New Features of the MPE XL User Interface

by Thomas Shem & Jeff Vance
Hewlett-Packard Company
19447 Pruneridge Avenue
Cupertino CA, 95014

Introduction

The User Interface to the MPE operating system is the sum of the components which allow users to direct the HP 3000 in its execution of given tasks. The MPE V User Interface, consisting of commands, job control words (JCWs) and intrinsics, provides the user with the basic mechanisms to accomplish jobs. The User Interface to MPE V, while functional in nature, can hardly be considered "user friendly". Although some operations can be accomplished in a straightforward manner, the user must frequently rely on programs to perform tasks.

Much of this has changed with the MPE XL User Interface. This new User Interface while designed to be compatible with MPE V, was also designed to provide a powerful, flexible, and productive environment for the general user. The experienced user will find many new features which expose simple straightforward solutions to previously tedious and complex problems. Tasks, which on MPE V, required programs to solve, may now be accomplished though the new User Interface.

This paper describes the new MPE XL User Interface, focusing on the extensions beyond MPE V/E. The command language aspect of the User Interface is emphasized, and examples are provided to illustrate some of the simple straightforward solutions previously thought tedious and complex on MPE V.

The Command Interpreter

The Command Interpreter (CI) is central to the user interface. It is the mechanism whereby users of the HP 3000 access system functionality. An examination of the MPE V CI and the MPE XL CI will show how the user environment has been enhanced. It will also show how those enhancements create an environment which assists user.

The MPE V CI is a line oriented interface through which user commands are routed to the appropriate command executor (see figure 1). The status of the command string routing and subsequent command execution are indicated through JCWs (job control words).

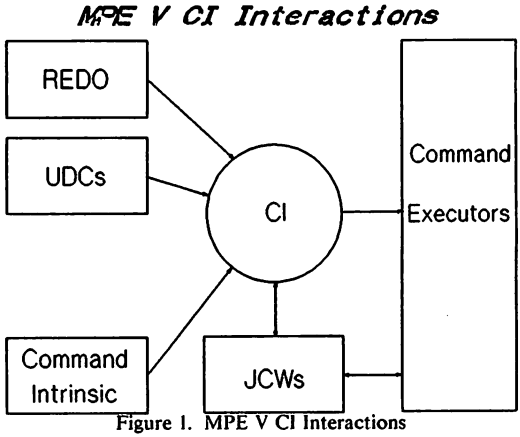

Figure 1. MPE V CI Interactions

User command strings may be fed to the CI through a variety of paths (see figure 1). Command strings can be input directly from a job or session, reissued or modified via the REDO command, invoked from a predefined UDC (user defined command), or issued from a user program or subsystems via the COMMAND intrinsic.

The MPE V Command Interpreter performs three major functions (see figure 2). Its first function is to interpret the input command string. The CI analyzes the command string for valid commands (either MPE commands or user defined commands). Another function of the CI is to determine if the user has the proper attributes to use the command. The CI compares the user capabilities against those assigned to the command. Lastly, the CI invokes the appropriate command executor. The executor completes the parsing of the command string, then performs the desired function.

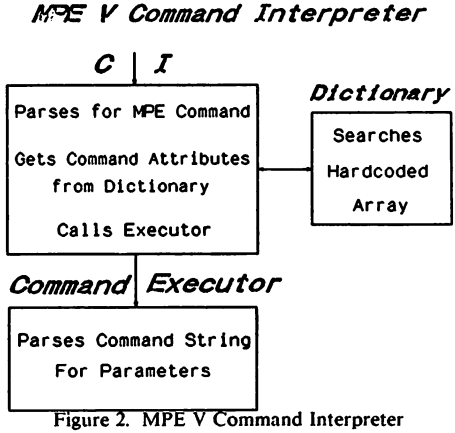

Figure 2. MPE V Command Interpreter

The MPE XL Command Interpreter performs the same basic functions as that of the MPE V Command Interpreter, since it was designed to be externally similar. The MPE XL CI has three major differences from the MPE V CI: it is implemented as an executable program instead of as a system process; the interfaces to the XL CI have been expanded and improved; and the structure of the CI has changed.

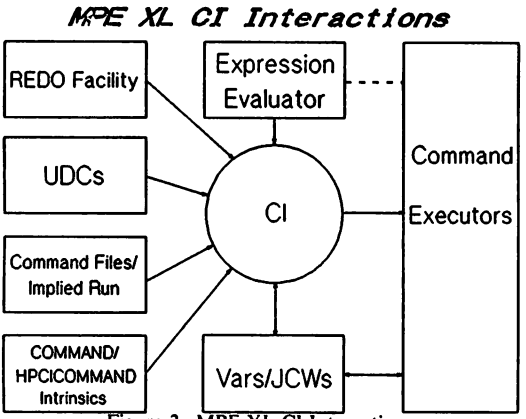

Figure 3. MPE XL CI Interactions

The MPE XL CI interacts with the same components as those on an MPE V system (see figure 3). However, these components have been specialized to facilitate usability. Command strings are still fed to the CI through a variety of paths. They can be input directly from a job or session, reissued or modified via the new REDO facility, invoked from predefined UDCs or command files (discussed later) or issued from the COMMAND or HPCICOMMAND intrinsics.

The XL CI has been restructured (see figure 4). It now consists of two parts; a centralized scanner/parser part and a command interpreter part. Command strings are first checked for correct syntax in the scanner/parser. The correct command executor is then invoked if the syntax is valid.

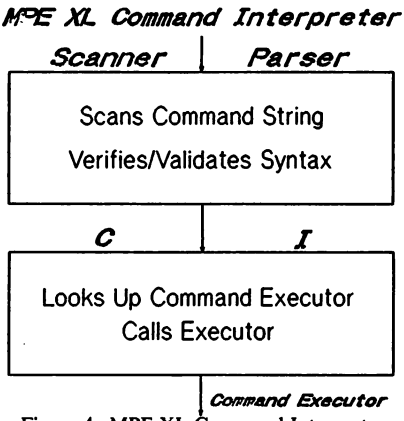

Figure 4. MPE XL Command Interpreter

The new structure of the MPE XL CI separates and specializes the parts of the system which deal with user interactions. This specialization gives the XL User Interface the ability to become a powerful tool which can be used when directing the system in a given task.

The User Environment

As can be seen from the two CI interaction diagrams (figures 1 and 3), many of the mechanisms existing in MPE V have been enhanced. These enhancements serve to make the XL CI more powerful, more flexible and much easier to use than its predecessor. A comparison of the differences in these features will show how they benefit the user, and serve to show how they interact to form a customizable user environment.

JCWs vs. Variables

Local to the user's sessions and jobs on MPE V are job control words. These words are temporary numeric indicators which provide statuses to the user whenever a command is executed (i.e. CIERROR). Additionally, users may use them to indicate the status of a program's execution. MPE V also provides several pre-assigned JCWs to indicate system information (HPMONTH, HPDAY, HPYEAR, etc.). JCWs are essential for controlling the flow of command execution within large batch jobs or complex User Defined Commands (UDCs).

In MPE XL, the idea of providing users with temporary variables has been expanded to include data types other than numeric. Variables can retain data in boolean form, 32-bit numeric form, string form, and JCW format.

These new variable types have also made it possible for MPE XL to provide more complete system information. The data in these predefined variables range from system global information (i.e. system time, system date, job count, and jobfence) to job or session system infomation (i.e. logon ID, capability lists, $STDIN ldev, $STDLIST ldev, interactive state, and CPU time used). Additionally, some variables are used to control the user environment. Modifying these variables

changes aspects of the session environment. For example, when the variable HPAUTOCONT is set to TRUE, the effect is the same as if a CONTINUE statement preceded every command.

To support the variables, three new commands are provided: the SHOWVAR command displays variable and JCW information information; the SETVAR command assigns values to variables; and the DELETEVAR command removes variables. The SETJCW and SHOWJCW commands can be thought of as a subset of the variable commands.

The value of a variable is referenced by preceding the variable name with a "!" (e.g. !HPACCOUNT). The referenced variable is then replaced with its value (e.g. !HPACCOUNT is replaced by SYS if the user is logged into the SYS account). This is called dereferencing the variable (specifically, explicit dereferencing).

Variables may be dereferenced in two additional ways, implicit and recursive. Implicit dereferencing is simply the substitution of a variable's value for its name.

An example of implicit dereferencing occurs in the statement:

```
IF VAR = 0 THEN ...
```

In the above example the variable name (VAR) is replaced by its value without requiring a "!".

Recursive dereferencing of variables occurs when a variable contains the name of another variable. The following statement personalizes the CI's prompt to contain your username followed by "(current system time):", e.g "TOM (12:01):". The predefined variables HPUSER, HPHOUR, and HPMINUTE contain the username, the current system hour, and the current system minute.

```
:setvar hpprompt "!hpuser (!!hphour:!!hpminute):"
```

The expression to be assigned to the global variable is evaluated in the following manner:

- the quotes cause the variable's type to be a string

- !hpuser is explicitly dereferenced as the user ID

- the " (" is evaluated as a string literal

- the doubled exclamation points ("!!") have a special meaning and are used to represent a single exclamation point (i.e. they are folded into one; however, in general an even number of exclamation points do not cause dereferencing while an odd number of exclamation points do cause dereferencing - pairs are then folded) and are evaluated as a string containing a "!".

- the "hphour:" is evaluated as a string (combining with the previous exclamation point to form a dynamic variable; a variable within a variable)

- the doubled exclamation points are again folded into a single exclamation point, then evaluated as a string

- the "hpminute):" is evaluated as a string.

The string stored into HPPROMPT then looks like this:

```
TOM (!hphour:!hpminute):
```

When the variable HPPROMPT is referenced (it is referenced by the CI before displaying the prompt), recursive dereferencing occurs on the !hphour and !hpminute to obtain the current values stored in the system. As can be seen in the example, recursive dereferencing becomes very handy when the value of a dynamic variable is needed.

REDO vs The REDO Facility

One of the most useful commands available to the user of MPE V is the REDO command. Users do all sorts of amazing things with this command: re-execute their last command; correct errors in their last command with the editing features of the command; capture a previously typed command with the enter key, edit the captured command string then re-execute it; and capture some data (such as the capability list from the ALTUSER cierror message), add a valid command to the list then execute it. The main benefit of this command is that it saves the user from excess typing. The main drawback is that the user is limited to the last command entered.

On MPE XL, this command has grown into a facility of its own. The REDO facility now encompasses a command history stack, two new commands and enhanced editing features. The REDO buffer has been replaced by a command history stack. Whereas in MPE V only one command was able to be saved, the history stack allows users to save up to 1000 previous commands (although the default is 20). The history stack was designed to be user configurable, so users can control the depth of their own command history stack through the variable HPREDOSIZE. The history stack is supported by the new LISTREDO and DO commands and also the enhanced REDO command.

The LISTREDO command gives the user the option of defining the range of history stack to be displayed, and the option of referencing previous commands in one of three manners. Commands may be listed either numbered relative to the top command in the stack, or numbered relative to the first command entered during the session, or unnumbered.

For example:

```
:LISTREDO ;ABS          :LISTREDO ;REL          :LISTREDO ;UNN
     1) commandone           -3) commandone          commandone
     2) commandtwo           -2) commandtwo          commandtwo
     3) LISTREDO ;ABS        -1) LISTREDO ;REL       LISTREDO ;UNN
```

Re-execution of commands from the history stack is accomplished with the DO and REDO commands. The two commands are identical in nature, except that the REDO command allows interactive editing. Previous commands may be executed by using a relative or absolute command number.

Additionally, prior commands may be retrieved by dereferencing an absolute or relative history stack command number (e.g. !1 or !-2). The absolute command number must be within the range available on the current history stack. The predefined variables HPREDOSIZE and HPCMDNUM are provided to show the current redo stack size and current absolute command number.

Editing of previous commands may be accomplished by appending an edit string to the DO or REDO commands, or by using the interactive method, similar to that available on MPE V, with the REDO command. The basic editing directives available on MPE V are all available: i (insert), r (replace), d (delete), and u (undo) Several new directives have also been added: d> (delete to end

of line), > (append to end of line), >d (delete from the end of line), >r (replace at the end of line), and c (change one string for another).

For example:

```
:DO ;EDIT=">dddd"
```

will result in the following, given any of the three previous :LISTREDO examples:

```
:LISTREDO                :LISTREDO                :LISTREDO
     1) commandone            1) commandone            1) commandone
     2) commandtwo            2) commandtwo            2) commandtwo
     3) LISTREDO ;ABS         3) LISTREDO ;REL         3) LISTREDO ;UNN
     4) LISTREDO              4) LISTREDO              4) LISTREDO
```

UDCs vs. Command Files

Whereas REDO is one of the most useful commands on MPE V, UDCs (User Defined Commands) are one of the most used feature. UDCs allow users to build a set of personalized commands since they are executed before MPE commands. They provide users with the ability to override or supersede MPE commands. UDCs offer a method for simplifying MPE commands. A sequence of commonly used commands can be bundled into one user command, simplifying user invocation and execution of tasks. Automatic invocation of a UDC at logon time can set up the user environment or be used to restrict users to a particular environment. Perhaps the most useful aspect of UDCs is that they provide a mechanism to avoid typing complex instructions.

With all of these useful functions, there wasn't much for MPE XL to improve upon. However, MPE XL has added several new features to UDC control and maintenance. Two new options have been added which control whether UDCs may recursively call themselves (OPTION RECURSION/ NORECURSION), and whether UDCs may be executed programmatically (OPTION PROGRAM/NOPROGRAM; more on this later). Maintenance of UDCs has been simplified via the new ;APPEND, and ;DELETE parameters to the SETCATALOG command. UDC files may be appended or deleted without having to uncatalog a session's current UDCs.

In MPE XL, an additional method for users to define their own set of customized commands has been provided through command files. Command files are simply files which contain one or more commands, much like UDCs. Commands within a command file may be MPE commands, UDCs, or filenames. They are similar to UDCs in all respects except for three differences:

- Command files are searched for after UDCs and MPE commands.

- Command files don't have the requirement of having to be cataloged.

- Command files do not support the control of recursion that UDCs provide via the RECURSION option, or the control for logon invocation via the LOGON option. In the case of recursion, the command file name need only be specified within the same command file in order to invoke recursion (UDCs require that the RECURSION option be specified). In the case of logon invocation, command files can not be invoked automatically at logon (except via a logon UDC).

Command files, like UDCs, may accept parameters by defining them in the header line and may use options. Invocation of command files, unlike UDCs, is executed by entering the filename.

Implied Run and Search Path

Program files like command files may now be invoked by entering the program filename. This is referred to as "implied run". The RUN command need no longer be specified to execute a program file in most cases. The optional INFO and PARM parameters are supported through the implied run. Use of any of the other optional parameters when invoking a program file will require that the RUN command be specified.

To enhance the user environment and to support invocation of command files and program files, a predefined variable HPPATH is provided to allow users to define a search path for the CI. The CI will follow the search path when a command has not matched a UDC or MPE command. The HPPATH variable virtually eliminates the need to explicitly specify a group or account with the command or program file name.

For example, suppose the HPPATH variable is set to "pub.sys,pub,!HPHGROUP, !HPGROUP" and a non-UDC/non-MPE command is entered. The CI will use the search path specified in the HPPATH variable to attempt to find a command or program file to execute. The CI will first look in the "pub.sys" group/account, then the "pub" group of the current logon account, then the user's home group, and finally the user's current logon group. Failing to find a command or program file at this point will result in an "UNKNOWN COMMAND NAME" message.

CI Flow Control Structures

The control of the sequence for execution of command statements in MPE V consists of the simple branching mechanism provided by the IF .. THEN .. ELSE .. ENDIF commands. As every programmer knows, branching mechanisms are adequate to get the job done, but just barely.

MPE XL expands upon this simple control structure by introducing a looping structure and a recursion structure. The WHILE .. DO .. ENDWHILE commands provide the users with a much needed control structure to repeat tasks. Another structure to accomplish looping as well as recursive executions is now supported through UDCs and command files. UDCs may invoke themselves if the RECURSION option is specified. Command files may also invoke recursion by using the filename within the command script. A predefined variable, HPUSERCMDEPTH, is provided to indicate the depth of nested UDCs and/or command files.

Additionally, a new RETURN command has beed provided to cause the execution of a User Command (UDC or command file) to return to the calling environment. User Commands invoked from the CI will return to the CI. Nested User Commands will return to calling UDC or command file.

Whereas in the past the command structure provided by MPE V did not allow users to get into complicated situations, the command structures provided by MPE XL can easily get the inexperienced user into a lot of trouble. Care should be used to prevent endless looping. Break will usually end an endless loop.

Expression Evaluator

In MPE V expressions are used in the IF .. THEN .. ELSE .. ENDIF control structure to indicate the truth or falsehood of the control structure. Expressions on MPE V consisted of the comparison of a JCW to another JCW, or a JCW to a fixed numeric value.

On MPE XL, evaluation of expressions has been separated into a new facility referred to as the expression evaluator, which provides the user with a rich set of features.

The expression evaluator supports a large set of functions. Many arithmetic operations are allowed, including: absolute value (ABS), modulo (MOD) and exponentiation (^). Many string operations are defined, such as: concatenation (+), length (LEN), ordinal (ORD), extraction (LFT, RHT, POS, STR), and case shifting (DWNS, UPS). Special variable functions include: existence (BOUND) and type (TYPEOF). Bit operations include: bitwise and (BAND), bitwise or (BOR), bitwise not (BNOT), bitwise exclusive or (BXOR), shift left (LSL), and shift right (LSR). Numeric conversion functions are: convert to octal (OCTAL) and convert to hexadecimal (HEX). Finally, some special functions provided via FINFO include: file existence (FINFO(0)), file creation date (FINFO(6)), file modification date (FINFO(8)), file code (FINFO(9)), foptions (FINFO(13)) and others.

Expressions may be implicit or explicit. Implicit expressions are only available in in four commands: CALC, SETVAR, IF, and WHILE. Explicit expressions may be dereferenced in any command by enclosing the expression within square brackets and preceding it with an exclamation mark (e.g. ![expression], referred to as "expression substitution"). For example, ![“li"+"stf"] will cause the LISTF command to be executed.

Mixed expressions are not allowed, for example, the expression "a" + 1 would result in an error. Standard precedence rules apply, with explicit variable dereferencing superseding all other operations.

Miscellaneous New Commands

Several new commands have also been provided to enhance the new User Interface. These include the CALC, CHGROUP, COPY, ECHO, INPUT, PRINT, and XEQ commands. A complete description of each of these commands can be found in the MPE XL Commands Reference Manual (Part Number 32650-90003); however, specific usage of some of these commands is described below:

- the CALC command provides easy access to the Expression Evaluator for quick evaluation of expressions.

- the CHGROUP command gives users an easy way to switch from one group to another, without logging off.

- the COPY command provides for simple fast file copies even in break mode.

- the ECHO command displays text to $STDLIST (this is particularly useful in conjunction with expression substitution, "![ ]" , and explicit variable dereferencing).

- the INPUT command can be used to prompt a user for data, and is a convienent way to load a string variable. Also, note the timed read feature.

- the PRINT command provides a quick way to display a file.

- the XEQ command will execute a command or program file, even if the file contains the same name as a UDC or MPE command.

COMMAND vs HPCICOMMAND

Programmatic invocation of MPE commands on MPE V is accomplished via the COMMAND intrinsic. Unfortunately, the COMMAND intrinsic does not perform all the functions available through the MPE V CI. On MPE XL, the introduction of the new HPCICOMMAND intrinsic provides users with a programmatic method for accessing most of the MPE XL CI functions, as well as command files, UDC invocation, and implied run.

The Command Language

The combination of commands, variables, expression evaluations, control structures, UDCs, and command files all come together to form a command language. With MPE V, end-users needed system programmers to build complex programs to assist in the operation of their HP3000. Now, non-programmer end-users, as well as experienced users and programmers, will be able to control their system without writing a program. The "language" of MPE XL exposes the power of the HP3000 at the level of the command interpreter instead of hiding it.

To illustrate the power available through the MPE XL command language, several examples are provided. These examples show how common operations which are tedious or cumbersome can be made simpler.

1. "CENTER" - this command file is an example of some of the string manipulation functions supported by the expression evaluator. It will echo to $STDLIST a centered string. It is used later in the "CALCIT" example.

```
Parm string
COMMENT
COMMENT +-------------------------------------------------+
COMMENT | CENTER                                          |
COMMENT | Note: cent_spc is loaded with a blank string    |
COMMENT +-------------------------------------------------+
COMMENT
setvar cent_spc "                                                "
echo ![lft(cent_spc,(80-len("!string"))/2)]!string
deletevar cent_spc
comment end of center

Usage:
:center 'Center this string'
                              Center this string
:
```

2. "FKEY" - this command file is an example of how escape functions to a terminal can be done. It will set up one function key.

```
PARM KEY,L1=" ",S1="UD",A1=2
COMMENT
COMMENT +-------------------------------------------------+
COMMENT | FKEY                                            |
COMMENT | Sets A SINGLE function key in one invocation.   |
COMMENT | The "L" parameter is the label, the "S" parameter |
COMMENT | is the string to be generated when the function |
COMMENT | key is pressed, and the "A" parameter is the    |
COMMENT | key attribute parameter, where 0=Normal, 1=Local, |
```

```
COMMENT | and 2=Transmit attributes for each key. The "KEY" |
COMMENT | is, of course, the key to set.                    |
COMMENT +-------------------------------------------------+
COMMENT
IF (HPJOBTYPE="S") AND (HPDUPLICATIVE=TRUE) THEN
  setvar esc chr(27)
    SETVAR vars "!S1"
    SETVAR varl "!L1"
    SETVAR lens LEN(vars)
    SETVAR lenl LEN(varl)
    SETVAR wkey "!esc"+"&f!a1"+"a!key"+"k"+&
      "!lenl"+"d"+"!lens"+&
      "L"+"!varl"+"!vars"
  echo !wkey
  ECHO !esc&jB
  DELETEVAR esc,vars,varl,lens,lenl,wkey
ENDIF
COMMENT end of fkey

Usage:
:fkey 1,test,'echo this is a test'
         now using the f1 key will perform the following:
:echo this is a test
this is a test
```

3. "TRIM" - this command file will trim all characters of a specified type from a variable name. It is an example of how string manipulation can be done.

```
parm varname,trimchar=" ",from=RIGHT
COMMENT
COMMENT +-------------------------------------------------+
COMMENT | TRIM                                            |
COMMENT | Trims all trimchar from varname, starting at from.|
COMMENT +-------------------------------------------------+
COMMENT
if not (bound(!varname)) then
   echo (TRIM): The variable !varname is not defined.
else
   if not(ups(lft('!from',1)) = 'L') then
      setvar trim_off 'RHT'
      setvar trim_save 'LFT'
   else
      setvar trim_off 'LFT'
      setvar trim_save 'RHT'
   endif
   while (len(!varname)>0) and (!trim_off(!varname,1)='!trimchar')
      setvar !varname !trim_save(!varname,len(!varname)-1)
      endwhile
   deletevar trim_off,trim_save
endif
comment end of trim
```

```
Usage:
:setvar string 'this string needs the question mark stripped off?????'
:trim string,?
:showvar string
STRING = this string needs the question mark stripped off
```

4. "ADDCAP" - this command file provides a simple method for adding capabilities to a "user"s existing capability list. Note the re-logging on option at the end of the command file.

```
parm cap=''
COMMENT
COMMENT +-------------------------------------------------------------------+
COMMENT |ADDCAP                                                             |
COMMENT |Adds a new capability to the "user"s existing capabilities. AM     |
COMMENT |is required to execute the :ALTUSER command. The new capability    |
COMMENT |is only available after re-logging on, which the user will be      |
COMMENT |prompted for.                                                      |
COMMENT +-------------------------------------------------------------------+
COMMENT
if ('!cap' = '') then
   echo (ADDCAP): Your capabilities are: !hpusercapf.
   return
endif
if (pos('![ups('!cap')]','!hpusercapf') <> 0) then
   echo (ADDCAP): You already have     : !cap.
   echo (ADDCAP): The capabilities are : !hpusercapf.
   return
endif
setvar addcap_temp "!hpusercapf,!cap"
setvar cierror 0
continue
altuser !hpuser;cap=!addcap_temp
if cierror <> 0 then
   echo (ADDCAP): The capabilities remain: !hpusercapf.
else
   setvar addcap_temp,ups(addcap_temp)
   echo (ADDCAP): !hpuser new capabilities are: !addcap_temp.
   setvar addcap_temp 'N'
   input addcap_temp,'(ADDCAP): Log off/on now (Y/N) ==>',10
   if cierror = -9003 then
      comment: Timed read expired.
      echo
      echo (ADDCAP): Timed 10-second read expired. Logon cancelled.
   else
      if not(lft(ups(addcap_temp),1) = 'Y') then
         echo (ADDCAP): New capabilities take effect at next logon.
      else
         hello !hpjobname,!hpuser.!hpaccount,!hpgroup
      endif
   endif
endif
```

```
deletevar addcap_temp
COMMENT end of addcap

Usage:
:listuser foo
**********
USER: FOO.UI

HOMEGROUP:                    PASSWORD: **
MAXPRI   : 150                LOC ATTR: $00000000
LOGON CNT: 0
CAP: AM,BA,IA

:addcap
(ADDCAP): Your capabilities are: AM,BA,IA

:addcap am
(ADDCAP): You already have     : AM
(ADDCAP): The capabilities are : AM,BA,IA

:addcap ds
(ADDCAP): FOO new capabilities are: AM,BA,IA,DS
(ADDCAP): Log off/on now (Y/N) ==>n
(ADDCAP): New capabilities take effect at next logon.
```

5. "FINFO" - this command file will display the file label information for a given file. Note the use of finfo in the IF..THEN control structure.

```
PARM file
COMMENT
COMMENT +-----------------------------------+
COMMENT | FINFO                             |
COMMENT | Use finfo to show file label info. |
COMMENT +-----------------------------------+
COMMENT
if not(finfo('!file',0)) then
   comment  File does not exist.
   if lft('!file',1) <> '*' and lft('!file',1) <> '$' then
      comment  Qualify file before reporting non-existence.
      if pos('.','!file') > 0 then
         if pos('.',rht('!file',len('!file')-pos('.','!file'))) > 0 then
            echo ![ups('!file')] does not exist.
         else
            echo ![ups('!file')].!hpaccount does not exist.
         endif
      else
         echo ![ups('!file')].!hpgroup.!hpaccount does not exist.
      endif
   else
      echo !file does not exist.
   endif
else
```

```
   comment ** formal file designator **
   echo (FINFO): Full file description for ![finfo('!file',1)] follows:
   comment ** creator and create/modify dates **
   echo    Created by ![finfo('!file',4)] on ![finfo('!file',6)].
   echo    Modified on ![finfo('!file',8)] at ![finfo('!file',24)].
   comment ** file code **
   if finfo('!file',9) = '' then
      echo    Fcode: ![finfo('!file',-9)].
   else
      echo    Fcode: ![finfo('!file',9)] (![finfo('!file',-9)]).
   endif
   comment ** rec size, eof, flimit **
   echo    Recsize: ![finfo('!file',14)], Eof: ![finfo('!file',19)], &
Flimit:![finfo('!file',12)].
   comment ** foptions **
   setvar _fopt finfo('!file',-13)
   echo    Foptions: ![finfo('!file',13)] (#!_fopt, ![octal(_fopt)],&
![hex(_fopt)]).
   deletevar _fopt
endif
COMMENT  End of finfo.

Usage:
:finfo a
(FINFO): Full description for A.GROUP.ACCT follows:
   Created by TOM on SUN, MAY 29, 1988.
   Modified on SUN, MAY 29, 1988 at 3:21 PM.
   Fcode: 0.
   Recsize: -80, Eof: 5, Flimit:5.
   Foptions: ASCII, FIXED, NOCCTL, STD (#5, %5,$5).
```

6. "PURGESP" - this command file will purge multiple spool files. Note the multiple entry points and the use of the Command Interpreter, CI.PUB.SYS.

```
parm user="@",grp="pub",entry_point="purgesp",command=""

COMMENT
COMMENT +-------------------------------------------------------------+
COMMENT | PURGESP                                                     |
COMMENT | This file will purge all spool files which belong to the    |
COMMENT | specified user (ie - @, @.@, mgr.test, etc.).               |
COMMENT | Only spool files in the READY state are purged.             |
COMMENT +-------------------------------------------------------------+
COMMENT

COMMENT << this entry point is the main body >>
if "!entry_point" = "purgesp" then
  COMMENT <<cleanup temporary files from a previous run of this file>>
   if FINFO ("spin.!grp", 0) then
      purge spin.!grp
   endif
```

```
if FINFO ("spin2.!grp", 0) then
   purge spin2.!grp,temp
endif

if FINFO ("spout.!grp", 0) then
   purge spout.!grp
endif

if FINFO ("purgein.!grp", 0) then
   purge purgein.!grp
endif

if FINFO ("purgein2.!grp", 0) then
   purge purgein2.!grp,temp
endif

COMMENT << build temporary files >>
build spin.!grp;rec=40,,f,ascii;disc=10000
build spout.!grp;rec=40,,f,ascii;disc=10000
build purgein.!grp;rec=40,,f,ascii;disc=10000
file spin2.!grp;rec=40,,f,ascii;disc=10000
file purgein2.!grp;rec=40,,f,ascii;disc=10000

COMMENT << generate the input file for spook >>
run ci.pub.sys;info="purgesp ,,'getspin','s !user'" &
              ;stdlist=spin.!grp;parm=3

COMMENT << extract only the lines that contain spook commands >>
print spin.!grp,*spin2.!grp;start=3

COMMENT << get a list of the spool files to purge >>
run spook.pub.sys;stdin=spin2.!grp &
                 ;stdlist=spout.!grp

COMMENT << convert the list of files to the format "p o####" >>
run ci.pub.sys;info="purgesp ,,getxddno" &
              ;stdin=spout.!grp &
              ;stdlist=purgein.!grp &
              ;parm=3

COMMENT << extract only the lines that contain xdd numbers >>
print purgein.!grp,*purgein2.!grp;start=3

COMMENT << purge the selected spool files >>
run spook.pub.sys;stdin=purgein2.!grp

COMMENT << clean up >>
purge spin.!grp
purge spin2.!grp,temp
purge spout.!grp
purge purgein.!grp
purge purgein2.!grp,temp
reset spin2.!grp
```

```
   reset purgein2.!grp
else

  COMMENT << this entry point creates the file 'spin' >>
  if "!entry_point" = "getspin" then
     echo !command
     echo exit
  else

    COMMENT << this entry point converts a spook-generated list of >>
    COMMENT << spool files to a list of just xdd numbers preceded  >>
    COMMENT << by a 'p' to tell spook to purge the file            >>
    if "!entry_point" = "getxddno" then
       setvar hpmsgfence 2
       setvar eof false
       setjcw cierror 0

       while not eof
          continue
          input line

          if cierror <> 900 then
             if LFT (line, 2) = "#O" then
                if POS ('READY', line) > 0 then
                   echo p ![STR (line, 3, 6)]
                endif
             endif
          else
             setvar eof true
          endif
       endwhile

       echo exit
       deletevar eof
       deletevar line
       setvar hpmsgfence 0
    endif
  endif
endif
COMMENT end of purgesp

Usage:
:spook5
SPOOKHPE A.02.50 (C) HEWLETT-PACKARD CO., 1985
>s
#FILE   #JOB    FNAME    STATE  OWNER
#O1777  #J758   $STDLIST READY  TOM.UI
#O1778  #J759   $STDLIST READY  TOM.UI
#O1779  #J760   $STDLIST READY  TOM.UI
#O1780  #J761   $STDLIST READY  TOM.UI
#O1781  #J762   $STDLIST READY  TOM.UI
#O1782  #J763   $STDLIST READY  TOM.UI
>e
```

```
END OF PROGRAM
:purgesp tom


END OF PROGRAM


END OF PROGRAM


END OF PROGRAM

SPOOK A.10.10  (C) HEWLETT-PACKARD CO., 1983
> #FILE   #JOB    DEV/CL   SECTORS      OWNER
#O1777  #758    LP            36      TOM.UI
> #FILE   #JOB    DEV/CL   SECTORS      OWNER
#O1778  #759    LP            36      TOM.UI
> #FILE   #JOB    DEV/CL   SECTORS      OWNER
#O1779  #760    LP            36      TOM.UI
> #FILE   #JOB    DEV/CL   SECTORS      OWNER
#O1780  #761    LP            36      TOM.UI
> #FILE   #JOB    DEV/CL   SECTORS      OWNER
#O1781  #762    LP            36      TOM.UI
> #FILE   #JOB    DEV/CL   SECTORS      OWNER
#O1782  #763    LP            36      TOM.UI
>
END OF PROGRAM
:
```

7. "CALCIT" - this command file will allow a user to interactively use the expression evaluator. Note the high usage of screen control escape characters.

```
PARM enh_ch=D
COMMENT
COMMENT +-----------------------------------------------+
COMMENT | CALCIT                                        |
COMMENT | Interactive calculator using :calc and :input. |
COMMENT +-----------------------------------------------+
COMMENT
echo ![chr(27)+'h'+chr(27)+'J']
center 'MPE XL Interactive Calculator'
center ': executes any MPE command!!!'
center 'Type EXIT or [RETURN] to exit'
echo
setvar calcit_esc chr(27)+'C'
setvar calcit_expr 1
while calcit_expr < 7
  setvar calcit_esc calcit_esc+calcit_esc
  setvar calcit_expr calcit_expr + 1
endwhile
setvar calcit_esc chr(27)+'A'+calcit_esc
setvar calcit_prompt lft(ups("!-1"),pos(' ',"!-1"+' ')-1)+' ==> '
```

```
setvar calcit_expr 'Hello'
while (not ups(calcit_expr) = 'EXIT') and (not calcit_expr = '')
  setvar calcit_expr ''
  input calcit_expr,"!calcit_prompt"
  setvar calcit_len len(calcit_expr)
  while (len(calcit_expr) > 0) AND (lft(calcit_expr,1) = chr(32))
    setvar calcit_expr rht(calcit_expr,len(calcit_expr) - 1)
  endwhile
  if (not ups(calcit_expr) = 'EXIT') and (not calcit_expr = '') then
  if lft(calcit_expr,1) = ':' then
    continue
    ![rht(calcit_expr,len(calcit_expr)-1)]
  else
    setvar hpmsgfence 2
    setvar cierror 0
    continue
    setvar hpresult !calcit_expr
    if not cierror = 0 then
      setvar hpmsgfence 0
      continue
     calc !calcit_expr
    else
     echo ![lft(calcit_esc,2+(len(calcit_prompt)+calcit_len)*2)+&
            ' = '+chr(27)+'&d!enh_ch']!hpresult
    endif
  setvar hpmsgfence 0
  endif
  endif
endwhile
deletevar calcit_@
comment end of calcit

Usage:
:calcit
                        MPE XL Interactive Calculator
                        : executes any MPE Command!!!
                        Type EXIT or [RETURN] to exit

CALCIT ==> :listf a

FILENAME

A

CALCIT ==> 5+7
CALCIT ==> 5+7 = 12
CALCIT ==> exit
:
```