MPE/XL PROGRAMMING

by Eugene Volokh
VESOFT, Inc.
1135 S. Beverly Dr.
Los Angeles, CA 90035

ABSTRACT

In 1983, I wrote a paper called "MPE PROGRAMMING" (presented at the INTEREX Montreal conference), which showed how you could do some remarkable things with MPE alone, without the aid of a custom-written program. MPE Programming was the art of writing system programs entirely in the "language" of CI commands (possibly with some help from standard, HP-supplied utilities).

The main advantages of MPE Programming were ease of writing and ease of maintenance. The idea was that a couple of dozen MPE commands in a job stream were easier to deal with than a custom-made SPL or COBOL program, especially since when you write a program, you'll have to always keep track not just of the job stream, but also the program's source and object files. UNIX, incidentally, has a very powerful "Command Interpreter Programming" facility (such programs are called "shell scripts"); UNIX users often write very many shell scripts to do things that would otherwise require some rather cumbersome C or PASCAL system programs.

Unfortunately, MPE/V (and earlier MPE versions) were not really designed for any sort of sophisticated MPE programming. Many of the tricks I showed in my original paper bordered, I must admit, on the perverse. For instance, to find out if you're in job mode or session mode (without writing a program that calls WHO), I suggested that you execute the :RESUME command.

Why the :RESUME command, of all things? Well (almost by accident), the :RESUME command returns one error condition if done in a job and another if done in a session (but not in break). We could then completely ignore the actual function of the :RESUME command, and look only at its "side effect" -- the value of the CIERROR JCW, which told us whether we were in a job or sesion.

Similarly, to see if a file existed, we'd do a :LISTF ;$NULL of it. This was not because we wanted to see information about this file (if we did, we wouldn't put on

the ;$NULL) -- rather, we wanted to see if the :LISTF succeeded or failed. If it failed with a CIERROR 907, this meant that the file didn't exist -- if it succeeded, the file did exist.

MPE/XL was intended to make many of these things a lot simpler to do -- instead of weird, indirect techniques, mechanisms would be provided for easily getting environment information (your logon mode, etc.), file information (does a file exist?), and so on. Seemingly using UNIX as a prototype (in spirit if not always in detail), MPE/XL sought to make MPE Programming a straightforward proposition.

To a large extent, HP succeeded -- MPE/XL has a number of new commands and features that let you do much more powerful things from the Command Interpreter. In some ways, though, some of the features seem at first glance to be more powerful than they really are, and quite a few things that you'd like to do remain tantalizingly out of your reach.

In the process of converting my MPEX/3000 and SECURITY/3000 products to MPE/XL -- and in the process of implementing most of the MPE/XL user interface features in the MPE/V version of MPEX (and in SECURITY/3000's STREAMX module), usable by "classic HP 3000" users -- I learned a good deal about the new MPE/XL features, their strengths and their weaknesses. This paper will try to objectively discuss both; to show you how to use the strengths to their utmost and how to work around some of the weaknesses.

THE NEW FEATURES OF MPE/XL

What exactly are the new MPE programming-related features of MPE/XL? There are several:

* First of all, MPE/XL supports VARIABLES. Think of them as JCWs that can have string values as well as integer values. (Actually, they can have boolean and 32-bit integer values, too.) E.g.

```
:SETVAR FNAME "FOO.DATA.PROD"
```

* MPE/XL PREDEFINES some variables to values such as your user name, your account name, your capabilities, etc. For instance,

```
:SHOWVAR @
HPACCOUNT = VESOFT
HPDATEF = TUE, FEB  9, 1988
HPGROUP = DEV
HPINPRI = 8
HPINTERACTIVE = TRUE
HPJOBCOUNT = 2
HPJOBLIMIT = 2
HPJOBFENCE = 7
HPJOBNAME = EUGENE
HPJOBNUM = 268
HPJOBTYPE = S
HPLDEVIN = 20
...
```

(Don't you wish you'd had this all along???)

* MPE/XL lets you SUBSTITUTE the values of variables (and even EXPRESSIONS involving the variables) into MPE commands -- just as you could always substitute the values of UDC parameters. For example,

```
:SETVAR FNAME "FOO.DATA.PROD"
:PURGE !FNAME
```

is equivalent to

```
:PURGE FOO.DATA.PROD
```

Then you could also say

```
:BUILD !FNAME;DISC=![100*NUMUSERS+25];REC=-64,,F,ASCII
```

it will build a new FOO.DATA.PROD file with room for 100*NUMUSERS+25 records (presumably NUMUSERS is an integer variable previously set with a :SETVAR).

* As shown in the above example, MPE/XL lets you use EXPRESSIONS in variable substitution, in the :SETVAR command, in the :IF command, and in the new :WHILE and :CALC commands:

```
:SETVAR EXPECTEDFLIMIT 100*NUMUSERS+25
:SETVAR FNAME "S"+MODULENAME+".PUB.SYS"
:SETVAR MODULENAME STR(FNAME,2,POS(".",FNAME)-2)
:IF HPACCOUNT<>"SYS" THEN
:IF POS("SM",HPUSERCAPF)=0 THEN  << user doesn't have SM >>
```

As you can see, the expressions can involve either numbers or strings, and a number of useful operators have been defined, such as:

+ to concatenate strings;
STR to extract substrings;
POS to find the position of one string in another;
UPS to upshift a string;

and many others.

* Perhaps the most useful of the defined operators is FINFO, which takes a filename and an option number and returns a piece of information about that file:

```
FINFO(filename,0)  = TRUE if file exists, FALSE if it does
FINFO(filename,1)  = string with fully-qualified filename
FINFO(filename,4)  = string containing file's creator
FINFO(filename,8)  = file's creation date, formatted strin
FINFO(filename,-8) = file's creation date, integer format
FINFO(filename,9)  = file's string filecode (e.g. "EDTCT")
FINFO(filename,-9) = file's integer filecode (e.g. 1052)
and much more.
```

For example, to check if a file exists, you can say

```
:IF FINFO('MYFILE',0) THEN
```

To check if a file's EOF is within 10% of its FLIMIT, you might enter

```
:IF FINFO('MYFILE',19)>=FINFO('MYFILE',12)*9/10 THEN
```

FINFO mode 19 gets you the EOF; FINFO mode 12 gets you the FLIMIT. (The mode numbers are taken from the FLABELINFO intrinsic -- one of the weaknesses of FINFO is that you have to remember these silly item numbers.)

* Commands have been added to OUTPUT and INPUT data:

```
:ECHO NOW WE'LL ASK YOU FOR A FILENAME.
:INPUT FNAME; PROMPT="Please enter the filename: "
:ECHO FNAME = !FNAME, FLIMIT = !![FINFO(FNAME,12)]
```

The :INPUT command can even have a timeout (wait for no more than X seconds) option.

* In addition to MPE/V control structures like :IF, :ELSE, and :ENDIF, MPE/XL implements the :WHILE / :ENDWHILE construct, e.g.

```
:SETJCW I = 295
:WHILE I < 314
:  ABORTJOB #J!I
:  SETJCW I = I+1
:ENDWHILE
```

* Instead of setting up UDCs, you can set up COMMAND FILES. If you want to define a command called S that does a :SHOWJOB, you can build a file called S.PUB.SYS that contains the lines:

```
PARM WHAT=" "
SHOWJOB JOB=@!WHAT
```

Now, whenever you type

```
:S J
```

(for example), MPE/XL will execute the file S.PUB.SYS passing "J" to it as a parameter. Same as a UDC, but no need to :SETCATALOG.

* Actually, whenever you type a command (like S in the example above) that isn't a normal MPE command, MPE/XL doesn't just check for it in PUB.SYS. It instead looks at the variable (remember those?) called HPPATH, and tries to find the file in the groups listed in the variable.

By default, HPPATH is set to

```
!HPGROUP,PUB,PUB.SYS
```

This means "first look in !HPGROUP (i.e. your group), then in the PUB group (of your own account), and then in PUB.SYS". You can change HPPATH to tell MPE/XL to look in UTIL.SYS, PUB.VESOFT, PUB.TELESUP, or what have you.

* In addition to letting you execute command files by just entering their names, you can also run a program just by entering its name (IMPLIED RUN). If you say

```
:SPOOK5
```

MPE/XL will search the groups specified in HPPATH -- if the first file it finds is SPOOK5.PUB.SYS (a program file), it'll run it just as if you'd said

```
:RUN SPOOK5.PUB.SYS
```

Similarly, to run a program in your own group, you can just say

```
:MYPROG
```

and MPE/XL will automatically supply the :RUN (remember, MPE/XL will look in HPPATH to determine which groups it should search -- by default, your group is one of them). If you say

```
:MYPROG "BANANA",5
```

it'll run MYPROG with INFO="BANANA" and PARM=5 (other :RUN command parameters are not available).

* Finally, a few odds and ends:

  - The :CALC command works as a general-purpose integer and string calculator.

  - Users can now redefine their own prompt by setting the HPPROMPT variable.

  - :SETCATALOG lets you add a new UDC file (or remove one) without retyping the names of all the other UDC files (which is cumbersome and risks accidentally unsetting an important file).

- You can :REDO not just the last command, but one of the last 20 commands (or even more than 20 if you so choose). This is actually a very powerful tool -- I'm only including it in "odds and ends" because it's not directly relevant to MPE/XL programming.

These are the features -- what are the benefits?

THINGS THAT ARE NOW EASY TO DO

#1. ENVIRONMENT VARIABLES

One example in my original "MPE Programming" paper involved a UDC finding out whether it's being executed in a session or in a job. This might, for instance, be a logon UDC that you use to set your function keys -- it outputs a whole bunch of escape sequences, which you want to see when you're online, but which will only garble your printout if printed in a job.

In MPE/V, if you recall, checking job/session mode was done this way:

```
SOFTKEYSINIT   << the logon UDC name >>
OPTION LOGON
SETJCW CIERROR=0
CONTINUE
RESUME
IF CIERROR<>978 THEN
  << initialize the softkeys >>
ENDIF
```

Very straightforward, isn't it? The :RESUME command, of course, is not used for :RESUMEing at all; rather, we count on it to generate an error condition -- error 978 if in batch, but a different error (warning 1686) if online.

MPE/XL makes this laughably simple:

```
SOFTKEYSINIT
OPTION LOGON
IF HPINTERACTIVE=1 THEN
  << initialize the softkeys >>
ENDIF
```

Essentially, MPE/XL automatically presets some JCWs to interesting values -- HPINTERACTIVE, HPLDEVIN (your terminal number), HPUSER (your logon user id), etc. This process actually started in MPE/V with the HPYEAR, HPMONTH, HPDATE, HPDAY, HPHOUR, and HPMINUTE JCWs, but MPE/XL has added a lot of new and useful ones.

Some more practical applications are readily apparent and others (the best kind) aren't. For instance, a really nice typing-saver is:

```
:NEWUSER JACK;CAP=!HPUSERCAPF
```

"HPUSERCAPF" stands for "USER CAPabilities, Formatted". It's a STRING variable that indicates which capabilities you currently have, e.g. "AM,AL,GL,ND,SF,PH,DS,IA,BA". The "!" before the "HPUSERCAPF" works much as it would before a UDC parameter -- it tells MPE to substitute in the VALUE of the HPUSERCAPF variable in place of its name.

Thus, the command might end up being:

```
:NEWUSER JACK;CAP=AM,AL,GL,ND,SF,PH,DS,IA,BA
```

You didn't have to type in all of those capabilities -- the !HPUSERCAPF automatically put in all the ones you have.

You might even say

```
:NEWUSER JACK;CAP=![HPUSERCAPF-"AM,"]
```

Saying ![xxx] tells MPE: "Evaluate the expression xxx and substitute in its result". Subtracting two strings in MPE/XL removes the first occurence of the second string from the first -- thus, the :NEWUSER command will become

```
:NEWUSER JACK;CAP=AL,GL,ND,SF,PH,DS,IA,BA
```

(since "AM,AL,GL,ND,SF,PH,DS,IA,BA"-"AM," is "AL,GL,...,BA").

Another nice example is:

```
:FILE SYSLIST=BK!HPYEAR!HPMONTH!HPDATE,NEW;DEV=DISC;SAVE
:STORE @.@.@; *T
```

This will do a system backup and send the listing to a disc file IDENTIFIED BY THE BACKUP DATE.

Thus, you can keep many of your backup listings online (so you could easily tell which tape set and reel number a file was on); each one will be stored in its own file.
For instance, on 20 November 1988, the above commands will be executed as:

```
:FILE SYSLIST=BK881120,NEW;DEV=DISC;SAVE
:STORE @.@.@; *T
```

Unfortunately, it's not quite this simple. (Almost, but not quite.) What if we do the :FILE SYSLIST= on the 9th of April? Then, we'd get

```
:FILE SYSLIST=BK8849;...
```

-- not quite what we want. We'd like the month and day to be zero-padded, so that the file names will be more comprehensible and a :LISTF will show them in the right order (i.e. not show BK8849 after BK88410 and BK881231). How can we do this? Well, how about

```
:FILE SYSLIST=BK![10000*HPYEAR+100*HPMONTH+HPDATE];...
```

Instead of substituting the month and the day in directly, we calculate the value 10000*HPYEAR+100*HPMONTH+HPDATE. Since this is arithmetic, not textual substitution, "zero-padding" will occur -- the 9th of April of 1988 will yield 880409. Then, we textually substitute the resulting value into the :FILE equation:

```
:FILE SYSLIST=BK880409;...
```

Even the additional power of MPE/XL doesn't remove the need for a little ingenuity.

Finally, one more useful little UDC:

```
HIPRI !JOBNUM
ALTJOB #J!JOBNUM;INPRI=14
SETVAR OLDJOBLIMIT HPJOBLIMIT
LIMIT ![HPJOBCOUNT+1]
LIMIT !OLDJOBLIMIT
DELETEVAR OLDJOBLIMIT
```

Three guesses as to what this does? Give up? Well, you :STREAM a job and find it at the bottom of the WAIT queue; you want it to execute, but you don't want to let any of the other WAITing jobs through.

This UDC:

* Alters the job to input priority 14 (the highest priority possible).

* Saves the old job limit (indicated by the built-in variable HPJOBLIMIT) in an MPE/XL variable (OLDJOBLIMIT).

* Sets the job limit to HPJOBCOUNT -- the number of currently executing jobs -- plus 1, thus letting the topmost WAITing job (the one you just :ALTJOBd) through.

* Sets the job limit back to what it was before.

* Just for cleanliness, deletes the OLDJOBLIMIT variable.

Voila! The one problem I can see is that the UDC expects only a job NUMBER, not the leading "#J" -- if a user types

```
HIPRI #J123
```

then the very first line will be

```
ALTJOB #J#J123;INPRI=14
```

-- MPE won't like this much. We'd like to let the user type either

```
HIPRI 123
```

or

```
HIPRI #J123
```

whichever he prefers.

The solution is again fairly simple, taking advantage of MPE/XL's provisions for strings and for string operators:

```
HIPRI !JOBNUM
IF UPS(LFT("!JOBNUM",2))="#J" THEN
  ALTJOB !JOBNUM;INPRI=14
ELSE
  ALTJOB #J!JOBNUM;INPRI=14
ENDIF
SETVAR OLDJOBLIMIT HPJOBLIMIT
LIMIT ![HPJOBCOUNT+1]
LIMIT !OLDJOBLIMIT
DELETEVAR OLDJOBLIMIT
```

The key here is the :IF expression -- it extracts the leftmost 2 characters of the string containing JOBNUM (LFT("!JOBNUM",2)), upshifts them (UPS(LFT("!JOBNUM",2))), and then compares them against "#J". If the characters are equal to "#J", then we just do an :ALTJOB !JOBNUM; if the characters are something else (presumably the start of the job number), then we insert a #J in front of them.

#2. FILE INFORMATION

One of the most valuable new features of the MPE/XL CI is the ability to obtain FILE INFORMATION. Remember the old MPE trick of finding out if a file exists or not?

```
SETJCW CIERROR=0
CONTINUE
LISTF MYFILE;$NULL
IF CIERROR=907 THEN
  << file doesn't exist >>
ELSE
  << file exists >>
ENDIF
```

Again, what we're doing here is executing a command (:LISTF) not for its main purpose, but rather for a side effect -- if we give :LISTF a file that doesn't exist, it'll set the CIERROR JCW to 907; if the file exists, CIERROR will remain 0.

MPE/XL is much more straightforward:

```
IF FINFO('MYFILE',0) THEN
  << file exists >>
ELSE
  << file doesn't exist >>
ENDIF
```

The FINFO function returns information about the file whose name is passed as the first parameter. The second parameter tells FINFO which information is to be gotten; 0 means a TRUE/FALSE flag indicating whether or not the file exists. Other values ask for other things, such as file code, EOF, FLIMIT, etc.

Applications for this abound. For instance, your job stream might check to see if a file has 100 or more free records:

```
:IF FINFO('DATAFILE',19) > FINFO('DATAFILE',12)-100 THEN
:  TELLOP File DATAFILE is &
:         ![FINFO('DATAFILE',19)*100/FINFO('DATAFILE',12)]% full!
:ELSE
...
```

FINFO(xxx,19) returns xxx's EOF; FINFO(xxx,12) returns xxx's FLIMIT; if EOF > FLIMIT-100, we send a message to the operator indicating how full the file is (again, the wonders of expression substitution).

Another application is that we can now :BUILD files that are the right size (rather than choose some number and hope that the file won't overflow) --

```
:BUILD NEWFILE;DISC=![FINFO('DFILE1',19)+FINFO('DFILE2',19)+100]
```

This builds NEWFILE to be large enough to fit all of DFILE1, all of DFILE2, and 100 more records on top of that. Unfortunately, note that we still can't figure out, say, the number of entries in an IMAGE database (which you might very well want to use in calculating a file limit) -- we're still restricted to the rather limited set of features that HP in its wisdom chose to provide to us.

There are, in fact, two pretty serious problems with FINFO:

* For one, there are still a number of things that FINFO just doesn't provide. To name a few:

  - The NUMBER OF SECTORS in a file. I found myself wanting to write a command file that compared the number of sectors a file occupied before and after a certain operation, but there was no way of getting this information.

  - The file's LAST ACCESS DATE/TIME and LAST RESTORE DATE/TIME (FINFO gives us the creation date and the last modify date, but not the last access date or the last restore date).

  - The file's security information -- :RELEASEd/:SECUREd flag, security matrix, etc. It would be quite nice, for instance, to check the access you're allowed to a file before running a program that might abort quite bizzarely if it isn't given the access it wants.

- Whether or not the file is currently IN USE (and if it is, in what mode).

- The NUMBER OF EXTENTS in a file, the number of user labels, and others.

In fact, if you look at the FINFO option numbers, you'll find that they're pretty much a subset of the option numbers of the FLABELINFO intrinsic, which also lets you obtain file information. Why a subset? Why not just implement all the FLABELINFO options (though even that would still leave some options out).

All the file attributes -- certainly all those listable with :LISTF ,2 and MPE/XL's new :LISTF ,3 -- should be easily obtainable from the CI.

* Perhaps more important than the omitted functions is the fact that

ALL THE FINFO OPTIONS ARE "MAGIC NUMBERS".

When you saw the command

```
:IF FINFO('DATAFILE',19) > FINFO('DATAFILE',12)-100 THEN
```

was it clear to you what FINFO(xxx,19) and FINFO(xxx,12) did? If HP is going to implement file access functions, why not have an FFLIMIT('DATAFILE'), an FEOF('DATAFILE'), an FFILECODE('DATAFILE') and so on? Or, if you want a single function, why not let the user say

```
FINFO('DATAFILE','FLIMIT')
```

or

```
FINFO('DATAFILE','EOF')
```

Sure, it would take a little bit of extra time to parse, but think of the advantages in clarity.

Of course, you can remedy this problem yourself by setting up (probably in a logon UDC) variables or JCWs that are set to the the appropriate FINFO values, e.g.

```
SETVAR FIFILECODE 9
SETVAR FIFLIMIT 12
SETVAR FIEOF 19
...
```

You'd probably have to set either 14 or 18 of these variables, and then you could say

```
:IF FINFO('DATAFILE',FIEOF)>FINFO('DATAFILE',FIFLIMIT)-100 THEN
```

Unfortunately, you and I both know most people won't do this -- they'll use the "magic numbers" and let you try to figure out what's going on.

Even if you set up all the variables and use them consistently, you'll lose one of the greatest advantages of command files: their stand-alone nature. Your "MPE programs" will now rely on your logon UDC and its SETVARs -- if it gets deleted, they'll stop working. If you want to copy your job stream or other MPE program onto some other machine, you'll have to be sure that the other machine has the same logon UDCs. The point is that HP shouldn't have made you (or let you) use "magic numbers" in the first place.

This might seem like looking a gift horse in the mouth -- for fifteen years, we had nothing, and now, when they give us something, we want more. However, it seems almost a shame that HP, having made the CI so much more powerful, didn't implement such reasonable and useful features.

#3. INPUT AND OUTPUT

A major shortcoming of MPE/V was the absence of any general output command. Why, to output a simple message, you had to have a UDC like

```
DISPLAY !STUFF
OPTION LIST
COMMENT !STUFF
```

The OPTION LIST would cause the UDC body -- in this case COMMENT followed by the DISPLAY parameters -- to be output; to output any message, you'd say

```
DISPLAY "HI THERE!"
```

Unfortunately, this would display not HI THERE!, but rather

```
COMMENT HI THERE!
```

To avoid the output of the "COMMENT ", you had to output special escape sequences to backspace the cursor and clear

the line -- of course, this wouldn't work on a printing terminal. All this bother just to display some text!

MPE/XL does things the right way -- it simply has an MPE command to do the job. Just say

```
ECHO HI THERE!
```

and that's it. The only thing I can complain about is the command name -- ECHO's pretty unintuitive. UNIX, of course, calls its command ECHO (along with calling its PURGE command RM and its text search command GREP), and MPE/XL borrowed the name. I'd rather HP called it DISPLAY or TYPE or OUTPUT or something like that, but it's hardly a big deal.

Of course, outputting variables and expressions can be easily done with the ECHO command -- just use the !xxx and ![xxx] syntaxes:

```
ECHO YOU'RE SIGNED ON AS !HPUSER.!HPACCOUNT, X = ![UPS(X)]
```

The only trick you need to know here is this: how do you output a string with leading blanks? Like all MPE commands, all blanks between the command name and the first parameter are skipped, so

```
ECHO HI THERE!
```

and

```
ECHO                HI THERE!
```

produce exactly the same output -- "HI THERE!" with no leading blanks. Stumped? Just say

```
CALC "    HI THERE!"
```

The CALC command takes an expression parameter (in this case, just a string constant), evaluates it, and outputs the result. Since the parameters start at the quote, all the blanks between the quote and the HI are NOT ignored, and are output. (Be careful, though, of using the CALC command for general output purposes -- it works quite well for strings and booleans, but for integers it outputs more than just the integer's value.)

In addition to the :ECHO command for output, MPE/XL also has an input command, fortunately called :INPUT. For instance, you might have a UDC that says:

```
MOVE !FROMFILE, !TOFILE
SETJCW CIERROR=0
IF FINFO("!TOFILE",0) THEN
  COMMENT Target file already exists!
  INPUT PROMPT="OK to purge !TOFILE? "; NAME=PURGEFLAG
  IF UPS(STR(PURGEFLAG,1,1))="Y" THEN
    PURGE !TOFILE
  ENDIF
ENDIF
RENAME !FROMFILE, !TOFILE
```

If TOFILE already exists, the UDC will ask the user if it's OK to purge it. UPS(STR(PURGEFLAG,1,1)) merely means "the upshifted first character of PURGEFLAG" -- this way, Y, YES, and YOYO will all be accepted as a YES answer.

Actually, there's one pretty big temptation with the :INPUT command that should be resisted. You should think twice (or more) before using the :INPUT command to prompt for UDC (or command file) PARAMETERS. For instance, a UDC such as

```
MOVEP
INPUT PROMPT="From file? "; NAME=FROMFILE
INPUT PROMPT="To file? "; NAME=TOFILE
SETJCW CIERROR=0
IF FINFO("!TOFILE",0) THEN
  COMMENT Target file already exists!
  INPUT PROMPT="OK to purge !TOFILE? "; NAME=PURGEFLAG
  IF UPS(STR(PURGEFLAG,1,1))="Y" THEN
    PURGE !TOFILE
  ENDIF
ENDIF
RENAME !FROMFILE, !TOFILE
```

may not be a very good idea at all. Unlike the parameterized UDC we showed above, this one can only be conveniently used directly from the CI. Say that you want to write another UDC that runs a program and renames one of its output files (LISTFILE) into LISTFILE.ARCHIVE. With the parameterized MOVE UDC, we could say:

```
...
RUN MYPROG
MOVE LISTFILE, LISTFILE.ARCHIVE
...
```

and then have the MOVE UDC prompt the user if LISTFILE.ARCHIVE already exists. The unparameterized MOVEP UDC can't be used here at all, since it always prompts the user for the input and output files, which in this case are fixed and should not be prompted for.

In other words, this is the same reason why the best third-generation language procedures take their input values as parameters rather than prompt for them -- a parameterized procedure is much more reusable than a prompting one.

One very interesting use of the :INPUT command, though, might be in cases such as this:

```
MOVE !FROMFILE=" ", !TOFILE=" "
IF "!FROMFILE"=" " THEN
  INPUT PROMPT="From file? "; NAME=VARFROMFILE
ELSE
  SETVAR VARFROMFILE "!FROMFILE"
ENDIF
IF "!TOFILE"=" " THEN
  INPUT PROMPT="To file? "; NAME=VARTOFILE
ELSE
  SETVAR VARTOFILE "!TOFILE"
ENDIF
SETJCW CIERROR=0
IF FINFO("!VARTOFILE",0) THEN
  COMMENT Target file already exists!
  INPUT PROMPT="OK to purge !VARTOFILE? "; NAME=PURGEFLAG
  IF UPS(STR(PURGEFLAG,1,1))="Y" THEN
    PURGE !VARTOFILE
  ENDIF
ENDIF
RENAME !VARFROMFILE, !VARTOFILE
```

This UDC can accept its input either from its parameters or from the terminal. If it's used from within another UDC or by a knowledgeable user, it can be passed parameters -- if a novice user is using it, he can just type

```
:MOVE
```

and be prompted for all the input (for instance, if he's unfamiliar with what parameters the UDC takes). Actually, this may not be so useful for a simple UDC like this, but a really complicated UDC with many parameters can be made much more convenient with "dual-mode" processing like this.

There are plenty of other uses for the :INPUT command -- menus, error processing ("Abort UDC or continue? "), etc. There are also a lot of rather devious, non-obvious uses for it, too (more about those later). The only thing that bears keeping in mind is that :INPUTs should not entirely take the place of parameter passing.

#4. :WHILE LOOPS

No programming language is really complete without some sort of looping capability. In MPE/V, you could sometimes make do with the pseudo-looping capabilities of EDITOR/3000 (for things like taking the output of one program and translating it into input for another) and the ability of :STREAMs to stream other jobs. For instance, one thing that we at VESOFT used to make multiple production tapes was a tape-making job stream that at the end streamed itself, thus forming a sort of "infinite loop". (This was before we implemented :WHILE and other MPE/XL functions in our STREAMX Version 2.0, which makes things much easier.)

In one respect, MPE/XL's :WHILE command gives you all the looping that you need (any loop, including the FOR x:=y TO z and the REPEAT ... UNTIL constructs, can be emulated with a :WHILE); however, as we'll discuss later, it falls tantalizingly short in some areas.

First the good news:

```
SETVAR JOBNUM 138
WHILE JOBNUM<=174 DO
  ABORTJOB #J!JOBNUM
  SETVAR JOBNUM JOBNUM+1
ENDWHILE
```

This is an example of how the :WHILE loop can iterate through a set of integers. This simply aborts a whole range of jobs, from #J138 to #J174. (Seems useless? Try submitting fifty jobs in one shot -- all of them with the same silly error! I did this the day before I wrote the paper; the :WHILE loop sure came in handy.) Similar things can be done in some other cases -- for instance, you can use this to purge LOG#####.PUB.SYS system log files IF you know the starting and ending log file numbers (unless you're willing to start at LOG0001 and work your way up).

Another example, taken roughly from Jeff Vance and John Korondy's excellent paper "DESIGN FEATURES OF THE MPE XL USER INTERFACE" (INTEREX Las Vegas 1987 Proceedings):

```
PRT F1, F2="", F3="", F4="", F5="", F6=""
COMMENT Prints F1, F2, F3, F4, F5, and F6 to the line printer
FILE OUT;DEV=LP
SETVAR I 1
SETVAR F7 ""              << to terminate the loop >>
WHILE '!"F!I"' <> ''
```

```
  IF FINFO('!"F!I"',0) THEN
    ECHO PRINTING !"F!I"
    PRINT !"F!I",*OUT
  ELSE
    ECHO ERROR: !F"!I" NOT FOUND, SKIPPED.
  ENDIF
  SETVAR I I+1
ENDWHILE
```

The WHILE loop here iterates through the 6 UDC parameters, making it unnecessary to repeat its contents once for each one. The construct !"F!I" is actually rather interesting. If I is 3, it gets translated into !"F3", which in turn gets replaced by the value of the F3 parameter.

Another example might be checking a parameter to make sure that it's, say, entirely alphabetic (in preparation for passing it to some program that will abort strangely and unpleasantly if there are any non-alphabetic characters in it):

```
SETVAR I 1
WHILE I<=LEN(PARM) AND UPS(STR(PARM,I,1))>="A" AND &
                       UPS(STR(PARM,I,1))<="Z" DO
  SETVAR I I+1
ENDWHILE
IF I>LEN(PARM) THEN
  COMMENT Hit the end of the string without finding a non-alpha
  RUN MYPROG;INFO="!PARM"
ELSE
  ECHO Error! Non-alphabetic character found:
  CALC "!PARM"
  SETVAR BLANKS ""
  SETVAR J 1
  WHILE J<I
    SETVAR BLANKS BLANKS+" "
    SETVAR J J+1
  ENDWHILE
  CALC BLANKS+"^"
ENDIF
```

Note the little "bell-and-whistle" -- if there's a non-alphabetic character, we use a :WHILE loop to concatenate together several blanks and an "^", so the output looks like:

```
Error! Non-alphabetic character found:
FOOBAR.XYZZY
      ^
```

Many parsing operations can actually be done more simply with the POS function (which finds the first occurence of

one string in another); however, some complicated operations (such as the ones we just showed) may require :WHILE loops.

Finally, one other place where :WHILE should find a lot of use is the :INPUT command:

```
INPUT PROMPT="OK to proceed (Y/N)? "; NAME=ANSWER
WHILE UPS(ANSWER)<>"Y" AND UPS(ANSWER)<>"YES" AND &
      UPS(ANSWER)<>"N" AND UPS(ANSWER)<>"NO" DO
  ECHO Error: Expected YES or NO.
  INPUT PROMPT="OK to proceed (Y/N)? "; NAME=ANSWER
ENDWHILE
```

Most good UDCs and command files that use :INPUT should have some sort of input error checking, and this kind of :WHILE loop is a convenient way of doing it.

With all this power, what's there to complain about? After all, with an :IF and a :WHILE any language is theoretically complete -- any algorithm can be implemented.

Well, not quite. Control structures can get you only as far as the data access primitives are able to take you. Take some of the iterative operations that you'd REALLY want to implement:

* WHILE there are files in a fileset, DO something to them.

* WHILE there are jobs left, ABORT them (in preparation for a backup).

* WHILE there are records in a fileset, DO some processing on them -- perhaps write some of the records into another file, or pass them as input to some other program.

You can't do any of this (straightforwardly) because MPE/XL doesn't provide you any functions to read files, to handle filesets, to find all jobs, etc. You'd like to be able to say:

```
:WHILE FRECORD('MYFILE',RECNUM)<>''
...
:ENDWHILE
```

where FRECORD would return you a particular record of the specified file; unfortunately, no FRECORD primitive exists. The :WHILE command is only as powerful as the conditions you can specify; unfortunately, at the moment, this seems mostly limited to numeric iteration and to checking command

success/failure.

Another thing you'd like to be able to do with :WHILE is to repeat a particular command every given number of seconds or minutes -- for instance, to have a job stream wait until a particular file is built or becomes accessible. Unless you're willing to spend lots of CPU time in the loop, you need to have some way of pausing for a given amount of time, e.g.

```
:WHILE NOT FINFO('MYFILE',0) DO
:  PAUSE 600    << 600 seconds >>
:ENDWHILE
```

Unfortunately, there is no :PAUSE command or PAUSE function provided by MPE/XL (although as we'll see shortly, there are some tricks you could do...).

#5. COMMAND FILES

Command files were implemented more for convenience than for additional power; however, they can be convenient indeed.

Simply put, a command file is a replacement for a UDC. If you want to implement a new command called DBSC to run DBSCHEMA, you used to have to write a UDC:

```
DBSC !TEXT="$STDIN", !LIST="$STDLIST"
FILE DBSTEXT=!TEXT
FILE DBLIST=!LIST
RUN DBSCHEMA.PUB.SYS;PARM=3
```

You'd add this UDC to your system UDC file, :SETCATALOG the file, and presto! you have a new command.

In MPE/XL, you could use a command file to do the same thing. You could build a file called DBSC.PUB.SYS that contains the text:

```
PARM !TEXT="$STDIN", !LIST="$STDLIST"
FILE DBSTEXT=!TEXT
FILE DBLIST=!LIST
RUN DBSCHEMA.PUB.SYS;PARM=3
```

Then, the very presence of the DBSC.PUB.SYS file will implement the new command -- no need to :SETCATALOG it. You can just say

```
DBSC MYSCHEMA, *LP
```

and MPE will check to see if DBSC.PUB.SYS exists, find that it does, and execute it much like it would have a :SETCATALOGed UDC.

Why is this so nice? Well, remember all the nonsense you had to go through to change a :SETCATALOGed UDC file? You had to build a new file with a different name, :SETCATALOG it in the old one's place, and even then it wouldn't take effect for another session until it logged off and logged back on! Most people ended up having several versions of the system UDC file, since you couldn't purge the old file until everybody who had been using it was logged off.

With command files, simply build the file, and there you have it. No need to worry about whether the UDC file is in use (unless the command is actually being executed at that very moment, it won't be in use); no need to choose a new name for the file; no need to remember to re-specify all the other UDC files on the :SETCATALOG.

In fact, the MPE/XL compiler commands are actually implemented this way -- :PASXL, for instance, is just a command file (PASXL.PUB.SYS) that sets up several file equations and runs PASCALXL.PUB.SYS (the actual compiler program file -- you still need programs for something!).

Whenever I give an example in this paper that involves UDCs, chances are very good that it will work with command files, too (in fact, you'd probably want to do it with command files). I only use UDCs in the examples to keep things as familiar as possible.

You could also implement account-wide commands by just putting the command files into your PUB group, and group-wide commands by putting them into your own groups. In fact, MPE/XL has a special variable called HPPATH that indicates where it is to search for command files; by default, HPPATH is set to "!HPGROUP,PUB,PUB.SYS", i.e. "search your group (!HPGROUP) first, then the PUB group, then the PUB.SYS group". You could actually change it to something else, e.g.

```
:SETVAR HPPATH "!HPGROUP,PUB,PUB.VESOFT,CMD.UTIL,PUB.SYS"
```

In fact, it's probably a good idea to keep your own command files not in PUB.SYS (where they'll just get lost among all the other files) but in a special group, say CMD.UTIL. This way, a simple

```
:LISTF @.CMD.UTIL
```

will show you all the system-wide command files that you've set up. Of course, you'll have to have a system-wide logon UDC that sets up the HPPATH variable to include CMD.UTIL.

A similar feature of MPE/XL is "implied run". Just entering a program file name will AUTOMATICALLY cause that program to be run; e.g.

```
:DBUTIL
```

will automatically do a

```
:RUN DBUTIL.PUB.SYS
```

WITHOUT your having to have a UDC or a command file for this purpose. You can also specify a parameter, which gets passed as the ;INFO= string to the program being run:

```
:MYPROG FOO
:PROG2 "TESTING ONE TWO THREE"
```

and also a second parameter, which gets passed as the ;PARM=:

```
:MYPROG ,10
:MYPROG FOOBAR,5
```

(Other parameters -- ;LIB=, ;STDIN=, ;STDLIST=, etc. can not be passed; you have to do a real :RUN for that.) Also note that MPE/XL looks for the program file in exactly the same places in which it looks for a command file: all those groups listed in the HPPATH variable.

These features are all very convenient, and can save you a good deal of effort and some typing. There is, however, one problem with both command files and implied :RUNs (and also UDCs) that limits their usefulness:

* THERE'S NO WAY FOR PASSING THE *ENTIRE REMAINDER OF THE COMMAND LINE* TO EITHER A COMMAND FILE, AN IMPLIED :RUN, OR A UDC.

For example, say that I want to implement a new command called :CHGUSER that executes my own CHGUSER.PUB.SYS command file. I want it to look much like MPE's :NEWUSER and :ALTUSER -- I'd like to let people say

```
:CHGUSER XYZZY;CAP=-BA,+DS,+PM;PASS=$RANDOM
```

The CHGUSER.PUB.SYS command file could then take the entire remainder of the line as a single parameter, and then perhaps pass it to some program that would process it.

Unfortunately, this simply can't be done! Since the parameter list includes ";"s, ","s, and "="s, MPE/XL views them as delimiters (it would view blanks as delimiters, too); there's no way of specifying in the command file that delimiter checking is to be turned OFF, and that the entire remainder of the command is to be passed as one parameter. Of course, you could require the user to enclose the parameter in quotes, but you'd rather not do that. (If you're thinking that declaring CAP=, PASS=, etc. as keywords to the command file will work, it won't -- look at the ","s in the CAP= parameter.)

In fact, MPE's own :FCOPY command couldn't be implemented as an auto-RUN or as a command file for this very reason -- each :FCOPY command always includes delimiters, and that won't work. I can see why HP doesn't like delimiters in an implied :RUN (so that the ;PARM= value can be specified as well as the ;INFO=), but why not have some sort of option for command files? Personally, I'd rather be able to pass the entire remainder of the command as one parameter than be able to specify a ;PARM= value.

In fact, UNIX does have a way of treating the parameter list (of either a program or a command file) as either a sequence of individual parameters or as one single string; UNIX programmers frequently use this feature. Again, this may be looking a gift horse in the mouth, but it would have been so easy for HP to implement something like this.

TRICKS

We've pretty much covered all the things you can do straightforwardly with MPE/XL. Of course, if this was all I had to say, I'd never have written this paper. People who know me know that I NEVER do things straightforwardly...

MPE/V had the (small) set of things you can do easily and the far larger set of things you could do if you really stood the system on its head. Similarly, MPE/XL has the larger set of things you can do easily, and the bigger still number of things you can do with a little bit of trickery. This is where the fun begins.

#1. PAUSING FOR X SECONDS

At a certain point in your job stream, a particular file may be in use. You don't want this to abort the job -- rather, you want the job to suspend until the file is no longer in use.

A first attempt at this might be:

```
WHILE FINFO('MYFILE',fileisinuseflag) DO
  PAUSE one minute
ENDWHILE
```

While the file is in use (surely there must be an FINFO option for this!), pause for a minute, and then check again. This shouldn't be too much of a load on the system (though without the :PAUSE this would be a heavy CPU hog indeed!).

Of course, you face two problems. First of all, there is no FINFO option to check to see if the file is in use or not. (OK, everybody, submit those SRs!) Old MPE programming hands, however, shouldn't despair:

```
FILE CHECKER=MYFILE;ACC=OUTKEEP;SAVE
SETJCW CIERROR=0
CONTINUE
PURGE *CHECKER
WHILE CIERROR<>0 DO
  PAUSE one minute
  SETJCW CIERROR=0
  CONTINUE
  PURGE *CHECKER
ENDWHILE
```

See what we're doing? The :FILE equation tells the file system to open the file with ;ACC=OUTKEEP (so the data won't get deleted) and close it with disposition ;SAVE (so the file itself won't get purged) -- the :PURGE command will thus not purge the file at all, but just try to open it with exclusive option. As long as the :PURGE is failing, we know that the file is in use (unless, of course, it doesn't exist or we're getting a security violation).

We do this check once before the :WHILE loop; then, if CIERROR<>0, we pause for a minute, do the check again, and keep going until the check succeeds.

The only problem that remains is, of course, that MPE/XL has no :PAUSE command -- without it, the entire exercise is academic.

What can we do? Well, one solution is to write a program. Call it PAUSE.PUB.SYS -- it'll take a ;PARM= value, convert it to a real number, and call the PAUSE intrinsic. Then, any of your command files could say

```
:RUN PAUSE.PUB.SYS;PARM=60
```

or just use the implied :RUN, as in

```
:PAUSE ,60
```

I don't like this. I don't like it for several reasons:

* The program, though not by any means difficult, is not trivial to write. If you know SPL, it's only a few lines; what if you only know COBOL? You can't even call the PAUSE intrinsic from COBOL (at least from COBOL '74), since COBOL can't handle real numbers (which the PAUSE intrinsic expects).

  From FORTRAN, you could call PAUSE, but you also need to call the GETINFO intrinsic (quick! do you know it's parameter sequence?). What if you had to write a program that checked to see if the file was in use? You'd have to call FOPEN, figure out the right foptions and aoptions bits (%1 and %100, if you're curious), and then use an intrinsic to set a JCW appropriately.

* Once you write it, you have to keep track of it. You put its object code into PAUSE.PUB.SYS -- where do you keep the source code? What if you lose it? Will you write documentation for it, or add a HELP option?

* Finally, the more external programs you use, the less self-contained the job stream will be. What if you move the job to one of your machines? You'll have to move the PAUSE program, too, and probably its source code and documentation, just to be safe.

  For vendors like VESOFT, the problem becomes even greater -- our installation job stream has to be able to run on a system where NONE of our software currently exists. We can't rely on your PAUSE.PUB.SYS or what have you.

You might agree with me or you might not. It's quite possible that the only problem with an external program file is that it somehow affects some silly esthetic sense of mine

-- that my mind is too twisted to appreciate a simple, straightforward solution. In any event, here's my answer to the problem:

```
:BUILD MSGFILE;TEMP;MSG
:FILE MSGFILE,OLDTEMP
:RUN FCOPY.PUB.SYS;STDIN=*MSGFILE;INFO=":INPUT DUMMY;WAIT=60"
```

Nice, eh? I build a temporary message file called MSGFILE, and then I run FCOPY with ;STDIN= redirected to it. Then, I tell FCOPY to execute an :INPUT command, telling it to WAIT for 60 seconds for input! (Of course, the only reason I use FCOPY here is to have it execute the MPE/XL command ":INPUT DUMMY;WAIT=60" -- FCOPY's convenient for this because we can pass the command to it as an INFO= string.)

Of course, the input will never come, since MSGFILE is empty; and, I must admit that the :INPUT ;WAIT= parameter was almost certainly intended to wait for TERMINAL input. However, it also works perfectly well when the input is coming from a $STDIN file that was redirected to a message file. When the 60 seconds are up, the :INPUT command will terminate and return control to FCOPY, which will then return back to the CI.

Now, our job stream is complete:

```
:BUILD MSGFILE;TEMP;MSG
:FILE MSGFILE,OLDTEMP
:FILE CHECKER=MYFILE;ACC=OUTKEEP;SAVE
:SETJCW CIERROR=0
:CONTINUE
:PURGE *CHECKER
:WHILE CIERROR<>0 DO
:  RUN FCOPY.PUB.SYS;STDIN=*MSGFILE;INFO=":INPUT DUMMY;WAIT=60"
:  SETJCW CIERROR=0
:  CONTINUE
:  PURGE *CHECKER
:ENDWHILE
```

Complete, of course, except for the many :COMMENTs that I'm sure that you, as a conscientious programmer, will certainly include...

Some may say that only a computer freak can think that the above solution is simpler than just running a program that loops doing FOPENs and PAUSEs.

They may be right.

#2. READING A FILE

The :REPORT command nicely shows you all the disc space used by each account on the system (actually, on MPE/XL 1.0 the disc space :REPORTed is sometimes erroneous, but I'm sure that'll be fixed soon). Unfortunately, it doesn't show you the total disc space used in the entire system, which is a useful piece of information. For instance, you might want to subtract the free and the used disc space counts from the total space on your discs, thus finding out how much lost space there is.

The :REPORT command can send its output to a file, which is good. But what can you do to read the file?

Well, let's start at the beginning. First, let's do a :REPORT into a disc file:

```
:FILE REPOUT;REC=-80,16,F,ASCII;NOCCTL;TEMP
:CONTINUE
:REPORT XXXXXXXX.@,*REPOUT
```

What's the XXXXXXXX.@ for? The :REPORT command usually outputs information on accounts and on groups; in our case, we don't want to have any group information at all. By specifying a group that we know doesn't exist in any account (I hope that you don't have a group called XXXXXXXX) we can make MPE output only the account information and no group information. It'll also print an error (NONEXISTENT GROUP), but that's OK.

Now, we have a temporary file called REPOUT, which contains two header lines and one line for each account. We'd like to extract the number of sectors used from each account line and add everything up. This is where the real trickery comes in.

One thing we might do is use EDITOR. The principle here is that we'll take the :REPORT listing, which looks like

```
ADMIN          15502      **     1046      **     8372      **
CUST            3062      **        0      **        0      **
DEV             7080      **       18      **        8      **
...
```

and "massage" it into a sequence of MPE/XL commands:

```
:SETVAR TOTALSPACE TOTALSPACE+    15502
:SETVAR TOTALSPACE TOTALSPACE+     3062
:SETVAR TOTALSPACE TOTALSPACE+     7080
...
```

We can then execute all these commands, and TOTALSPACE will be the total used disc space count.

Doing this is simple (?):

```
:PURGE REPOUT,TEMP
:FILE REPOUT;REC=-80,16,F,ASCII;NOCCTL;TEMP
:CONTINUE
:REPORT XXXXXXXX.@,*REPOUT
:SETVAR TOTALSPACE 0
:EDITOR
/TEXT REPOUT
/DELETE 1/2              << delete the header lines >>
/CHANGE 23/72,"",ALL     << delete everything right of the count >>
/CHANGE 1/8,":SETVAR TOTALSPACE TOTALSPACE+"  << delete the left >>
<< now, each line looks like: >>
<< :SETVAR TOTALSPACE TOTALSPACE+    15502 >>
/KEEP REPUSE,UNN
/USE REPUSE              << execute the :SETVARs >>
/EXIT
```

Now, the TOTALSPACE variable is set to the total disc space!

This is very much like what we did in pre-MPE/XL "MPE PROGRAMMING" -- we used EDITOR as a means of taking a program's or a command's output and making it another program's (in this case, also EDITOR's) input. In fact, UNIX's "sed" editor is very frequently used for this purpose by UNIX programmers (although it's much more adapted to this than EDITOR/3000 is).

The trouble with this solution is that it's inherently limited to plain textual substitution. What if we wanted to sum the disc space of all accounts that used more than 20,000 sectors? EDITOR has no command that can easily check the value of a particular field in a line. What we'd really like to do is use all the power of MPE/XL's :WHILE loop and expressions to process the :REPORT listing one line at a time.

As I mentioned before, MPE/XL unfortunately has no "get a record from a file" function. However, not all is lost.

Let's set up two command files. One (TOTSPACE) will look like this:

```
FILE REPOUT;REC=-80,16,F,ASCII;NOCCTL;TEMP
SETVAR OLDMSGFENCE HPMSGFENCE
SETVAR HPMSGFENCE 2
PURGE REPOUT,TEMP
CONTINUE
REPORT XXXXXXXX.@,*REPOUT
SETVAR HPMSGFENCE OLDMSGFENCE
FILE REPOUT,OLDTEMP
CONTINUE
RUN CI.PUB.SYS;PARM=3;INFO="TOTSPAC2";STDIN=*REPOUT;STDLIST=$NULL
ECHO TOTAL USED DISC SPACE = !TOTALSPACE
```

There are two new things here. One is

```
SETVAR OLDMSGFENCE HPMSGFENCE
SETVAR HPMSGFENCE 2
CONTINUE
REPORT XXXXXXXX.@,*REPOUT
SETVAR HPMSGFENCE OLDMSGFENCE
```

What's all this HPMSGFENCE stuff? Well, remember that the REPORT XXXXXXXX.@,*REPOUT command will almost certainly output an error message (NONEXISTENT GROUP). This is to be expected, and we don't want the user to have to see this.

So, we set the HPMSGFENCE variable to 2, indicating that error message are not to be displayed (setting it to 1 would inhibit display of warnings, but still print errors). However, since we want to reset HPMSGFENCE to its old value later, we save the old value of HPMSGFENCE, set the value to 1, do the command, and then reset the old value.

Personally, I think that this is a bit more effort than required. In MPEX, I simply added a new command called %NOMSG; saying

```
%NOMSG REPORT XXXXXXXX.@,*REPOUT
```

makes MPEX execute the :REPORT command without printing any messages. Similarly, HP could have had a :NOMSG command (for suppressing errors and warnings) and a :NOWARN command (for suppressing only warnings). This would have saved all the bother of the saving of the old HPMSGFENCE, setting it, and resetting it. In fact, to be really clean, I should even do a

```
:DELETEVAR OLDMSGFENCE
```

after doing the :SETVAR HPMSGFENCE OLDMSGFENCE.

In any case, the HPMSGFENCE solution is better than no solution at all -- in MPE/V, the warning message would always be displayed, and users might get quite confused by it.

The only other little trick (in this command file) is

```
RUN CI.PUB.SYS;PARM=3;INFO="TOTSPAC2";STDIN=*REPOUT;STDLIST=$NULL
```

What on earth does this mean?

In MPE/XL, the CI is not some special piece of code kept in the system SL. Rather, it's a normal program file called CI.PUB.SYS -- when a job or a session starts up, the system creates a new CI.PUB.SYS process on the job/session's behalf. However, CI.PUB.SYS is also :RUNable just like any other program; you can run it interactively by saying

```
:RUN CI.PUB.SYS
```

or just

```
:CI
```

Alternatively, you can run it and tell it to execute exactly one command:

```
:RUN CI.PUB.SYS;PARM=3;INFO="command to be executed"
```

(;PARM=3 tells the CI not to display the :WELCOME message and to only process the ;INFO= command, rather than prompt for more commands -- other ;PARM= values do different things.)

In our case, we're running CI.PUB.SYS with ;INFO="TOTSPAC2" (telling it to execute our TOTSPAC2 command file), and with ;STDIN= redirected to our :REPORT command output file. We redirect ;STDLIST= to $NULL, since the CI will otherwise echo its ;INFO= command -- ":TOTSPAC2" -- before executing it.

Now we can see what TOTSPAC2 contains:

```
:INPUT DUMMY          << to skip the first header line >>
:INPUT DUMMY          << to skip the second header line >>
:SETVAR TOTALSPACE 0
:SETVAR HPMSGFENCE 2  << to ignore any error messages >>
:WHILE TRUE DO        << loop until we get an error >>
:  INPUT REPORTLINE   << get a :REPORT detail line >>
   << extract the disc space -- 15 columns starting with >>
```

```
   << column 9 -- and add it to TOTALSPACE >>
:  SETVAR TOTALSPACE TOTALSPACE + ![STR(REPORTLINE,9,15)]
:ENDWHILE
```

See the trick? CI.PUB.SYS's ;STDIN= is redirected to a disc file, so all :INPUT commands will read from that disc file. For each line we read in, we extract the account disc space (STR(REPORTLINE,9,15)), and do a

```
:SETVAR TOTALSPACE TOTALSPACE + extracted_account_disc_space
```

When we run out of input lines, the :INPUT command will get an EOF condition, and the command file will stop executing. TOTALSPACE is now set to the total disc space.

Both the EDITOR and the two-command-files solution can be used online, though both require two files (the first approach would require a disc file that contains all the required EDITOR commands). In a job, the EDITOR approach can be completely self-contained, since the EDITOR commands can just be put into the job stream; the second approach can also be self-contained if you create the TOTSPAC2 command file within the job (by using EDITOR or FCOPY).

Finally, one more variation on the same theme:

```
FILE REPOUT;REC=-248,,U,ASCII;NOCCTL;MSG;TEMP
SETVAR OLDMSGFENCE HPMSGFENCE
SETVAR HPMSGFENCE 2
CONTINUE
PURGE REPOUT,TEMP
REPORT XXXXXXXX.@,*REPOUT
SETVAR HPMSGFENCE OLDMSGFENCE
FILE REPOUT,OLDTEMP
CONTINUE
RUN CI.PUB.SYS;PARM=3;INFO="INPUT DUMMY";STDIN=*REPOUT;STDLIST=$NULL
RUN CI.PUB.SYS;PARM=3;INFO="INPUT DUMMY";STDIN=*REPOUT;STDLIST=$NULL
SETVAR TOTALSPACE 0
WHILE FINFO('*REPOUT',19)>0 DO
  RUN CI.PUB.SYS;PARM=3;INFO="INPUT REPORTLINE";STDIN=*REPOUT;&
                                                STDLIST=$NULL
  SETVAR TOTALSPACE TOTALSPACE + ![STR(REPORTLINE,9,15)]
ENDWHILE
ECHO TOTAL USED DISC SPACE = !TOTALSPACE
```

Intuitively obvious, eh?

* The :REPORT command output is sent to a MESSAGE FILE.

* To read a line from the file, we say

```
RUN CI.PUB.SYS;PARM=3;INFO="INPUT REPORTLINE";STDIN=*REPOUT;&
                                              STDLIST=$NULL
```

  This essentially tells the CI to read into REPORTLINE the first record from *REPOUT -- since it's a message file, the record will be read and deleted; the next read will read the next record.

* We loop while FINFO('*REPOUT',19) -- REPOUT's end of file -- is greater than 0. When the file is emptied out, we stop.

This is entirely self-contained, and in some respects more versatile (we can, for instance, prompt the user for input in the middle of the :WHILE loop, since our $STDIN is not redirected). The output-to-a-message-file and run-the-CI-to-get-each-record constructs are essentially a poor man's FREAD function. On the other hand, this approach runs CI.PUB.SYS once for each file -- even on a Spectrum this'll take some time!

One other glitch: each one of those :RUNs prints out one of those pesky "END OF PROGRAM" messages. In MPE/XL, you can actually avoid them -- as long as you use an implied :RUN rather than an explicit :RUN command. We can't use an implied :RUN because we need to redirect the STDIN and STDLIST. This is another good argument for using the two-command-file solution, which does only one :RUN and thus prints out only one END OF PROGRAM message.

#3. A PSCREEN COMMAND FILE

One of the most useful contributed programs for the HP 3000 is PSCREEN. (If you've been living in Katmandu for the past ten years, you might not know that it prints the contents of your screen to the line printer.) It works by outputting an ESCAPE-"d" sequence to the terminal, which causes almost any HP terminal to send back (as input) the contents of the current line on the screen. PSCREEN sends one ESCAPE-"d" for each line, picks up the output transmitted by the terminal, and prints it to the line printer.

Now, PSCREEN is already up and running, so there's really no reason to implement it as a command file; however, it's quite interesting to try it, both as an example of the power of MPE/XL and of the trickery you need to resort to in order to work around some restrictions on that power.

The process of reading the data from the terminal is actually quite straightforward:

```
CALC CHR(27)+'H'
WHILE there are more lines on the screen DO
  INPUT CURRENTLINE;PROMPT=!![CHR(27)+"d"]
ENDWHILE
```

CHR(27) means a character with the ascii value 27 -- the escape character. "!![CHR(27)+'d']" is the string ESCAPE-d, which when sent to the terminal (by the ;PROMPT=) will cause the terminal to input (into CURRENTLINE) the current line on the screen. The CALC command outputs ESCAPE-H (home up) to send the cursor to the top of the screen.

(Actually, it turns out that we can't just display the home up sequence in the :CALC since :CALC will then output a carriage return and line feed, and we'll skip the first line on the screen; instead, we have to incorporate the ESCAPE-H into the first :INPUT command prompt.)

The only twist here (one that the "real" PSCREEN has to deal with, too) is finding out how many lines there are on the screen. If we send an ESCAPE-d after we've already read the last data line, the terminal will just send us a blank line, and will be happy to do this forever.

There are two ways of solving this problem. One is to output (at the very beginning) some sort of "marker" to the terminal, e.g. "*** PSCREEN END OF MEMORY ***"; then, we can keep INPUTing until we get this marker line, at which point we know we're done. (We should also then erase the tag line so that subsequent PSCREENs won't run into it.)

Another solution is to ask the terminal itself. If we say

```
INPUT PROMPT="!![CHR(27)+'F'+CHR(27)+'a']";NAME=CURSORPOS
```

then the terminal will be sent an ESCAPE-F (HOME DOWN, i.e. go to the end of memory) and an ESCAPE-a. The ESCAPE-a will ask it to transmit information on the current cursor position, in the format "!&a888c999R", where the "!" is an escape character, the "888" is the column number, and the "999" is the row number. This string will be input into the

variable CURSORPOS. Then, the value of the expression

```
![STR(CURSORPOS,8,3)]
```

will be the row number of the bottom of the screen.

The old PSCREEN uses the first approach (write a marker), probably because it's more resilient; I suspect that some old terminal over some strange datacomm connection can't handle the ESCAPE-a sequence right.

In any event, reading the data from the screen isn't that hard. The question is: how can we output it to the printer?

As we showed in our previous discussion, it's quite hard to read data from a file into a variable. It's harder still to output the data from a variable to a file.

The solution lies in running CI.PUB.SYS with ;STDLIST= redirected, thus letting the :ECHO command output to a file rather than to the terminal. (This is much like doing file input by running CI.PUB.SYS with ;STDIN= redirected.) Here's what the full PSCREEN script actually looks like:

```
SETVAR PSCREENTERM "*** PSCREEN MARKER ***"
ECHO !PSCREENTERM
SETVAR PSCREENLINE 0
INPUT PSCREEN!PSCREENLINE;PROMPT="![CHR(27)+'H'+CHR(27)+'d']"
WHILE PSCREEN!PSCREENLINE <> PSCREENTERM DO
  SETVAR PSCREENLINE PSCREENLINE+1
  INPUT PSCREEN!PSCREENLINE;PROMPT="![CHR(27)+'d']"
ENDWHILE
CALC CHR(27)+"A"+CHR(27)+"K"   << clear the PSCREEN MARKER line >>
FILE PSCROUT;DEV=LP
RUN CI.PUB.SYS;PARM=3;INFO="PSCREENX";STDLIST=*PSCROUT
RESET PSCROUT
DELETEVAR PSCREEN@
```

Note that we're reading all the lines into variables called PSCREEN0, PSCREEN1, PSCREEN2, PSCREEN3, etc. These variables will then be read by the PSCREENX command file, which looks like this:

```
SETVAR PSCREENI 0
WHILE PSCREENI<PSCREENLINE DO
  ECHO ![PSCREEN!PSCREENI]
  SETVAR PSCREENI PSCREENI+1
ENDWHILE
```

There it is, in all its glory! Again, the PSCREEN program works just fine -- probably even better than these command files -- but this is just an example of the kind of things you can do.

One little glitch you'll run into with these command files is that the first line of every printout will read ":PSCREENX". That's because CI.PUB.SYS will echo its ;INFO= command to the ;STDLIST= file. For PSCREEN, this should be fairly harmless; however, what if you simply want to write the contents of a variable to a disc file without the echoing getting in the way?

The solution is this:

```
PURGE TEMPOUT,TEMP
BUILD TEMPOUT;NOCCTL;REC=-508,,V,ASCII;TEMP
FILE TEMPOUT,OLDTEMP;SHR;GMULTI;ACC=APPEND
RUN CI.PUB.SYS;INFO="ECHO !MYVAR";STDLIST=*TEMPOUT
FILE TEMPOUT,OLDTEMP
FILE DISCFILE;ACC=APPEND
PRINT *TEMPOUT;OUT=*DISCFILE;START=3
```

We run the CI and tell it to echo the variable MYVAR to a temporary file called TEMPOUT. Then we do a :PRINT command (a new feature of MPE/XL) that appends to DISCFILE the contents of TEMPOUT starting with record #3. Record #1 is CI.PUB.SYS's echo of the ":" prompt; record #2 is its echo of the "ECHO !MYVAR" command; record #3 is the actual contents MYVAR variable.

What a bother, and relatively slow, too (that's why we ran the CI only once in the PSCREEN script). A built-in MPE/XL FWRITE function would have been so much simpler...

#4. EXPRESSIONS AND PROGRAMS

One of the most interesting possibilities of the MPE/XL command interface has nothing to do with command files (or UDCs or job streams) at all. I've never seen it implemented before, so it might have a good deal of practical problems; however, I think that it has a lot of potential for power.

Consider a program that prints the contents of one of your specially-formatted data files. If it were a database, you could use QUERY, with its fairly sophisticated selection

conditions -- you could specify exactly what records you want to select.

However, if you're writing a special custom-made program, how can you let the user specify the records to be selected? There are 1,000 records in the file (17 pages at 60 lines per page), and the user only wants a few of them. If you don't put in some sort of selection condition, the user won't be happy; if you put in the ability to select on one particular field, I'll bet you that the user will start asking for selection on another field. What about ANDs? ORs? Arithmetic expressions (SALARY<>BASERATE+BONUSRATE)? Soon they'll be asking for you to write your own expression parser!

What you really want is a GENERALIZED EXPRESSION PARSER, usable by any subsystem that wants to have user-specified selection conditions (and user-specified output formats). You could tell it about the variables that you have defined -- e.g., define one variable for each field in the file, plus some other variables for some calculated values that the user may find handy. Then, you tell it to evaluate a user-supplied expression.

Think of all the various programs that could use this!

* V/3000 could have used this for the input field validity checks (rather than having its own parser);

* QUERY could have used this for the >FIND command (rather than having its own parser, which, incidentally, can't handle parenthesized expressions);

* MPE/V could have used it for the :IF command logical expressions;

* LISTLOG could have used it to let you select log records;

* QUERY could have used it to output expression values in >REPORTs (rather than have that silly assembly-language-style register mechanism);

* EDITOR or FCOPY could have implemented a smart string search mechanism (find all line that contain "ABC" OR "DEF").

HP could have saved itself man-years of extra effort, while at the same time standardizing those expression evaluators that exist AND implementing expression evaluation in a lot of places that need it! Not to mention the uses

that you and I could put it to!

The point here is that with MPE/XL you can -- in a way -- do this yourself. Take that file-reader-and-printer program of yours and prompt the user for a selection condition. Then, for each file record, use the HPCIPUTVAR intrinsic (or pass the COMMAND intrinsic a :SETVAR command) to set AN MPE/XL VARIABLE FOR EACH FIELD IN THE RECORD. Now, do a

```
:SETVAR SELECTIONRESULT expression_input_by_the_user
```

Finally, do an HPCIGETVAR to get the value of the SELECTIONRESULT variable; if it's TRUE, the record should be selected -- if it's FALSE, rejected.

In other words, you're using the :SETVAR commands expression handling to do the work for you. You set MPE/XL variables for all the fields in your record, and the user can then use those variables inside the selection condition. The condition can use all the MPE/XL functions -- =, <>, <, >, +, -, STR, POS, UPS, etc.; it can reference integer, string, or boolean variables. A sample run of the program might be:

```
:RUN SELFILE

SELFILE Version 1.5 -- this program prints selected records from
the PS010 KSAM file; please enter your selection condition:

   >UPS(STATUS)<>"XX" AND WORK_HOURS*HOURLY_SALARY>=10000
```

Meantime, the program is doing:

```
FOR each record from PS010 DO
  BEGIN
  :SETVAR STATUS value_of_status_field_file
  :SETVAR NAME value_of_name_field
  :SETVAR WORK_HOURS value_of_work_hours_field
  :SETVAR HOURLY_SALARY value_of_hourly_salary_field
  :SETVAR DEPARTMENT value_of_department_field
  ...
  :SETVAR SELECTIONRESULT &
         UPS(STATUS)<>"XX" AND WORK_HOURS*HOURLY_SALARY>=10000
  IF value of SELECTIONRESULT variable = TRUE THEN
    output the record;
  END;
```

(The :SETVAR commands in the pseudo-code should probably be calls to the HPCIPUTVAR intrinsic.)

There are several non-trivial problems with this approach:

* You're restricted to INTEGER, STRING, and BOOLEAN variables -- no dates, reals, etc.

* You're restricted to those functions that MPE/XL provides, which are rather limited (though fairly powerful).

* Most importantly, all those intrinsic calls will take some time! If you're reading through a 100,000 record file, you might encounter some serious performance problems.

As I said, to the best of my knowledge nobody's ever implemented this sort of facility -- for all I know, it may just not be practically feasible. However, I suspect that for quick-and-dirty query programs (and also input checking, output formatting, etc.) where performance is not a major consideration, it can be very powerful. You can use it to give a lot of control to the user, with very little programming effort on your own part.

CONCLUSION

The MPE/XL user interface is much more powerful and much more convenient than the "classic MPE" interface. (I didn't even mention some features, like multi-line :REDO, which are convenient indeed.) It lets you easily do many things that used to require a lot of effort; however, some key features are unfortunately missing.

Fortunately, with a little bit of ingenuity, even the apparently "impossible" can be achieved -- I'd be happy if all this paper did was let you know that there are possibilities to MPE/XL beyond those that are apparent at first glance. We HP programmers did some pretty amazing things with the limited capabilities that classic MPE offered us -- with MPE/XL, we should be able to write some very powerful stuff.

One thing that the new MPE/XL features should do is whet the appetites of all the poor people who still have to stick with MPE/V (or, heaven forbid, MPE/IV!) for some time in the future. After seeing all those wonderful things on the new machines, how can we bear to live with the old stuff?

There is actually a product out now (called Chameleon, from Taurus Software, Inc.) that implements MPE/XL functionality on MPE/V; VESOFT's own MPEX/3000 Version 2.0 release, tentatively scheduled for a June 1988 release, should do the same (in addition, of course, to all the other stuff that MPEX has always done -- fileset handling, %ALTFILE, new %LISTF modes, hook, etc.). MPEX Version 1.6 has, for the past year, already implemented the multi-line :REDO feature, both in MPEX, and in other programs, such as EDITOR, QUERY, TDP, QEDIT, etc.

VESOFT's STREAMX also implements many MPE/XL-like features (including variables, :WHILE loops, expressions, etc.) for job stream submission, an area unfortunately neglected by HP. Personally, I think that variable input, expression evaluation, input checking, etc. are even more useful at job stream SUBMISSION time than they are in session mode and at job stream execution time.

Finally, there are several other papers available about MPE/XL, all of which I can recommend highly. Jeff Vance & John Korondy of HP had the "Design Features of the MPE/XL User Interface" paper in the 1987 INTEREX Las Vegas proceedings; David T. Elward published the "Winning with MPE/XL" paper in the October and November 1988 HP Chronicles. Also, the MPE/XL Commands Manual actually has a lot of useful documentation on command files (including some very interesting MPE/XL Programming examples!) -- I've seen several versions, and it seems that the most recent ones have the most information. And, of course, the recently released "Beyond RISC!" book is an indispensable tool for anybody who deals or will be dealing with Precision Architecture machines.

Thanks to Rob Apgood of Strategic Systems, Inc. and Gavin Scott of American Data Industries for their input on this paper; thanks especially to Gavin for letting me test out all the examples on the computer in the two hours between the time I finished writing it and the time I had to Federal Express it up to BARUG.

Finally, any errors in this paper are NOT the fault of the author, but were rather caused by cosmic rays hitting the disc drives and modifying the data...