MPE/XL Variables and Command Files

Brett Clemons
Softwarewizardry, Inc.
Tampa, FL. 33615

Abstract. With the introduction of Hewlett-Packard's Precision Architecture came a new command interpreter - MPE/XL. One of the most powerful new enhancements to the basic MPE command interpreter is a new type of file - the command file. Command files can be used in place of, or in conjunction with, UDC's to produce a versatile tool at the disposal of the veteran or novice programmer alike. The most exciting aspect is the introduction of variables, or 'vars' to allow you the ability to customize your command interpreter to perform very sophisticated tasks with a single keystroke. Examples include compiles of wildcard filesets, file purges or compressions, as well as analysis of files in a wildcard fileset. Variables, as well as a few new MPE/XL commands, allow the programmer to place control of the command interpreter where it belongs - with the user.

I. Introduction.

The evolution of the MPE command interpreter has been slow and gradual. Existing commands were enhanced, but no major change occurred in the basic 'core' of the command interpreter until recently. With the advent of MPE/XL Precision Architecture command interpreter, something new has evolved from the old command interpreter. For the purposes of this paper, MPE/XL shall be termed a command interpreter, which more technically expresses what it is.

Hewlett-Packard's new approach to command interpreters has breathed new life into the old MPE command interpreter, bringing with it something which allows for the command interpreter itself to be used as an extension the programming languages, or as a programming language itself. Many of the elements found in fundamental programming languages are present in the new MPE/XL command interpreter. For example, variables allow you to store temporary information in any of the formats that may be found in a programming language. In addition, testing of variables in Boolean expressions allow for a much expanded analysis of conditions that may occur external to the programs. Also, several commands have been added to allow recursive performance of a block of commands. Previously, a set of commands that you wanted to execute could only be stored in a file, and :SETCATALOGed. This file contained User Defined Commands or UDCs. Command files, which are meant to expand upon UDC's, and not replace them, allow the execution of a block of commands - and command files

are easy to set up and maintain.

This paper shall also investigate more advanced topics using command files and variables, as well as some new MPE/XL commands that add flavor to command files.

II. Variables - the new JCWs

Under the MPE command interpreter, the only way of storing temporary information external to a program was in a file, or with Job Control Words (JCWs). JCWs are not eliminated with MPE/XL, but are enhanced with a new form of storage called variables.

Variables names identify the variable referenced in the commands. Variable names start with an alphabetic character or underscore (_) character, and contain from one to 255 characters. Of course, variable names must be unique.

There are three several different types of variables. Variable types are defined with the use of the new MPE/XL command SETVAR. SETVAR allows the definition of a variable name explicitly with the SETVAR command; the variable type is implicit with the initial value assigned to the variable. SETVAR is also used to redefine the value of an existing variable. The basic type of variables are Boolean, Integer and Strings. The type of a variable is set at time of definition, depending upon the information to be stored in the variable. For example, to define a boolean variable, use the following command construct :

```
                          TRUE
:SETVAR varname,(         )
                          FALSE
```

The second type of variable, integer variables, allow you to define variables containing numeric values, and are defined at definition time with a

```
:SETVAR varname,integer-value.
```

The third type of variable, string variables, are defined with a

```
:SETVAR varname, 'string-value '.
```

String variables may contain any valid string, from none to 256 valid alphanumeric characters.

Variables are removed with the DELETEVAR command. When a variable is defined it retains its characteristics initially defined with the SETVAR command until the DELETEVAR command

is used to remove the variable from the user's job or session. Its form is

```
:DELETEVAR varset
```

and as shown in the syntax, may contain wildcards (@, #,etc) to specify a set of variables to be deleted.

The final variable command is the SHOWVAR command. SHOWVAR lists variables to $STDLIST, and uses wildcards in the same way as the LISTF command does. This command has the format

```
:SHOWVAR [varset]
```

If the varset parameter is omitted, only user-defined variables are displayed; if @ is used for varset, then all variables will be shown. Other wildcards may be used to list any subset of variables. For example,

```
SAVE_@
HP@
SAVE_VAR_#@
```

are all valid subsets that may be used to represent a one or more variables that are defined.

Variables can be used in a variety of ways, most usefully in command files, but they are not excluded from use in UDCs or jobs.

III. Using Variables

When MPE/XL parses a command line, the Expression Evaluator (a part of MPE/XL) first looks for variables in the command line. The process of substituting a variable's value in a command line is called dereferencing; dereferencing takes precedence over all other operations in an MPE/XL command line, including the recognition of the command name itself! There are two methods of dereferencing variables. The first, implicit dereferencing, is where the variable name is placed in the command line and MPE/XL substitutes the value of the variable before parsing the command line. For example, when the following commands are entered

```
:SETVAR INDEX1 17
:IF INDEX1 <= 20 THEN
```

the expression evaluator responds with the familiar

```
*** EXPRESSION TRUE
```

and will continue executing commands until a matching ENDIF is encountered, because the value of INDEX1 is 17, which is less than 20.

In the second case of dereferencing, explicit dereferencing, the variable name is preceded by one or more exclamation points (!) which directs MPE/XL to substitute the value of the variables represented at that place in the command line. The most important thing about explicit dereferencing is that MPE/XL substitutes a pair of exclamation points with a single exclamation point, and a single exclamation point forces MPE/XL to perform value substitution. For example

```
:SETVAR var1,'stringvalue'
:SETVAR var2,'!!var1'
:SHOWVAR var2
 VAR2 = !VAR1
:ECHO !var2
 stringvalue
```

This example also shows how variables can be set to any valid expression, including other variables. However, expression types may not be mixed as in

```
:SETVAR VAR_VALUE 17 + 'foo'
```

The Expression Evaluator would flag this command as an error.

Functions allow the manipulation of text in command line. Again, the part of MPE/XL responsible for evaluation of the results of functions is the Expression Evaluator. There are several functions, just a few of which are

```
len - string length function
str - string extraction
ups - upshift string
```

These powerful functions allow complicated variables to be built and examined in command files.

IV. Global Variables

Variables may be defined by the user, but MPE/XL maintains its own set of variables. All global variables start with HP (what else?), except CIERROR and JCW, which, under the old command interpreters, were JCW names. Global variables may be read only or read/write, and may not be deleted.

Jobs and sessions begin with the Global variables defined with initial values. Global variables allow for lots of things in the user environment to be tested or displayed:

```
HPJOBLIMIT - is the system's job limit
HPMONTH - is the month according to MPE/XL
HPCIERRMSG - is the error message that corresponds
 to another variable, CIERROR
```

HPMSGFENCE - allows MPE/XL error messages to be suppressed.

One global variable of particular interest is HPPATH. This variable defines the 'path' that MPE/XL will search for command files (and program files, too). Initially, it has the value !hpgroup,pub,pub.sys. This means that MPE/XL will first look for command files or programs in your current group (!hpgroup), the the public group of your logon account (pub) and last in pub.sys.

V. Defining command files

Ever since MPE III, users have had the ability to define their own commands. The way this was accomplished in the past was through the User Defined Commands (UDC's), which are familiar to most users. However, a new way of storing user commands was introduced with MPE/XL. This is the command file. Command files and UDCs are both very similar and very different. One of the main differences is that multiple UDCs are defined in a single MPE file and the SETCATALOG command is used to invoke the UDCs at the appropriate level - system, account or user. On the other hand, a command file represents a single user command, and that command is invoked by virtue of the name of the file itself. In other words, to execute the commands in the file COMMFILE, merely say

```
:COMMFILE
```

Kind of simple, isn't it? Optional parameters may be added after the command file name, but must follow the rules outlined in the header portion of the command file. Command file command lines may contain commands that are valid MPE/XL commands, or user commands in UDCs or command files. Command lines may even contain the name of program files, since with MPE/XL, the :RUN command is implied if the command name is a valid program file. In MPE/XL, user commands may even invoke themselves (unless disallowed with the OPTION NORECURSION).

Command files are much more simple to create than UDCs. Simply use you favorite brand of text editor, and place commands in an MPE file. To execute command files in a different group or account, you must have the appropriate access to that file, in the same fashion as UDC's.

VI. Command File Structure

The basic structure of command files is simple,and very much like UDCs. The first line is the optional parameter line. Up to 63 parameters may be specified. The syntax for this line is

```
parm parm1[='defaultvalue'],...parm63
```

The second line is the options line. These lines specify how the command file will be accessed or the basic environment the command file will operate in:

```
option option1[,option2[,...]]
```

Some available options are BREAK/NOBREAK, HELP/NOHELP, etc. Remaining lines are command lines. All MPE/XL commands except :DO and :REDO are valid and may be used in user commands.

Parameters for command files are specified in one of two ways. New MPE/XL users will feel comfortable with the POSITIONAL parameter sequence, in with the parameters in the command are specified in the same sequence as in the PARM line of the command file. The other way is by using the KEYWORD construct. A user file (named COPYFILE) containing the lines

```
parm filein='$stdin', fileout='$stdlist'
fcopy from=!filein;to=!fileout;new
```

may be invoked by any of these valid user commands:

```
:copyfile oldfile, b
:copyfile oldfile b
:copyfile fileout=b,filein=oldfile
```

VII. Programmatic Access to Command Files

One of the really nice things about command files (and also UDCs) is that with MPE/XL, all user commands may be invoked from programs. Those that cannot are those that have the OPTION NOPROGRAM specified in the header portion of the command file. Hewlett-Packard had the foresight to provide MPE/XL users with the new HPCICOMMAND intrinsic, which allows any valid user command to be invoked from a program. This powerful new intrinsic allows the distribution of programming power to the command interpreter, and will be discussed in a subsequent section. The consequences are that any command, MPE/XL or user, can be executed from a program.

VIII. New MPE/XL commands to support user commands

Several new commands were introduced in MPE/XL that allow maximum utilization of command files. Although command files can be written without the use of these commands, the power of MPE/XL really comes through when these are used.

The first of the new commands is the WHILE and the ENDWHILE commands, which have the the syntax

```
:WHILE boolean-expression [DO]
        < commands executed as long as condition
          is true >
```

```
:ENDWHILE
```

The WHILE and ENWHILE commands allow for multiple repetition of of a block of commands while a condition is true. As long as the condition is true, the commands will continue to be executed.

Another command of great use in commands files is the new COPY command. The COPY command is the much modified FCOPY command, with a much more simplistic syntax:

```
                                                    { ASK }
:COPY [FROM=]fromfile[;TO=tofile] [;{ YES }]
                                                    { NO  }
```

COPY perform a multi-record, no-buffered file copy of files much the same way that FCOPY does. There are some restrictions: fromfile and tofile may not be system defined files or spool files. The options determine whether the user is asked to replace the tofile if it exists - ASK will prompt the user; YES will replace the file and a NO option will leave the tofile intact if it exists.

The PRINT command is very similar to the COPY command. This command's syntax is

```
:PRINT [FILE=]file
     [;OUT = outfile ]
     [;START = startrecord]
     [;END = endrecord ]
     [;PAGE = linesinapage ]
     [;{UNN }
       {NUM}
```

Although primarily for printing files to $STDLIST, once you realize that any file may be specified for the file parameter (unlike the COPY command) and about any file may specified for the outfile, then the PRINT command is not unlike the FCOPY command except for real neat feature - it may called while in BREAK mode! PRINT and COPY are commands, and unlike FCOPY which is a program, may be used in BREAK mode.

The INPUT command allows interactive acceptance of variable values. The format for the INPUT command is

```
:INPUT [NAME=]variable-name
     [;PROMPT = promptstring]
     [;WAIT = waitseconds ]
```

This command allows the changing or the creation of variables, and optionally will prompt the user with a prompt string, and wait a given number of seconds for the user to respond. If the user does not respond in that time, command file execution continues, but CIERROR will be set with a

value of 9003.

Another really useful command, introduced previously, is the ECHO command which has the format

:ECHO [message]

Echo does not perform implicit dereferencing, but instead requires explicit dereferencing of variable names. One thing to remember about the ECHO command is that a carriage return is always generated after the message is displayed, and if message is null, only a carriage return is generated.

Other MPE/XL commands are the CALC command (used to generate the result of an expression to $STDLIST and to HPRESULT), and the RETURN command (returns to previous level of Command file).

IX. Command File Examples

Our first example will be to use three existing variables to create a fourth. The three pre-existing variables are global variables and contain the year (last two digits), the month (digits, not name) and the day of the month, respectively. We use these to create a new variable, HPYYMMDD:

```
SETVAR HPYYMMDD,!HPYEAR * 10000
SETVAR HPYYMMDD,!HPYYMMDD + (!HPMONTH * 100)
SETVAR HPYYMMDD,!HPYYMMDD + !HPDATE
```

The next example uses the PAGE option of the PRINT command to print a file without pausing:

```
PARM FILENAME
PRINT !FILENAME;PAGE=0
```

This example shows the use of the parameter line, and the use of the parameter in the command file to cause the parameter value to be replaced with the value (REQUIRED) specified when the command was executed.

The next example shows the easy way to recover lost filespace by 'squeezing' the end of file to the file limit.

The following example is the command file SQUEEZE:

```
PARM FILENAME
COMMENT
COMMENT  This specifies that the command file will have ONE
COMMENT  required parameter - the filename to be squeezed.
COMMENT
IF NOT FINFO('!FILENAME',0) THEN
COMMENT
COMMENT  This line tests for the presence of the file
```

```
COMMENT  specified in the parm line. The FINFO is a function
COMMENT  requiring two parameters : the first the filename in
COMMENT  string form, and the second, the function number.
COMMENT  Zero as a parameter queries for the existence of a
COMMENT  file, and returns a Boolean result.
COMMENT
    ECHO !FILENAME does not exist - cannot squeeze
ELSE
COMMENT  ^
COMMENT  If the file does not exist, no need to even try any
COMMENT  of this stuff.
COMMENT
    SETVAR END_OF_FILE,!(FINFO(!FILENAME,19)
COMMENT  ^
COMMENT  This command line will set a variable to the number
COMMENT  of records in the file (end of file) by using
COMMENT  the FINFO function with a parameter of 19.
COMMENT
    FILE NEWFILE;DISC=!END_OF_FILE;SAVE
COMMENT  ^
COMMENT  This command line will set a variable to the number
COMMENT  of records in the file (end of file) by using FINFO
COMMENT  with a parameter of 19.
COMMENT
    SETVAR CIERROR,0
    COPY !FILENAME,*NEWFILE;YES
COMMENT  ^
COMMENT  These lines prime CIERROR to zero and copy the file
COMMENT  from the oldfile to the newfile.
COMMENT
    IF CIERROR = 0 THEN
        PURGE !FILENAME
        IF CIERROR = 0 THEN
            RENAME NEWFILE,!FILENAME
        ELSE
            ECHO !FILENAME was not purged nor renamed
        ENDIF
    ELSE
        ECHO Copy of !FILENAME failed
    ENDIF
COMMENT  ^
COMMENT  If the COPY command succeeds, purge the oldfile and
COMMENT  rename the newly created file to the name of the old
COMMENT  file. Otherwise, tell the user the copy and rename
COMMENT  has failed.
COMMENT
ENDIF
COMMENT * end of the command file.
```

Now for a really good example of the power of command files. First, lets consider the mundane output from the LISTF command. Typically, it has the format that we are all familiar with:

```
ACCOUNT= TESTACCT    GROUP= TESTGRP

FILENAME  CODE  ------------LOGICAL RECORD-----------
                    SIZE  TYP        EOF      LIMIT R/B

DISCFILE             80B  FA         597        597   3
CRLFILE   NMRL     128W   FB         689        689   1
COBIEXT   EDICT   1276B   VA         785        785   1
V23AXFRM  VFORM    256B   FA       27958      50000   1
W23AXWSP  ISR     1024B   FA           3         39   1
XL        NMXL     128W   FB       44260    4096000   1
Y23AXUSL  NMPRG    128W   FB        2031       2031   1
UOOAAX89  USL      128W   FB         266       1023   1
```

Careful examination of the output will show that if we redirect this output to a disc file, we could use fileset wildcards in some command files. PURGESET is such a command file. It invokes another command file (XEQFILE), which itself invokes other command files. The result is a wildcard file purge.

The following example is the command file PURGESET.

```
PARM FILESET
COMMENT
COMMENT This command file will purge filesets. The
COMMENT only parameter is the desired fileset.
COMMENT
SETVAR SAVE_HPAUTOCONT,HPAUTOCONT
SETVAR SAVE_HPMSGFENCE,HPMSGFENCE
COMMENT
COMMENT In these lines, we save off the current value
COMMENT of HPAUTOCONT, the autocontinue var, and
COMMENT HPMSGFENCE, the var that determines if MPE/XL
COMMENT error messages are displayed.
COMMENT
SETVAR HPAUTOCONT,TRUE
SETVAR HPMSGFENCE,2
COMMENT
COMMENT  Here, we'll set HPAUTOCONT to true, meaning that
COMMENT  we won't have to preface every command line with
COMMENT  a :CONTINUE; it's implied. HPMSGFENCE = 2 tells
COMMENT  MPE/XL to override printing of error messages.
COMMENT
SETVAR FILE_SET,"!FILESET"
ECHO Please wait..Determining value of fileset
COMMENT
COMMENT Save off our original fileset and tell the user
COMMENT to hang on a sec.
COMMENT
SETVAR CIERROR,0
FILE TEMPFILE;TEMP;REC=-80,1,F,ASCII;NOCCTL
FILE OLDTEMP=TEMPFILE,OLDTEMP
```

```
LISTF !FILE_SET,1;*TEMPFILE
COMMENT
COMMENT Here is the LISTF of our fileset into a temporary
COMMENT file. The ,1 format will give us lots of good info
COMMENT about each file, as shown above.
COMMENT
IF CIERROR = 0 THEN
    RUN EDITOR.PUB.SYS;STDIN=PRGSTDIN;STDLIST=$NULL
ELSE
    ECHO Fileset !FILE_SET Is Invalid.....
    ECHO CIERROR is !CIERROR which means:
    ECHO !HPCIERRMSG
ENDIF
COMMENT
COMMENT  In these lines, if CIERROR is zero, run EDIT/3000
COMMENT  with a redirected $STDIN, else tell the user what
COMMENT  went wrong.
COMMENT
RESET TEMPFILE
RESET OLDTEMP
XEQ XEQCOMM
COMMENT
COMMENT  Reset the tempfile file equations and XEQute the
COMMENT  text file that EDIT/3000 created previously.
COMMENT
DELETEVAR FILE_@
DELETEVAR GOTFILE
DELETEVAR QUALNAME
DELETEVAR LISTF_@
DELETEVAR YES_@
COMMENT
COMMENT Cleanup. Delete vars created in this and other
COMMENT command files.
COMMENT
PURGE XEQCOMM
PURGE TEMPFILE,TEMP
COMMENT
COMMENT Purge the XEQ file and the temporary file.
COMMENT
SETVAR HPAUTOCONT,SAVE_HPAUTOCONT
SETVAR HPMSGFENCE,SAVE_HPMSGFENCE
COMMENT
COMMENT Reset these vars to their previous values.
COMMENT Delete the SAVE_ vars
DELETEVAR SAVE_@
```

The following example is the MPE file used as the $STDIN for EDITOR.PUB.SYS in the command file PURGESET. This file is PRGSTDIN:

```
TEXT *OLDTEMP
CHANGE 50 TO :": IN ALL
CHANGE 1 TO :PURGEFLE ": IN ALL
KEEP XEQCOMM,UNN
```

```
EXIT
```

The following command file is invoked from PURGESET. It has a single parameter, the line from the LISTF command that was prefixed with the name of this command file (PURGEFLE):

```
PARM LISTF_LINE_IN
COMMENT
COMMENT The single parameter 'passed' to this command file
COMMENT file from PURGESET is the line of the LISTF file,
COMMENT TEMPFILE.
COMMENT
COMMENT REMEMBER : HPMSGFENCE = 2 and HPAUTOCONT = TRUE
COMMENT
SETVAR LISTF_LINE,"!LISTF_LINE_IN"
COMMENT
COMMENT Save off the parameter passed for examination.
COMMENT
SETVAR FILE_NAME,"![STR(LISTF_LINE,1,8)]"
COMMENT
COMMENT Examine the LISTF file. There are three types of
COMMENT lines in that file:
COMMENT    1. header lines.
COMMENT    2. blank lines
COMMENT    3. filename lines.
COMMENT Lets save off the first eight character (the STR
COMMENT function, discussed previously, does that) into
COMMENT a variable called 'FILE_NAME'.
COMMENT
IF "!FILE_NAME" = "ACCOUNT=" THEN
COMMENT
COMMENT This LISTF line is a header line. It will contain
COMMENT the group and account name of those files that
COMMENT will be listed in the succeeding LISTF lines.
COMMENT
    SETVAR GOTFILE,FALSE
    SETVAR LISTF_ACCOUNT,"![STR(LISTF_LINE,11,8)]"
    SETVAR LISTF_GROUP,"![STR(LISTF_LINE,31,8)]"
COMMENT
COMMENT Lets set a variable to say let us know that this
COMMENT is not a file. Also, extract the account and
COMMENT group in variables.
ELSE
    IF ("!FILE_NAME" = "FILENAME") OR &
       ("!FILE_NAME" = "        ") THEN
        SETVAR GOTFILE,FALSE
    ELSE
COMMENT
COMMENT These lines are of no concern to us. We ignore them.
COMMENT
        SETVAR GOTFILE,TRUE
        SETVAR QUALNAME,"!FILE_NAME" + "." + "!LISTF_GROUP"
        SETVAR QUALNAME,"!QUALNAME" + "." + "!LISTF_ACCOUNT"
        STRIP QUALNAME
```

```
    ENDIF
ENDIF
COMMENT  ^
COMMENT  If it is not a header line or a blank line, it is
COMMENT  the name of a file. Lets fully qualify the filename
COMMENT  such that QUALNAME contains file.group.account.
COMMENT  If QUALNAME contains imbedded blanks (file,group or
COMMENT  account < 8 chars), STRIP will strip out blanks.
COMMENT
IF GOTFILE THEN
COMMENT ^
COMMENT If our Boolean var is set, the line just processed
COMMENT contains a filename, which was preceeded by blank and
COMMENT header lines (ALWAYS).
COMMENT
    SETVAR YES_NO_PROMPT,'Purge ' + '!QUALNAME' + ' (Y,N)?'
    SETVAR YES_NO," "
COMMENT ^
COMMENT Setup the variable names to contain the userprompt
COMMENT and the user's response.
COMMENT
    WHILE ("!YES_NO" <> "Y") AND ("!YES_NO" <> "N") DO
        INPUT YES_NO,PROMPT="!YES_NO_PROMPT"
    ENDWHILE
COMMENT ^
COMMENT These lines will ask the user if the file should
COMMENT be purged. User must respond Y or N.
COMMENT
    IF ("!YES_NO" = "Y") THEN
        ECHO    << Purging !QUALNAME >>
        SETVAR CIERROR,0
        PURGE !QUALNAME
COMMENT ^
COMMENT User responded Y. Attempt to purge the file.
COMMENT
        IF CIERROR <> 0 THEN
            ECHO Purge of !QUALNAME failed.......
        ENDIF
COMMENT ^
COMMENT Attempt failed. Tell the user why and continue to
COMMENT next file.
COMMENT
    ENDIF
ENDIF
COMMENT * end of command file.

The next command file is STRIP. It will parse a variable
value and remove ALL leading,trailing and imbedded blanks.

PARM VARNAME
SETVAR LITVAR,"!VARNAME"
SETVAR SAVEVAR,!VARNAME
WHILE POS( " ","!SAVEVAR") > 0
    SETVAR SAVEVAR,"!SAVEVAR" - " "
```

```
ENDWHILE
SETVAR !LITVAR,"!SAVEVAR"
DELETEVAR LITVAR
DELETEVAR SAVEVAR
```

Once we have this command file under our belt, we can really start to get fancy. With a little imagination, we can select files in our fileset by specific attributes, such as file code, file size, etc. The mechanics are not difficult, but are not included because of space considerations. However, I can include a command file to set the attributal variables related to a file. This command file is FILEATTR:

```
PARM LISTF_LINE
OPTION NOLIST
SETVAR FILE_CODE,STR(LISTF_LINE,11,6)
COMMENT ^  file code of file (string)
SETVAR FILE_CODE_I,FINFO('!QUALNAME',-9)
COMMENT ^  file code of file (integer)
SETVAR CREATOR,FINFO('!QUALNAME',4)
COMMENT ^  file creator (string)
SETVAR DATE_CREATED_STR,FINFO('!QUALNAME',6)
COMMENT ^  date created (string) )
SETVAR REC_SIZE,STR(LISTF_LINE,17,5)
COMMENT ^  record size (bytes or words,string)
SETVAR ASCII_BINARY,STR(LISTF_LINE,26,1)
COMMENT ^  file format (ascii or binary,string)
SETVAR FIXED_VARIABLE,STR(LISTF_LINE,25,1)
COMMENT ^  record format (fixed or variable,string)
SETVAR BYTES_WORDS,STR(LISTF_LINE,22,1)
COMMENT ^  record units of measure (bytes or words,string)
SETVAR EOF,STR(LISTF_LINE,30,9)
COMMENT ^  number of records in the file (string)
SETVAR FILE_LIMIT,STR(LISTF_LINE,40,10)
COMMENT ^  maximum number of records in file (string)
SETVAR FILE_LIMIT_I,FINFO('!QUALNAME',12)
COMMENT ^  max records in file (integer)
SETVAR FOPTIONS,FINFO('!QUALNAME',13)
COMMENT ^  file options (string)
SETVAR FOPTIONS_I,FINFO('!QUALNAME',-13)
COMMENT ^  integer foptions
SETVAR LAST_MOD_DATE_YYYYMMDD,FINFO('!QUALNAME',-8)
COMMENT ^  last modification date (integer)
SETVAR INT_1,LAST_MOD_DATE_YYYYMMDD / 1000000
SETVAR INT_2,INT_1 * 1000000
SETVAR LAST_MOD_DATE_YYMMDD,LAST_MOD_DATE_YYYYMMDD - INT_2
COMMENT ^ last mod date YYMMDD
STRIP FOPTIONS
STRIP FILE_CODE
STRIP REC_SIZE
STRIP EOF
STRIP FILE_LIMIT
COMMENT ^ Strip the blanks from these vars. Once STRIPped,
COMMENT   their variable class will change to integer.
```

X. MPE/XL as a Programming Language

MPE has been thought of by some as being capable of being a programming language. Previous versions of MPE fell short of that mark, but probably not by much.

By that same token, lets examine MPE/XL in comparison to both older versions of MPE and other programming languages.

Although MPE/XL does not have a compiler, it is not necessarily a prerequisite of a programming language to have a compiler. Recursive abilities are found in MPE/XL (WHILE..ENDWHILE) and other programming languages; storage of numeric and non-numeric literals (SETVAR) are also present in MPE/XL and other compilers. Examination of stored literals (IF statement with explicit variable dereferencing) and intermediate literals (IF statement with STR function of literal constant, for example) are present in MPE/XL as well as compiler languages. The ability to call programs written in other languages (RUN statements) or in the same language is characteristic of some programming languages. Programs written in MPE/XL are callable from programs written in other programming languages (HPCICOMMAND intrinsic is needed). In addition, MPE/XL lends itself well to structuring and development with modularity, a trademark of some programming languages. MPE/XL can also accept input external to the program (INPUT command), produce output (PRINT, ECHO) and can perform arithmetic operations (CALC command). Older versions of MPE had but a few of these features.

Indeed, there may now be little argument that MPE/XL is a programming language. However, to make MPE/XL a REALLY powerful programming language, I offer Hewlett-Packard a 'wish list'.

XI. Some Things to Make MPE/XL Really Neat

While MPE/XL is a vast improvement over MPE, I would like to see several things implemented to make MPE/XL a bona fide programming language.

The first would be the ability to call any intrinsic from a command file (except those that use procedure labels). All the new HP intrinsics (HPFOPEN, HPFCLOSE) could be called in the same manner as calling other command files. For example, to call HPFOPEN, you might use command syntax like this:

```
HPFOPEN filenum,filename,fileoptions
```

This seems like a natural extension to MPE/XL, and something that should follow all the progress made with MPE/XL.

Several additional functions that would be nice to have might be a leading/trailing spaces 'trimming' function similar to the 'STRIP' command file presented previously. A 'STUFF' function to place a string within another string would be great. How about a square root function for the FORTRAN programmers?

These 'wishes' are not to be taken as a criticism of MPE/XL because they do not have these 'goodies'. MPE/XL is an outstanding example of a natural software evolution, and the absence of these functions or structure in no way detracts from the total product.

Acknowledgments

Many thanks to the people of Collier-Jackson, Inc in Tampa, for allowing me the experience of programming on a SPECTRUM machine. Special thanks to Barry Lemrow at CJI for ideas and suggestions, as well as his review of this paper before publication.

Special thanks to Hugh McKee of Hewlett-Packard, who also reviewed this paper before publication, and urged many months ago, that it should be written.