

Format: $ED.PRINTER.DISP arguments

Arguments:

1	printername
2	qname/startswitches (eg LQP/NL)
3	device (eg CON20)
4	UPPER (To convert Lower-To-Upper case) or /NULL
5	lpp (eg 66)
6	cpl (eg 80)
7	headers (0, 1 or 2)
8	even pagination (ON or OFF)
9	cleanupfile (If Laser Printer) or /NULL
10	defaultform (eg QP.DEF.FORM) or /NULL
11-	additional qnames (if any) feeding this device

File $ED.PRINTER.DISP.CLI

```
WRITE [$$ERASE_PAGE],,,,,,,Printer Parameters for the&
     [$$UNDERLINE %1%] Printer
WRITE
WRITE [$$DIM,,,Queuename:] %2\%&
          [$$CURSOR/LINE=3/COL=35][$$DIM START switches:] %2/%
WRITE [$$DIM,,,,,,Device:] %3%&
          [$$CURSOR/LINE=4/COL=33][$$DIM Convert to UPPER:]&
          [!EQUAL,UPPER,%4\%]Yes[!ELSE]No[!END]
WRITE [$$DIM,,,,,,,,,CPL:] %5%&
          [$$CURSOR/LINE=5/COL=46][$$DIM LPP:] %6%
WRITE [$$DIM,,,,,Headers:] %7%&
          [$$CURSOR/LINE=6/COL=34][$$DIM EVEN Pagination:] %8%
WRITE [$$DIM Cleanupfile:] %9\%&
          [$$CURSOR/LINE=7/COL=38][$$DIM Defaultform:] %10\%
WRITE [$$DIM,,,Other Q's:] %11-%
WRITE
```

In this macro we display the values provided to us as parameters two up on the screen, with descriptive information dimmed and values highlighted and aligned. Alignment of the Left Column is achieved with commas (or null arguments) to force alignment of the Colons (:), however the variable length nature of the values does not allow us to align the second column in the same way. This is achieved with our **$$CURSOR** User Pseudo Macro. Our **$$ERASE_PAGE** UPM is used to clear the screen (and position on Line 1), while **$$UNDERLINE** and **$$DIM** have obvious purposes.

Although the use of these **!ASCII** User Pseudo Macros makes execution of the macro slower, we are convinced that you (as non-authors of this macro) could maintain it with little difficulty, but would find its equivalent with all the !ASCII expansions substituted in directly, a different proposition altogether.

$ED.PRINTER.EDIT

This macro should only ever be called from $ED.PRINTER.1
It displays the current Printer Parameters.

Format: $ED.PRINTER.EDIT arguments

Arguments:

1	printername
2	qname/startswitches (eg LQP/NL)
3	device (eg CON20)
4	UPPER (To convert Lower-To-Upper case) or /NULL
5	lpp (eg 66)
6	cpl (eg 80)
7	headers (0, 1 or 2)
8	even pagination (ON or OFF)
9	cleanupfile (If Laser Printer) or /NULL
10	defaultform (eg QP.DEF.FORM) or /NULL
11-	additional qnames (if any) feeding this device

File $ED.PRINTER.EDIT.CLI

```
DELETE/2=IGNORE :$SYSPARAMS:%1%.TMP.CLI
LISTFILE :$SYSPARAMS:%1%.TMP.CLI
$FILECHECK/TYPE=QUE @%2\%
$ED.PRINTER.Q [!STRING]
$ED.PRINTER.SW [!STRING],%2/%
$FILECHECK/TYPE=CON/TYPE=LPU @%3%
$ED.PRINTER.DEV %1% [!STRING]
$ED.PRINTER.YN %4% UPPER /NULL Convert to UPPER Case
$NUM/DEF=%5%/MIN=1 Enter LPP
WRITE/L [!VAR0] <&>
$NUM/DEF=%6%/MIN=1 Enter CPL
WRITE/L [!VAR0] <&>
$NUM/DEF=%7%/MIN=0/MAX=2 No Headers
WRITE/L [!VAR0] <&>
$ED.PRINTER.YN %8% ON OFF EVEN Pagination
$FILECHECK :UTIL:FORMS:%9\%
$ED.PRINTER.CLNUP %9%,[!STRING]
$FILECHECK :UTIL:FORMS:%10\%
$ED.PRINTER.FORMS %10%,[!STRING]
$ED.PRINTER.QS %11-%
LISTFILE/G
```

This macro deletes any existing temporary version of the parameter file and sets the **LISTFILE** to it, so that the entry and validation macros that follow can **WRITE/L** the new parameter values to the file, without opening the output file more than once. It is closed on completion with **LISTFILE/G.**

We have used our **$NUM** macro to prompt for and validate the numeric parameters. Note we did not use $NUMBER (the environment preserving version), because this is taken care of at a higher level and we are not interested in the contents of the STRING which $NUM always clears. Descriptions of the other entry and validation macros follow more or less in their order of appearance above.

$FILECHECK

This macro returns in the CLI STRING the name of the file whose pathname is passed as an argument, if the file actually exists and satisfies the optional FILESTATUS switches (eg /TYPE=type), otherwise the STRING is returned empty. The macro is designed to overcome the fact that the !FILENAME Pseudo Macro does not support the switches of the FILESTATUS command.

Format: $FILECHECK{/switches} pathname

Optional Switches:

/switches Any valid switches for the FILESTATUS command

Argument:

pathname Pathname to file whose existence, subject to the restrictions imposed by the switches, you want checked.

File $FILECHECK.CLI

```
DELETE/2=IGNORE :UDD:[!USERNAME]:[!PID]F.TMP
FILESTATUS%0/%/L=:UDD:[!USERNAME]:[!PID]F.TMP/1=IGNORE %1%
[!UEQ,0,[!SIZE :UDD:[!USERNAME]:[!PID]F.TMP]]
     STRING/K
[!ELSE]
     STRING [!EFILENAME %1%]
[!END]
DELETE :UDD:[!USERNAME]:[!PID]F.TMP
```

If **!FILENAMES** supported the same switches as the CLI **FILESTATUS** command (or at least the ones that affect the selection of files, such as **/TYPE=type** or **/BEFORE/TLM=datetime** etc, we wouldn't need this macro and in **$ED.PRINTER.EDIT**, instead of:

```
$FILECHECK/TYPE=QUE @%2\%
$ED.PRINTER.Q [!STRING]
```

we would write

```
$ED.PRINTER.Q [!EFILENAME [!FILENAME/TYPE=QUE @%2\%]]
```

However **!FILENAMES** does not support any switches, so we did write this macro and you may notice that we also made it a general purpose multi-user macro (by putting **!PID** in the temporary file name) because we may find in useful outside the context of **$ED.PRINTER.**

Notice too, that the **FILESTATUS** command is another exception to the DG HELP file quoted general rule about severity levels. Despite the fact that FILESTATUS is not an environment changing command, its errors give rise to a **CLASS1** exception (we discovered it by the suck-it-and-see method after first unsuccessfully trying a /2=IGNORE switch).

File $ED.PRINTER.Q.CLI

```
STRING [$$ARG1 [!READ/S Enter Qname&
     [!NEQUAL,,%1%][$$IN_SQUARE %1%] [!END][$$BELL]]]
[!EQUAL,,%1%[!STRING]]
     WRITE
     WRITE ERROR - Must enter a Qname
     WRITE
     %0-%
[!ELSE]
     [!EQUAL,,[!STRING]]
          STRING %1%
     [!ELSE]
          $FILECHECK/TYPE=QUE @[!STRING]
          [!EQUAL,,[!STRING]]
               WRITE
               WRITE ERROR - Not a known Q
               WRITE
               %0-%
          [!END]
     [!END]
[!END]
```

This macro prompts for, accepts, and validates the Printer Queue name. If a null response is given **AND** there was no valid default, we must force entry of a value, so an error is given and we re-prompt with a recursive call. Note the concatenation of the default (**%1%**) and the response (**[!STRING]**) to effect this complex logic operation. You will also note that consistently throughout these macros, we display options in round brackets, and defaults in square brackets after the prompt text, where the default is always the current value. Therefore when a null response is given when there was a valid dafault, the default is then accepted without validation.

Validation is achieved using our **$FILECHECK** macro, where Queuenames must exist in the Peripheral Directory (**@** or :PER) and be of TYPE **QUE**. Note this is not a failsafe validation because it permits the entry of a queue which is not a **PRINT** queue (eg. BATCH_INPUT), but it is probably good enough.

Most of our $ED.PRINTER entry and validation macros write their results immediately to the **LISTFILE**, but in this case we merely return the result in the STRING, because we prefer to get the START command switches first (which are appended as switches to the Qname). Since **$ED.PRINTER.SW** (entry and validation of START switches) is going to write both the Qname parameter and the START switches to the LISTFILE we pass the Qname to it as an argument, so that **$ED.PRINTER.SW** has unencumbered use of the CLI STRING thus in **$ED.PRINTER.EDIT:**

```
$FILECHECK/TYPE=QUE @%2\%;          Comment validates default
$ED.PRINTER.Q [!STRING];            Comment result in STRING
$ED.PRINTER.SW [!STRING],%2/%;      Comment Qname passed as Arg1
```

File $ED.PRINTER.SW.CLI

```
STRING [$$ARG1 [!READ/S Enter Switches for START Command&
     (/NL /NL/8BIT /8BIT None)&
     [$$IN_SQUARE [!NEQUAL,,%2%]%2%[!ELSE]None[!END]] [$$BELL]]]
[!EQUAL,,[!STRING]]
     WRITE/L %1%%2% <&>
[!ELSE]
     [!EQUAL,None,[!STRING]]
          WRITE/L %1% <&>
     [!ELSE][!EQUAL,/NL,[!STRING]]
          WRITE/L %1%/NL <&>
     [!ELSE][!EQUAL,/NL/8BIT,[!STRING]]
          WRITE/L %1%/NL/8BIT <&>
     [!ELSE][!EQUAL,/8BIT,[!STRING]]
          WRITE/L %1%/8BIT <&>
     [!ELSE]
          WRITE
          WRITE ERROR - Invalid switches Choose one of those shown
          WRITE
          %0-%
     [!END][!END][!END][!END]
[!END]
```

Validation here is achieved by the most straightforward of all means, namely testing the response against all the valid responses. Note the introduction of a special response namely **None**, to indicate no switches. This is necessary because the null response means to accept the default value. In writing the valid response to the LISTFILE with **WRITE/L** (remembering to include the Qname) note the way '**value <&>**' is used to force a space to be written to delimit this argument from the next in the User Pseudo Macro, and of course, the ampersand to ensure that the NEWLINE written by WRITE will be ignored when the UPM is expanded.

File $ED.PRINTER.DEV.CLI

```
STRING [$$ARG1 [!READ/S Enter Device&
     [!NEQUAL,,%2%][$$IN_SQUARE %2%] [!END][$$BELL]@]]
[!EQUAL,,%2%[!STRING]]
     WRITE
     WRITE ERROR - Must enter a Device
     WRITE
     %0-%
[!ELSE]
     [!EQUAL,,[!STRING]]
          WRITE/L %2% <&>
     [!ELSE]
          $FILECHECK/TYPE=CON/TYPE=LPU @[!STRING]
          [!EQUAL,,[!STRING]]
               WRITE
               WRITE ERROR - Not a valid Printer Device
               WRITE
               %0-%
          [!ELSE]
               [!NEQUAL,,[!PATHNAME :$SYSPARAMS:[!STRING].IN_USE.CLI]]
                    WRITE
                    WRITE ERROR - Device already used for&
                    Printer [:$SYSPARAMS:[!STRING].IN_USE]
                    WRITE
                    %0-%
               [!ELSE]
                    WRITE/L [!STRING] <&>
                    DELETE/2=IGNORE :$SYSPARAMS:%2%.IN_USE.CLI
                    WRITE/L=:$SYSPARAMS:[!STRING].IN_USE.CLI %1%<&>
               [!END]
          [!END]
     [!END]
[!END]
```

This macro is practically identical to the **$ED.PRINTER.Q** macro except that validation requires a file of TYPE **CON** (Console) or **LPU** (Line Printer Unit) in the Peripheral Directory (**@**). Unlike $ED.PRINTER.Q the validated response is written immediately to the LISTFILE. There is however one major additional check necessary, when validating devices. No two processes can own the one device at the one time. As it is outside the scope of this macro to control whether or not the Device is used elsewhere (eg. as an EXEC Enabled Console - if UP starts Print Co-ops before consoles are enabled EXEC will give warnings on attempts to enable consoles already used by a Printer), but we must ensure that we can't START two Print Co-operatives to the one device. Thus we maintain a further Parameter File (User Pseudo Macro) in :$SYSPARAMS, being **device.IN_USE.CLI** containing the name of the Printer using that device. Each time we select a new device we must delete the "in use" flag file for the device we previously used (if we had one), and create one for the new device (provided of course some other Printer wasn't already using it). In order to write the **printername** to this **device.IN_USE.CLI** file, it must be passed as an argument by **$ED.PRINTER.EDIT**. The existence of such parameter files is completely hidden from the User. They are wholly maintained by this macro and **$ED.PRINTER.DEL** which must delete the associated **device.IN_USE.CLI** file on successful deletion of the **printername.PRM.CLI** file.

$ED.PRINTER.YN

Asks Yes/No Question re the state of a Parameter and Writes the current value to the Listfile if the User gives any invalid response or the default one otherwise the new value as specified is written.

Format: $ED.PRINTER.YN arguments

Arguments:

1	current value of parameter
2	value of parameter for Y answer
3	value of parameter for N answer
4-	prompt text

File $ED.PRINTER.YN.CLI

```
[!EQUAL,[!EQUAL,%1%,%2%]N[!ELSE]Y[!END],[$$ARG1 &
[!READ/S %4-% (Y N) &
[$$IN_SQUARE [!EQUAL,%1%,%2%]Y[!ELSE]N[!END]] [$$BELL]]]]
     WRITE/L [!EQUAL,%1%,%2%]%3%[!ELSE]%2%[!END] <&>
[!ELSE]
     WRITE/L %1% <&>
[!END]
```

Rather than having a validation macro tell you to enter "Y or N or NEWLINE to accept the Default" after an invalid response to a Yes/No Question, it is sometimes useful to assume the default on any invalid response. When the default or any other response other than the non-default response is given the default is assumed. When the non-default response is entered the non-default value is written. This is trivial when the response (Y or N) is also the value written to the parameter file (as in the **Satisfactory?** question of **$ED.PRINTER.1**). However in the case of whether or not to "convert to **UPPER** case" (in case this is an upper case only printer) we have chosen to write values to the parameter file **UPPER** (for Y) and **/NULL** (for N) that simplify the coding of **$UP.PRINTER**. We have a similar problem when determining whether or not to enable "**EVEN** pagination", where **ON** is written to the parameter file for Y, and **OFF** for N. So we wrote this common macro to handle both cases where the arguments above enable us to:

1) determine the non-default response
2) determine what to write for a non-default response of Y
3) determine what to write for a non-default response of N
4) specify our own prompt text

Thus the calling sequences to use this macro for both these Questions are:

$ED.PRINTER.YN %4% UPPER /NULL Convert to UPPER Case

and

$ED.PRINTER.YN %8% ON OFF EVEN Pagination

This is non-trivial and you may have to try it out to see how it actually achieves the desired effect.

File $ED.PRINTER.CLNUP.CLI

```
[!EQUAL,%1%,/NULL]
     [!EQUAL,Y,[$$ARG1 [!READ/S Do you want to specify a Cleanup File&
          (Y N) [$$IN_SQUARE N] [$$BELL]]]]
          %0%
     [!ELSE]
          WRITE/L /NULL <&>
     [!END]
[!ELSE]
     [!EQUAL,N,[$$ARG1 [!READ/S Do you still want a Cleanup File&
          (Y N) [$$IN_SQUARE Y] [$$BELL]]]]
          WRITE/L /NULL <&>
     [!ELSE]
          STRING [$$ARG1 [!READ/S Enter CLEANUP File name&
          [!NEQUAL,,%2%][$$IN_SQUARE %2%] [!END][$$BELL]]]
          [!EQUAL,,[!STRING]]
               WRITE/L %2% <&>
          [!ELSE]
               [!EQUAL,,[!PATHNAME :UTIL:FORMS:[!STRING]]]
                    WRITE
                    WRITE ERROR - Cleanup File does not exist in&
                                  :UTIL:FORMS
                    WRITE
                    %0-%
               [!ELSE]
                    [!UEQ,0,[!SIZE :UTIL:FORMS:[!STRING]]]
                         WRITE
                         WRITE ERROR - Cleanup File is empty no point&
                                       having it
                         WRITE
                         %0-%
                    [!ELSE]
                         WRITE/L [!STRING] <&>
                    [!END]
               [!END]
          [!END]
     [!END]
[!END]
```

If a Cleanup File is required validation consists of checking that the specified file exists in **:UTIL:FORMS** and that it is a non-empty file (in case the User accidently entered a form name). If the user made an error entering the Cleanup File name we give him an escape clause by allowing him to revert to not having one at all, rather make him have repeated incorrect guesses. In this way when he gets back to CLI he can get a list of **+.CLEANUP** files in :UTIL:FORMS. Alternatively we could have displayed a list from which to choose, but we would lose some generality in so doing, since we would have to make some (perhaps invalid) assumption about the format of the names of the cleanup files on your System.

File $ED.PRINTER.FORMS.CLI

```
[!EQUAL,%1%,/NULL]
     [!EQUAL,Y,[$$ARG1 [!READ/S Do you want to specify DEFAULTFORMS&
          (Y N) [$$IN_SQUARE N] [$$BELL]]]]
          WRITE DEFAULTFORMS override LPP & CPL settings
          %0%
     [!ELSE]
          WRITE/L /NULL <&>
     [!END]
[!ELSE]
     [!EQUAL,N,[$$ARG1 [!READ/S Do you still want DEFAULTFORMS&
          (Y N) [$$IN_SQUARE Y] [$$BELL]]]]
          WRITE/L /NULL <&>
     [!ELSE]
          STRING [$$ARG1 [!READ/S Enter DEFAULTFORM name&
          [!NEQUAL,,%2%][$$IN_SQUARE %2%] [!END][$$BELL]]]
          [!EQUAL,,[!STRING]]
               WRITE/L %2% <&>
          [!ELSE]
               [!EQUAL,,[!PATHNAME :UTIL:FORMS:[!STRING]]]
                    WRITE
                    WRITE ERROR - Defaultform does not exist in&
                                  :UTIL:FORMS
                    WRITE
                    %0-%
               [!ELSE]
                    [!UNE,0,[!SIZE :UTIL:FORMS:[!STRING]]]
                         WRITE
                         WRITE ERROR - [!STRING] is not an FCU created&
                                       form
                         WRITE
                         %0-%
                    [!ELSE]
                         WRITE/L [!STRING] <&>
                    [!END]
               [!END]
          [!END]
     [!END]
[!END]
```

This macro is identical in structure to **$ED.PRINTER.CLNUP**, with the difference existing only in the test for validity, where this time the file must be an **empty** file in **:UTIL:FORMS**. A more failsafe check would also insist on the file having a UDA (User Data Area), which is where **FCU** (The Forms Control Utility) writes its information, but in practice very few empty files in :UTIL:FORMS are not FCU FORMS files.

File $ED.PRINTER.QS.CLI

```
[!EQUAL,,%1%]
     [!EQUAL,Y,[$$ARG1 [!READ/S Do you want to specify additional&
     Print Queues for this Printer (Y N) [$$IN_SQUARE N] [$$BELL]]]]
          $ED.PRINTER.QS.1
     [!END]
[!ELSE]
     [!EQUAL,Y,[$$ARG1 [!READ/S Do you want to CHANGE additional&
     Print Queues for this Printer (Y N) [$$IN_SQUARE N] [$$BELL]]]]
          $ED.PRINTER.QS.1 %-%
     [!ELSE]
          WRITE/L %-%<&>
     [!END]
[!END]
```

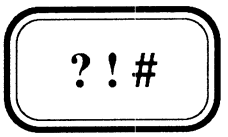
If there are currently no additional queues and the user wants some, **$ED.PRINTER.QS** calls **$ED.PRINTER.QS.1**, otherwise there is nothing to do since a non-existent argument-10 in the parameter file (argument-11 in the macros) indicates no additional queues. If there are additional queues and the User wants to change them (possibly to re-set to having no additional queues) **$ED.PRINTER.QS.1** is again called, this time with the existing list as arguments (which are used in turn as defaults until the user elects to finish, otherwise the exisiting list is written to the LISTFILE.

File $ED.PRINTER.QS.1.CLI

```
[!EQUAL,Y,&
[$$ARG1 [!READ/S Any more Q's (Y N) [$$IN_SQUARE N] [$$BELL]]]]
     $ED.PRINTER.Q %1%
     WRITE/L [!STRING] <&>
     %0% %2-%
[!END]
```

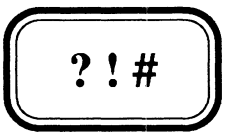
If the user wants to specify more queues, our existing **$ED.PRINTER.Q** macro is used to validate. The valid Qname is then written and a recursive call made (using the next Qname from the previous list as the default) to get the next one (if any more are required).

THIS PAGE INTENTIONALLY LEFT BLANK



CHAPTER 28

Initial IPC Macros



CHAPTER 28

Initial IPC Macros

28.1 What is an Initial IPC?

An **IPC** is an Inter-Process Communication. Processes on the System may communicate with each other by sending and receiving IPCs. To use this feature programmers have to write programs that use the appropriate System Calls to send and/or receive IPCs. CLI does not have a command for receiving an IPC but it can send one with the command **CONTROL**. An initial IPC is a message sent to a process on initiation. The process receiving the initial IPC will have been programmed to do something with it. In the case of CLI, it has been programmed to treat its initial IPC as a set of commands or macros to be carried out before giving control to the User. When as is normally the case, CLI is specified as a User's initial program (with PREDITOR), a pathname to a file being the **Initial IPC** file is also specified. This file holds the initial IPC message for CLI. Since CLI's buffer for holding its initial IPC message is 512 characters, the contents of this initial IPC file, or **Initial IPC Macro** as we will call it in future, **must not be more than 512 characters.**

28.2 Why have an Initial IPC Macro?

The main reason for having an initial IPC macro is to set the User up with the appropriate initial environment, which includes:

- Having the correct **DEFACL** so that files created in the session can be accessed by all those who need access to them. The default of **[!USERNAME],OWARE** is rarely appropriate.

- Having the correct **SEARCHLIST** so that all the Software and Databases the user needs access to are accessible. The default, being whatever SEARCHLIST EXEC has, is almost never appropriate.

- Having the correct WORKING **DIRECTORY**, however this is used less frequently because the default of **:UDD:[!USERNAME]** very often is correct.

However from a System Manager's point of view it can serve a far more important purpose of giving him total control over the access to the System. Very often in a production environment the User never actually gets to interact with CLI. CLI and the Initial IPC Macro serve the sole purpose of setting up the correct environment and then **CHAIN**ing to the User's application system main menu processor. Although each user can be set up with their own Initial IPC Macro (this practice is common), setting up the appropriate environment for that User, we believe in an approach where every User on the System is channelled through the one Initial IPC Macro. At first this may seem impossible since not every User needs the same environment, but by choosing well planned structured Usernames all the information about each User's environment can be buried in the Username which the universal Initial IPC Macro will obviously have to decode.

28.3 A Universal Initial IPC Macro

The first command in any initial IPC macro, should be the command,

PROMPT BYE

This means that any attempt by the User to break out of the initial IPC before it has completed (say with a **CTRL-C CTRL-A**, which would otherwise present the user with the CLI prompt), will cause the User to be immediately logged off. CLI always executes the commands in the PROMPT buffer before presenting the user with the prompt for a command, so by placing the command **BYE** in the prompt buffer, the User is logged off. Of course for users who legitimately require CLI access you will have to clear the prompt buffer (with a PROMPT/K) somewhere in the initial IPC macro before giving them control, but by always issuing this command first you can make sure no user gets control until you're ready for him to. Furthermore no batch job will run if the CLI PROMPT is set to BYE so we must also make sure we clear the PROMPT buffer if this is a Batch Job. Since the User can't interrupt a Batch Job we can do this immediately.

Bearing in mind the restriction on the size of the initial IPC macro of 512 characters, and the danger of forgetting about this at some stage in the future, we suggest that the bulk of the code be handled by a subordinate macro, which since it is not being passed as an IPC can be as large as you like (provided it doesn't blow CLI memory). Thus we might code a universal initial IPC macro like this:

File $IIPC.CLI

```
PROMPT BYE
[!EQUAL,BATCH,[!LOGON]]
     PROMPT/K
[!END]
SEARCHLIST :UTIL
$LOGONCHECK
$IIPC.1 [$$USER_DECODE]
```

Note that before our initial IPC macro can invoke any other macro we have to make sure that the directory where such macros are located is on the user's SEARCHLIST. Although AOS/VS can support up to eight directories on a SEARCHLIST we do not like long searchlists, and believe that in most environments you need only two directories on your searchlist, namely, your application system's software directory and :UTIL. This does not mean we believe all DG software and other 3rd party products (such as ORACLE POWERHOUSE etc) necessary for the running of your software should be lumped together in :UTIL. What it does means however is that entry level access to them, should be through :UTIL where a series of linkfiles exists to where the software really is. Thus we assume that all our **$** macros and **$$** User Pseudo Macros are accessible through :UTIL, and place this directory on the SEARCHLIST before invoking any of them. In case you've forgotten the trick in Chapter 20, let's assume you have bought the companion Magnetic Tape with all the example Macros in this Manual on it, but decide to go against our advice to load them in :UTIL, and decide to put them in a directory called :$MACROS. In order for them to be accessible through :UTIL without actually being in :UTIL you must create a set of linkfiles in :UTIL, which can be done as follows:

```
) SUPERUSER ON
*) DIRECTORY :$MACROS
*) CREATE/LINK :<UTIL $MACROS>:([!EFILENAME [!FILENAMES +]])
```

Now getting back to our initial IPC macro, **$LOGONCHECK** is a special macro that prevents any single Username from logging on at a console more than once. This stops users using another person's log-on, and in these days of hackers and the like this is a feature we're sure you'll find useful even if you don't like the rest of our initial IPC macro (your auditors will like it too). **$$USER_DECODE** is a User Pseudo Macro that decodes the user's username, so that the real initial IPC macro (ie the one that does the bulk of the work) **$IIPC.1**, can determine what it has to do on this user's behalf. Before looking at $$USER_DECODE, we will have to determine a structure for our usernames. We favour usernames whose components are separated from each other by some special character. Although we prefer periods (full-stops or dots) to be used because this simplifies separation of the components using the **!ENAME** and **!EEXTENSION** Pseudo macros, we are aware that this can cause some problems. For example Application Systems that use ORACLE have a problem because Users defined in an ORACLE database cannot have imbedded dots. If your AOS/VS Usernames had imbedded dots, you would have to force a separate logon to the database. So we will code our $$USER_DECODE macro to recognize both dots and underscores as component delimiters (but not a mixture of them). First of all we have to decide what these components are. This will vary from System to System and if your structure differs from the assumptions we are about to make you will have to modify both **$$USER_DECODE** and **$IIPC.1** before you could use our **$IIPC** macro. Note however that **$LOGONCHECK** is completely independant of this structure and immediately useful on any AOS/VS System.

We will assume usernames have the following structure:

sys.{id}{.type} or **sys_{id}{_type}**

where

sys identifies the User's Application System. **$$IIPC.1** is going to use this information in establishing the initial SEARCHLIST and DEFACL for the User.

id is an identifier uniquely identifying this user within his Application System. If there is only one user of a particular type this may be omitted.

type identifies the type of user. When omitted the user is assumed to be a production user without access to CLI, with two other types recogized, namely **MGR** for application manager (Project Leader) and **DEV** for development user (Programmer). In a mixed environment there would typically be one MGR, several DEV's, and many production users, while at a production site there would be one MGR, many production users but no DEV users. Both MGR and DEV users have access to CLI.

eg. XYZ..MGR XYZ.ME.DEV XYZ.FRED.DEV XYZ.YOU XYZ.MARY XYZ.ME
or ORA__MGR ORA_ME_DEV ORA_FRED_DEV ORA_YOU ORA_MARY ORA_ME

$$USER_DECODE will return a list of arguments being the various components of the Username and an indication of the component delimiter thus:

. sys .id .type or **_ sys _id _type**

$$USER_DECODE does not expect an argument since it assumes that what we have to break down is **[!USERNAME]** but we allow it to take a Username as an argument if for no other reason other than to enable us to test it, without having to become different Users.

File $$USER DECODE.CLI

```
[$$USER_DECODE.1 [!EQUAL,,%1%][!USERNAME][!ELSE]%1%[!END]]&
```

File $$USER DECODE.1.CLI

```
[!EQUAL,%1%,[!ENAME %1%]]_[!ELSE].[!END] &
[$$ARG_SPLIT[!EQUAL,%1%,[!ENAME %1%]]/DEL=_[!END] %1%]&
```

$$USER_DECODE.1 assumes the existence of a general purpose argument splitting UPM **$$ARG_SPLIT**, which takes a **/DEL=?** switch to specify the character upon which its argument is to be split. If this switch is not supplied a period (.) is assumed. If the username and the username with its extension stripped off (**[!ENAME %1%]**) are the same then it must be of the form where underscore (_) is the delimiter.

File $$ARG SPLIT.CLI

```
[!EQUAL,,%0/DEL=%][$$EXT_SPLIT %1%]&
[!ELSE][$$ARG_SPLIT.1%0/% [!EXPLODE %1%]]&
[!END]&
```

File $$ARG SPLIT.1.CLI

```
[!NEQUAL,,%1%]&
[!EQUAL,%0/DEL=%,%1%],[!END]%1%[%0% %2-%]&
[!END]&
```

File $$EXT SPLIT.CLI

```
[!EQUAL,%1%,[!ENAME %1%]]%-%[!ELSE]&
[%0% [!ENAME %1%] [!EEXTENSION %1%][!NEQUAL,,%2%] %2-%[!END]]&
[!END]&
```

If there is no delimiter specified **$$EXT_SPLIT** is used, otherwise **$$ARG_SPLIT** explodes the characters of its argument apart, and **$$ARG_SPLIT.1** joins them back together again, but expanding a comma whenever it sees a delimiter. **$$EXT_SPLIT** on the other hand recursively splits its first argument into its root and extension, until it has none. The split executes noticeably faster when it uses **$$EXT_SPLIT.** Note all this without the use of CLI STRINGs.

Before looking at **$IIPC.1** which is too dependant on our assumptions to be useful immediately in your own environments, let's look at **$LOGONCHECK** which stops users logging on under the same username more than one at a time.

$LOGONCHECK.CLI

This macro terminates this process if this User is already logged on, unless it is running in Batch. An appropriate message is displayed informing the User why access to the system has been denied.

Format: $LOGONCHECK

File $LOGONCHECK.CLI

```
[!NEQUAL,[!LOGON],BATCH]
     $LOGONCHECK.1 :UDD:[!USERNAME]:[!USERNAME]_LOGGED_ON
[!END]
```

File $LOGONCHECK.1.CLI

```
[!NEQUAL,,[!PATHNAME %1%]]
     [!EQUAL,,%2%]
          %0% %1%,[%1%]
     [!ELSE]
          PUSH; PROMPT POP
          DELETE/2=IGNORE :UDD:[!USERNAME]:?[!PID]WHO.TMP
          WHO/L=:UDD:[!USERNAME]:?[!PID]WHO.TMP %2%
          STRING [:UDD:[!USERNAME]:?[!PID]WHO.TMP]
          DELETE :UDD:[!USERNAME]:?[!PID]WHO.TMP
          [!EQUAL,[!USERNAME],[$LOGONCHECK_WHO [!STRING]]]
               [!NEQUAL,[!CONSOLE],%3%]
                    [!UNE,[!PID],%2%]
                         WRITE [$$BELL]
                         WRITE SORRY - You cannot Log on as someone&
                              is already logged on Under your Name
                         WRITE,,,,,,,,,[!STRING]
                         WRITE
                         BYE
                    [!ELSE]
                         DELETE %1%
                         WRITE/L=%1% [!PID],[!CONSOLE]<&>
                    [!END]
               [!END]
          [!ELSE]
               DELETE %1%
               WRITE/L=%1% [!PID],[!CONSOLE]<&>
          [!END]
          POP
     [!END]
[!ELSE]
     WRITE/L=%1% [!PID],[!CONSOLE]<&>
[!END]
```

File $LOGONCHECK WHO.CLI

```
%3%&
```

Basically after checking that this is not a Batch Job (in which case we ignore the check), if a flag-file (**[!USERNAME]_LOGGED_ON** in **:UDD:[!USERNAME]**) already exists the User is already logged on somewhere so we terminate with a **BYE** command. If it does not then we create it to prevent subsequent Log-ons. If only it were so simple. The problem is that we cannot guarantee to get control back after the User has logged off, in order to delete this flag-file at the end of the session. The User might Log Off with a **CTRL-C CTRL-B**, or **$IIPC.1** might **CHAIN** to the application, in which case the flag-file would remain and the user would not be able to get back on. So as well as checking the existence of the file, if it exists we have to find out why? If it is because the User really is logged on then we terminate, but if it exists as a leftover from a previous log-on we can ignore it and let the user log on. To help us make this decision we store some information in the flag-file (which now becomes a User Pseudo Macro), being this process's **PID** and **CONSOLE**. Note that although the Username wouldn't normally need to be in the flag-file's name (since we've placed it in a directory unique for each user), some environments have the **:UDD:[!USERNAME]** directories as links to a common directory which then becomes the Initial Working Directory for all users linked to it. For our macro to work in such an environment the username must be in the flag-file name.

If the flag-file does not exist we simply create it. If it does then we call ourselves with its name and contents as arguments. Thus in the main body of the code for **$LOGONCHECK.1**, arg1 is the pathname to the flagfile, arg2 is the PID of the User who last logged on at a console under this username, and arg3 is the console at which it was done. First we find out who if anyone is logged on at that PID now. If that PID's username is different to ours, that person who last logged on as us can't still be logged on, so we delete the old flag-file and create a new one with our PID and CONSOLE in it. However if the usernames are the same and our console is the same as that of the last person to log on as us, then since only one process can own a console it is us and we are still logged so there is nothing to do (in other words the same process ran $LOGONCHECK twice, or this process is a son of the original creator of the flag-file). But if the console is different and our PID is the same, then we just happened to log on at a different console and by pure chance got allocated the same PID, in which case we delete the old flag-file and create a new one. Otherwise that person is still logged on as us so we terminate after giving an appropriate message.

That explanation of what **$LOGONCHECK** is doing and why, is pretty long-winded and probably unsatisfactory, but $LOGONCHECK really does work, but no doubt you'll have to try it for youselves before you are convinced. The macro is almost certainly not 100% failsafe, (for example the time window between flag-file deletion and creation of a new one potentionally allows a second person to log on), but we have not yet been able to crash it in a production environment.

Now getting back to **$IIPC.1**, we are also going to have to make certain assumptions about **DIRECTORY structures** and the ACLs of these directories in order to establish SEARCHLISTs and DEFACLs. We will make the following assumptions:

- A directory (or at least a link to a directory) called **sys** exists in **:PROD** for each Production Application System installed. A similar directory (or linkfile) exists in **:DEV** for each Development Application System installed.

- **:PROD:sys** contains at least one arbitrarily named directory, and perhaps several, where each such directory is a different version of the Application System software. (Ditto **:DEV:sys**)

- **:PROD:sys** contains a Linkfile **REV** linking to whichever of its subdirectories is the current revision of the software. (Ditto **:DEV:sys**)

- For users of type **DEV**, access is only ever to the **:DEV** version of the system (ie **:DEV:sys:REV**).

- For users of type **MGR** access is to the **:DEV** version, unless it does not exist in which case access is to the **:PROD** version.

- For production users access is to the **:PROD** (ie **:PROD:sys:REV**), unless it does not exist in which access is to the **:DEV** version.

- The ACL for **:PROD** (and **:DEV**) is **+,RE**

- The ACL for **:PROD:sys** (and **:DEV:sys**) is **sys.-.MGR,OWARE sys.+,RE** while the files in these directories have an ACL of **+,RE**. The implication is that only **MGR** users can place files in these directories.

- Each **:PROD:sys:rev** directory may contain several subdirectories at least one of which is called **EXEC**, which contains the Application System's executable software and which is placed on all Users SEARCHLISTs. (Ditto **:DEV:sys:rev**)

- Each User is assumed to be already in the correct Working Directory namely **:UDD:[!USERNAME]**.

- If the directory **:PROD:sys:rev:DATA** also exists it is also placed on the User's searchlist. Note it may of course be a linkfile to another directory. This is assumed to contain the Application System's Databases and its ACL is most likely to be **sys.-.MGR,OWARE sys.+,WARE**. (Ditto **:DEV:sys:rev:DATA**)

- The DEFACL of production Users will be **sys.+,OWARE**

- The DEFACL of **MGR** users the default of **[!USERNAME],OWARE**

- The DEFACL of **DEV** users will be **sys.-.MGR,OWARE [!USERNAME],OWARE**

In all of the above underscores replace periods in the ACLs and DEFACLs if the Usernames are of the form **sys_{id}{_type}**.

Once the SEARCHLIST and DEFACL have been set up according to the above rules, the prompt buffer will be cleared for non-production users (ie MGR and DEV). For non-MGR Users this is also a covenient place to implement a primitive but none-the-less extremely effective and useful Application Lock-Out. If the file **sys_DISABLED** exists anywhere in either the user's working directory or on his searchlist, we terminate logging the user off and informing him that "Access to the sys System is currently Disabled". Such a file could be created by the System Manager in :UTIL whenever he needed to disable access to the System, but more commonly the Application System's Backup/Recovery Job (which would not be allowed to begin until all Users of the System are logged off), would probably create this file in **:PROD:sys:rev:DATA** as its first step to prevent subsequent log-ons to the System, and then delete it on successful completion. There is no need for Operations to be involved in locking out the Users, say by Disabling the Consoles. By looking for the **sys_DISABLED** flag in the Searchlist we can disable access at various levels, including individual Users (by creating it in **:UDD:[!USERNAME]**), a particular software revision (in **sys:rev:EXEC**), a particular Database (in **sys:rev:DATA**), or system wide (in **:UTIL**). Any **sys_DISABLED** flag should have an ACL of **[!USERNAME],OWARE +,RE**, so that only its creator can delete it but anybody can see it.

Finally if the file **STARTUP.CLI** exists anywhere in the user's current working directory or on the searchlist that macro will be invoked. Typically such a macro will exist in the **sys:rev:EXEC** directory and will contain the command to fire up the production system. eg:

```
CHAIN MAINMENU
```

However by placing a **STARTUP** macro in a User's **:UDD:[!USERNAME]** directory the Application **MGR** can do different things for different users since the version in the Working Directory will be the one found and therefore run. Similarly the **MGR** himself or any of the **DEV** users can set up their own **STARTUP** macro in their **:UDD:[!USERNAME]** directory to override the default one in **sys:rev:EXEC**. The **STARTUP** macro will be passed the same arguments as **$IIPC.1** received from **$IIPC**, namely:

```
? sys ?id ?type
```

By investigating arg4 the **STARTUP** macro can be coded to do different things for the different types of Users.

To simplify setting up the Searchlist we will write a little UPM to return either **:PROD** or **:DEV** according to the rules described on the previous page, which we have called **$$IIPC.ENV**. Since it needs to know what type of user this is and whether or not **:PROD** and/or **:DEV** exist, it is invoked as follows:

```
[$$IIPC.ENV ?,?type,[!PATHNAME :PROD],[!PATHNAME :DEV]]
```

and looks like this:

File $$IIPC.ENV.CLI

```
:[!EQUAL,%2%,%1%DEV]DEV[!ELSE]&
[!EQUAL,,%2%][!EQUAL,,%3%]DEV[!ELSE]PROD[!END]&
[!ELSE][!EQUAL,,%4%]PROD[!ELSE]DEV[!END]&
[!END]&
```

$IIPC.1

Initialization Macro for all Users. Assumes username format sys?{id}{?type} ? = "." or "_", and directory structures :env:sys:rev:<EXEC{ DATA}> where sys is from username, rev is determined by link :env:sys:REV, and env is PROD or DEV as appropriate. These directories if they exist are placed on the Searchlist ahead of :UTIL. Once Searchlist is established the Application Lock-Out sys_DISABLED is checked, and if okay the macro STARTUP.CLI is run if it exists.

Format: $IIPC.1 ? sys ?{id}{ ?type}

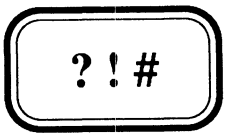
Arguments:

?	? = "." if username contains one otherwise "_"
sys	Application System
?id	Identifier (unique within System and Type)
?type	Type = ?MGR or ?DEV or null for Production User

File $IIPC.1.CLI

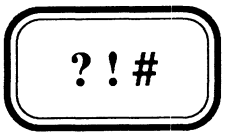
```
Comment Our macro requires Usernames to have at least one delimiter
Comment Ignore other Usernames except for running STARTUP macro.

[!NEQUAL,,%3%]
     [!EQUAL,,%4%]
          DEFACL %2%%1%+,OWARE
     [!ELSE]
          DEFACL [!EQUAL,%1%DEV,%4%]&
%2%%1%-%1%MGR,OWARE [!END][!USERNAME],OWARE
     [!END]
     SEARCHLIST/1=IGNORE &
[$$IIPC.ENV %1%,%4%,[!PATHNAME :PROD],[!PATHNAME :DEV]]&
:%2%:$REV$:(DATA EXEC) [!SEARCHLIST]
     [!NEQUAL,,[!PATHNAME %2%_DISABLED]]
          WRITE [$$BELL]
          [!EQUAL,%4%,%1%MGR]
               WRITE WARNING - Disable flag [!PATHNAME %2%_DISABLED]&
                               exists
               WRITE
          [!ELSE]
               WRITE SORRY - Access to the %2% System is currently&
                               Disabled
               WRITE
               BYE
          [!END]
     [!END]
     [!NEQUAL,,%4%]
          PROMPT/K
     [!END]
[!END]
[!NEQUAL,,[!PATHNAME STARTUP.CLI]]
     STARTUP %-%
[!END]
```

Notice how we cater for both delimiters by concatenating **%1%** and **DEV** before comparing them with **%4%** (the ?type portion of the username). Testing for a production user is considerably simpler since we just test for a null ?type. In establishing the searchlists the **/1=IGNORE** gets around any problem we might have with non-existant directories. The process is split into two searchlist commands "**(DATA EXEC)**", so that if either one exists they are placed on the searchlist and if both exist EXEC comes first (by inserting it at the head of the existing searchlist, after first attempting to put DATA on). Only **MGR** users are allowed to log on when a **sys_DISABLED** flag exists, and even they are at least given a warning message showing the full pathname to it. Notice that for a Username that does not conform to our format (doesn't have at least one "." or "_"), unless a **STARTUP.CLI** macro exists in their **:UDD:[!USERNAME]** directory or **:UTIL** (the only directory on the searchlist at the time for such users) they will be logged off, because the Prompt Buffer still has a **BYE** command in it. This could easily be changed if not considered desirable but we are reasonably happy with it.



CHAPTER 29

Variable Length Argument Lists



CHAPTER 29

Variable Length Argument Lists

29.1 Single Variable Length Lists

If single variable length argument lists were all we wanted to talk about in this Chapter we wouldn't bother to write the Chapter at all, since CLI makes dealing with them almost trivial. We have in fact been using them in examples throughout the Manual. Because of CLI's **%n-%** argument expansion syntax, the only thing you have to remember is that your variable length list must come last. All a called macro needs to know about a variable length argument list is which argument it starts on. All our recursive Macro and User Pseudo Macro examples are of course examples of macros handling variable length argument lists.

29.2 Multiple Variable Length Lists

CLI does not handle multiple variable length argument lists **well**. Most people including some very experienced and talented CLI Macro writers would have left the last word of the previous sentence off. However there just does happen to be a way to handle multiple variable length lists, but it requires the use of an **undocumented feature**, which we cannot (nor can DG) guarantee will remain.

Before describing this undocumented feature we will describe a method of handling multiple variable length argument lists, that does not use any undocumented feature, but which unfortunately only works in a limited set of circumstances. The technique is really quite simple in that for each variable length list we need (over and above one list), we simply enclose the list in round brackets to tell CLI to treat it as one argument. In this way each such list is in fact a single argument and can be referred to as a single unit. However you must remember that all such macros must be invoked using the square bracket syntax, because outside square brackets, round brackets have an entirely different meaning (which is to generate multiple commands - See Chapter 9 if you've forgotten). Let's look at an example. Assume we have a macro **$SETACL**, which has two variable length lists. One is the list of filenames whose ACL we want to set, the other is the ACL we want to set it to.

$SETACL

Sets the specified list of files and/or templates to the specified ACL. It must be invoked with square brackets.

Format: [$SETACL (templates) acl]

Arguments:

(templates) specifies the list of files whose ACL you want set. The round brackets must be included.

acl The ACL to which you want these files set

File $SETACL.CLI

```
ACL %1% %2-%
```

Clearly this is a contrived example to highlight a point, since the macro itself is really nothing more than the ACL command itself, so who would use $SETACL when we can just as easily use ACL. The macro could have been equivalently written **ACL %-%**, but we wanted to emphasize the fact that **%1%** is in fact our first variable length argument list, while **%2-%** is the second in normal format. The ACL command after substitution of its arguments becomes:

ACL (templates) acl

This causes the ACL command to be invoked once for each template.

The limited set of circumstances under which this technique works is where when finally you actually want to use the variable length lists passed as single arguments (eg %1% ie **(templates)**), you require that they generate multiple commands, one for each item in the list, as we did in the above example.

Notice that we could not have reversed the roles of our two lists, because there is no way we could have generated a satisfactory command. For example if the syntax of **$SETACL** was **[$SETACL (acl) templates]** the following:

ACL (%2-%) %1%

would not work, since this becomes:

ACL (templates) (acl)

For example:

ACL (+.CLI +.PR) (+,RE)

which would generate two garbage ACL commands, namely

ACL +.CLI +
ACL +.PR RE

However enough of contrived examples let's look at a meaningful and useful example of this technique. Let's consider a generalized CLI string input and validation routine for non-numeric input. Such a macro, which we will call **$INPUT**, will require two variable length lists, one being the list of valid responses, the other being the prompt text. The most important decision to make concerns which list to include in round brackets, and we will of course only be able to use this technique if we can ensure that the list we choose to enclose in round brackets, will only ever be used with the brackets on. It is our intention to display the prompt to the user in the following fashion:

prompt text (valid responses) [default]

in line with generally accepted DG conventions. This clearly points to the valid response list being the one to enclose in round brackets.

$INPUT

This macro returns in the CLI STRING the User's response to the supplied prompt text, provided a valid response was entered. Prompt is:

prompt text (valid resp's){ [default]}

Format: [$INPUT{/switches} (valid resp's) prompt text]

Optional Switches:

/A{ppend}	Append the response to the existing [!STRING]
/BELL	Rings the Bell on Prompt (otherwise only on errors)
/DEF=default	Set the default response for when user hits Newline
/HLP=path	Pathname to a HELP file typed if user enters ?
/NULL=resp	Sets the null response (to distinguish from default)

Arguments:

prompt text	Prompt text
valid resp's	List of valid responses (including Default & Null)

eg: **[$INPUT/DEF=6250/HLP=DENSITY.HELP (800 1600 6250) Select Density]**

File $INPUT.CLI

```
[!EQUAL,,%0/RESPONSE%]
     PUSH; PROMPT POP
     [$INPUT.ASK%0/% %-%]
[!ELSE]
     POP
     [!NEQUAL,,%0/A/AP/AP/APP/APPE/APPEN/APPEND%]
          [!NEQUAL,,%0/RESPONSE=%]
               STRING [!STRING]%0/RESPONSE=%
          [!END]
     [!ELSE]
          [!EQUAL,,%0/RESPONSE=%]
               STRING/K
          [!ELSE]
               STRING %0/RESPONSE=%
          [!END]
     [!END]
[!END]
```

In the same way as we did with our environment preserving **$NUMBER** macro, we **PUSH** in a top level macro and call a subordinate to do a recursive prompt-and-validate-till-valid cycle. We cannot do an immediate **POP** on return because we will then lose the user response held in the CLI STRING. When it has a valid response the subordinate macro, **$INPUT.ASK** calls this macro with an undocumented **/RESPONSE=value** switch, which enables us to **POP** because we get hold of the response through this switch. The current environment's STRING is then set appropriately. Note how we deliberately take special steps to avoid issuing a **STRING** command without an argument which would cause the current STRING value to be displayed. Note also that in order to preserve our two variable length lists **$INPUT.ASK** must be invoked using the square bracket syntax, otherwise $INPUT.ASK would be invoked once for each valid response.

File $INPUT.ASK.CLI

```
STRING [$$ARG1 [!READ/S %2-% %1% &
[!NEQUAL,,%0/DEF%][$$IN_SQUARE %0/DEF=%] [!END]&
[!NEQUAL,,%0/BELL%][$$BELL][!END]]]

[!EQUAL,,[!STRING]]
     [!NEQUAL,,%0/DEF%]
          $INPUT%0/%/RESPONSE=%0/DEF=%
     [!ELSE]
          WRITE
          WRITE ERROR - There is no default response select from&
               options listed[!NEQUAL,,%0/HLP=%] or ? for HELP[!END]
          WRITE [!EQUAL,,%0/BELL%][$$BELL][!END]
          [%0-%]
     [!END]
[!ELSE]
     [!EQUAL,**,[!NEQUAL,,%0/HLP=%]*[!END][!EQUAL,?,[!STRING]]*[!END]]
          WRITE [!EQUAL,,%0/BELL%][$$BELL][!END]
          TYPE %0/HLP=%
          [%0-%]
     [!ELSE]
          VAR0 0
          $INPUT.CHK [!STRING] %1%
          [!UEQ,0,[!VAR0]]
               WRITE
               WRITE ERROR - Invalid response select from options&
                     listed[!NEQUAL,,%0/HLP=%] or ? for HELP[!END]
               WRITE [!EQUAL,,%0/BELL%][$$BELL][!END]
               [%0-%]
          [!ELSE]

     $INPUT%0/%/RESPONSE=[!NEQUAL,[!STRING],%0/NULL=%][!STRING][!END]

          [!END]
     [!END]
[!END]
```

File $INPUT.CHK.CLI

```
[!EQUAL,%1%,%2%]
     VAR0 1
[!END]
```

The reference to our two variable length lists in the **!READ** pseudo as **%2-% %1%**, ie **prompt text (valid resp's)**, should be fairly obvious, but our subsequent use of the valid responses in a check to see if the user's response matches one of them is less so. We had to arrange for the validation to work in such a way that we could call a validation macro once for each valid response, and be able to determine at the end whether or not the user entered one of them. You should now be able to see how **$INPUT.CHK** achieves this. The recursive calls after the errors must be made using the square bracket syntax for the same reason as in the original call of **$INPUT.ASK**. Notice also our use of the special technique of Chapter 13 to effect the complex logic expression:

IF a HELP File is provided AND the User entered a ?

29.3 The Undocumented Feature

Our example where **$SETACL** has a syntax of **[$SETACL (acl) templates]**, could in fact be made to work if CLI had a **!ROUNDOFF** Pseudo Macro to remove the round brackets. **$SETACL** could then be written:

```
ACL (%2-%) [!ROUNDOFF %1%]
```

CLI does not have such a Pseudo Macro, but we could and can solve the problem by writing our own User Pseudo Macro **$$ROUNDOFF**. But we are not going to for several reasons, one it's too hard (we tried for a while and gave up), two it would be horribly inefficient, and three there is an undocumented feature of CLI, that makes removing round brackets a piece of cake.

Believe it or not round brackets around round brackets takes them off (provided the command in which you attempt to take them off is not enclosed in square brackets).

Thus we could code our new format **$SETACL** thus:

```
ACL (%2-%) (%1%)
```

Which becomes:

```
ACL (templates) ((acl))
```

Which due to this **undocumented feature** becomes:

```
ACL (templates) acl
```

Which is the correct command.

We could also reverse the roles of our two variable length lists in **$INPUT**, giving it the syntax **[$INPUT (prompt text) valid resp's]**, however the trick in **$INPUT.ASK** to remove the brackets from the prompt text is not simply a matter of:

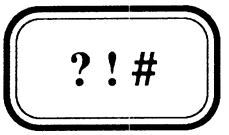
```
STRING [$$ARG1 [!READ/S (%1%) (%2-%) &
```

because inside square brackets, the brackets around brackets trick does not work. Thus we would have to code it as follows:

```
STRING (%1%)
STRING [$$ARG1 [!READ/S [!STRING] (%2-%) &
```

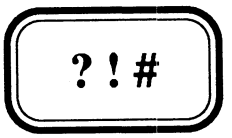
which places an arbitrary restriction on the length of the prompt text of 128 characters (the maximum length of the CLI STRING variable), which the original version did not. The original version is also better because it doesn't rely on any undocumented feature. This feature however can be extremely useful and has remained a feature of AOS/VS CLI since its original release, so hopefully it will not be taken away. Perhaps we can even force DG to document it, so that it will be guaranteed to remain.

THIS PAGE INTENTIONALLY LEFT BLANK



CHAPTER 30

Examples



CHAPTER 30

Examples

30.1 A Simple Menu Processor

Well, if you're still with us, there's nothing else we can think of to teach you about writing CLI Macros, so we thought we'd toss in a couple of our favourites which we think you might find useful. Just about everyone who has got into writing non-trivial CLI macros has attempted to write a menu processor of sorts. Usually they are extremely application specific, and nearly always difficult to maintain. If you are one of these people, we are sure you could have made good use of **$INPUT** from the previous Chapter had you had it at the time, to significantly simplify the selection of an option and validation part of your code. We propose to show you a simple generalized menu processor that makes defining menus and the code to process them trivial. Despite its apparent usefulness in coding such a utility we are not going to make use of **$INPUT**.

Our generalized CLI macro menu processor, called **$MENU**, basically takes a **menuname** as an argument. All you have to do to use it is create a text file, **menuname.MENU** for displaying the menu options on the screen, and a Macro for each valid option called **menuname.option.CLI** and the command **$MENU menuname** does the rest.

$MENU

Enables you to process Menus with CLI macros. All you need to do is create a Text file "menuname.MENU" with the menu screen in it (best if it starts with a Form-Feed Blank first page), then a suite of action macros for each option, "menuname.option.CLI", and run it. $MENU recognises either \ or / as the Exit option. Global options, "GLOBAL_OP.option.CLI" can be set up. Unknown switches are passed on to the option macros.

Format: $MENU{/switches} menuname{,prompt text}

Switches:

/Line=n	Specifies screen line for prompt (Default Line 22)
/Col=n	Specifies screen column for prompt (Default Col 2)
/Autoexit	Forces exit to previous level after running an Option

Arguments:

menuname	Identifies the menu text file and action macros
prompt text	Used when prompting selection (Default "Select Option:")

File $MENU.CLI

```
[!EQUAL,,[!PATHNAME %1%.MENU]]
     WRITE
     WRITE ERROR - '%1%.MENU' does not exist
     WRITE [$$BELL]
     PAUSE 3
[!ELSE]
     TYPE %1%.MENU
     [$MENU.CHOICE%0/%&
[!EQUAL,,%0/LINE=%]/LINE=22[!END][!EQUAL,,%0/COL=%]/COL=2[!END]&
 %1% [!EQUAL,,%2%]Select option:[!ELSE]%2-%[!END]]
[!END]
```

$MENU is unlikely to be invoked with a **menuname** that doesn't exist but just in case this is checked. The **PAUSE** is in case it is on a submenu, where the screen would be cleared on return. We agree this is a less than satisfactory way of ensuring a User sees a message, but we really only expect to get this message in a development environment when **$MENU** would be called directly from the CLI during testing, and in which case forcing the user to hit NEWLINE simply to return him to CLI seems like overkill. **$MENU** assumes that the screen is cleared by each option macro and menu. Therefore all **menuname.MENU** files should have a Form-Feed character as the first character in the file (ie a blank first page in SED). Option macros (**menuname.option.CLI**), that don't execute a program that clears the screen anyway, will be expected to do so with say, **WRITE [$$ERASE_PAGE]**, although this is not essential.

After displaying the menu by TYPing out the **menuname.MENU** file, **$MENU** calls **$MENU.CHOICE** to accept the user's selection in a loop until the EXIT option is chosen. Since most menus use numeric options it was decided to use **** as the fixed exit option, since it is on the end of the line of numeric keys across the top of the keyboard. However since some people have difficulty distinguishing **** from **/**, **$MENU** recognizes both as meaning EXIT. **$MENU** supplies **$MENU.CHOICE** with the defaults for any required information that was not supplied by the caller such as the prompt text, and the line and column on which to display it.

File $MENU.CHOICE.CLI

```
[$MENU.PROCESS%0/%&
     %1%,[$$ARG1 [!READ/S [$$CURSOR%0/%]%2-%[$$ERASE_EOL] ]],%2-%]
```

File $MENU.PROCESS.CLI

```
[!NEQUAL,\,%2%][!NEQUAL,/,%2%]
     [!EQUAL,,[!PATHNAME %1%.%2%.CLI][!PATHNAME GLOBAL_OP.%2%.CLI]]
          WRITE
          WRITE [$$CURSOR/LINE=24/COL=1]Invalid selection. Choose a&
          valid option or \ to Exit[$$BELL][$$CURSOR/LINE=23/COL=1]
          [$MENU.CHOICE%0/% %1%,%3-%]
     [!ELSE]
          [!EQUAL,,[!PATHNAME %1%.%2%.CLI]]
               [GLOBAL_OP.%2%%0\LINE\COL\AUTOEXIT%]
          [!ELSE]
               [%1%.%2%%0\LINE\COL\AUTOEXIT%]
          [!END]
          [!EQUAL,,%0/AUTOEXIT%]
               [$MENU%0/% %1%,%3-%]
          [!END]
     [!END]
[!END][!END]
```

$MENU.CHOICE displays the prompt, accepts the user's response, and calls **$MENU.PROCESS** to process the user's selection. If either of the recognized EXIT options is chosen there is nothing to do except return to the caller. If anything else was entered we have to validate that it is in fact a valid option for this menu. This is typically the messiest part of the code of most CLI menu processors, but for us it is trivial. Simply if the option macro, **menuname.option.CLI** (or the global option macro **GLOBAL_OP.option.CLI**) exists it's valid otherwise it is not.

If an invalid option was chosen, the user is told about it with an error message on Line 24, before reprompting by calling **$MENU.CHOICE.** Note the repositioning of the cursor after the error text to prevent the NEWLINE that the CLI WRITE command sends to the terminal, causing the screen to scroll. In this way there is no need to redisplay the actual menu (by calling **$MENU**) because it will remain on the screen until a valid option is chosen.

When a valid option is chosen **$MENU.PROCESS** simply invokes the associated option macro passing on any non-$MENU switches to it. A global option macro is invoked only if the corresponding option macro for the menu does not exist. On return from the action macro we reconstitute the caller's original call of **$MENU** to get his next choice on this menu (which may of course be to EXIT), unless the **/AUTOEXIT** switch was given in which case we do an automatic EXIT by doing nothing.

Simple isn't it. Now let's look at how we might define a menu and its options for use with **$MENU**. Those of you who purchase the associated Mag-Tape with all this manual's examples will also find this dummy example menu on the Tape. For want of a better name we have called our menu **FRED**. Note **^L** denotes **CTRL-L** (ie Form-Feed).

File FRED.MENU

```
^L              Fred Menu
                ---------

          1.    Sub menu

          2.    A choice

          \.    Exit
```

File FRED.1.CLI

```
$MENU/AUTOEXIT/LINE=6%0/% FREDSUB
```

File FRED.2.CLI

```
WRITE [$$ERASE_PAGE]
WRITE Hi ! I'm option 2 on the FRED menu
WRITE
@NULL [!READ/S Hit NEWLINE to continue [$$BELL]]
```

File FREDSUB.MENU

```
^L              Fred's SUB-menu
                --------------
          N.    do Nothing
          S.    do Something
```

File FREDSUB.N.CLI

```
WRITE [$$ERASE_PAGE]
WRITE I just did Nothin!
WRITE
@NULL [!READ/S Hit NEWLINE to continue [$$BELL]]
```

File FREDSUB.S.CLI

```
WRITE [$$ERASE_PAGE]
STRING [!READ/S OK - so U want me to do something well - What?
[$$BELL]]
WRITE
WRITE Alright here we go ....
PAUSE 2
WRITE
WRITE [!STRING]
WRITE
@NULL [!READ/S There! I did it. Satisified ? [$$BELL]]
WRITE
WRITE [$$BLINK STIFF!]
[!NEQUAL,,%0/%]
     WRITE
     WRITE Oh! and by the way I got the following switches: %0/%
[!END]
PAUSE 2
WRITE
WRITE Back to menu now
PAUSE 2
```

The **/AUTOEXIT** switch in **FRED.1.CLI** means that after choosing an option on **FREDSUB** menu (after having chosen option 1 on **FRED**'s menu), we will automatically exit back to the **FRED** main menu.

The beauty of **$MENU** is its simplicity. Coding of the menus and their options is straightforward, obvious and maintenance is easy. Options are not restricted to the numerics or single characters as in some systems, but are of course restricted to valid AOS/VS filename characters. The global option feature enables you to set up standard options that can be invoked from any menu. For example:

File GLOBAL_OP.BYE.CLI

```
WRITE [$$ERASE_PAGE]
BYE
```

This means we could choose the option **BYE** from either of the menus **FRED** or its submenu **FREDSUB** (despite the fact that neither of the menu screens tells us we can) and in both cases we would be logged off. If **GLOBAL_OP.BYE.CLI** exists in :UTIL you could override it for an entire application by placing your own GLOBAL_OP.BYE.CLI macro in the application's software directory, or for an individual menu, by creating a **menuname.BYE.CLI** option macro.

30.2 A Day-of-the-Week User Pseudo Macro

In this example we are going to describe a User Peudo Macro **$$DOW** that returns the Day-of-the-Week in words (ie Monday, Tuesday etc.). The code is based on the commonly used algorithm:

Day-No = (Days-since-01-JAN-1900 **MOD** 7) + 1

where

MOD is the remainder on division by or MODULUS function

Day-No 1=Sunday, 2=Monday, ... 7=Saturday

We suspect that although we aren't the first to write a CLI Macro or User Pseudo Macro to do this, we suspect we are the first to write it without the use of CLI **STRING**s or CLI Numeric **VAR**iables.

File $$DOW.CLI

```
[$$DAY [!UMOD [$$DAYS [!EQUAL,,%1%][!DATE][!ELSE]%1%[!END]],7]]&
```

File $$DAY.CLI

```
[!UEQ,0,%1%]Sunday[!END]&
[!UEQ,1,%1%]Monday[!END]&
[!UEQ,2,%1%]Tuesday[!END]&
[!UEQ,3,%1%]Wednesday[!END]&
[!UEQ,4,%1%]Thursday[!END]&
[!UEQ,5,%1%]Friday[!END]&
[!UEQ,6,%1%]Saturday[!END]&
```

These two User Pseudo Macros should be an obvious implementation of the above algorithm, provided the **$$DAYS** User Pseudo Macro returns the number of days since 1st January 1900. **$$DAYS** takes as its argument the date in CLI-format, which is DD-Mon-YY, and we assume that this is the format of the argument we received. If no argument was specified we assume the caller wants to know what today is, so we call **$$DAYS** with **[!DATE]** as the argument.

File $$DAYS.CLI

```
[$$DAYS.1 [!EXPLODE %1%]]&
```

File $$DAYS.1.CLI

```
[$$DAYS.1.1 %1%%2%,[$$MM %4%%5%%6%],%8%%9%]&
```

File $$DAYS.1.1.CLI

```
[!UADD [!UADD %1% [$$DAYS_TO_BOM %2% %3%]] [$$DAYS_TO_BOY %3%]]&
```

$$DAYS calls **$$DAYS.1** with the exploded date, which in turn calls **$$DAYS.1.1** with three arguments being **DD,MM,YY**. The month number is returned by our **$$MM** User Pseudo Macro described in Chapter 19. Now the number of days since 1st January 1900 is:

DD + Days-to-Bom-MM-in-year-YY + Days-to-Boy-for-year-YY

File $$DAYS TO BOM.CLI

```
[!ULT,%1%,3]&
[!UEQ,1,%1%]0[!END]&
[!UEQ,2,%1%]31[!END]&
[!ELSE]&
[!UADD &
[!UEQ,3,%1%]59[!END]&
[!UEQ,4,%1%]90[!END]&
[!UEQ,5,%1%]120[!END]&
[!UEQ,6,%1%]151[!END]&
[!UEQ,7,%1%]181[!END]&
[!UEQ,8,%1%]212[!END]&
[!UEQ,9,%1%]243[!END]&
[!UEQ,10,%1%]273[!END]&
[!UEQ,11,%1%]304[!END]&
[!UEQ,12,%1%]334[!END]&
,[!UEQ,0,%2%]0[!ELSE][!UEQ,0,[!UMOD %2% 4]]1[!ELSE]0[!END][!END]]&
[!END]&
```

File $$DAYS TO BOY.CLI

```
[!UEQ,0,%1%]0[!ELSE]&
[!UADD [!UMUL %1% 365] [!UDIV [!USUB %1% 1] 4]][!END]&
```

The days to beginning of month (**$$DAYS_TO_BOM**) for January and February is fixed, but for later months it needs an extra day added if this is a leap year, ie **YY MOD 4 = 0** (except YY=0 which was not a leap year). The days to beginning of year (**$$DAYS_TO_BOY**) since 1st January 1900 is: (except when YY=0, when of course there were no days before that date)

(YY x 365) + (YY-1)/4

where

(YY-1)/4 is the number of leap years prior to this year since 1st January 1900

This algorithm for **$$DAYS** (and therefore **$$DOW**) assumes **YY** is in the 20th Century and will have to be modified at the turn of the century (ie 2000) which unlike 1900 will be a leap year. We could use **$$DOW** perhaps in an initial IPC macro as follows:

WRITE Gidday! Today is [$$DOW] [!DATE]

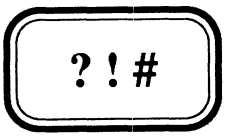
which when we tried it today gave us

Gidday! Today is Thursday 23-MAR-89

which of course it was. **$$DOW** returns its result surprisingly quickly.

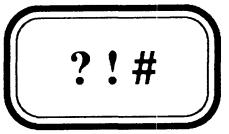
The **$$DAYS** User Pseudo Macro is of course a Julian Days User Pseudo Macro and could be used to calculate the difference between two dates. Eg. a dump macro might do an incremental dump by default, and a full dump whenever it's more than six days since the last time it did one:

```
MYDUMP[!ULT,[!USUB [$$DAYS [!DATE]] [$$DAYS [LAST_FULL_DATE]]],7]&
/AFTER/TLM=[LAST_DUMP_DATE][!END]%0/% %1-%
```



APPENDIX A

SPEED Macros



APPENDIX A

SPEED Macros

A.1 What is SPEED?

SPEED is a character oriented text editor. It is designed to minimize keystrokes for users who do a lot of text editing. Therefore nearly every character on the keyboard denotes a separate command. Without regular practice using it is difficult, but when you become familiar with it and some of its more esoteric features it can be an extremely powerful tool. **SPEED** is fully documented in the Data General manual "SPEED Text Editor (AOS and AOS/VS) User's Manual" (093-000197).

It is not a programming language, but like CLI, since it has features that enable input, output, data manipulation, arithmetic, task repetition and conditional logic, it can be used as a **programming language**. It is beyond the scope of this manual to teach you how to use SPEED as a text-editor, let alone as a programming language, (although if there is sufficient demand for it, we might make this the subject matter of another manual). Our aim is merely to introduce you to some of its programming like features, and show you a few examples of some of the uses to which we have put it.

A.2 SPEED's programming features

SPEED has a number of features that make it effectively a programming language. Some of these are described below. There is no particular significance to the order in which they appear.

Numeric Variables

SPEED has 10 numeric variables available for temporary storage of numeric data. These variables are denoted by the expression **Vi** where i=0-9. They are integers stored in a 16-bit word and as such depending on context, can be used to represent values in the range -32768 to +32767 or in the range 0 to 65535. Their values can be set with the expression **nVSi** which sets **Vi** to the value **n** and also represents that value. The expression **VIi** will increment **Vi** by one and also represents that incremented value, while **VDi** will decrement **Vi** by one and represents the decremented value. In addition SPEED has some other rather useful numeric variables:

- **.** The current value of the character pointer ie the number of characters in the editing buffer left of (or before) the character pointer.
- **Z** The number of characters in the editing buffer.
- **VN** The number of lines in the editing buffer.
- **VL** The number of the current line ie the number of NEWLINE characters left of the character pointer plus one.
- **VC** The ASCII value of the character immediately following the character pointer.
- **:** Although not representing a value in isolation, when it prefixes a **S**earch or **C**hange command the entire Search or Change command then represents a value being zero if the search was unsuccessful and one if it was successful.

Arithmetic

SPEED can do integer arithmetic on its numeric variables using the normal operators **+**, **-**, *****, and **/**. It also supports special operators for bitwise Boolean functions **AND**, **OR** (inclusive and exclusive), and **NOT**. There are no operator precedence rules, so all expressions are evaluated left-to-right. This means care must be taken in ordering expressions. Take for example the common integer arithmetic expression for the **MODULUS** function (remainder on division by):

m MOD n = m - m/n*n

Let's say **m** is **.**, ie the character pointer and **n** is **16** (chosen because we actually use this in a later example), then in **SPEED** we could evaluate **. MOD 16** leaving the result in **V0** thus:

./16*16VS0$$.-V0VS0

where underlined dollars represent ESCape characters. Whereas the expression:

.-./16*16VS0

would always result in setting **V0** to zero since .-. is zero.

Conditional Logic

SPEED commands can be executed conditionally with the expression:

n"Xspeed commands'

where **X** may be **E**, **N**, **G**, or **L** as follows:

when
X is **E** "speed commands" are executed if **n = 0**
X is **N** "speed commands" are executed if **n not = 0**
X is **G** "speed commands" are executed if **n greater than 0**
X is **L** "speed commands" are executed if **n less than 0**

There is no **else** expression but one can usually be formed eg:

V0"Ecommands if V0 = 0' **!else! V0"N**commands if V0 not = 0'

Labels and GO TO's

Incidentally **!string!** is SPEED's way of defining a Label which you can **GO TO** with the command **Ostring$**, but we writers of structured code have no need of GO TO statements and therefore we use Labels as **comments**, as we did in the preceding example.

String Variables

Strings are part and parcel of any text editor, which of course can be **I**nserted, **S**earched for and **C**hanged. In addition they can be stored in temporary buffers with SPEED's **BT** (buffer transfer) or **BC** (buffer copy) commands, where 36 buffers **0-9,A-Z** are available. Buffer A's contents can be expanded with **^BA** (ie CTRL-B A).

Iteration

SPEED commands can be repeated a specified number of times with the expression:

n<speed commands>

where "speed commands" are executed **n** times.

In addition they can be repeated indefinitely until some condition becomes true with:

<commands before test **n;** commands after test>

where the loop is exited when **n less than or** = **0.** Once the loop is entered "commands before test" are executed, while "commands after test" are only executed if the loop is not exited, as a result of the evaluation of the expression **n;**. The sense of the test can be reversed with:

<commands before test **n:;** commands after test>

Inside iterative loops all **S**earch and **C**hange commands are considered to be **:**-modified searches and therefore represent the value zero (failure) or one (success), and when **;** or **:;** appear in such loops unprefixed by **n**, then the loop is exited conditionally on the value of the last such search, thus:

<commands before test **;** commands after test>

exits the loop when the last **S**earch was unsuccessful, while

<commands before test **:;** commands after test>

exits the loop when the last **S**earch was successful.

For example:

<CFRED$FREDA$;> changes all "FRED"s to "FREDA"s

while

<Z-.;commands> exits when we are positioned at the end of the editing buffer

Special variations on the theme exist for SPEED's I/O commands where **Y;** attempts to **Y**ank in the next page and exits the Loop when there are no more pages to Yank in. Similar Loop exits exist for the **A**ppend command (**A;**) and the **R**oll command (**R;**), which Rolls out the current buffer contents to the output file and Yanks in the next one from the input file.

Hopefully it will be clear to those of you who are programmers, that with features like these, you could write some pretty mean Programs. We have not explained all the programming like features, and virtually none of SPEED's basic editing commands with which you would need to be familiar in order to actually write some programs in SPEED. What we have done is explain enough of the constructs to explain our examples.

A.3 A Few Words of Warning

Although SPEED can be used to edit virtually any file, whether it is a text file or not, we advise you to be extremely cautious when editing non-text files. The main reason is that SPEED gobbles up ASCII null characters, therefore **NEVER use SPEED to Edit a File with ASCII nulls in it**, unless you actually want them discarded.

The technique described in the previous section for exiting a Loop when you have reached the end of the buffer (**<Z-.; .... >**), will fail if there are more than 32767 characters in the buffer because SPEED's twos complement 16-bit arithmetic will treat numbers larger than 32767 as negative numbers and thus the expression **Z-.** will always be either negative or zero. Provided your buffer does not contain more than 32767 lines then **<VN-VL; .... >** should get around most such situations. Otherwise you'll have to switch to **W**indow **M**ode specifying a buffer of so-many lines rather than the default which treats a Form-Feed character as the buffer delimiter. Switching to Window Mode is a pain though, because you can't specify the Window Size from the command line and this therefore prevents you specifying the File to Edit from the command line.

A.4 What is a SPEED Macro?

A SPEED Macro is a file containing SPEED commands, usually designed to perform some editing function on a particular type of file. Normally you interact with SPEED to edit a file but if you have some standard often repeated editing procedure, by placing the appropriate commands in a SPEED macro, you can invoke the whole procedure from the command line without any interaction. A SPEED macro is invoked thus:

```
X SPEED/I=speedmacro{ pathname}
```

Where **speedmacro** is the pathname to the file containing the SPEED commands and **pathname** is the optional (since the file to edit may be specified in the SPEED macro itself - it has to when using Window Mode) pathname to the file to be edited. To edit a whole bunch of files with the same SPEED macro one could for example:

```
X SPEED/I=speedmacro ([!FILENAMES +.CLI])
```

All our Shalless Software SPEED macros are named in accordance with our conventions which are that they start with a dollar (**$**) and end with the extension **.SPD**. To simplify use of the SPEED macro for users we always create a CLI macro to invoke it, and we suggest you also adopt this convention. One of the main reasons for doing this is that more often than not we invoke our SPEED macros with the CLI PROCESS command rather than the EXECUTE (X or XEQ) command. This enables us to discard SPEED's "Lower case input encountered" and "Line too long" warnings which to Users of our Macro (especially those who don't know they are using it) merely causes confusion. Typically this is done as follows:

```
PROCESS/BLOCK/DEFAULT/OUTPUT=@NULL SPEED/I=speedmacro{ pathname}
```

However each invocation of a Process (whether by X or PROCESS) will cause at least one NEWLINE character to be sent to @OUTPUT and this should be taken into account when designing screen output.

A.5 $SPEEDTYPE (An Incestuous Example)

Our first example of a SPEED macro, only exists for the incestuous purpose of enabling us to type out SPEED macros. SPEED macros contain several (sometimes many) non-printing control characters, at least including several ESCape characters. As a result SPEED macros cannot be typed out (using the CLI TYPE command) or printed. Attempts to do so will result in all the control characters being absent (when printed) or gibberish on your screen (when typed). SPEED itself echoes these chacacters on the screen by showing ESCapes as underlined dollars (**$**), and other control characters as a two character sequence, prefixing the control character with an up-arrow (eg CTRL-E as **^E**). One method of enabling us to type or print SPEED macros would be to write a SPEED macro to do such substitutions, but why go to all the trouble when SPEED already does such substitutions. All we have to do is to create a SPEED macro that types out the contents of a file, and invoke it in such a way that we control where the output goes. Thus:

File $SPEEDTYPE.SPD

```
FR@DATA$Y#TFCH
```

which basically does the following; opens @DATA for input (FR), Yanks in its first (and assumed only) page, Types out the entire buffer contents, Closes the file (FC), and Halts. Then we simply create a **$SPEEDTYPE** CLI macro as follows:

File $SPEEDTYPE.CLI

```
[!EQUAL,,[!PATHNAME %1%]]
     WRITE
     WRITE ERROR - Cannot find File %1%
     WRITE [$$BELL]
[!ELSE]
     PROCESS/BLOCK/DEFAULT/IOC/DATA=%1% SPEED/I=$SPEEDTYPE.SPD
[!END]
```

Although this now enables us to type out SPEED macros on the screen we need a slight variation on the theme, where output is this time directed to a **.LS** file to give us a printable version of a SPEED macro. This CLI macro **$SPEEDLIST** uses the same SPEED macro (ie $SPEEDTYPE.SPD) to accomplish its task, but invokes SPEED in a slightly different way:

File $SPEEDLIST.CLI

```
[!EQUAL,,[!PATHNAME %1%]]
     WRITE
     WRITE ERROR - Cannot find File %1%[$$BELL]
[!ELSE]
     (DELETE/2=IGNORE CREATE/TYPE=TXT) [!PATHNAME %1%].LS
     PROCESS/BLOCK/DEFAULT/IOC/OUTPUT=[!PATHNAME %1%].LS/DATA=%1%&
          SPEED/I=$SPEEDTYPE.SPD
     WRITE Printable Version of Speed Macro %1% in [!PATHNAME %1%].LS
[!END]
WRITE
```

These are extremely useful tools for anyone contemplating writing SPEED macros, or who just wants to look at someone else's macros.

A.6 $FLIST (The Single Most Useful Macro in this Manual)

Many is the time that we (prior to writing $FLIST), wished the CLI's **!FILENAMES** Pseudo Macro supported the same switches as the FILESTATUS command. Sometimes it's for no other reason than to get the filenames matching the supplied templates processed in alphabetical order (/S), while on others we have wanted to select only those of a particular type (eg. only /TYPE=LNK or /TYPE=\LNK), or only those that have been modified since a certain date (/AFTER/TLM=DD-Mon-YY). With **$FLIST** we give you that functionality. **$FLIST** uses the CLI FILESTATUS command to give us the appropriate list of files, then a SPEED macro is used to edit that list to make it a User Pseudo Macro format list, which we invoke in place of the !FILENAMES Pseudo Macro we wished we had.

$FLIST

Produces a Filestatus List of filenames according to the supplied Filestatus switches and templates. Output is to the file specified by the required '/L=filename' switch. The output is in Pseudo macro format suitable for use in macro calls thus: MYMACRO ([filename])

Format: $FLIST/L=filename{/switches} templates

Required Switch:

/L=filename	Specifies output file which is always deleted if it already exists, before being used.

Optional Switches:

other	Any valid Filestatus switch except /Nheader & /Cpl

Arguments:

templates	Any valid Filestatus templates

File $FLIST.CLI

```
[!EQUAL,,%0/L=%]
     WRITE
     WRITE ERROR - Must specify output file with /L=file switch[$$BELL]
     WRITE
[!ELSE]
     DELETE/2=IGNORE %0/L=%
     FILESTATUS/NHEADER/CPL=1%0\NHEADER\CPL% %-%
     PROCESS/BLOCK/DEFAULT/OUTPUT=@NULL SPEED/I=$FLIST.SPD %0/L=%
[!END]
```

File $FLIST.SPD

```
C^E$$$<C
^E$ &
$;><C=$$;>ZJ-1MI&$FUH
```

CTRL-E (**^E**) is SPEED's special match control character for matching any string of spaces or tabs, thus the SPEED macro eliminates the leading spaces at the beginning of each line (/CPL=1 guarantees 1 line/file), putting **<space>&<NL>** between each file and **&<NL>** after the last one.

Having stated that we think **$FLIST** is the single most useful macro in this manual, before describing any more SPEED macros, we would like to show you some of the uses to which we have put **$FLIST**, the resultant macros in some cases becoming extremely useful macros in their own right. One simple application in any programming development environment enables us to print off compilation listings of our programs in alphabetical order to simplify filing of them. Thus:

```
$FLIST/S/L=LISTINGS +.LS
QPRINT [LISTINGS]
DELETE LISTINGS
```

In our next example we will show a use of the /TYPE=type switch. Many people have written DISKSPACE macros to tell them about availability of disk space on their system. Most of these are extremely environment specific, and have to be altered each time a disk is added to or removed from the system. Our **$DISKSPACE** macro, makes no assumptions about the names of your disks or how many there are. Its only assumption (which in 90% of cases is correct), is that all the disks on your system hang off the System Disk (:). It uses **$FLIST/TYPE=LDU** to find out dynamically what disks are attached at the time.

$DISKSPACE

Displays % full, maximum and available Diskspace for all Disks initialized from the System's Root Directory (:). For disks initialized elsewhere, or in fact any Control Point Directory (CPD), you merely supply the Pathname to it as an argument. You must have R access to the Disk or CPD for it to be reported.

Format: $DISKSPACE{/switches} {pathname/s}

Switches:

/L{=filename}	Redirects output to current Listfile or a specified file

Arguments:

pathname/s	Pathname/s to a specific Disk or CPD, when omitted all Disks initialized into : are reported

File $DISKSPACE.CLI

```
PUSH; PROMPT POP
SUPERUSER/1=IGNORE/2=IGNORE ON
[!EQUAL,,%1%]
     $FLIST/TYPE=LDU/L=:UDD:[!USERNAME]:?[!PID]DISKS.CLI :+
     $DISKSPACE.1%0/% (: [:UDD:[!USERNAME]:?[!PID]DISKS])
     DELETE/2=IGNORE :UDD:[!USERNAME]:?[!PID]DISKS.CLI
[!ELSE]
     WRITE
     $DISKSPACE.1%0/% (%-%)
[!END]
WRITE
POP
```

File $DISKSPACE.1.CLI

```
DELETE/2=IGNORE :UDD:[!USERNAME]:?[!PID]DSPACE.CLI
SPACE/L=:UDD:[!USERNAME]:?[!PID]DSPACE.CLI %1%
$DISKSPACE.1.1%0/% %1%,[:UDD:[!USERNAME]:?[!PID]DSPACE]
DELETE :UDD:[!USERNAME]:?[!PID]DSPACE.CLI
```

File $DISKSPACE.1.1.CLI

```
[!EQUAL,%2%,Warning:]
     WRITE %1%[$$TAB][$$DIM %2-%]
[!ELSE]
     $DISKSPACE.1.1.1%0/% %1% &
[!UGT,%3%,9999][!UDIV %5% [!UDIV %3% 100]] &
[!ELSE][!UDIV [!UMUL %5% 100] %3%] [!END]&
[!ULT,%3%,2048]%7% %3% blocks&
[!ELSE][!UDIV %7% 2048] [!UDIV %3% 2048] mb[!END]
[!END]
```

File $DISKSPACE.1.1.1.CLI

```
WRITE%0/% %1%[$$TAB][$$DIM is],&
[!ULT,%2%,10],%2%[!ELSE]&
[!UGT,%2%,79][$$BLINK [$$UNDERLINE %2%]][!ELSE]&
[!UGT,%2%,69][$$UNDERLINE %2%][!ELSE]%2%[!END][!END][!END]&
[$$DIM %% full -,[!ULT,%3%,10],[!END][!ULT,%3%,100],[!END]]&
%3% [$$DIM %5% available out of] %4% [$$DIM %5%]
```

$DISKSPACE displays % full, amount available for use, and total. Amounts are displayed in **mb** (Megabytes) unless the total space is less than 1mb (2048 blocks) in which case, they are displayed in **blocks** (512 bytes). Percentages over 79 are underlined and blinked, otherwise if they are over 69 they are underlined. Notice how the output of the SPACE command is directed to a file, whose contents is expanded as a User Pseudo Macro, so that the appropriate arithmetic can be done, to enable a more meaningful display than that of the SPACE commnad itself. Users who do not have the SUPERUSER privilege will need R access to the System Root Directory (:) for the Default (no arguments) version of $DISKSPACE to work.

A generalized version of this macro (ie without knowing what disks are on the system) would not have been possible without **$FLIST**.

Although **$FLIST** will normally be used to return only a list of files matching supplied templates and subject to the restrictions imposed by the various FILESTATUS switches used, it can also be used to return other information that FILESTATUS returns as well as a file's name. In next example we plan to use **$FLIST** to return along with the file's name, its type (/TYPE), its elementsize (/ELEMENTSIZE), its size (/LENGTH) and whether or not it has a User Data Area (/UDA). We will then use this information to calculate the file's **OPT**imum **EL**ement**SIZE**, so that we can make a copy of the file of the same type but with the new optimum elementsize (which we eventually use to replace the original). The macro is called **$OPT_ELSIZE**.

There is a two-fold benefit to be gained from running **$OPT_ELSIZE** across various files on a system. For large files **$OPT_ELSIZE** will attempt to make them contiguous single element files, or if you specified a maximum (using the **$DEF_ELSIZE** macro), the maximum elementsize. The result will be to improve access performance by reducing the number of AOS/VS index levels the file has. We decided it would be a dangerous practice to change the elementsize of files whose elementsize is already greater than the maximum, so such files will be considered to already be at their optimum. For small files (such as CLI macros which are typically created with a text editor most of which create the file with an elementsize of 4 blocks), the result is often (since most are less than 512 characters) to reduce them to an elementsize of 1 block (provided your System Default elementsize is 1 block). This can reduce the amount of disk space occupied by small files to 25% of that which they originally occupied.

AOS/VS only allows files to have an elementsize that is a multiple of the System Default (which is usually 1 or 4), and so after calculating the optimum size we then need to round this up to the nearest multiple of the System Default elementsize. To assist us in doing this (and because we thought we might be able to use it elsewhere we first wrote a general purpose User Pseudo Macro to round a number (the first argument) up to the nearest multiple of another number (the second argument), which we called naturally enough **$$ROUND_UP:**

File $$ROUND_UP.CLI

```
[!EQUAL,,%2%]%1%[!ELSE][!UEQ,%2%,1]%1%[!ELSE]&
[!UADD [!UMUL [!UDIV [!USUB %1% 1] %2%] %2%] %2%]&
[!END][!END]&
```

Before **$OPT_ELSIZE** can calculate a file's optimum elementsize it must know the System Default elementsize. Since CLI does not provide us with a mechanism to determine it we will have to find it out by other means. **$OPT_ELSIZE** will assume that a User Pseudo Macro **DEF_ELSIZE.CLI** exists in **:$SYSPARAMS** specifying the System Default, while the macro **$DEF_ELSIZE** will create this User Pseudo Macro by attempting to CREATE a file with an elementsize of 1 block. It then uses **$FLIST/ELEMENTSIZE** to find out the actual elementsize of this file, which will necessarily be the System Default. To prevent "insufficient contiguous disk blocks" errors **$DEF_ELSIZE** also recommends you set a maximum elementsize as well (suggesting a default of 64).

$DEF_ELSIZE

This macro calculates the System Default Elementsize and saves it in :$SYSPARAMS:DEF_ELSIZE.CLI for use by $OPT_ELSIZE. This is necessary because AOS/VS doesn't allow files to have elementsizes that aren't multiples of the System Default. In addition it lets you set the maximum elementsize to which $OPT_ELSIZE can change a file. If you don't set it, $OPT_ELSIZE will attempt to make every file contiguous, which may result in errors because of "insufficient contiguous disk blocks". The maximum if set is saved in :$SYSPARAMS:MAX_ELSIZE.CLI.

Format: $DEF_ELSIZE

File $DEF_ELSIZE.CLI

```
(DELETE/2=IGNORE CREATE/ELEMENTSIZE=1) :$SYSPARAMS:DEF_ELSIZE.CLI
$FLIST/ELEMENTSIZE/L=:UDD:[!USERNAME]:?[!PID]ELT.CLI&
     :$SYSPARAMS:DEF_ELSIZE.CLI
$DEF_ELSIZE.1 [:UDD:[!USERNAME]:?[!PID]ELT]
DELETE :UDD:[!USERNAME]:?[!PID]ELT.CLI
ACL :$SYSPARAMS:***_ELSIZE.CLI OP,OWARE [!USERNAME],OWARE +,RE
```

File $DEF_ELSIZE.1.CLI

```
WRITE Setting :$SYSPARAMS:DEF_ELSIZE.CLI to System Default Elementsize&
          of %2% Blocks
WRITE/L=:$SYSPARAMS:DEF_ELSIZE.CLI %2%<&>
WRITE
WRITE You may also specify a Maximum Elementsize for use by $OPT_ELSIZE
WRITE If you do not $OPT_ELSIZE will attempt to make all files&
          contiguous.
WRITE
PUSH; PROMPT POP
$YN/DEF=N Do you wish to
DELETE/2=IGNORE :$SYSPARAMS:MAX_ELSIZE.CLI
WRITE
[!EQUAL,N,[!STRING]]
     WRITE OKAY - No upper limit imposed on $OPT_ELSIZE
[!ELSE]
     $NUM/DEF=64/MIN=%2% Enter a Maximum Elementsize as&
          a multiple of %2%
     [!UNE,0,[!UMOD [!VAR0] %2%]]
          VAR0 [$$ROUND_UP [!VAR0] %2%]
     [!END]
     WRITE
     WRITE Setting :$SYSPARAMS:MAX_ELSIZE.CLI to Maximum Elementsize&
          of [!VAR0] Blocks
     WRITE/L=:$SYSPARAMS:MAX_ELSIZE.CLI [!VAR0]<&>
[!END]
WRITE
POP
```

Although **$DEF_ELSIZE** suggests the user select a maximum elementsize that is a multiple of the system default, if it wasn't we use our **$$ROUND_UP** User Pseudo Macro, to make sure it is. Having defined $DEF_ELSIZE to establish the System Default elementsize, let's now look at the originally proposed macro **$OPT_ELSIZE. $OPT_ELSIZE** calls $OPT_ELSIZE.1 once for each file matching the supplied templates, after first establishing and passing on as switches the System Default elementsize, and the Maximum elementsize that **$OPT_ELSIZE** is allowed to use. **$OPT_ELSIZE.1** then uses **$FLIST** to get the file's type, elementsize, length and whether or not it has a UDA, and then by expanding the listfile as a User Pseudo Macro passes this information on to **$OPT_ELSIZE.1.1** to do the work. Since only one file is involved it may appear that the output of the FILESTATUS command could be used directly without the use of **$FLIST**, however FILESTATUS has a horrible habit of throwing out an extra NEWLINE when it deems the filename is too long for it to maintain it's columnar display, which would cause premature command termination and a mis-matched brackets error.

$OPT_ELSIZE

This macro will attempt to change the Elementsize of files matching the supplied templates, so that they will be at their OPTimum ELementSIZE. Files of type CPD, DIR or LDU are untouched as are those that have a UDA. The Optimum size is calculated as bytes/512+1 rounded up to nearest multiple of the System Default Elementsize, which must be calculated by $DEF_ELSIZE before $OPT_ELSIZE can be run. $DEF_ELSIZE also lets you define a maximum elementsize, which, if defined, can be overidden with a /MAX=size switch. Files with elementsizes = the calculated optimum, or > than the maximum will not have their elementsizes changed. As a result most files, will either be made contiguous single element files or be set to the maximum size.

Format: $OPT_ELSIZE{/MAX=size} template/s

Optional Switch:

/MAX=size Overrides maximum elementsize set by $DEF_ELSIZE

Arguments:

template/s Templates of files to be processed

File $OPT ELSIZE.CLI

```
[!EQUAL,,[!PATHNAME :$SYSPARAMS:DEF_ELSIZE.CLI]]
     WRITE
     WRITE ERROR - Can't calculate Optimum Elementsize without knowing&
                   System Default
     WRITE,,,,,,,,,which I expect to find in&
                   :$SYSPARAMS:DEF_ELSIZE.CLI which doesn't
     WRITE,,,,,,,,,exist. You may create it using $DEF_ELSIZE.
     WRITE [$$BELL]
[!ELSE]
     PUSH; PROMPT POP
     $OPT_ELSIZE.1%0\DEF%/DEF=[:$SYSPARAMS:DEF_ELSIZE]&
[!EQUAL,,%0/MAX=%][!NEQUAL,,[!PATHNAME :$SYSPARAMS:MAX_ELSIZE.CLI]]&
/MAX=[:$SYSPARAMS:MAX_ELSIZE][!END][!END] ([!FILENAMES %-%])
     WRITE
     POP
[!END]
```

File $OPT ELSIZE.1.CLI

```
[!EQUAL,,%1%]
     WRITE
     WRITE ERROR - No files matching supplied template/s[$$BELL]
[!ELSE]
     $FLIST/TYPE/ELEMENTSIZE/LENGTH/UDA&
/L=:UDD:[!USERNAME]:?[!PID]ELT.CLI %1%
     $OPT_ELSIZE.1.1%0/% [:UDD:[!USERNAME]:?[!PID]ELT]
     DELETE :UDD:[!USERNAME]:?[!PID]ELT.CLI
[!END]
```

File $OPT ELSIZE.1.1.CLI

```
[!EQUAL,*,&
[!EQUAL,%2%,CPD]*[!END]&
[!EQUAL,%2%,DIR]*[!END]&
[!EQUAL,%2%,LDU]*[!END]]
     WRITE WARNING - Can't change elementsize of file of Type %2%&
                   - File %1%
[!ELSE][!EQUAL,UDA,%5%]
     WRITE WARNING - Won't change elementsize of file with a UDA&
                   - File %1%
[!ELSE]
     VAR0 [$$ROUND_UP [!UADD [!UDIV %4% 512] 1] %0/DEF=%]
     [!NEQUAL,,%0/MAX=%]
          [!UGT,[!VAR0],%0/MAX=%]
               VAR0 %0/MAX=%
          [!END]
     [!END]
     [!NEQUAL,,[!UGE,%3%,%0/MAX=%]*[!END][!UEQ,%3%,[!VAR0]]*[!END]]
          WRITE File already at optimum elementsize %3% - File %1%
     [!ELSE]
          CREATE/TYPE=%2%/ELEMENTSIZE=[!VAR0]/2=WARNING %1%.?
          [!EQUAL,,[!FILENAME %1%.?]]
               WRITE WARNING - Unable to create file with new&
                         elementsize [!VAR0] - File %1%
          [!ELSE]
               COPY/A/2=WARNING %1%.? %1%
               [!UNE,[!SIZE %1%.?],[!SIZE %1%]]
                    WRITE WARNING - Unable to copy data to file with&
                              new elementsize [!VAR0] - File %1%
                    DELETE %1%.?
               [!ELSE]
                    ACL %1%.? [!ACL %1%]
                    DELETE %1%
                    RENAME %1%.? [!EFILENAME %1%]
                    WRITE New elementsize of %1% is [!VAR0]
               [!END]
          [!END]
     [!END]
[!END]
[!END]
```

Note that the original file is only deleted and the new one with the new elementsize renamed, if the new one is exactly the same size as the original. This guards against possible "insufficient contiguous disk blocks" or "CPD max size exceeded" or ACL errors.

A.7 $HELP (A generalized HELP facility)

Okay enough of the examples of the uses of **$FLIST**, however we are certain you can put **$FLIST**, **$DISKSPACE** and **$OPT_ELSIZE** to good use. Nonetheless this Appendix is supposed to be about SPEED macros so lets look at some more of those. This example again involves the editing of the output produced by CLI's FILESTATUS command. This time we wanted a generalized **$HELP** facility to give HELP on all our Macros and User Pseudo Macros, described in this Manual and distributed on the separately purchasable associated Magnetic Tape. It is because of our desire to provide such a facility and our belief that such HELP should exist in one place only, that we prefer to **TYPE** out the contents of a HELP file, rather than use **CREATE/M @OUTPUT**, when providing support for **/H**{ELP} switches in our macros (see Chapter 12 where the relative merits of these two techniques were discussed). You will find that a lot of the macros described in this manual differ from the version distributed with the associated tape, in that on the tape they contain support for this switch. For your interest we added the code to support this switch to these macros, using the following SPEED macro, which incidently is not distributed with the Tape. First we created a file:

File HELP.TXT

```
[!NEQUAL,,%0/H/HE/HEL/HELP%]
     TYPE [!PATHNAME $HELP]:[!EQUAL,.CLI,[!EEXTENSION %0\%]]&
[!ENAME %0\%][!ELSE][!EFILENAME %0\%][!END].HELP

[!ELSE]
```

And then a SPEED macro to insert the contents of this file at the beginning of the macro and to add the necessary **[!END]** at the end. In addition the SPEED macro looks to see if this has already been done, so as not to attempt to do it twice on the same macro. Our SPEED macro looked something like this:

File ADDHELP.SPD

```
:S%0/H/HE/HEL/HELP%$"EI^FHELP.TXT$$ZJI
[!END]
$'FUH
```

which means if **%0/H/HE/HEL/HELP%** can't be found, Insert the contents of HELP.TXT, Jump to the end of the buffer, and insert the **[!END]**. If it was found nothing is done. Then to add **/H**{ELP} support to a macro we simply typed:

```
X SPEED/I=ADDHELP.SPD macroname.CLI
```

Clearly from the above code the **/H**{ELP} switch on a macro expects a directory called **$HELP** to exist somewhere on the user's searchlist, and for a file to exist in it called **macroname.HELP**. Notice the messing around we have to go to guarantee isolation of the **macroname** from Argument 0. It is not simply a matter of **%0\%**, because the macro can be invoked in several ways, with or without the .CLI extension, and with or without directory prefixes specifying its full pathname, all of which have to be removed to isolate **macroname**.

Now we want a generalized HELP facility, through a macro, not surprisingly called **$HELP**, which when called without arguments gives us an alphabetic list of topics available, and when called with an argument, either gives us HELP on that topic, tells us none is available on that topic, or gives us a list of the topics it knows about which start with the characters supplied. In addition we want to allow the user to omit the Shalless Software **$** prefix on macros and the **$$** prefix on User Pseudo Macros and still give the appropriate HELP.

$HELP

Provides HELP on use of Shalless Software Macros and User Pseudo Macros, where help files macroname.HELP are assumed to exist in a directory $HELP which must exist somewhere on the user's searchlist. A complete list of topics is given, if no argument is supplied. All macroname.HELP files are designed to fit on one screen.

Format: $HELP{ topics}

Arguments:

topics — Any topic(s) on which you want HELP. Topics may be abbreviated, and leading $'s or $$'s omitted. When more than one HELP file matches the supplied topic an alphabetic list of matching topics is given.

$HELP uses a macro called **$HELP.LIST** which assumes $HELP is the Working Directory and expects the following arguments:

argument-1 — A template or list of templates (surrounded by round brackets so that it is treated as a single argument, and where the call must therefore be enclosed in square brackets) of files to look for in the $HELP directory.

argument-2 — A term for use by $HELP.LIST (eg topic) when telling the user to type "$HELP term" to get help.

argument-3.. — A description of what the topics being displayed are (eg Topics starting with <$HELP's argument-1>)

$HELP.LIST then gets a sorted list of files matching the template/s into a temporary listfile using the command:

```
FILESTATUS/NHEADER/S/L=listfile (%1%)
```

Note the use of the brackets around brackets trick to force their removal. If argument-1 didn't have brackets there is still only one invocation of the FILESTATUS command. **$HELP.LIST** then uses the SPEED macro **$HELP.SPD** to format the output of the FILESTATUS command for display. This involves removal of the = prefixes on the filenames as well as their **.HELP** extensions, and finally the padding of each name with spaces up to 16 characters to format for 5 column display on an 80 column screen. These macros and the associated SPEED macro are shown on the following pages without further explanation.

File $HELP.CLI

```
[!EQUAL,,[!PATHNAME $HELP]]
     WRITE
     WRITE ERROR - Cannot find $HELP Directory
     WRITE,,,,,,,,,[$$DIM $macroname/H may give you some joy!]
     WRITE [$$BELL]
[!ELSE]
     PUSH; PROMPT POP
     DIRECTORY [!PATHNAME $HELP]
     [!EQUAL,,%1%]
          $HELP.LIST/NOBELL $+.HELP\$$+ $macro Shalless Software Macros
          $HELP.LIST $$+.HELP $$upm Shalless Software UPMs
     [!ELSE][!NEQUAL,,%2%]
          [$HELP.LIST (<<,$,$$><%1-%>>+.HELP) topic Topics with those&
               prefixes]
     [!ELSE][!EQUAL,$,%1%]
          $HELP.LIST $+.HELP\$$+ $macro Shalless Software Macros
     [!ELSE][!EQUAL,$$,%1%]
          $HELP.LIST $$+.HELP $$upm Shalless Software UPMs
     [!ELSE]
          VAR0 [$$ARGS [!FILENAMES <,$,$$>%1%+.HELP]]
          [!UEQ,1,[!VAR0]]
               STRING [!FILENAMES <,$,$$>%1%+.HELP]
               TYPE [!STRING]
          [!ELSE][!UEQ,0,[!VAR0]]
               WRITE
               WRITE No $HELP available on %1% - Type $HELP for a&
                    list of topics
               WRITE [$$BELL]
          [!ELSE]
               [!UEQ,1,[$$ARGS [!FILENAMES <,$,$$>%1%.HELP]]]
                    STRING [!FILENAMES <,$,$$>%1%.HELP]
                    TYPE [!STRING]
               [!ELSE][!UEQ,1,[$$ARGS [!FILENAMES %1%.HELP]]]
                    TYPE %1%.HELP
               [!ELSE]
                    [$HELP.LIST (<,$,$$>%1%+.HELP) topic Topics&
                              starting with %1%]
               [!END][!END]
          [!END][!END]
     [!END][!END][!END][!END]
     POP
[!END]
```

File $HELP.LIST.CLI

```
DELETE/2=IGNORE :UDD:[!USERNAME]:?[!PID]$H.TMP
FILESTATUS/NHEADER/S/L=:UDD:[!USERNAME]:?[!PID]$H.TMP (%1%)
PROCESS/BLOCK/DEFAULT/OUTPUT=@NULL SPEED/I=$HELP.SPD&
          :UDD:[!USERNAME]:?[!PID]$H.TMP
WRITE [$$UNDERLINE [$$DIM&
          HELP is available on these %3-% by typing $HELP %2%]]
TYPE :UDD:[!USERNAME]:?[!PID]$H.TMP
DELETE :UDD:[!USERNAME]:?[!PID]$H.TMP
[!EQUAL,,%0/NOBELL%]
     WRITE [$$BELL]
[!END]
```

File $HELP.SPD

```
<C^E=$ $;>1D<C.HELP^E$ $;><C.HELP
$$;>
<S^E$;./16*16VS0$$.-V0VS0"N16-V0<I $>'>
J<80MZ-.;10I>FUH
```

After getting rid of all the unwanted junk we are left with a file containing a list of topic names separated from each other by a single space. To pad each topic name to 16 characters with spaces we use the previously discussed technique for evaluating **. MOD 16** which is saved in **V0**. We then insert **16-V0** spaces, repeating the process until we can't find any more spaces. Finally, in case the user has a wide screen with CPL > 80, insert a NEWLINE every 80 characters.

A.8 $SMAC (The "piece de resistance" of SPEED macros)

Now for our "piece de resistance" of SPEED macros. You need to be a bit of a SPEED whiz to understand how this one works, but you don't even have to know what SPEED is in order to use it. This SPEED macro was originally developed by the author when he was a training instructor at Data General Australia. In the AOS System Manager course he used to present the UP macro on the Training Department's System as an example. By the time each new course had come around, he discovered someone had been playing with the macro, and had inserted some new code or features, but had done so without maintaining the previously neat layout of the macro's imbedded **[!conditonal,a,b]-[!ELSE]-[!END]** structure. Eventually after getting sick of fixing it up manually and invariably getting it wrong, he decided to write a SPEED macro to fix it up automatically, and so we give you **$SMAC**. It is particularly useful for macro writers who reckon that a Tab indents each level too far but are too lazy to type the necessary spaces. As well as laying your macro out neatly it effectively gives you a primitive syntax check by clearly showing any **[!conditional,a,b]** / **[!END]** imbalance.

$SMAC

This macro edits the specified CLI macro, to properly indent the imbedded [!conditional,a,b]-[!ELSE]-[!END] structure. Indentation is by Tabs or the specified no of spaces. $SMAC does not add lines, but conditional Pseudo Macros within a line, will be reflected in the indentation of subsequent ones. Lines starting with) will not be indented, (for /M), but lines between the command/M and), will be (which may be undesirable). Minimum abbreviation of conditional Pseudo Macros is three characters (eg !END or !UEQ), except !EQUAL and !NEQUAL, which are recognized as !EQ and !NE.

Format: $SMAC{/I=n} macroname

Optional Switch:

/I=n	Specifies to indent each level n spaces (Default: Tab)

Argument:

macroname	Macro to edit (DO NOT USE $SMAC ON USER PSEUDO MACROS)

Note the reason why **$SMAC** should not be used on User Pseudo Macros is that typically they are never indented, and if they were, unwanted spaces or tabs would appear in their resultant expansions. Note since we have now explained where our HELP files are, **$SMAC.CLI** is shown as it appears on the associated Magnetic Tape, with support for a **/H**{ELP} switch.

File $SMAC.CLI

```
[!NEQUAL,,%0/H/HE/HEL/HELP%]
     TYPE [!PATHNAME $HELP]:[!EQUAL,.CLI,[!EEXTENSION %0\%]]&
[!ENAME %0\%][!ELSE][!EFILENAME %0\%][!END].HELP

[!ELSE]

[!EQUAL,,[!PATHNAME %1%]]
     [!EQUAL,.CLI,[!EEXTENSION %1%]]
          WRITE
          WRITE ERROR - Cannot find File %1%
          WRITE [$$BELL]
     [!ELSE]
          %0% %1%.CLI
     [!END]
[!ELSE]
     DELETE/2=IGNORE :UDD:[!USERNAME]:?[!PID]SMAC.TMP
     WRITE/L=:UDD:[!USERNAME]:?[!PID]SMAC.TMP&
          [!EQUAL,,%0/I=%]0[!ELSE]%0/I=%[!END]
     WRITE
     WRITE Starting $SMAC edit of %1%
     PROCESS/BLOCK/DEFAULT/IOC/DATA=:UDD:[!USERNAME]:?[!PID]SMAC.TMP&
          SPEED/I=$SMAC.SPD %1%
     DELETE :UDD:[!USERNAME]:?[!PID]SMAC.TMP
[!END]

[!END]
```

File $SMAC.SPD

```
BST$^F@DATA$VS0<I $> V0"EI     $'BS0$0VS1$
<<Z-.;
  C^T$$VC-10VS0"E1M'
  V0"N
   VC-41"N 1WP$:.,.+5S[!ELS$VS9"E:.,.+5S[!END$VS9'V9"N-5M'0WP$
   V1-V9<I^BT$>'
  <1S[!^\ENU^\$;
   1<:.-1,.+1SNE$"NVI1$':; :.-2,.SEQ$"NVI1$':;1M
     :.-3,.SUEQ$"NVI1$':;  :.-3,.SUNE$"NVI1$':;
     :.-3,.SULT$"NVI1$':;  :.-3,.SULE$"NVI1$':;
     :.-3,.SUGT$"NVI1$':;  :.-3,.SUGE$"NVI1$':;
     :.-3,.SEND$;VD1$>
   >
  '

>R;>BKT$FU$@T"
Structural Indentation Complete
"H$
```

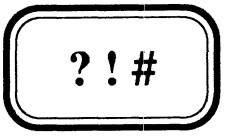
The number of spaces to indent each level (or zero if indentation is to be by Tabs) is passed to SPEED through the generic DATAFILE @DATA, since SPEED does not support a facility for getting hold of the arguments, other than its assumption that argument-1 is the name of the file on which it is to do an **FO** command followed by a **Y**ank. A Tab or the appropriate number of spaces is then stored in internal buffer T (actually buffers are just SPEED temporary files) for later use.

Each line within each page is processed one at a time. Firstly any existing indentation is stripped. Then any NEWLINE only (ASCII 10 decimal) lines skipped as well as lines beginning with "**)**" (ASCII 41 decimal). Then the appropriate indentation is inserted at the beginning of the line. This is done by inserting the contents of Buffer T **V1-V9** times. Where **V1** is a variable holding the current indentation level (initially zero) and **V9** is 1 if this line begins with **[!ELSE]** or **[!END]** otherwise 0. Then the rest of the line is scanned for **[!conditional]**'s, where **V1** is incremented each time one is found, and **[!END]**'s where **V1** is decremented for each one found, so that **V1** is set up with the correct value for indenting the next line.

If after running **$SMAC** on a macro of yours the last line is not hard up on the left-hand-side of the page, then your macro is missing at least one **[!END]**. If on the other hand your macro has two or more **[!END]**'s aligned at the same level, then you are missing at least one [!conditional]. Extraneous **[!ELSE]**'s will also be obvious while missing ones will not (bad luck). We hope you like $SMAC.

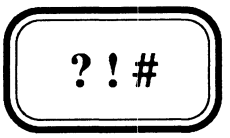
A final word of warning about it though. We couldn't be bothered making it smart enough to detect **command/M**'s and leaving lines between them and the closing "**)**" unchanged from the original file. In all of our uses of **command/M** the indentation of subsequent lines (provided that the "**)**" wasn't indented which it is not) did not affect performance of the macro, since such files are usually command files for some utility which treats leading spaces or Tabs as white space and ignores them. This may not always be the case, and you may have to go back and re-edit your macro to return these lines to their original form. On the other hand you may want to make **$SMAC** smarter. If you do send us a copy for the next revision of this Manual (we'll even mention your name).

SPEED may well be a text editor but as you can see, it is also a **programming language.**



APPENDIX B

SORT/MERGE Command Files



APPENDIX B

SORT/MERGE Command Files

B.1 What is SORT/MERGE?

SORT/MERGE is the AOS/VS utility for SORTing and MERGing files. It does not come automatically with AOS/VS (as does SPEED) but must be purchased separately. We cannot imagine how any System, whether it be a production, development or whatever site, could manage without it. Apart from the standard features you would expect from such a package, it has powerful editing and reformatting features, as well special features for supporting INFOS files. Although it will not be discussed in this Manual SORT/MERGE also comes with a built-in Report Writer. **SORT/MERGE** is fully documented in "Sort/Merge with Report Writer User's Manual (AOS and AOS/VS)" (093-000155).

Although SORT/MERGE (excluding its built-in Report Writer which is in itself a specialized programming language for writing reports), does not support temporary variables, arithmetic or user (ie screen) I/O, it can still be used as a **programming language**. It is beyond the scope of this manual to teach you how to use SORT/MERGE. Our aim is merely to introduce you to some of its programming like features, and show you a few examples of some of the uses to which we have put it.

B.2 SORT/MERGE's programming features

When you use SORT/MERGE you may choose to SORT, MERGE or merely COPY records from one or more input files to an output file. The first level at which you can program SORT/MERGE is in the way you define the Record Format of the input file(s). Except for INFOS files, whose Record Formats cannot be specified (they are always Variable Length), any AOS/VS file can be specified to have any record format you like, possibly overriding its AOS/VS Record Format. For example a file with Fixed Length Record Format may be viewed as Datasensitive and vice versa. When defining a file to have Datasensitive Records you can specify any character or group of characters as being delimiters. For example if you wanted to process a text file a page at a time rather than a line at a time you could specify the Form-Feed character (ie "<14>") as the sole delimiter.

The real programmability of SORT/MERGE, however, lies in its special massage statements that enable you to manipulate the data within records, both on the input phase, prior to them being processed (ie SORTed, MERGEd or COPY'd), and on the output phase, after they have been processed. An imperative statement (either **SORT**, **MERGE** or **COPY** or some variant of them) determines how records are processed. Massage statements placed before the imperative statement apply on the input phase, while those placed after it apply on the output phase. What these massage statements enable you to do, will be briefly described:

COMPRESS

Enables you to remove unwanted characters from a record.

INSERT

Enables you to insert a particular character or character string before, after or in place of an existing string in a record. A special feature enables you to insert the record's RECORDCOUNT which can be particularly useful in generating unique keys.

PAD

Enables you to convert variable-length records to fixed length by padding with a selected character to a specified number of characters.

REFORMAT

Enables you to reformat the fields of a record to relocate, delete or repeat them.

REPLACE

Enables you to make global replacements throughout a record of character strings with other character strings (including an empty string - ie you can use REPLACE to delete strings).

TRANSLATE

Enables you to convert characters from one character-set to another. SORT/MERGE enables you to define your own conversion tables, or you may use the predefined ones for converting from ASCII to EBCDIC or vice-versa or from LOWER to UPPER case. When SORTing, the location of this massaging statement (before or after the imperative statement) will determine whether records are sorted in the collating sequence of the character set you converted from (before), or the one you converted to (after).

IF

Enables you to conditionally SKIP records, STOP processing or use any of the other massage statements. Basic conditions include tests for equality, inequality, less than (or equal), greater than (or equal) and either does or does not appear in. Complex conditions may be built with AND and OR operators. Fields within a record may be compared with particular character strings or other fields within the record, while the RECORDCOUNT internal variable may be compared with a specific value.

These massaging statements can be used to build some pretty powerful yet concise programs, that might otherwise require significant programming effort.

B.3 A Few Words of Warning

Excluding the features of the built-in Report Writer which we do not intend to discuss, SORT/MERGE does not have any internal variables apart from the special RECORDCOUNT variable, nor any temporary variables (apart from the fields of an input record) which therefore means you cannot recall information from a previous record or accumulate values.

SORT/MERGE cannot create more output records than it receives input records (although it can create less, with its conditional SKIP, and its DELETING DUPLICATES features).

If records from an input file are specified to be fixed-length (whether or not they really are), and the input file does not contain an exact multiple of the record-length characters, the last incomplete record of the input file will not be processed. To guard against loss of data in such situations we recommend you use COPY/A (the CLI command) to tack record-length minus one pad characters onto the back of the file, before processing with SORT/MERGE.

B.4 What is a SORT/MERGE Command File?

Although SORT/MERGE can be driven interactively, and entirely from the command line if the entire record is the sort-key, it is normally driven by commands held in a SORT/MERGE Command File. Apart from an ability to specify the input and output files from the command line there is no mechanism for passing dynamic or parametric information to SORT/MERGE. Therefore although it is possible to build fixed SORT/MERGE command files, which only differ in the files they process, you will usually find that most general purpose SORT/MERGE command files are created as temporary files.

If you wish to define the Record Format to be other than that with which a file is defined, it is not even possible to specify the input files from the command line. The /RECORD switch of the FILESTATUS command will tell you a file's defined Record Format. The examples we will be presenting in the next section deal mainly with files we wish to treat as DATASENSITIVE, but whose defined Record Formats may be anything from DATASENSITIVE to DYNAMIC to undefined, depending on how they were created. For this reason we cannot specify the input files in the command line, and therefore our SORT/MERGE command files are nearly always temporary files created within a CLI macro, so that CLI's argument and switch expansion syntax can be used to enable dynamic specification of the input files.

As with SPEED, when building a general purpose SORT/MERGE command file, we recommend you always encompass it in a CLI macro to simplify its use. In the examples that follow we use the **/S** switch to suppress SORT/MERGE's statistics display, so that the user is not even aware that SORT/MERGE is being used.

B.5 Examples

We have already shown you one example of a SORT/MERGE macro in Chapter 21 when describing the special CLI switch **/M**, and its use with the CLI CREATE and EXECUTE (XEQ or X) commands. That macro, **$FIXSORT**, is not in itself particularly useful and we do not intend to dwell on it here.

It is often possible to use SORT/MERGE to perform the identical editing function that might otherwise have been done with SPEED or a program written in a real programming language. Consider our "single most useful" macro **$FLIST**. It uses SPEED to edit the one-line per file output of the FILESTATUS command. The aim of the editing was to remove the leading spaces in front of the first filename, separate each filename with an argument delimiter, remove any "=" prefixes and to precede each NEWLINE with an ampersand "&", in order for the file to be in User Pseudo Macro format. If our current Working Directory contained the following files; FILE1, FILE2 and FILE3, the output of the following FILESTATUS command is shown:

```
) FILESTATUS/S/NHEADER FILE*
  =FILE1          =FILE2          =FILE3
)
```

while

```
) $FLIST/S/L=FILES.LS FILE*
```

would produce:

File FILES.LS

```
FILE1 &
FILE2 &
FILE3&
```

To use SORT/MERGE to perform this function, we have a problem because we can't tell when we've got the last record and the last record needs a different structure to the rest. However we can use SORT/MERGE's internal **RECORDCOUNT** variable to detect when we've got the first record. If we also observe that the following is a functionally equivalent User Pseudo Macro format:

```
FILE1&
  FILE2&
  FILE3&
```

Then we can write a SORT/MERGE equivalent of **$FLIST**, which we'll call say **$FLIST2**, as follows:

File $FLIST2.CLI

```
[!NEQUAL,,%0/H/HE/HEL/HELP%]
     TYPE [!PATHNAME $HELP]:[!EQUAL,.CLI,[!EEXTENSION %0\%]]&
[!ENAME %0\%][!ELSE][!EFILENAME %0\%][!END].HELP

[!ELSE]
     [!EQUAL,,%0/L=%]
          WRITE
          WRITE ERROR - Must specify output file with /L=file&
               switch[$$BELL]
          WRITE
     [!ELSE]
          (DELETE/2=IGNORE CREATE/DATASENSITIVE) %0/L=%
          FILESTATUS/NHEADER/CPL=1%0\NHEADER\CPL% %-%
          SORT/C=$FLIST.SMC/O/S INTO %0/L=% FROM %0/L=%
     [!END]
[!END]
```

File $FLIST.SMC

```
IF 1/1 = "<12>" THEN SKIP.
IF RECORDCOUNT=1 THEN REPLACE ANY " " IN 1/LAST WITH "".
REPLACE "=" IN 1/LAST WITH "".
REPLACE "<12>" IN 1/LAST WITH "&<12>".
COPY.
END.
```

Note that by forcing the output file **%0/L=%** to be created first as a DATASENSITIVE Record Format file, we are able to specify input and output files (note **/O** overwrite switch to enable SORT/MERGE to output over the top of an existing file, which in this case is also the input file), in the command line, meaning that we can use a permanent SORT/MERGE command file **$FLIST.SMC**.

The sequence **<n>** in a literal string is recognized by SORT/MERGE as meaning the single character with the ASCII value octal **n**. Therefore **"<12>"** is the NEWLINE character. **SKIP**ping records that contain only a NEWLINE is probably an unnecessary precaution. All spaces are removed from record one, any "=" characters are removed from all records, and all NEWLINEs (which as a default DATASENSITIVE delimiter will always be the LAST character in a record) are preceded with an ampersand. It is certainly much easier for a layman to understand this code than its SPEED equivalent, but we have stuck to our SPEED version of **$FLIST**, because it runs faster. The size of the files being processed and the type of processing being performed will affect the performance of SPEED and SORT/MERGE versions of the same task, differently and SPEED will not always outstrip SORT/MERGE.

Our next example is virtually trivial to code but highlights one of SORT/MERGE's most useful features, which is its ability to translate from one character set to another, using either a translation Table you define yourself with the TABLE statement, or one of the predefined ones. The predefined tables **EBCDIC_TO_ASCII** and **ASCII_TO_EBCDIC** are particularly useful when reading tapes from, or preparing tapes for another manufacturer's equipment. In this example we are going to use the predefined table **LOWER_TO_UPPER** to convert a text file to Upper case only alphabetic characters.

$TO_UPPER

This macro translates Lower Case ASCII characters in the text file supplied as the argument to UPPER case. Lines may be up to 136 characters. For files with lines longer than this, you must supply a /MAX=length switch. It uses SORT/MERGE.

Format: $TO_UPPER{/MAX=length} pathname

Optional Switch:

/MAX=length	Specifies maximum line length (default is 136)

Argument:

pathname	Pathname to the file to be translated to UPPER case

File $TO UPPER.CLI

```
[!NEQUAL,,%0/H/HE/HEL/HELP%]
     TYPE [!PATHNAME $HELP]:[!EQUAL,.CLI,[!EEXTENSION %0\%]]&
[!ENAME %0\%][!ELSE][!EFILENAME %0\%][!END].HELP

[!ELSE]
     [!EQUAL,,%0/MAX=%]
          %0\%/MAX=136 %1%
     [!ELSE]
          DELETE/2=IGNORE ?[!PID]UP.TMP
          CREATE/M ?[!PID]UP.TMP
INPUT FILE IS "%1%", RECORDS ARE DATASENS UPTO %0/MAX=% CHARACTERS.
OUTPUT FILE IS "%1%".
TRANSLATE 1/LAST USING LOWER_TO_UPPER.
COPY.
END.
)
          WRITE
          WRITE Translating Lower Case chars in [$$UNDERLINE %1%]&
               TO UPPER.
          SORT/C=?[!PID]UP.TMP/O/S
          DELETE ?[!PID]UP.TMP
          WRITE Done!
          WRITE
     [!END]
[!END]
```

Another particularly useful feature, of SORT/MERGE's tables is the ability to specify a table to be used as the collating sequence for **SORT**ing, without actually translating the characters of the record. For example the following Sort Key specification:

KEY 1/20 COLLATED BY LOWER_TO_UPPER.

will sort the records of a file into alphabetic sequence of the first 20 characters regardless of case, but without actually translating any lower case characters to upper case.

Our next example highlights the use of SORT/MERGE's conditional **SKIP**ping of records and **STOP**ping of processing features. In a development environment programmers are renowned for accidently deleting the source file of a program they are developing. More often than not they delete new sources that they just spent a good portion of the day entering, but which haven't yet been backed up to tape by the system's nightly backup. Normally there is no option for them but to sit down and type the whole damn thing in again. However you'd be surprised how many times the listing file, generated by the most recent attempt at compiling the program, still remains on disk after their little accident. If only there was a tool to reconstruct the original source from the listing file. We can use SORT/MERGE to provide just such a tool, which of course will be different depending on which programming language is being used. Our example **$COB_RECON** reconstructs sources from **.LS** listing files produced by Data General's **32-Bit COBOL Compiler**. In order to produce such a tool for other languages you must carefully analyse the differences between the source and the listing.

$COB_RECON

This macro reconstructs COBOL Source files from the Listing File produced by Data General's 32-Bit COBOL Compiler. It assumes the Listing File exists in the file "pathname.LS" and recreates the file "pathname" from this unless you specify a /EXT=.ext switch when it recreates "pathname.ext". It won't proceed if the Source file already exists. Before using it on your System test it, to see that it reconstructs a source identical to the original (you may need to change it slightly, in your environment, depending on compiler options chosen).

Format: $COB_RECON{/EXT=.ext} pathname

Optional Switch:

/EXT=.ext	To reconstruct "pathname.ext" rather than "pathname"

Argument:

pathname	Pathname to .LS Listing File (without the .LS ext)

In the case of the 32-Bit COBOL Compiler we have observed the following:

1. The first three lines of the listing (actually it's 4 because of an ASCII null character, which is one of the default datasensitive delimiters, after the filename) can be discarded.

2. Form-Feed characters which are also default datasensitive delimiters, occur in Column 7 if they were part of the original source, or in Column 1 (after a NewLine) if they were generated by the compiler because of a line beginning with **/**. To reconstruct source we must SKIP Form-Feeds generated by /'s.

3. All source lines are echoed in the listing prefixed by a right-justified line number in Columns 1 to 5, followed by an ASCII Tab character, or if the line was copied in from another source file via a COBOL COPY statement (Viz. INCLUDE in some other languages), a Capital **C** in Column 6 and then the Tab character. To reconstruct source we must SKIP lines copied in from other source files.

4. Depending on the Compiler Options chosen there will be various tables, error listings, and/or cross-reference listings attached to the end of the listing, which must be discarded.

File $COB RECON.CLI

```
[!NEQUAL,,%0/H/HE/HEL/HELP%]
     TYPE [!PATHNAME $HELP]:[!EQUAL,.CLI,[!EEXTENSION %0\%]]&
[!ENAME %0\%][!ELSE][!EFILENAME %0\%][!END].HELP

[!ELSE]

[!EQUAL,,[!PATHNAME %1%.LS]]
     WRITE
     WRITE ERROR - Can't find %1%.LS from which to reconstruct Source
     WRITE [$$BELL]
[!ELSE]
     [!NEQUAL,,[!PATHNAME %1%%0/EXT=%]]
          WRITE
          WRITE ERROR - %1%%0/EXT=% already exists. If you really want&
               to persist with
          WRITE,,,,,,,,,Source reconstruction DELETE or RENAME it&
               before trying again.
          WRITE [$$BELL]
     [!ELSE]
          (DELETE/2=IGNORE CREATE/M) :UDD:[!USERNAME]:?[!PID]COBR.SMC
INPUT FILE IS "%1%.LS",RECORDS ARE DATASENSITIVE UPTO 256 CHARACTERS.
OUTPUT FILE IS "%1%%0/EXT=%".
IF RECORDCOUNT < 5 THEN SKIP.
IF 1/1 = "<14>" THEN SKIP.
IF 1/1 < " " THEN STOP.
IF 6/6 <> "<11>" THEN SKIP.
COPY.
REFORMAT 7/LAST.
END.
)
          WRITE
          WRITE Reconstructing %1%%0/EXT=% from %1%
          SORT/S/C=:UDD:[!USERNAME]:?[!PID]COBR.SMC
          DELETE :UDD:[!USERNAME]:?[!PID]COBR.SMC
          WRITE Done!
     [!END]
[!END]
[!END]
```

Looking at the various conditional statements, we can see that:

IF RECORDCOUNT < 5 THEN SKIP.

will discard the initial compiler generated lines, while

IF 1/1 = "<14>" THEN SKIP.

discards the compiler generated Form-Feeds forced by lines beginning with **/**. The statement

IF 1/1 < " " THEN STOP.

says that as soon as a line beginning with any Control Character (most likely to be a NewLine) is reached, the end of the listing lines generated directly from the source has been reached, and anything that follows must be compiler generated junk at the end so **STOP** processing (because the source should now be fully reconstructed). It is important to note that the order in which these statements appear is important. If the previous two lines were reversed, the first Form-Feed generated by a **/**, would cause processing to cease and leave us with an incomplete source file. Furthermore the following statement must be placed after these two, because it makes reference to field **6/6**, ie Column 6, which would not exist in records whose only character is a delimiter. Reference to fields beyond the end of the record (unless you take other precautions such as PADding), will cause SORT/MERGE to terminate prematurely with a Run-Time error. That line then:

IF 6/6 <> "<11>" THEN SKIP.

which SKIPs all lines that do not have a Tab in column 6, serves the twofold purpose of discarding any lines copied in from other files by virtue of a **COPY** statement, and also any lines at the end of the listing that escape our **STOP** processing test. Finally all records that get passed to the output phase are stripped of their leading 6 characters with:

REFORMAT 7/LAST.

We have tested **$COB_RECON** with several 32-Bit COBOL listings using various options and have been able to reconstruct source files identical to the originals. We have not tested it with listings produced from **CARD** Format sources, and expect that it would have to be altered to cope with these, but nobody uses CARD Format sources these days, do they??

Well we reckon that should be enough to whet your appetite.

B.6 SORT/MERGE and INFOS Files

If we lose you in this Section, don't worry about it, our aim here is to give sophisticated **INFOS** users a few tips from our experience, about using SORT/MERGE with INFOS files. Although we do not intend to give many actual examples, we will state however that we have built some pretty useful INFOS utilities using SORT/MERGE. For example with the aid of a COBOL program called **POOH2**, which DG once (perhaps they still do - we haven't looked for a while) published in its Commercial Newsletter, which read the INFOS DB Volume and produced as output a Sequential file containing the Data Base records, we built a SORT/MERGE and associated CLI macro, to **REBUILD** a simple **ISAM** file from the DB Volume. As well as overcoming problems encountered with corrupt indexes for which we didn't have a valid tape backup copy, it also had the benefit of reclaiming a considerable amount of diskspace. This is because **INFOS**, never releases space back to AOS/VS for re-use, even with **SPACE MANAGEMENT** turned on. SPACE MANAGEMENT only enables re-use internally by **INFOS**. Thus every INFOS file will grow to the maximum size it has ever attained, and even after considerable purging of records the file will still occupy the same amount of Disk space.

The following tips are presented for the benefit of sophisticated INFOS users, who believe that they might be able to use SORT/MERGE's INFOS hooks to save them considerable programming effort. There is no particular significance to their order.

1. When the Key is not included in the Data Base record remember to include both a **,RECORD** and a **,KEY** phrase. Often the length of the key is fixed but the record variable, and therefore although on first reading of the syntax definition you might think you must specify the **,RECORD** phrase first, the order of these phrases is up to you, and determines what order the retrieved fields (note you can also retrieve the Partial Record if you have one) are placed in SORT/MERGE's record buffer. Therefore when the Key is not in the Record, your PATH IS clause might look like this:

   ,PATH IS *,KEY,RECORD

   Note that even when the Key is also variable length SORT/MERGE gives you the option to PAD it to a fixed length on retrieval so that you can tell where the Key ends and the Record begins, eg:

   ,PATH IS *,KEY PADDED TO 10 CHARACTERS WITH "#",RECORD

2. If you use **ANSI-Standard COBOL Alternate Keys**, rather than creating your own DBAM subindexes, note that these are actually implemented, and created by COBOL's **OPEN OUTPUT** statement, as **DBAM** files, where the Primary Keys are held in a subindex whose key is a 16-Bit (2-Byte) field containing binary zero, while the first specified Alternate Keys are held in a subindex whose key is a 16-Bit field containing binary one. For each additional Alternate you simply increment the subindex key by one in their order of specification in the **SELECT** clause. To visit every key in Primary Key sequence using SORT/MERGE, the appropriate **PATH IS** clause for the **INPUT INFOS INDEX** statement is:

   ,PATH IS "<0><0>",*

To visit every key in the sequence of the first Alternate Key the corresponding clause is:

```
,PATH IS "<0><1>",*
```

3. The compulsory **RECORDS ARE** clause of the **INPUT INFOS INDEX** statement really only serves the purpose of informing SORT/MERGE how big its Record Buffer needs to be, since INFOS always writes records in Variable Length Format. When retrieving more than just the Record (eg. the Key and/or the Partial Record), remember to allow enough room in the **VARIABLE UPTO** phrase for the sum of the length of these fields, allowing for the maximum record length.

4. Note the **TRIM "literal" FROM KEYS** clause of the **OUTPUT INFOS** statement, enables you to remove the PAD characters you may have used in retrieving Keys as shown under Point 1 above.

5. Note that **OUTPUT INFOS INDEX** means that for each key written a Data Base Record is to be written, while **OUTPUT INFOS INVERSION** means to merely write a Key with a pointer to the existing Data Base Record determined from the pointer associated with the Input Record.

6. SORT/MERGE cannot create an INFOS **SUBINDEX**. These must be created externally, say with INQUIRE. It can however write the top level keys for which SUBINDEXes must be externally defined. To rebuild an entire DBAM structure with SORT/MERGE you would need a minimum of one pass for each level of index.

It is difficult to dream up meaningful general purpose examples to demonstrate how we might use SORT/MERGE with INFOS files so we'll dream up a specific example. Actually we didn't dream it up it's a real life example. Someone had developed a System for controlling an in-house work Football Tipping competition. The System consisted of a single Program that read in a Fixtures Data Base, and maintained a Users Data Base. There are 7 matches (14 teams) in a round and 22 rounds in a season. The Users Data Base (**USER** & **USER.DB**), was a simple ISAM file where the records of the file were 214 characters, whose COBOL record description was as follows:

```
01   USER-RECORD.
     03   USER-CODE                    PIC X(3).
     03   USER-NAME                    PIC X(20).
     03   USER-PWORD                   PIC X(15).
     03   USER-ROUND                   OCCURS 22 TIMES.
          05   USER-PAID               PIC X.
          05   USER-TIPS               OCCURS 7 TIMES.
               07   USER-SELECTION     PIC X.
```

USER-CODE was the Key to the User records and was unique. USER-CODE usually consisted of a User's three initials, while the USER-NAME was his full name entered with the surname first. USER-PWORD was the user's Password and a Hard-Coded Master User maintained the Data-Base and the record associated with him held the Results. All reports and screen based results analyses were reported in **USER-CODE** sequence. After the first Round it was decided that Reports would be better in **USER-NAME** sequence to simplify the process of a User finding himself on the ladder.

The Program was modified to process the **USER** Data Base as an ANSI-Standard Alternate Key Structured File where **USER-CODE** was the Primary Key and **USER-NAME** was the Alternate Key. This of course required Data Base conversion, which was done with the following macro:

File CONVERT.CLI

```
WRITE
WRITE Coverting USER Data Base to New Format:
IRENAME <,OLD_>USER(,.DB)
WRITE
WRITE Creating New Format USER Data Base:
ICREATE/B=USER.ICR.TR
INQUIRE/B=USER.INQ.TR
WRITE Loading Data from OLD_USER to USER
SORT/S/C=USER.PASS1.SMC
SORT/S/C=USER.PASS2.SMC
WRITE Done!
WRITE
```

The appropriate SORT/MERGE commands files were:

File USER.PASS1.SMC

```
INPUT INFOS INDEX IS "OLD_USER", PATH IS *, RECORD
,RECORDS ARE 214 CHARACTERS.
OUTPUT INFOS INDEX IS "USER", RECORD IS 1/LAST, PATH IS "<0><0>",*.
KEY 1/3.
COPY.
END.
```

File USER.PASS2.SMC

```
INPUT INFOS INDEX IS "USER", PATH IS "<0><0>",*, RECORD
,RECORDS ARE 214 CHARACTERS.
OUTPUT INFOS INVERSION IS "USER", PATH IS "<0><1>",*.
KEY 4/23.
COPY.
END.
```

Note that in **PASS1** we take input from the original IRENAMEd Data Base **OLD_USER** and output both key and record to the Primary Key INDEX of the new Data Base **USER.** The new Data Base must already exist (this is done by **ICREATE/B=USER.ICR.TR**) and the Primary Key and Alternate Key Subindexes must have been created (**INQUIRE/B=USER.INQ.TR** does this). In **PASS2** we take input from the newly written Primary Subindex (since we need to pick up the pointer to the Data Base record), and output just the keys (OUTPUT INFOS INVERSION) to the Alternate Subindex. The new program was able to successfully process the new Data Base. We have not included this example on the separately purchasable Magnetic Tape because it is too specific and useless without the Program that maintains the Data Base and the Fixtures Data Base as well.

For your interest the two trailfiles created with corresponding **/T=trailfile** switches, in a test environment are shown following:

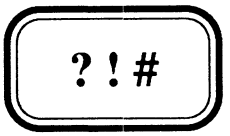
File USER.ICR.TR

```
ICREATE/T=NEW.ICR.TR

****** INFOS FILE CREATION     04/03/89  23:12:38 ******

NAME OF FILE TO BE CREATED: USER
ACCESS METHOD (I=ISAM, D=DBAM) [D]:

****** DEFINE INDEX FILE ******

      MAXIMUM NUMBER OF INDEX LEVELS [2]:
      PAGE SIZE (BYTES) [2048]:
      PARTIAL RECORD LENGTH [0]:
      ROOT NODE SIZE [2042]:
      MAXIMUM KEY LENGTH [255]:
      ALLOW DUPLICATE KEYS IN THIS INDEX? (Y OR [N]):
      ENABLE SPACE MANAGEMENT? (Y OR [N]):
      ENABLE KEY COMPRESSION? (Y OR [N]):
      OPTIMIZE RECORD DISTRIBUTION? (Y OR [N]):

      ****** DEFINE INDEX VOLUME(S) ******

      NUMBER OF VOLUMES TO DEFINE [1]:
      VOLUME 1 NAME [VOL01]:
           SPECIFY MAXIMUM SIZE? (Y OR [N]):
           SPECIFY FILE ELEMENT SIZE? (Y OR [N]):

****** DEFINE DATABASE FILE ******

      DATABASE FILE NAME [USER.DB]:
      PAGE SIZE (BYTES) [2048]:
      ENABLE SPACE MANAGEMENT? (Y OR [N]):
      ENABLE DATA RECORD COMPRESSION? (Y OR [N]):
      OPTIMIZE RECORD DISTRIBUTION? (Y OR [N]):

      ****** DEFINE DATABASE VOLUME(S) ******

      NUMBER OF VOLUMES TO DEFINE [1]:
      VOLUME 1 NAME [VOL01]:
           SPECIFY MAXIMUM SIZE? (Y OR [N]):
           SPECIFY FILE ELEMENT SIZE? (Y OR [N]):
```

All responses in the above are default except the highlighted NAME OF FILE TO BE CREATED, namely **USER**. In the Inquire trailfile that follows it too shows non-default responses highlighted but as you will see there are considerably more:

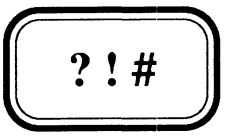
File USER.INQ.TR

```
INQUIRE/T=USER.INQ.TR
INDEX: USER
NUMBER OF LOCKS:  [0]:
ENTER C TO GET LIST OF COMMANDS

COMMANDS: F
FORMATS IN FILE? (Y OR [N]):
NUMBER OF LEVELS TO BE DESCRIBED:  [1]:
LEVEL # 0
KEY
     NUMERIC? (Y OR [N]): Y
     RADIX (8,10, OR 16):  [8]: 10
     BYTES (B) OR WORDS(W):  [B]: W
PARTIAL RECORD
     NUMERIC? (Y OR [N]):
NUMBER OF FIELDS IN DATABASE RECORD:  [1]:
FIELD #1
     NUMERIC? (Y OR [N]):
     STARTING CHARACTER POSITION (1-N):  [1]:
     FIELD LENGTH, BYTES:  [0]:
SAVE FORMATS IN FILE? (Y OR [N]):

COMMANDS: K W B
NUMERIC KEY (DECIMAL WORDS)
      ****** ENTER CARRIAGE RETURN TO TERMINATE LOOP ******
     WORD #1: 0
     WORD #2:

COMMANDS: S
ROOT NODE SIZE: 2042
MAXIMUM KEY LENGTH:  [255]:
PARTIAL RECORD LENGTH:  [0]:
SUBINDICES ALLOWED? (Y OR [N]):
ALLOW DUPLICATE KEYS IN THIS INDEX? (Y OR [N]):

COMMANDS: K W B
NUMERIC KEY (DECIMAL WORDS)
      ****** ENTER CARRIAGE RETURN TO TERMINATE LOOP ******
     WORD #1: 1
     WORD #2:

COMMANDS: S
USE PREVIOUS INFO? (Y OR [N]): N
ROOT NODE SIZE: 2042
MAXIMUM KEY LENGTH:  [255]:
PARTIAL RECORD LENGTH:  [0]:
SUBINDICES ALLOWED? (Y OR [N]):
ALLOW DUPLICATE KEYS IN THIS INDEX? (Y OR [N]): Y

COMMANDS: CLOSE
INDEX:
```



APPENDIX C

Example Macros on Magnetic Tape



APPENDIX C

Example Macros on Magnetic Tape

C.1 The Files on Tape

All the example macros described in this Manual (except where explicitly stated otherwise) are available on Magnetic Tape. This Tape can be purchased separately, and is delivered free with any orders of ten or more Manuals. The following files, listed alphabetically rather than in DUMP order are all present on the tape:

$$ARG1.CLI
$$ARGS.CLI
$$ARG_SPLIT.1.CLI
$$ARG_SPLIT.CLI
$$BELL.CLI
$$BLINK.CLI
$$COMMA.CLI
$$COMMENTS.CLI
$$CR.CLI
$$CURSOR.CLI
$$DATETIME.CLI
$$DAY.CLI
$$DAYS.1.1.CLI
$$DAYS.1.CLI
$$DAYS.CLI
$$DAYS_TO_BOM.CLI
$$DAYS_TO_BOY.CLI
$$DIM.CLI
$$DOW.CLI
$$ERASE_EOL.CLI
$$ERASE_PAGE.CLI
$$EXT_SPLIT.CLI
$$HHMMSS.1.CLI
$$HHMMSS.CLI
$$IIPC.ENV.CLI
$$IN_ANGLE.CLI
$$IN_ROUND.CLI
$$IN_SQUARE.CLI
$$LENGTH.CLI
$$MM.CLI
$$NL.CLI
$$ROUND_UP.CLI
$$TAB.CLI
$$UNDERLINE.CLI
$$USER_DECODE.1.CLI
$$USER_DECODE.CLI
$$YYMMDD.1.CLI
$$YYMMDD.CLI
$ARGTEST.1.CLI
$ARGTEST.CLI
$COB_RECON.CLI
$COUNTDOWN.CLI
$DEF_ELSIZE.1.CLI
$DEF_ELSIZE.CLI
$DELETE.CLI
$DISKSPACE.1.1.1.CLI
$DISKSPACE.1.1.CLI
$DISKSPACE.1.CLI
$DISKSPACE.CLI
$ED.PRINTER.1.CLI
$ED.PRINTER.CLI
$ED.PRINTER.CLNUP.CLI
$ED.PRINTER.CREATE.CLI
$ED.PRINTER.DEL.CLI
$ED.PRINTER.DEV.CLI
$ED.PRINTER.DISP.CLI
$ED.PRINTER.EDIT.CLI
$ED.PRINTER.FORMS.CLI
$ED.PRINTER.Q.CLI
$ED.PRINTER.QS.1.CLI
$ED.PRINTER.QS.CLI
$ED.PRINTER.SW.CLI
$ED.PRINTER.YN.CLI
$FILECHECK.CLI
$FIXSORT.CLI
$FIXSORT2.CLI
$FLIST.CLI
$FLIST.SMC
$FLIST.SPD
$FLIST2.CLI
$HELP:#
$HELP.CLI
$HELP.LIST.CLI
$HELP.SPD
$IIPC.1.CLI
$IIPC.CLI
$INPUT.ASK.CLI
$INPUT.CHK.CLI
$INPUT.CLI
$LOGONCHECK.1.CLI
$LOGONCHECK.CLI
$LOGONCHECK_WHO.CLI
$LOOP.CLI
$LOOP4.CLI
$LOOP_COUNT.1.CLI
$LOOP_COUNT.CLI
$LOOP_CREATE.CLI
$MENU.CHOICE.CLI
$MENU.CLI
$MENU.PROCESS.CLI
$NEST.1.1.1.CLI
$NEST.1.1.CLI
$NEST.1.CLI
$NEST.CLI
$NEST2.CLI
$NOTOUTOFMEMORY.CLI
$NUM.CLI
$NUMBER.CLI
$OPT_ELSIZE.1.1.CLI
$OPT_ELSIZE.1.CLI
$OPT_ELSIZE.CLI
$OUTDEMO.CLI
$OUTOFMEMORY1.CLI
$OUTOFMEMORY2.CLI
$PSED.1.CLI
$PSED.CLI
$READ.CLI
$READIT.CLI
$SETACL.CLI
$SMAC.CLI
$SMAC.SPD
$SPEEDLIST.CLI
$SPEEDTYPE.CLI
$SPEEDTYPE.SPD
$SWTEST.1.CLI
$SWTEST.CLI
$TEST_A.CLI
$TO_UPPER.CLI
$UP.PRINTER.1.CLI
$UP.PRINTER.CLI
$YN.CLI
FRED.1.CLI
FRED.2.CLI
FRED.MENU
FREDSUB.MENU
FREDSUB.N.CLI
FREDSUB.S.CLI
GLOBAL_OP.BYE.CLI

The **$HELP** directory used by the **$HELP** macro contains the following HELP files:

$$ARG1.HELP	$$ARGS.HELP	$$ARG_SPLIT.HELP
$$BELL.HELP	$$BLINK.HELP	$$COMMA.HELP
$$CR.HELP	$$CURSOR.HELP	$$DATETIME.HELP
$$DAY.HELP	$$DAYS.HELP	$$DAYS_TO_BOM.HELP
$$DAYS_TO_BOY.HELP	$$DIM.HELP	$$DOW.HELP
$$ERASE_EOL.HELP	$$ERASE_PAGE.HELP	$$EXT_SPLIT.HELP
$$HHMMSS.HELP	$$IN_ANGLE.HELP	$$IN_ROUND.HELP
$$IN_SQUARE.HELP	$$LENGTH.HELP	$$MM.HELP
$$NL.HELP	$$ROUND_UP.HELP	$$TAB.HELP
$$UNDERLINE.HELP	$$USER_DECODE.HELP	$$YYMMDD.HELP
$ARGTEST.HELP	$COB_RECON.HELP	$COUNTDOWN.HELP
$DEF_ELSIZE.HELP	$DELETE.HELP	$DISKSPACE.HELP
$ED.PRINTER.DISP.HELP	$ED.PRINTER.HELP	$ED.PRINTER.YN.HELP
$FILECHECK.HELP	$FIXSORT.HELP	$FLIST.HELP
$FLIST2.HELP	$HELP.HELP	$IIPC.1.HELP
$IIPC.HELP	$INPUT.HELP	$LOGONCHECK.HELP
$LOOP.HELP	$LOOP_COUNT.HELP	$LOOP_CREATE.HELP
$MENU.HELP	$NEST.HELP	$NEST2.HELP
$NOTOUTOFMEMORY.HELP	$NUM.HELP	$NUMBER.HELP
$OPT_ELSIZE.HELP	$OUTDEMO.HELP	$OUTOFMEMORY1.HELP
$OUTOFMEMORY2.HELP	$PSED.HELP	$READ.HELP
$READIT.HELP	$SETACL.HELP	$SMAC.HELP
$SPEEDLIST.HELP	$SPEEDTYPE.HELP	$SWTEST.HELP
$TEST_A.HELP	$TO_UPPER.HELP	$UP.PRINTER.HELP
$YN.HELP		

C.2 Loading Instructions

The **$UP.PRINTER**, **$ED.PRINTER**, **$DEF_ELSIZE**, and **$OPT_ELSIZE** Macros require that the directory **:$SYSPARAMS** exists, which we suggest you create as follows:

```
) SUPERUSER ON; Comment you'll have to be OP to do it
*) CREATE/MAX=1000 :$SYSPARAMS
*) ACL :$SYSPARAMS OP,OWARE +,RE
*) SUPERUSER OFF
)
```

The files on Tape can be loaded in any directory which is on everybody's SEARCHLIST and we recommend **:UTIL** for this. To load:

```
) SUPERUSER ON; Comment you'll have to be OP to do it
*) DIRECTORY :UTIL; Comment or wherever you want to Load them
*) LOAD/V/BUFF=8192 @MTxn:0; Comment after mounting @ 1600 bpi
*) SUPERUSER OFF
)
```

where **x** will depend on what kind of Tape Drive you have and **n** on the Unit number (eg @MTD0:0). On systems where the default elementsize is 4 blocks these files will occupy about 850 Disk blocks, and about 350 blocks on those with a default elementsize of 1 (we ran $OPT_ELSIZE across all these files). The Top of Page 28-3 shows you the technique for linking these macros through a directory like **:UTIL**, after loading them in some alternate directory, such as **:$MACROS**. You should run **$DEF_ELSIZE** as soon as possible so that users can run **$OPT_ELSIZE**.

C.3 A Summary of the Macros

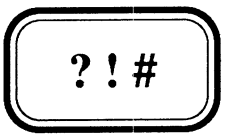
In the summaries shown below we believe you will find the highlighted macros particularly useful. In some cases we believe they are so useful as to be worth the purchase price of the Manual on their own.

Macros

$ARGTEST	Dummy macro for Demo purposes only
$COB_RECON	**Reconstruct COBOL source from Listing**
$COUNTDOWN	Dummy macro for Demo purposes only
$DEF_ELSIZE	Determines System Default Elementsize
$DELETE	Deletes regardless of Permanence
$DISKSPACE	**Displays %, Available & Total Space**
$ED.PRINTER	**Maintains Printer Parameter Files**
$FILECHECK	FILESTATUS switches existence check
$FIXSORT	Dummy macro for Demo purposes only
$FLIST	**!FILENAMES with FILESTATUS switches**
$FLIST2	Same as $FLIST using SORT/MERGE
$HELP	**Gives HELP on all these**
$IIPC.1	Sample generalized Initial IPC
$IIPC	Sample generalized Initial IPC
$INPUT	**String Input & Validity Checker**
$LOGONCHECK	**Stops Multiple Logons per Username**
$LOOP	Dummy macro for Demo purposes only
$LOOP_COUNT	Cyclic Loop Counter
$LOOP_CREATE	Initializes Cyclic Loop Counters
$MENU	**CLI Macro Menu Processor**
$NEST	Dummy macro for Demo purposes only
$NEST2	Dummy macro for Demo purposes only
$NOTOUTOFMEMORY	Dummy macro for Demo purposes only
$NUM	**Numeric Input & Validity Checker**
$NUMBER	**Numeric Input & Validity Checker**
$OPT_ELSIZE	**Elementsize Optimizer**
$OUTDEMO	Dummy macro for Demo purposes only
$OUTOFMEMORY1	Dummy macro for Demo purposes only
$OUTOFMEMORY2	Dummy macro for Demo purposes only
$PSED	Runs SED retaining permanence & no .ED
$READ	Failsafe CLI string Input
$READIT	Used by $READ
$SETACL	Dummy macro for Demo purposes only
$SMAC	**CLI MACro Structural Indentation**
$SPEEDLIST	Lists printable copy of SPEED macros
$SPEEDTYPE	Types printable copy of SPEED macros
$SWTEST	Dummy macro for Demo purposes only
$TEST_A	Dummy macro for Demo purposes only
$TO_UPPER	Coverts Text files to Upper Case
$UP.PRINTER	**Brings Up Printer Co-operatives**
$YN	Y/N Asker and Validity Checker

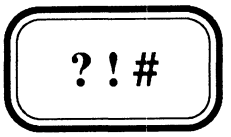
User Pseudo Macros

$$ARG1	Returns Argument 1
$$ARGS	**Returns Count of Arguments**
$$ARG_SPLIT	Returns Argument Split into Components
$$BELL	Rings Bell in WRITE's
$$BLINK	Blinks Text in WRITE's
$$COMMA	Displays a comma in WRITE's
$$CR	Displays a <CR> in WRITE's
$$CURSOR	**Positions Cursor on Screen in WRITE's**
$$DATETIME	**Returns current YYMMDD_HHMMSS**
$$DAY	Returns Day of Week from n = 0 - 6
$$DAYS	**Returns No Days since 1st January 1900**
$$DAYS_TO_BOM	Returns Days to Beginning of MM in YY
$$DAYS_TO_BOY	Returns Days to Beginning of YY
$$DIM	Dims Text in WRITE's
$$DOW	**Returns Day of Week given Date**
$$ERASE_EOL	Zaps to End of Line in WRITE's
$$ERASE_PAGE	Clears Screen in WRITE's
$$EXT_SPLIT	Returns Argument Split into extensions
$$HHMMSS	Returns HHMMSS from Time
$$IN_ANGLE	Text in Angle brackets in WRITE's
$$IN_ROUND	Text in Round brackets in WRITE's
$$IN_SQUARE	Text in Square brackets in WRITE's
$$LENGTH	Returns no of characters in Argument
$$MM	Returns MM from MON (eg 01 from JAN)
$$NL	Displays <NL> in WRITE's
$$ROUND_UP	Returns n rounded up to mutiple of m
$$TAB	Displays Tab in WRITE's
$$UNDERLINE	Underlines Text in WRITE's
$$USER_DECODE	Breaks Username into components
$$YYMMDD	**Returns YYMMDD from Date**



APPENDIX I

INDEX



APPENDIX I

INDEX

Note that **f** after a page reference indicates this and the following page, while **ff** indicates this and two or more of the following pages within the same Chapter. Reference to CLI Commands and Pseudo Macros is made where they are first described, and occasionally where they are used if such use is considered a significant departure from previous or standard usage.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

S (continued)

T

U

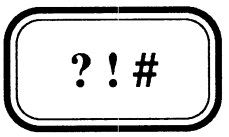
V

W

X

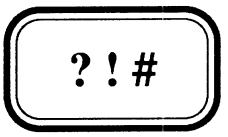
Y

Z



APPENDIX X

The Macros that generated the Index



APPENDIX X

The Macros that generated the Index

X.1 Index Generation

The preceding Index is a little different to most you will see. We were thinking about how we might build an Index for this Manual, and had almost decided, that because of its tutorial and hierarchic nature, the Manual didn't really need one (actually we couldn't be bothered poring over the manual building tables of what should be indexed and where it is referenced), when we realised we could use the tools we have just described in the Manual, namely **CLI** Macros, **SPEED** Macros, and **SORT/MERGE** Command Files, to automatically generate an Index. As an afterthought we thought our readers, might be interested to see how these tools could be used to solve such a problem, and so we added this Appendix. The macros are designed to solve our particular problem, and are therefore not of any general purpose use, but the techniques used are. For this reason these macros are shown only in the Manual, and are not present on the associated Magnetic Tape.

Bearing in mind that each Chapter of this Manual is actually a separate (so that CEO's automatic Page Numbering could be used to reset the page to 1 for each Chapter) **CEO** document, the steps involved were as follows:

1) Export each Chapter to an AOS/VS TXT file with a 2-character filename representing the Chapter no (01 02 .. 30 AA BB CC).

2) Use a CLI and SPEED macro to put correctly formatted Page numbers on Line 1 of each Page of each Chapter. (Note although CEO's EXPORT facility allows you to include printer information from the Headers & Footers which would automatically include generated Page Nos, we found that some of our pages were split into two pages, where they print as one and therefore these page numbers could not be relied upon. When we exported without printer information the AOS/VS TXT files had the correct number of pages.)

3) Use a CLI macro to JOIN the various Chapters to form a single file to contain the entire Manual, remembering to include a Form-Feed character between each Chapter.

4) Use SPEED macro to prepare the file representing the Manual for Searching. This includes removing all NEWLINES replacing them with Spaces (so that phrases split across lines will be found) and reducing all strings of spaces and tabs to a single space.

5) Use a CLI and SPEED macro to search for and generate the Index for each string or phrase you want entered in the Index.

6) Use a CLI macro and SORT/MERGE command file to Sort the Index.

7) Manually edit it to remove any unwanted references, and finally IMPORT it back into CEO for inclusion in the Manual.

This procedure is not fast, but who cares? It didn't take long to develop, it works and it's a once-off job. We ran it overnight for a couple of nights (ie Step 5 which is the time comsuming part).

X.2 The Index Macros

The following Macro adds Page numbers to the Chapter file specified as the argument.

File NO PAGE.CLI

```
NO_PAGE.1 %1% [!EXPLODE %1%]
```

File NO PAGE.1.CLI

```
WRITE Numbering Pages for %1%
DELETE/2=IGNORE CHAP.INP
[!EQUAL,*,&
[!EQUAL,A,%2%]*[!END][!EQUAL,B,%2%]*[!END][!EQUAL,C,%2%]*[!END]]
     WRITE/L=CHAP.INP,,%2%
[!ELSE]
     [!EQUAL,0,%2%]
          WRITE/L=CHAP.INP,,%3%
     [!ELSE]
          WRITE/L=CHAP.INP %2%%3%
     [!END]
[!END]
PROCESS/BLOCK/OUTPUT=@NULL/DATA=CHAP.INP/LIST SPEED/I=NO_PAGE.SPD %1%
DELETE CHAP.INP
```

File NO PAGE.SPD

```
0vs0$$$<i^F@data$$-1di-$vi0\10ir;>fuh
```

The Generic File @DATA contains the Chapter (or Appendix) number, which is inserted at the top of each page followed by a hyphen, the page number and a NEWLINE. The expression **vi0** increments variable zero and inserts into the buffer the ASCII character string representing the incremented value, and since this is performed once for each page it represents the page number.

The following Macro uses COPY/A and a special file FF containing a Form-Feed character to join the various chapters. The macro was called using **JOIN MANUAL [CHAPTERS]**, where CHAPTERS was formed using **$FLIST/S/L=CHAPTERS **\FF**.**

File JOIN.CLI

```
[!NEQUAL,,%2%]
     [!EQUAL,,[!FILENAME %1%]]
          WRITE Creating %1% by copying %2%
          CREATE/ELEMENT=16 %1%
          COPY/A %1% %2%
     [!ELSE]
          WRITE Appending %2% to %1%
          COPY/A %1% FF %2%
     [!END]
     %0-1%,%3-%
[!END]
```

This SPEED macro replaces all NEWLINEs (except the first on the Page number line) with spaces and reduces all strings of spaces and tabs (^**E**) to a single space. It also displays the page number line for each page as it processes it, as well as the number of pages processed.

File PREP.SPD

```
0VS0$$<1TVI0=1L<C
^E$ $;><C^E$ $;>R;>FUH
```

The **INDEX** macro adds the item supplied as the argument to the Index by searching for it throughout the file **MANUAL**, and reporting to the list file **INDEX.LS**, a line containing the item and the page number for each occurrence of it (once it is found in a page, it does not search further within that page but moves on to the next one). Because of various restrictions imposed by specifying the searchstring as a CLI argument (eg. no spaces, no brackets, etc), the associated SPEED macro **INDEX.SPD**, has been written to recognise the following special characters in the searchstring:

~	(Tilde) used to represent a **space** in a searchstring
{¦	(Open curly - vertical bar) used to represent **[**.
¦}	(Vertical bar - close curly) used to represent **]**.
{/	(Open curly - slash) used to represent **(**.
\}	(Backslash - close curly) used to represent **)**.
^	(Uparrow ie Shift-6) used to match any **digit** in the searchstring. Replaced by **n** when reported.
¦text¦	(Text enclosed in vertical bars) means to include this text as part of the searchstring but not to report it.
{text}	(Text enclosed in curly brackets) means to report this text but not to include it as part of the searchstring.
'string'	(Searchstring enclosed in single quotes) makes the search case-sensitive, otherwise it is not. (Actually only the terminating quote is necessary, the leading one being optional.

File INDEX.CLI

```
[!NEQUAL,,%1%]
     [!EQUAL,,[!FILENAME INDEX.LS]]
          CREATE/ELEMENTSIZE=8 INDEX.LS
     [!END]
     DELETE/2=IGNORE ITEM
     WRITE/L=ITEM %1%
     WRITE Adding "%1%" to INDEX.LS
     PROCESS/BLOCK/OUTPUT=@NULL/DATA=ITEM/LIST=INDEX.LS&
           SPEED/I=INDEX.SPD
     DELETE ITEM
[!END]
```

File INDEX.SPD

```
frmanual$$fw@list$$
bsa$i^F@data$$-1d:.-1,.s'$"n-1djvc-39"e1d'1ws'
j<c~$ $;><c{¦$[$;><c¦}$]$;><c{/$($;><c\}$)$;>
#bcxbsx<c¦$$;><c^$^\0123456789^\$;><C{^X^N}}$$;>
bsavc-32"e1d'<c^$n$;><c¦^X^N¦¦$$;><s{$;-1m.vs0$$s{^X^N}}$$;-1djv0m1d>
1m:.-1,.s^\!/^\$vs0"e:.-1,.s$$vs0"e2vs0'v0-1"e:.,.+1s$$vs0$$1-v0vs0''
jv0<i $>zji  $40-.<i.$>i  $bs0
<y;q^Bx$;bsa#pwbs0$j1pw>fch
```

A brief line by line description of what this SPEED macro does follows:

Line 1 — Open the file **manual** for input and the Listfile (**@list**) for output (ie INDEX.LS).

Line 2 — Read into Buffer-A the searchstring held in the Datafile (**@data**) and turn case-sensitive search mode on (**1ws**) if the searchstring is enclosed in (or ends with) single quotes. Delete these single quotes if present.

Line 3 — Make special character substitutions (**space**, **[**, **]**, **(**, and **)** for **~**, **{¦**, **¦}**, **{/**, and **\}** respectively).

Line 4 — Copy to Buffer-X to set up searchstring, by removing vertical bars (**¦**), replacing **^** with a special control string to match any digit, and to remove any text surrounded by curly brackets (**<c{^X^N}}$$;>**).

Line 5 — Back to Buffer-A to set up report string, by deleting a leading space (**vc-32"e1d'**), replacing **^** with **n**, removing text surrounded by vertical bars (**<c¦^X^N¦¦$$;>**), and removing the curly bracket pairs from the text they enclose (**<s{$;-1m.vs0$$s{^X^N}}$$;-1djv0m1d>**).

Line 6 — Set up Variable zero to contain the appropriate number of spaces to insert before the searchstring to force correct alphabetic sorting from column 3. Thus for strings beginning with **!**, **/**, or a single **$**, **v0** is set to 1, for strings beginning with **$$** it is set to 0, while for others it is set to 2.

Line 7 — Insert that number of spaces at the beginning of Buffer-A, two spaces at the end of the report string, then pad to 40 characters with periods (**40-.<i.$>**), and finally two more spaces after which the page number will appear in the output file if the searchstring in Buffer-X is found. Switch back to the main Buffer (**bs0**).

Line 8 — Search across pages for the contents of Buffer-X (**q^Bx$**) and each time we find it, output the contents of Buffer-A (**bsa#pw**) followed by the page reference in line one (**bs0$j1pw**). Finally close files and Halt.

Depending on the number of occurrences of the searchstring, and the system workload this process was observed to take anything from 1 to 5 minutes for each searchstring.

Below are some examples of the actual searchstrings we specified:

```
'{Input~-}~Response~Evaluation'
'{Input~-}~Numeric¦~Input¦'
'{CLI~-~Environment~Preservation}¦Preserving~the~Environment¦'
{Logic~-~Complex}~OR~{/inclusive\}
'$FLIST¦.CLI¦{~Macro}'
~/^={severity Switch}
'$$DOW¦.CLI¦{~User~Pseudo~Macro}'
'¦{¦¦!VARn{~Pseudo~Macro}'
'X¦EQ¦{~Command}'
```

Finally the following macro is used to sort the generated Index into alphabetical sequence. Note that sorting is insensitive to case but no actual case conversion takes place. Note also the importance of the placement of massaging statements with respect to the imperative statement (STABLE SORT). On input single digit page numbers (**-n**) are replaced with (**-0n**), so that **-2** will sort before **-10**, while on output (after the imperative) these inserted **0**'s are removed. In a similar fashion Appendix Letters **A**, **B**, and **C** are duplicated in the preceding character position (which would have been a space), so that **<space>A-1** sorts after **10-1**, with the space being restored on output.

File SORTINDEX.CLI

```
WRITE
WRITE Sorting INDEX.LS
(DELETE/2=IGNORE CREATE/M) SINDEX.SMC
INPUT FILE IS "INDEX.LS",RECORDS ARE DATASENS UPTO 50 CHARACTERS.
OUTPUT FILE IS "INDEX.LS".
KEY 3/LAST COLLATED BY LOWER_TO_UPPER.
IF 47/47 = "<12>" THEN INSERT "0" BEFORE 46.
IF 44/44 = "A" OR 44/44 = "B" OR 44/44 = "C" THEN
   REFORMAT 1/42,44/44,44/LAST.
STABLE SORT.
REPLACE "0" IN 46/46 WITH "".
IF 44/44 = "A" OR 44/44 = "B" OR 44/44 = "C" THEN
   INSERT " " IN 43/43.
END.
)
SORT/O/C=SINDEX.SMC/S
DELETE SINDEX.SMC
WRITE Done!
```

THIS PAGE INTENTIONALLY LEFT BLANK

? ! # Hook-Bang-Crunch

by Greg Shalless

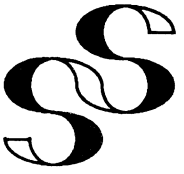
This book / manual is designed to teach its readers how to use the **CLI** (Command Line Interpreter) provided with Data General's AOS/VS Operating System, as a Programming Language. Its name is **Hook-Bang-Crunch** because this is **DG-ese** for the three characters **?**, **!** and **#**, which occur commonly in CLI macros, and after all **? ! #** is what complex CLI macros are to most people. Hopefully it will help to remove some of the mystery for you.

The author has had some eight years experience with Data General's AOS/VS Operating System, firstly as a training instructor with Data General Australia and subsequently as a Consultant in a variety of roles including Application Development, Systems Programming and Standards Manual preparation. He brings much of his depth of experience and skill as a CLI macro writer to you in this book. It is liberally spiced with meaningful and practical example macros, all of which can be separately purchased on Mag-Tape.

This book can be read and understood by Computer Operators who have not yet tried to write a macro, and yet it covers sufficient ground to teach hot-shot Data General SE's a thing or two as well. Special features include techniques for writing your own Pseudo-Macros, writing macros without the use of more than one STRING variable, correct use of PUSH, POP and PROMPT POP, and how to avoid "CLI out of memory" errors. In addition there are special appendices on DG's two other hidden programming languages, namely the text editor **SPEED** and the **SORT/MERGE** utility.

It is our firm belief that this book will become a necessary tool-of-trade for all users of Data General's AOS/VS Command Line Interpreter.

ISBN 0 7316 6442 6



Shalless Software Pty Ltd