HP WORLD 2004
Solutions and Technology Conference & Expo

CI Programming For Stability

Jeff Vance, HP-vCSY
jeff.vance@hp.com
Hewlett-Packard

Outline
(read the notes too!)

- UDCs and scripts (parameters, entry points)
- Variables
- Expressions and functions
- I/O redirection and file I/O
- Error handling
- Script cleanup techniques
- Debugging
- Converting a quick'n'dirty script to near production quality
- Examples
- Appendix

Alternatives

- 3GL (C, COBOL, Java, Pascal, compiled Basic, etc.)
- 4GL (Speedware, Transact, Powerhouse, Visual Basic, etc.)
- Interpretive (CI, Basic, other scripting languages)

Common CI “programming” command

IF, ELSEIF, ELSE, ENDIF ESCAPE, RETURN	branching
WHILE, ENDWHILE	looping
ECHO, INPUT	terminal, console I/O, file I/O
SETVAR, DELETEVAR SHOWVAR	create/modify/delete/display a variable
ERRCLEAR	sets CI error variables to 0
RUN	invoke a program
XEQ	invoke a program or script
PAUSE	sleep; job synchronization
OPTION recursion	only way to get recursion in UDCs
# or COMMENT	comment

UDCs

- User Defined Command files (UDCs) - a single file that contains 1 or more command definitions, separated by a row of asterisks (***)

- Features:
  - simple way to execute several commands via one command
  - allow built-in MPE commands to be overridden
  - can be invoked each time the user logs on
  - require lock and (read or eXecute) access to the file
  - cataloged (defined to the system) for easy viewing and prevention of accidental deletion -- see SETCATALOG and SHOWCATALOG commands
  - can be defined for each user or account or at the system level
  - more difficult to modify since file is usually opened by users

Command files (scripts)

- Command file - a file that contains a single command definition

- Features:
  - similar usage as UDCs
  - searched for after UDCs and built-in commands using HPPATH
    - default path is: logon-group, PUB.logon-acct, PUB.SYS, ARPA.SYS
  - require read or eXecute access
  - easy to modify since file is only in use while it is being executed
  - very similar to unix scripts or DOS bat files

UDC / script comparisons

- Similarities:
  - ASCII, NOCCTL, numbered or unnumbered, max 511 byte record width
  - optional parameter line ok - max of 255 arguments
  - optional options, e.g. HELP, NOBREAK, RECURSION
  - optional body (actual commands)
    - no inline data, unlike Unix ‘here’ files :(
  - can protect file contents by allowing eXecute access-only security, i.e., denying read access

UDC / script comparisons (cont)

- Differences:
  - scripts can be variable record width files
  - UDCs require lock access, scripts don’t
  - script names can be in POSIX syntax, UDC filenames must be in MPE syntax
  - UDC name cannot exceed 16 chars, script name length follows rules for MPE and POSIX named files
  - EOF for a script is the real eof, end of a UDC command is one or more asterisks, starting in column one

UDC / script exit

- EOF -- real EOF for scripts, a row of asterisks (starting in column 1) for UDCs
- :BYE, :EOJ, :EXIT -- terminate the CI too, to use BYE or EOJ must be the root CI
- :RETURN -- useful for entry point exit, error handling, help text - jumps back one call level
- :ESCAPE -- useful to jump all the back to the CI, or an active :CONTINUE. In a job without a :CONTINUE, :escape terminates the job. Sessions are not terminated by :escape. Can optionally set CIERROR and HPCIERR variables to an error number

Recommendation

- UDCs provide a repository and are easier to locate, but they are more difficult to change after they have been cataloged. They are also more difficult to purge (deliberately or accidentally).
- Scripts can be located anywhere but are easier to maintain if they are kept in one or a few groups / directories. Scripts are easier to modify and delete.
- My experience has been to use scripts as my first choice and only use UDCs to override built-in MPE commands.
- Use ESCAPE rather than RETURN for script errors that demand user attention/intervention

Parameters

- Syntax: ParmName [= value]
  - supplying a value means the parameter is optional. If no value is defined the parameter is considered required.
  - max parm name is 255 bytes, chars A-Z, 0-9, “_”
  - max parm value is limited by the CI’s command buffer size (currently 511 characters)
  - all parm values are un-typed, regardless of quoting
  - Parms are separated by a space, comma or semicolon
  - default value may be a: number, string, !variable, ![expression], an earlier defined parm (!parm)
  - all parameters must be explicitly referenced in the UDC/script body, e.g. !parmname
  - the scope of a parm is the body of the UDC/script

Parameters (cont)

- all parameters are passed "by value", meaning the parm value cannot be changed within the UDC/script
- a parm value can be the name of a CI variable, thus it is possible for a UDC/script to accept a variable name, via a parm, and modify that variable's value, e.g.

```
SUM a, b, result_var
setvar !result_var !a + !b
*****
```

SUM is a UDC name

```
:SUM 10, 2^10, x
:showvar  x                         X = 1034

:setvar I  10
:setvar J  12
:SUM  i, j, x                       inside SUM: setvar x, i + j
:showvar  x                         X = 22
```

ANYPARM parameter

- all delimiters ignored
- must be last parameter defined in UDC/script
- only one ANYPARM allowed
- only way to capture user entered delimiters, without requiring user to quote everything
- example:

```
TELLT  user
ANYPARM msg = “”
# prepends timestamp and highlights msg text
tell !user; at !hptimef: ![chr(27)]&dB !msg

:TELLT  op.sys Hi,, what’s up; system seems fast!
FROM S68 JEFF.UI/3:27 PM: HI,, what’s up; system seems…
```

- `anyparm()` function is useful with ANYPARM parameters

Entry points

- simple convention for executing the same UDC/script starting in different “sections” or subroutines
- a UDC/script invokes itself recursively passing in the name of an entry (subroutine) to execute
- the script detects that it should execute an alternate entry and skips all the code not relevant to that entry.
- most useful when combined with I/O redirection, but can provide the appearance of generic subroutines
- benefits are: fewer script files to maintain, slight performance gain since MPE opens an already opened file faster, can use variables already defined in script
- UDCs need OPTION RECURSION to use multiple entry points

Entry points (cont)

- two approaches for alternate entries:
  - define a parm to be the entry point name, defaulting to the main part of the code, for example: “main”
  - the UDC/script invokes itself recursively in the main code, and may use I/O redirection here too
  - each entry point returns when done (via :RETURN command)

--------------------------- or ----------------------------------

  - test HPSTDIN or HPINTERACTIVE variable to detect if script/UDC has I/O redirected.
  - if TRUE then assume UDC/script invoked itself.
  - limited only to entry points used when $STDLIST or $STDIN are redirected
  - limited to a single alternate entry point, may not work well in jobs

Entry points (cont)

- generic approach:

```
PARM  p1 ...  entry=main                      # default entry is "main"
if "!entry" = "main" then
    ... initialize etc...
    xeq !HPFILE !p1, ... entry=go             # run same script, different
  entry
    ...  cleanup etc...
    return
elseif "!entry" = "go"  then...
    # execute the GO subroutine ...
    return
elseif "!entry" =  ...
    ...
endif
```

Entry points (cont)

- I/O redirection specific approach:

```
PARM  p1 …            # no “entry” parm defined
if HPSTDIN = “$STDIN” then
     # assume “main” entry -- initialize etc…
     xeq !HPFILE !p1, …  <somefile
     …  (cleanup etc…)
    return
else                  # no elseif since only 1 alternate
     # execute the entry to read “somefile”
     setvar eof FINFO(hpstdin, “eof”)
     …
     return
endif
```

Recommendations

- Comment all parameters and their expected and default values. Equally important for entry points since args may be used differently and input and/or output may have been redirected.
- Define good default parm values and allow some obvious value for the first parm ("?") to signify script-specific help. Sometimes an absent first parm should imply help text needs to be displayed.
- Choose parameter names which do not collide with the variable names in the script/UDC.
- Use "entry points" to make scripts more structured and for file I/O. The parameter based alternate entry approach is superior from a flexibility perspective since it works in all environments and is easily expanded.

CI variables

- 100 predefined "HP" variables* in MPE/iX release 7.0
- user can create and modify their own variables via :SETVAR
- variable types are: integer (signed 32 bits), Boolean and string (up 1024 characters)
- variable names can be up 255 alphanumeric alphanumeric and "_" (cannot start with number)
- predefined variable cannot be deleted, some allow write access
  - :SHOWVAR @ ; HP -- shows all predefined variables
- can see user defined variables for another job/session (need SM)
  - :SHOWVAR @ ; job=#S or #Jnnn
- the bound( ) function returns true if the named variable exists
- variables deleted when job / session terminates
- :HELP variables and :HELP VariableName

Predefined variables

- HPAUTOCONT - set TRUE causes CI to behave as if each command is protected by a :continue.
- HPCMDTRACE - set TRUE causes UDC / scripts to echo each command line as long as OPTION NOHELP not specified. Useful for debugging.
- HPCPUMSECS - tracks the number of milliseconds of CPU time used by the process. useful for measuring script performance.
- HPCWD - current working directory in POSIX syntax.
- HPDATETIME - contains the date/time in CenturyYearMonthDateHourMinuteSecondMicrosecond format.
- HPDOY - the day number of the year from 1..365.
- HPFILE - the name of the executing script or UDC file.
- HPINTERACTIVE - TRUE means $STDIN and $STDLIST do not form an interactive pair, useful to test if it is ok to prompt the user.
- HPLASTJOB - the job ID of the job you most recently streamed, useful for a default parm value in UDCs that alter priority, show processes, etc.

Predefined variables (cont)

- HPLASTSPID - the $STDLIST spoolfile ID of the last job streamed, useful in :print !hplastspid.out.hpspool
- HPLOCIPADDR - IP address for your system.
- HPMAXPIN - the maximum number of processes supported on your system.
- HPPATH - list of group[.acct] or directory names used to search for script and program files
- HPPIN - the Process Identification Number (PIN) for the current process.
- HPPROMPT - the CI's command prompt, useful to contain other info like: !!HPCWD, !!HPCMDNUM, !!HPGROUP, etc.
- HPSPOOLID - the $STDLIST spoolfile ID -- if executing in a job.
- HPSTDIN - the filename for $STDIN, useful in script "subroutines" where input has been redirected to a disk file
- HPSTREAMEDBY - the "Jobname,User.Acct (jobIDnum)" of the job/session that streamed the current job.
- HPUSERCAPF - formatted user capabilities, useful to test if user has desired capability, e.g. if pos("SM",hpusercapf) > 0 then

Recommendations

- Define your own variables to not appears as HP variables and chose unique names, e.g. I, J, K, NAME, TEMP are not meaningful names for any variable which survives the scope of its creation. NUM_CUSTOMERS, PAYROLL_FILENAME, etc. are more descriptive names.
- Don't define parameters with the same names as your variables and vice-versa -- just not worth the extra confusion.
- In general don't use HPAUTOCONT since it can mask errors in your script/UDC.
- Be careful using the date/time variables. Remember your script could be running when the clock just passes midnight, or the month or year just advances.
- Use formatted vs. numeric variables. E.g. HPUSERCAPF is preferred to HPUSERCAP.
- Use HPFILE to avoid hard-coding the name of your script.
- Use HPINTERACTIVE to avoid prompting in a job.

CI functions

- functions are invoked by their name, accept zero or more parms and return a value in place of their name and arguments
- file oriented functions:
  - BASENAME, DIRNAME, FINFO, FSYNTAX, FQUALIFY
- string parsing functions:
  - ALPHA, ALPHANUM, DELIMPOS, DWNS, EDIT, LEN, LFT, LTRIM, NUMERIC, PMATCH, POS, REPL, RHT, RPT, RTRIM, STR, UPS, WORD, WORDCNT, XWORD
- conversion functions:
  - CHR, DECIMAL, HEX, OCTAL, ORD
- arithmetic functions
  - ABS, MAX, MIN, MOD, ODD

CI functions (cont)

- job/process functions:
  - JINFO, JOBCNT, PINFO
- misc. functions:
  - ANYPARM, BOUND, INPUT, SETVAR, TYPEOF
- new to 7.5: devinfo, volinfo, spoolinfo:
  Return info about devices, volumesets, and spoolfiles. See Jazz for details.
- new to 7.5: user defined functions:
  Function name is a filename. HPPATH is used to locate the function file. RETURN command accepts an expression used as the function return. HPRESULT variable holds the function return.

  Examples:
  - if myFunc(a, b, c) then …
  - if compare(result) < compare(last) then …
  - if get_user_data(start, end) = 0 then …
  - if get_device_info(ldev, "state") = "READY" then …

CI expressions

- an expression is any variable, constant or function with or without an operator, e.g.:
  - MYVAR, "a"+"b", x^10*y/(j mod 6),
  - false, (x > lim) or (input() ="y")
- 5 commands accept implicit expressions:
  :calc, :if, :elseif, :setvar, :while
- ![expression] can be used explicitly in any command:

```
:build afile; rec=-80; disc= ![100+varX]
:build bfile; disc= ![ finfo("afile","eof")*3]     # file b is 3x larger
```

- examples:

```
:print ![input("File name? ")]
:setvar reply ups(rtrim(ltrim(reply)))
```

Partial expression evaluation

- The CI evaluates the minimal amount of a Boolean expression needed to determine the end result. For example:

```
if true or  x                    # "x" side not evaluated
if false and  x                  # "x" side not evaluated
if bound(z) and z > 1 then # if "z" not defined it won't be referenced
```

- Partial evaluation can cause some mysterious results
  - CI scripts may run differently in an MPEX environment since (last I heard) MPEX does not support partial evaluation. In this case break up complex expressions.

File I/O

- why not use INPUT in WHILE to read a flat file?, e.g.:

```
while not eof do
    input varname  < filename
endwhile
```

- three main alternatives:
  - write to (create) and read from a MSG file via I/O redirection
  - use :PRINT and I/O redirection to read file 1 record at a time
  - use entry points and I/O redirection
- MSG files work because each read is destructive, so when INPUT <file reads the 1st record it automatically gets the next record.
- PRINT works because start and end record numbers can be selected.
- once in an entry point where I/O has been redirected, you can easily read a file.

File I/O - MSG file

```
PARM fileset=./@
# This script reads LISTFILE,6 output and measures CPU millisecs
# using a MSG file
setvar savecpu hpcpumsecs
errclear
file msg=/tmp/LISTFILE.msg; MSG
continue
listfile !fileset,6 >*msg
if hpcierr = 0 then
   # read listfile names into a variable
   setvar cntr setvar(eof, finfo('*msg', "eof"))
   while setvar(cntr, cntr-1) >= 0 do
       input rec <*msg
   endwhile
endif
echo ![hpcpumsecs - savecpu] msecs to read !eof records.
deletevar cntr, eof, rec
```

```
:readmsg
259 msecs to read 22 records

:readmsg @.pub.sys
15845 msecs to read 1515
```

File I/O - :print

```
PARM fileset=./@
# This script reads a file produced by LISTFILE,6 and measures CPU msecs
# using PRINT as an intermediate step
setvar savecpu hpcpumsecs
errclear
continue
listfile !fileset,6 > lftemp
if hpcierr = 0 then
   # read listfile names into a variable
   setvar cntr 0
   setvar eof finfo('lftemp',"eof")
   while setvar(cntr, cntr+1) <= eof do
      print lftemp; start=!cntr;end=!cntr > lftemp1
      input rec <lftemp1
   endwhile
endif
echo ![hpcpumsecs - savecpu] msecs to read !eof records.
deletevar cntr,eof,rec
```

```
:readprnt
735 msecs to read 22 records
3 times slower than MSG files

:readprnt @.pub.sys
74478 msecs to read 1515 recs
over 4 times slower than MSG files!
```

File I/O - entry points

```
PARM fileset=./@, entry="main”
# This script reads a file produced by LISTFILE,6 and measures CPU msecs
# using entry points and script redirection
if "!entry" = "main" then
   setvar savecpu hpcpumsecs
   errclear
   continue
   listfile !fileset,6 > lftemp
   if hpcierr = 0 then
      xeq !hpfile !fileset entry=read <lftemp
   endif
   echo ![hpcpumsecs - savecpu] msecs to read !eof records.
   deletevar cntr,eof,rec
   purge lftemp;temp
   return
```

. . . (continued on next slide)

File I/O - entry points (cont)

```
else
   # read listfile names into a variable
   setvar cntr setvar(eof, finfo(hpstdin, "eof"))
   while setvar(cntr,cntr-1) >= 0 and setvar(rec, input()) <>
chr(1) do
   endwhile
   return
endif
```

:readntry
90 msecs to read 22 records.
---> Almost 3 times faster than MSG files
---> 8 times faster than the PRINT method!

:readntry @.pub.sys
2400 msecs to read 1515 records.
---> Over 6 times faster than MSG files
---> 31 times faster than using PRINT!

Recommendations

- Use variable names naturally (implicitly) – no explicit referencing unless necessary.
- Use the more powerful string parsing functions (word, xword, wordcnt, delimpos, edit) where possible.
- Enter :help functions and see if there are any surprises.
- Recognize partial evaluation, test the “skipped” clauses.
- Use “entry points” to make scripts more structured and for file I/O.
- Use MSG files for simple or one-time tasks, or for reading small files.
- Always, always write comments and log changes.
- Assume your quick’n’dirty script will stay in production longer than you!

Error handling

- use HPAUTOCONT variable judiciously. This is better:

```
continue
command
if hpcierr > 0 then
    echo something…
    return   -- or --  escape
endif …
```

- RETURN vs. ESCAPE
  - :return goes back ONE level
  - :escape goes back to the CI level in a session, to an active CONTINUE, or can abort a job
- HPCIERRMSG - variable contains the error text for the value of CIERROR JCW / variable
- :ERRCLEAR - sets HPCIERR, CIERROR, HPFSERR, HPCIERRCOL variables to zero

Cleanup

- delete variables “local” to the UDC / script
  - :deletevar _”prefix”_@
- purge scratch files
- reset “local” file equations
- don’t do the above if still debugging!
- better to build in a way to preserve files, variables, etc. on the fly
  - use a central cleanup “entry” routine
  - use a variable to control the cleanup related commands

Debugging

- Some common problems:
  - syntax error (unmatched parenthesis), variable name typo, reliance on a var that has not been initialized, hitting eof, using an HFS file for I/O redirection and then referencing FINFO(hpstdin) -- CI bug!, entry name typo (case sensitive!), off-by-one on loop counters, unexpected user input, re-using the same var in two places that are executed together (popular in entry points), reading from terminal but $stdin is already redirected, a skipped portion of an expression or skipped commands now being executed with different data…
- Trickier problems to find:
  - echoing a literal ">" without escaping,word() by index but index out of bounds, "array" index increment and reference in same loop, unmatched endwhile or endif, creating files that could contain CI metachars, date calculations that cross day, month, year boundaries…

Quick'n'dirty à production

- Real example taken from a request on 3000-L to report all program files with PM capability.
- Need to consider NM and CM program files.
- Wanted a free solution.

The quick solution

```
purge progf
purge versf
build progf;msg;rec=-80,,f,ascii
build versf;msg;rec=-80,,f,ascii
file x=progf,old
file y=versf,old
listfile @.@.@,6; seleq=[code=PROG]    >*x
listfile @.@.@,6; seleq=[code=NMPRG] >>*x
setvar peof finfo('*x','eof')
while setvar(peof,peof-1) >= 0 do
   input progname <*x
   version !progname >*y
   setvar veof finfo('*y','eof')
   while setvar(veof,veof-1) >= 0 do
      input vrec <*y
      if pos("CAPABILITIES:",vrec)=1 or pos("CAP:",vrec)=1 then
         setvar veof 0
         if pos("PM",xword(vrec,':')) > 0 then
            echo !progname has PM capabilty
         endif
      endif
   endwhile
endwhile
```

What’s wrong?

- Let’s add some comments in the beginning and accept a parameter so the user can specify which files they are interested in.
- Let’s also start adding some error handling

```
PARM fileset=@.@.@
# Reports NM and CM program files which have PM capability. Since two
# LISTFILEs are done to get the full list of NMPRG and PROG files the final output
# will not be in alphabetic order.  Note: HFS syntax is not supported by VERSION.
purge progf
purge versf
build progf;msg;rec=-80,,f,ascii
build versf;msg;rec=-80,,f,ascii
file x=progf,old
file y=versf,old
continue
listfile !fileset, 6;seleq=[code=NMPRG]  >*x
continue
listfile !fileset, 6;seleq=[code=PROG]   >>*x
...
```

Pass two...

- Let’s add some real error handling and make the output more user friendly

```
PARM fileset=@.@.@
# (same comments in the beginning as previous version...)
purge progf >$null
purge versf >$null
(same BUILD and FILE eq as before...)
errclear
continue
listfile !fileset, 6;seleq=[code=NMPRG]  >*x
if hpcierr <> 0 then
   echo !hpcierrmsg
   return
endif
continue
listfile !fileset, 6;seleq=[code=PROG]   >>*x
if hpcierr <> 0 then
   ditto...
...
```

Pass three...

- Let's try to get the error handling nailed...

```
...
errclear
continue
listfile !fileset,6;seleq=[code=NMPRG] >*x
if hpcierr > 0 then
    # print progf which contains the error
    print *x
    return
elseif hpcierr < 0 then
    # hide warning and erase the contents of progf (the warn text).
    print *x >$null
    errclear
endif
continue
listfile !fileset,6;seleq=[code=PROG] >>*x
if hpcierr > 0 then
    # got an error, maybe the progf file is full?  Cannot display progf as
    # above since it could contain NMPRG files. Also cannot print a subset of
    # progf since FPOINT fails on MSG files.
    echo !hpcierrmsg
    return
elseif hpcierr < 0 then
    # It would be nice to remove the last two records from progf, but
    # since it is a MSG file we cannot use :PRINT ;start=eof to do this.
    # Ignore the warn but remember the warn text is in progf!
endif
...
```

Production version

```
PARM fileset=@.@.@
# Reports NM and CM program files which have PM capability. Since two LISTFILEs
# are done to get the full list of NMPRG and PROG files the final output will
# not be in alphabetic order.  Note: HFS syntax is not supported by VERSION.
if word(fsyntax('!fileset')) = "POSIX" then
   echo POSIX syntax names are not supported by the VERSION utility
   return
endif
# build the MSG files to hold LISTFILE and VERSION output
purge progf >$null
purge versf >$null
build progf;msg;rec=-80,,f,ascii
build versf;msg;rec=-80,,f,ascii
file x=progf,old
file y=versf,old

# first list NM program files to a MSG file
errclear
continue
listfile !fileset,6;seleq=[code=NMPRG] >*x
if hpcierr > 0 then
   # print progf which contains the error
   print *x
   return
elseif hpcierr < 0 then
   # hide warning and erase the contents of progf (the warn text).
   print *x >$null
   errclear
endif
...
```

Production version (cont)

```
# Now append CM program files to the same MSG file (progf).
# This means that the output will not be in alphabetic order!
continue
listfile !fileset,6;seleq=[code=PROG] >>*x
setvar peof finfo('*x','eof')
if hpcierr > 0 then
    # got an error, maybe the progf file is full?  Cannot display progf as
    # above since it could contain NMPRG files. Also cannot print a subset of
    # progf since FPOINT fails on MSG files.
    echo !hpcierrmsg
    return
elseif hpcierr < 0 then
    # It would be nice to remove the last two records from progf, but
    # since it is a MSG file we cannot use :PRINT ;start=eof to do this.
    # Ignore the warn but remember the warn text is in progf!
    setvar peof peof-2
endif

echo
echo The following programs (out of !peof) have PM capability:
echo
setvar pcnt 0
errclear
```

(… the read WHILE loop follows…)

Production version (cont)

```
# read the combined LISTFILE,6 output and pass each filename to VERSION
while setvar(peof,peof-1) >= 0 do
    input progname <*x
    setvar progname rtrim(progname)
    continue
    version !progname >*y
    if hpcierr = 0 then
        setvar veof finfo('*y','eof')
        while setvar(veof,veof-1) >= 0 do
            input vrec <*y
            if pos("CAPABILITIES:",vrec) = 1 or pos("CAP:",vrec) = 1 then
                setvar veof 0
                if pos("PM",xword(vrec,':')) > 0 then
                    echo    !progname
                    setvar pcnt pcnt+1
                endif
            endif
        endwhile
    endif
endwhile
echo
echo !pcnt programs have PM
```

PM program check output

:progcap @.@.vance

```
The following programs (out of 22) have PM capability:
   LARSPING.PUB.VANCE
   LINKEDDB.PUB.VANCE
   MOVER.PUB.VANCE
   RYDER.PUB.VANCE
   SWINVENP.PUB.VANCE
   JINFO.TEST.VANCE
   JOBINFO.TEST.VANCE
   SIUDBP.TMP.VANCE
   SIUDBP.TMP1.VANCE
   SIUDBP.TMP2.VANCE
   SIUDBP.UDCS.VANCE
11 programs have PM
```

Examples

- We start off with some simple, but perhaps still novel examples.
- A few more complex examples are given with emphasis on techniques for getting more out of MPE.
- There are many more examples at the end of the Appendix.
- Many of the longer examples are on Jazz
  http://jazz.external.hp.com/src/scripts/

Simple examples

display last N records of a file (no process creation)

-
  ```
  PARM file, last=12
  print !file; start= -!last
  ```
  "Tail" script

display CI error text for a CI error number

-
  ```
  PARM cierr= !cierror
  setvar save_err  cierror
  setvar cierror  !cierr
  showvar HPCIERRMSG
  setvar cierror  save_err
  deletevar save_err
  ```
  "Cierr" script

alter priority of job just streamed -- great for online compiles ;-)

-
  ```
  PARM job=!HPLASTJOB; pri=CS
  altproc job=!job; pri=!pri
  ```
  "Altp" script

HP WORLD 2004 Solutions and Technology Conference & Expo

Brief file, group, user, dir listings

-
  ```
  PARM fileset=./@                    "LF"
  listfile !fileset,6
  ```

-
  ```
  PARM group=@                        "LG"
  listgroup !group; format=brief
  ```

-
  ```
  PARM user=@                         "LU"
  listuser !user; format=brief
  ```

-
  ```
  PARM dir=./@                        "LD"
  setvar _dir  "!dir"
  if delimpos(_dir, "./") <> 1 then
     # convert MPE name to POSIX name
     setvar _dir  dirname( fqualify(_dir)) + "/" + basename(_dir)
  endif
  listfile !_dir, 6; seleq=[object=HFSDIR] ;tree
  ```

Displaying spoolfiles

- PRINTSP script:

```
PARM  job=!HPLASTJOB
# Prints spoolfile for a job, default is the last job you streamed
if  "!job" = "" then
     echo No job to print
     return
endif
setvar hplastjob "!job"
if hplastspid  = "" then
     echo No $STDLIST spoolfile to print for "!job".
     return
endif
print !HPLASTSPID.out.hpspool
```

- :stream scopejob

```
#J324
:printsp
:JOB SCOPEJOB,MANAGER.SYS,SCOPE.
 Priority = DS; Inpri = 8; Time = UNLIMITED. . .
```

Powerfail script

- UPS configuration file, UPSCNFIG.PUB.SYS):

Contents:

```
powerfail_message_routing = all_terminals
powerfail_low_battery     = keep_running
powerfail_command_file    = prodshut.opsys.sys
powerfail_grace_period    = 300
```

- PRODSHUT.OPSYS.SYS script example:

```
warn @; Powerfail detected by UPS. Orderly shutdown BEGIN…
warn @; ***** Please logoff immediately! *****
if jobcnt("prod1J,usr.acct", jobID) > 0 then
    stream hipriJ
    pause 60; job=!hplastjob
    abortjob !jobID
endif
errclear
pause 180; job=@s
if cierror = 9032 then
    warn @;System going down in 2 minutes!
    pause 120
endif
shutdown
```

Testing remote command execution

```
ANYPARM cmd
# Script that executes a command in a remote session and returns the
# CIERROR and HPCIERR values for that command back to the local
# environment.
purge rmstatus; temp >$null
build rmstatus;rec=-80,,f,ascii; temp
remote file rmstatus=rmstatus:$back,oldtemp
continue
remote !cmd
remote echo setvar cierror !!cierror    >*rmstatus
remote echo setvar hpcierr !!hpcierr >>*rmstatus
xeq rmstatus
echo remote CIERROR=!cierror, remote HPCIERR=!hpcierr
```

```
:rem  listfile 4abc,2
First character in file name not alphabetic.(CIERR 530)
remote CIERROR=530, remote HPCIERR=530
```

Synchronize jobs

```
!JOB jobZero,…
!limit +2
!stream job1
!pause job=!hplastjob
!stream job2
!errclear
!pause  600, !hplastjob
!if hpcierr = -9032 then
!    tellop Job ”!hplastjob” has exceeded the 10 minute limit
!    eoj
!endif
!stream job3
!pause job=!hplastjob; WAIT
!input  reply, “’Reply ‘Y’ for !hplastjob”;  readcnt=1; CONSOLE
!if dwns(reply) = “y” then
   . . .
```

Parsing HPPATH

```
setvar x 0
while setvar(token, &
        word("!hppath",",; ",setvar(x, x+1))) <> "" do
   if delimpos(token,"/.") = 1 then
      # we have a POSIX path element

   else
      # we have an MPE path element

   endif
endwhile
```

- Why was HPPATH explicitly referenced?

“Where” script output

:where @sh@

```
SHOWME                  USER    UDC in SYS52801.UDC.SYS
SH                      SYSTEM  UDC in HPPXUDC.PUB.SYS
SH.PUB.VANCE            NMPRG
SHOWVOL.PUB.VANCE       script
BASHELP.PUB.SYS         PROG
HSHELL.PUB.SYS          script
PUSH.SCRIPTS.SYS        script
RSH.HPBIN.SYS           NMPRG
SH.HPBIN.SYS            NMPRG
/bin/csh                NMPRG
/bin/ksh                symlink  --> /SYS/HPBIN/SH
/bin/remsh              symlink  --> /ENM/PUB/REMSH
/bin/rsh                symlink  --> /SYS/HPBIN/RSH
/bin/sh                 symlink  --> /SYS/HPBIN/SH
```

Appendix

- CI limits
- Recent CI enhancements
- Redo/do features
- COMMAND and HPCICOMMAND inrtrinsics
- More on UDCs and scripts
- More on CI variables, including compound variables and “arrays”
- Expressions, JINFO, JOBCNT, and PINFO CI functions
- More on I/O redirection
- More examples...

CI limits

- command buffer — 511 bytes
  - applies to interactive, batch, UDCs, scripts, COMMAND and HPCICOMMAND intrinsics, NM and CM
  - CM command parms limited to 255 bytes due to MYCOMMAND intrinsic, eg. info= string

nested IFs and WHILEs	100
nested UDCs and scripts	30 each
length of string variable value	1024 bytes
length of CI variable name	255 bytes
max number of CI variables	10,800 (approx)
typical number of CI variables	8,300 (approx)
length of UDC name	16 bytes
length of script name	255 bytes
max number of UDC/script parms	255
length of user function name*	255 bytes

“Recent” CI enhancements

- extended POSIX filename characters
- new CI functions: anyparm, basename, dirname, fqualify, fsyntax, jobcnt, jinfo, pinfo, wordcnt, xword
- new CI variables: hpdatetime, hpdoy, hphhmmssmmm, hpleapyear, hpmaxpin, hpyyyymmdd
- new CI commands: abortproc, newci, newjobq, purgejobq, shutdown
- enhanced commands: INPUT from console, FOS store-to-disk, :SHOWVAR to see another job/sessions’ variables, :COPY to= a directory, :ALTJOB HIPRI and jobq=, :LIMIT +-N
- :HELP shows all CI variables, functions, ONLINEINFO, NEW
- user functions, e.g. if myFunc(a, true,10) > b then ...

Redo

- delete a word
  - dw, >dw, dwddw, dwiXYZ
- delete up to a special character
  - d., d/, d*, d/iXYZ, d.d
- delete to end-of-line
  - d>
- delete two or more non-adjacent characters
  - d d
- upshift/downshift a character or word
  - ^, ^w, v, vw, >^, >v, ^>, v>
- append to end-of-line
  - >XYZ
- replace starting at end of line
  - >rXYZ
- change one string to another
  - c/ABCD/XYZ, c:123::
- undo last or all edits
  - u or u twice in a row
- available in CI, VOLUTIL, STAGEMAN, DEBUG others...

COMMAND intrinsic

- COMMAND is a programmatic system call (intrinsic)
  syntax: COMMAND (cmdimage, error, parm)
- implemented in native mode (NM, PA-RISC mode)
- use COMMAND for system level services, like:
  - building, altering, copying purging a file
- no UDC search (a UDC cannot intercept “cmdimage”)
- no command file or implied program file search
- returns command error number and error location (for positive parmnum), or file system error number for negative parmnum

HPCICOMMAND intrinsic

- HPCICOMMAND is an intrinsic
  syntax: HPCICOMMAND (cmdimage,error,parm [,msglevel])
- implemented in native mode (NM, PA-RISC mode)
- use HPCICOMMAND for a “window” to the CI, e.g.:
  - providing a command interface to a program, “:cmdname”
- UDCs searched first
- command file and implied program files searched
- returns command error number and error location or file system error number.
- Msglevel controls CI errors/warnings -- similar to the HPMSGFENCE variable

UDCs vs. scripts

- option logon
  - UDCs only (a script can be executed from an “option logon” UDC)
  - logon UDCs executed in this order:
    - 1. System level 2. Account level 3. User level (opposite of the non-logon execution order!)

- CI command search order:
  - A. UDCs (1. User level 2. Account level 3. System level)
    - thus UDCs can override built-in commands
  - B. built-in MPE commands, e.g. LISTFILE
  - C. script and program files. HPPATH variable used to qualify unqualified filenames
  - :XEQ command allows script to be same name as UDC or built-in command, e.g. :xeq listf.scripts.sys

UDCs vs. scripts (cont.)

- performance
  - logon time:
    9 UDC files, 379 UDCs, 6050 lines: 1/2 sec.

    most overhead in opening and cataloging the UDC files
    - to make logons faster remove unneeded UDCs
  - execution time:
    identical (within 1 msec) for simple UDCs vs scripts, however:
    - factorial script:
      :fac 12   157 msec
    - factorial UDC (option recursion):
      :facudc 12   100 msec
    - file close logging impacts performance for scripts more since they are opened/closed for each invocation

UDCs vs. scripts (cont.)

- maintenance / flexibility / security
  - SETCATALOG opens UDC file, cannot edit without un-cataloging file, but difficult to accidentally purge UDC file
  - UDC commands grouped together in same file, easier to view and organize
  - UDC file can be lockword protected but users don't need to know lockword to execute a UDC

  - scripts opened while being executed (no cataloging), can be purged and edited more easily than UDCs
  - scripts can live anywhere on system. Convention is to place general scripts in a common location that grants read or eXecute access to all, e.g. "XEQ.SYS" group
  - if script protected by lockword then it must be supplied each time the script is executed

UDC search order

File:UDCUSER.udc.finance

1. Invoke UDCC, which calls UDCA with the argument “ghi”
2. UDCA is found, starting after the UDCC definition (option NOrecursion default)
3. ~~The line “p1=ghi” is echoed~~

4. Invoke UDCB, which calls UDCA passing the arg “def”. The recursion option causes the first UDCA to be found. This calls UDCC and follows the path at step 1 above
5. The line “p1=def” is echoed

```
UDCA  p1 = abc
option NOrecursion
udcC !p1
***
UDCB  p1 = def
option recursion
udcA !p1
***
UDCC p1 = ghi
udcA !p1
***
UDCA p1 = xyz
echo  p1=!p1
***
```

Script search order

- scripts and programs are searched for after the command is known not to be a UDC or built-in command
- same order for scripts and for program files
- fully or partially qualified names are executed without qualification
- unqualified names are combined with HPPATH elements to form qualified filenames:
  - first match is executed – could be a script, could be a program file
  - filecode = 1029, 1030 for program files
  - EOF > 0 and filecode in 0..1023 for script files
  - to execute POSIX named scripts with HPPATH qualification, a POSIX named directory must be present in HPPATH

UDC file layout

filename: AUDC.PUB.SYS

header:	**UDCcommandname** [parm1] [p2 [= value]] [**ANYPARM** parm4 [= value]] [**OPTION** option_list]
body:	any MPE command, UDC or script (option list or option recursion supported in body too)
end-of-UDC:	********** (end of this command definition)
header:	**NextUDCcommand** [parm1] [**PARM** P2, P3 = value] [**OPTION** option _list]
body:	any MPE command etc…

Script file layout

filename: PRNT.SCRIPTS.SYS

header:

```
[ PARM  parm1, parm2  [= value ] ]
[ ANYPARM parm3 [ = value ] ]
[ OPTION  option_list ]
```

body:

```
any MPE command, UDC or script
(:option list or :option recursion supported in body too)
```

eof

filename: LG.SCRIPTS.SYS

header:

```
PARM ...
OPTION  nohelp ...
any MPE command etc...
```

body:

Variable scoping

- all CI variables are job/session global, except the following: HPAUTOCONT, HPCMDTRACE, HPERRDUMP, HPERRSTOLIST, HPMSGFENCE, which are local to an instance of the CI
- thus it is easy to set "persistent" variables via a logon UDC
- need care in name of UDC and script "local" variables to not collide with existing job/session variables
  - _scriptName_varname -- for all script variable names. Use:deletevar _scriptName_@ at end of script
  - Can create unique variable names by using !HPPIN, !HPCIDEPTH, !HPUSERCMDEPTH as part of the name, e.g.
    :setvar _script_xyz_!hppin , value
- save original value of some "environment" variables
  - :setvar _script_savemsgfence hpmsgfence
    :setvar hpmsgfence 2

Variable referencing

- two ways to reference a variable:
  - explicit -- !varName
  - implicit -- varName
- some CI commands expect variables (and expressions) as their arguments, e.g.
  - :CALC, :IF, :ELSEIF, :SETVAR, :WHILE
  - use implicit referencing here, e.g.
    :if (HPUSER = "MANAGER") then
- most CI commands don't expect variable names (e.g. BUILD, ECHO, LISTF)
  - use explicit referencing here, e.g.
    :echo You are logged on as: !HPUSER.!HPACCOUNT
  - note: all UDC/script parameters must be explicitly referenced
- all CI functions accept variable names, thus implicit referencing works
  - :while JINFO (HPLASTJOB, "exists") do… better than ...
    :while JINFO ("!HPLASTJOB", "exists") do

Explicit referencing - !varname

- processed by the CI early, before command name is known
  - can cause hard-to-detect bugs in scripts - array example
- lose variable type -- strings need to be quoted, e.g..
  “!varName”
- **!!** (two exclamation marks) used to “escape” the meaning of “!”, multiple “!’s” are folded 2 into 1
  - even number of “!” --> don’t reference variable’s value
  - odd number of “!” --> reference the variable’s value
- useful to convert an ASCII number to an integer, e.g.

```
setvar int “123”        or        input foo, “enter a number”
 if  !int > 0 then ...             if  !foo = 321 then ...
```

- the only way to reference UDC or script parameters
- the only way for most CI commands to reference variables

Implicit referencing - just varname

- evaluated during the execution of the command -- later than explicit referencing
- makes for more readable scripts
- variable type is preserved -- no need for quotes, like: "!varname"
- only 5 commands accept implicit referencing: CALC, ELSEIF, IF, SETVAR, WHILE -- all others require explicit referencing
- all CI function parameters accept implicit referencing
- variables inside ![expression] may be implicitly referenced
- performance differences:
  - "!HPUSER.!HPACCOUNT" = "OP.SYS" 4340 msec
  - HPUSER + "." + HPACCOUNT = "OP.SYS" 4370 msec
  - HPUSER = "OP" and HPACCOUNT = "SYS" 4455 msec*

(*with user match true)

I prefer the last choice since many times :IF will not need to evaluate the expression after the AND

Compound variables

• :setvar a “!!b”	# B is not referenced, 2!’s fold to 1
• :setvar b “123”	
• :showvar a, b	A=“!b” B=123
• :echo b is !b, a is !a	b is 123, a is 123
• :setvar a123 “xyz”	
• :echo Compound var "a!!b": !"a!b”	Compound var "a!b": xyz

- :setvar J 2
  :setvar VAL2 “bar”
  :setvar VAL3 “foo”

• :calc VAL!J	bar
• :calc VAL![J]	bar
• :calc VAL![decimal(J)]	bar
• :calc VAL![setvar(J,J+1)]	foo

Variables arrays

- simple convention using standard CI variables

-

varname0	= number of elements in the array
varname1…varnameN	= array elements, 1 .. !varname0
varname!J	= name of element J
!"varname!J"	= value of element J

- :showvar buffer@

```
BUFFER0 = 6
BUFFER1 = aaa
BUFFER2 = bbb
BUFFER3 = ccc
BUFFER4 = ddd
BUFFER5 = eee
BUFFER6 = fff
```

Variable array example

- centering output:

“Center” script

```
PARM count=5
setvar cnt 0
while setvar(cnt,cnt+1) <= !count do
   setvar string!cnt,input("Enter string !cnt: ")
endwhile
setvar cnt 0
while setvar(cnt,cnt+1) <= !count do
   echo ![rpt(" ",39-len(string!cnt))]!"string!cnt”
endwhile
```

```
:center

  Enter string 1: The great thing about Open Source
  Enter string 2: software is that you can
  Enter string 3: have any color
  Enter string 4: "screen of death”
  Enter string 5: that you want.


        The great thing about Open Source
            software is that you can
                 have any color
                "screen of death”
                  that you want.
```

Filling variables arrays -- wrong!

- example 1: # array name is “rec”

```
setvar j 0
setvar looping true
while looping do
    input name, “Enter name “
    if name = “” then
        setvar looping false
    else
        setvar j j+1
       setvar rec!j  name
    endif
endwhile
setvar rec0  j
```

- :xeq exmpl1
    - infinite loop!, won’t end until <break>

Filling variables arrays (cont)

- example 2:

```
setvar j 0
setvar looping true
while looping do
      setvar NAME  “ ”
    input name, “Enter name “
    if name = “” then
        setvar looping false
    else
        setvar j j+1
      setvar rec!j  name
    endif
endwhile
setvar rec0  j
```

- <u>:xeq exmpl2   <datafile</u>   (datafile has 20 text records)

  (“enter name” prompt shown 20 times snipped…)

```
End of file on input. (CIERR 900)
 input name, "enter name “
Error executing commands in WHILE loop. (CIERR 10310)
```

Filling variables arrays (cont)

- example 3:

```
setvar j 0
if HPINTERACTIVE then
    setvar prompt  “’Name = ‘”
    setvar limit 2^30
      setvar test  ‘name= “” ‘
else
    setvar prompt “”
    setvar limit FINFO (HPSTDIN, ”eof”)
      setvar test “false”
endif
while (j < limit) do
    setvar name  “”
    input name , !prompt
    if  !test  then
        setvar limit 0                # exit interactive input
    else
        setvar j j+1
        setvar rec!j  name
    endif
endwhile
setvar rec0  j
```

Filling variables arrays (cont)

- :xeq exmpl3 <datafile

- :showvar rec@

```
REC1 = line1
REC2 = line2
…
REC20 = line20
REC0 = 20
```

- performance:
  - Script as is: 100 records: 530 millisecs
  - Script modified for file input only (shown in notes):

    100 records: 380 millisecs

Filling variables arrays (cont)

- can we fill arrays (and read files) faster?
- example 4:

```
setvar rec0  0
setvar limit FINFO (HPSTDIN, ”eof”)
while setvar(rec0, rec0+1) <= limit and  &
        setvar(rec![rec0+1], input()) <> chr(1) do
endwhile
setvar rec0 rec0-1
```

- performance (:xeq exmpl4 <datafile):
  - 100 records: 185 millisecs (twice as fast!)

CI expressions

- operators:
  - + (ints and strings), -, *, / , ^, (), <, <=, >, >=, =, AND, BAND, BNOT, BOR, BXOR, CSL, CSR, LSL, LSR, MOD, NOT, OR, XOR
- precedence (high to low):
  - 1) variable dereferencing
  - 2) unary + or -
  - 3) bit operators (csr, lsl…)
  - 4) exponentiation (^)
  - 5) *, / , mod
  - 6) +, -
  - 7) <, <=, =, >, >=
  - 8) logical operators (not, or…)
  - left to right evaluation, except exponentiation is r-to-l

JINFO function

syntax: JINFO (“[#]S|Jnnnn”, “item” [,status])
where jobID can be “[#]J|Snnn” or “0”, meaning “me”

- 63 unique items: Exists, CPUSec, IPAddr, JobQ, Command, JobUserAcctGroup, JobState, StreamedBy, Waiting ...
- status parm is a variable name. If passed, CI sets status to JINFO error return -- normal CI error handling bypassed
- can see non-sensitive data for any job on system
- can see sensitive data on: “you”; on other jobs w/ same user.acct if jobsecurity is LOW; on other jobs in same acct if AM cap; on any job if SM or OP cap

JOBCNT function

syntax: JOBCNT (“job_spec” [,joblist_var])

- “Job_Spec” can be:
  - “user.account”
  - “jobname,user.account”
  - “@J”, “@S”, “@”
  - “@J:[jobname,]user.acct” or “@S:[jobname,]user.acct”
  - wildcarding is supported
  - use empty jobname (“,”) to select jobs without jobnames
  - omit jobname to match any jobname

PINFO function

syntax: PINFO (pin, “item” [,status])
where PIN can be a string, “[#P]nnn[.tin]”, or a simple integer, “0” is “me”

- 66 unique items: Alive, IPAddr, Parent, Child, Children, Proctype, WorkGroup, SecondaryThreads, NumOpenFiles, ProgramName, etc.
- status parm is a variable name. If passed, CI sets status to PINFO error return -- normal CI error handling bypassed
- can see non-sensitive data for any user process on system
- follows SHOWPROC’s rules for sensitive data

CI I/O redirection

- `> name` - redirect output from $STDLIST to "name"
  - "name" will be overwritten if it already exists
  - file will be saved as "name";rec=-256,,v,ascii;disc=10000;TEMP
  - file name can be MPE or POSIX syntax
- `>> name` - redirect, append output from $STDLIST to "name"
  - same file attributes for "name" if it is created
- `< name` - redirect input from $STDIN to "name"
  - "name" must exist (TEMP files looked for before PERM files)

- I/O redirection has no meaning if the command does not do I/O to $STDIN or $STDLIST
- available on all commands, except:
  - IF, ELSEIF, SETVAR, CALC, WHILE, COMMENT, SETJCW, TELL, TELLOP, WARN.

CI I/O redirection (cont)

- how it works:
  - CI ensures the command is not one of the excluded commands
  - CI scans the command line looking for <, >, >> followed by a possible filename (after explicit variable resolution has already occurred)
    - text inside quotes is excluded from this scan
    - text inside square brackets is excluded from this scan
  - filename is opened and “exchanged” for the $STDIN or $STDLIST
  - after the command completes the redirection is undone

- examples:
  - INPUT varname < filename
  - ECHO   The next answer is: !result >>filename
  - LISTFILE ./@,6 > filename
  - PURGEACCT myacct <Yesfile
  - PURGE foo@ ;temp ;noconfirm >$null
  - ECHO You need to include !<THIS!> too!

String manipulations

Assume variable X = "ab c;de,,fg;hij=k lmn,op=qr" and 500 iterations for timing tests

- Parse out all tokens in a string variable:

```
• setvar j 0
  while j<= len(x) do
        setvar tok word(x, , , j, j+1)
  endwhile                                   2136 millisecs
```

OR

```
• setvar j 0
  while setvar(j, j+1) <= wordcnt(x) do
        setvar tok word(x, , j)
  endwhile                                   2298 msecs
```

OR

```
• setvar j 0                                 # fails on null token
  while setvar(tok, word(x, , setvar(j, j+1))) <= “” do
  endwhile                                   1686 msecs
```

String manipulations (cont)

Assume variable X = "ab c;de,,fg;hij=k lmn,op=qr”

- Extract the first N tokens from a string var
  - setvar toks lft(x, delimpos(x, , N) -1)   # includes all token delimiters

  OR

  -
    ```
    setvar j 0                                # original delimiters replaced by single space
    setvar toks “”
    while setvar(j, j+1) <= N do
          setvar toks toks + word(x, , j) + “ “
    endwhile
    ```
- Extract the last N tokens from a string var
  - setvar toks rht(x, -delimpos(x, , -N)-1)   # includes all token delimiters

  OR

  -
    ```
    setvar j 0                       # original delimiters replaced by single space
    setvar toks “”
    while setvar(j, j+1) <= N do
          setvar toks word(x, , -j) + “ “ + toks
    endwhile
    ```

String manipulations (cont)

Assume variable X = "ab c;de,,fg;hij=k lmn,op=qr" and 500 iterations for timing tests

- Test for word "hi" somewhere in a string var
  - pos("hi", x) is wrong, e.g. "high", "highest" will also match
  - word(x, , , , pos("hi", x)) = "hi" works correctly
- Count tokens in a string var
  - setvar cnt wordcnt(x)
- Remove Nth token from a string var
  - setvar y lft(x, delimpos(x, , N-1)) + rht(x, -delimpos(x, , N) -1)
    # note: removes the right hand delimiter from X after extraction # 526 msecs

  OR

  - setvar y xword(x, , N) # note: same as above # 364 msecs
- Remove N consecutive tokens from a string var
  - # assume we are removing tokens 5,6,7 so N=3 and START=5:

    setvar y lft(x, delimpos(x, , START-1)) + rht(x, -delimpos(x, , START+N-1) -1)

Customize jobs using variables

```
PARM p1="my value", p2="something“
# create a simple job passing parms and variables to the job
setvar testvar1 true
setvar testvar2 46
setvar testvar3 "abc“
echo !!job jeff.vance;outclass=,2            >tmpjob
echo !!setvar myP1 "!p1"                    >>tmpjob
echo !!setvar myP2 "!p2"                    >>tmpjob
echo !!setvar myVar1 !testvar1              >>tmpjob
echo !!setvar myVar2 !testvar2              >>tmpjob
echo !!setvar myVar3 "!testvar3"            >>tmpjob
echo !!showvar my@                          >>tmpjob
echo !!eoj                                  >>tmpjob
stream tmpjob
```

New location (group, CWD)

- CD script

```
PARM dir=“”
setvar d  “!dir”
# “-” means go to prior CWD
if d = ‘-’ and bound(save_chdir) then
   setvar d  save_chdir
elseif fsyntax(d) = “MPE”  then                      # MPE syntax?
   if finfo(“./”+d, “exists”) then                   # HFS dir?
      setvar d  “./” + d
   elseif finfo(“../”+ups(d), “exists”) then         # MPE group?
      setvar d  “../” + ups(d)
   elseif finfo(ups(d), “exists”) then               # MPE dir name?
      setvar d  ups(d)
   endif
endif
setvar save_chdir HPCWD
chdir !d
```

Columnar output

- before:

```
setvar j 0
while setvar(j,j+1) < 4 do
    setvar a rpt("a", j)
    setvar b rpt("b", (4-j)*2)
    echo !a xx !b xx
endwhile
```

output:

```
a xx bbbbbb xx
aa xx bbbb xx
aaa xx bb xx
```

- after:

```
while ...
   setvar a ; setvar b...same way...
   echo !a ![rpt("  ", 3-len(a))]xx &
            ![rpt("  ", 6-len(b))] !b xx
endwhile
```

```
a   xx bbbbbb xx
aa  xx   bbbb xx
aaa xx     bb xx
```

MPE version

“Vers” script

```
PARM vers_parm=!hprelversion
# react to MPE version string
setvar vers "!vers_parm”
# convert to integer, e.g.. "C.65.02" => 6502
setvar vers ![str(vers,3,2) + rht(vers,2)\
if vers >= 7500 then
   echo On 7.5!
elseif vers >= 7000 then
   echo  On 7.0!
elseif vers >= 6500 then
   echo On 6.5!
elseif vers >= 6000 then
   echo On 6.0!
endif
```

INFO= example

-
  ```
  ANYPARM info=![""]                    # "anyrun" script
  run volutil.pub.sys; info=":!info"
  ```
  - :anyrun echo "Hi there!"
    ```
    run volutil.pub.sys;info=":echo "Hi there!""
                                 ^
    Expected semicolon or carriage return. (CIERR 687)
    ```

-
  ```
  ANYPARM info=![""]
  setvar _inf repl('!info', '"', '""')  # double up quotes in :RUN
  run volutil.pub.sys;info=":!_inf "
  ```
  - :anyrun echo "Hi there!"
    ```
    Volume Utility A.02.00, (C) Hewlett-Packard Co.,
    1987. All Rights...
    volutil: :echo "Hi there!"
    "Hi there!"
    ```

- is this correct now?

INFO= example (cont)

-
  ```
  ANYPARM info=![""]
  setvar _inf anyparm(!info)              # note info parm is not quoted
  setvar _inf repl(_inf, '"', '""')
  run volutil.pub.sys;info=":_!inf "
  ```

- :anyrun echo "Hi there, ‘buddy’!”
  ```
  Volume Utility A.02.00, (C) Hewlett-Packard Co.,
  1987. All Rights...
  volutil: :echo "Hi there, ‘buddy’!”
  "Hi there, ‘buddy’!”
  ```

Random names

```
• PARM varname, minlen=4, maxlen=8
  # This script returns in the variable specified as "varname" a `random’
  # name consisting of letters and numbers - cannot start with a number.
  # At least "minlen" characters long and not more than "maxlen" chars.

  ## expression for a `random' letter:
  setvar letter  "chr( (hpcpumsecs mod 26) + ord('A') )”

  ## expression for a `random' number:
  setvar number  "chr((hpcpumsecs mod 10) + ord('0'))"
  ## first character must be a letter
  setvar !varname  !letter

  ## now fill in the rest, must have at least "minlen" chars , up to "maxlen"
  setvar i 1
  setvar limit  min( (hpcpumsecs mod !maxlen) + !minlen,  !maxlen)
  while setvar(i,i+1) <= limit do
     if odd(hpcpumsecs) then
         setvar !varname  !varname + !letter
      else
         setvar !varname  !varname + !number
      endif
  endwhile
```

PRNT - print file based on HPPATH

```
PARM filename
# This command file prints the first MPE filename found in HPPATH.
setvar _prnt_i 0
setvar _prnt_match false
while not (_prnt_match) and &
    setvar(_prnt_tok,word(“!hppath”,',; ',setvar(_prnt_i,_prnt_i+1)))<>””do
    if delimpos(_prnt_tok,'./') <> 1 then
        # skip HFS path elements, we have an MPE syntax element
        setvar _prnt_match (finfo("!filename.!_prnt_tok",'exists'))
    endif
endwhile
if _prnt_match then
    setvar _prnt_f fqualify("!filename.!_prnt_tok")
    echo !_prnt_f
    continue
    print !_prnt_f,!out ;page=22
else
    echo ![ups("!filename")] was not found in your HPPATH.
endif
```

Scan history (redo) stack

```
PARM cmdstr entry=main
# Script scans the redo stack, from top-of-stack (TOS), backwards towards the
# beginning, searching for the 1st cmd line that contains "cmdstr“ anywhere.
if '!entry' = 'main' then
   listredo ;unn >lrtmp
   # create variables for each command line in the redo stack
   xeq !hpfile "!cmdstr" entry='listredo' <lrtmp
   # scan above variables for first match on "cmdstr“
   xeq !hpfile "!cmdstr" entry='match‘
   # match or not?
   if _rdo_line = "" then
      echo "!cmdstr" not found in history stack.
   else
      # do an interactive command redo feature
      echo Edit command line for REDO:
      echo !_rdo_line
      setvar _rdo_edit input()
      while _rdo_edit <> "" do
          setvar _rdo_line edit(_rdo_line,_rdo_edit)
          echo !_rdo_line
          setvar _rdo_edit input()
       endwhile
      # execute the command
      continue
      !_rdo_line
   endif
   deletevar _rdo_@
   return
```

Scan history stack (cont)

```
elseif '!entry' = 'listredo' then
   # Fill variable "array" so redo stack can be searched from TOS down.
   # Input comes from output of LISTREDO ;unn command.
   # Skip TOS redo line since it invoked this script!
   setvar _rdo_x 0
   setvar _rdo_size finfo(hpstdin,'eof')-1
   while setvar(_rdo_x,_rdo_x+1) <= _rdo_size do
      setvar _rdo_!_rdo_x input()
   endwhile
   return

elseif '!entry' = 'match' then
   # Find redo entry (now in variable "array") that matches user's string.
   # Search from last array element down to the first. Return _rdo_line as
   # "" for no match, or the matching cmd.
   setvar _rdo_txt dwns("!cmdstr")
   setvar _rdo_x _rdo_size+1
   while setvar(_rdo_x,_rdo_x-1) > 0 and &
          pos(_rdo_txt,dwns(_rdo_![_rdo_x-1])) = 0 do
   endwhile
   if _rdo_x > 0 then
      # match
      setvar _rdo_line _rdo_!_rdo_x
   else
      setvar _rdo_line "“
   endif
   return
endif
```

Scan history stack (cont)

```
:listredo
        1) listf,6
        2) Showtime
        3) run editor
        4) run edit.pub.sys
        5) hpedit rem
        6) listredo ;unn
        7) showjob
        8) me
        9) spme
        10) showproc 0
        11) listredo
:rdo sys
Edit command line for REDO:
   run edit.pub.sys
       ihp
   run hpedit.pub.sys
HP EDIT  HP32656A.02.33 (c) COPYRIGHT Hewlett-Packard Co
   ...
```

Where is a “command”?

```
PARM cmd=“”, entry=main
# This script finds all occurrences of "cmd" as a UDC, script or program in
# HPPATH.  Wildcards are supported for UDC, program and command file
names.
# Note: a cmd name like "foo.sh" is treated as a POSIX name, not a qualified
#          MPE name.
if "!entry" = "main" then
   errclear
   setvar _wh_cmd "!cmd”
   if delimpos(_wh_cmd,”/.") = 1 then
       echo WHERE requires the POSIX cmd to be unqualified.
       return
   endif

   # see if the command could be a UDC (wildcards are supported)
   setvar _wh_udc_ok (delimpos(_wh_cmd,'._') = 0)
   # see if the command could be an MPE filename (wildcards ok, and
   #  MPE names cannot be qualified at all)
   setvar _wh_mpe_ok (delimpos(_wh_cmd,'._') = 0)
   ## All command values are assumed to be ok as a POSIX filename.
   ## The dash (-) char is excluded above since it could be in a [a-z] pattern
```

. . . continued . . .

Where (cont)

```
...
   # check for UDCs first
   if _wh_udc_ok then
      continue
      showcatalog >whereudc
      if cierror = 0 then
         xeq !hpfile !_wh_cmd entry=process_udcs <whereudc
      endif
   endif

   # Now check for command/program files
   if word(setvar(_wh_syn,fsyntax("./"+_wh_cmd))) = "ERROR" then
      # illegal name, could be a longer UDC name, in any event there
      # no need to check for command/program files.
      deletevar _wh_@
      return
   endif
   setvar _wh_wild pos("WILD",_wh_syn) > 0
   ... continued ...
```

Where (cont)

```
. . .
 # loop through hppath
 setvar _wh_i 0
 while setvar(_wh_tok,word(hppath,”,; “,setvar(_wh_i,_wh_i+1)))<>”” do
   if delimpos(_wh_tok,”/.”) = 1 then
     # we have a POSIX path element
     setvar _wh_tok "!_wh_tok/!_wh_cmd”
   elseif _wh_mpe_ok then
     # we have an MPE syntax HPPATH element with an unqualified _tok
     setvar _wh_tok "!_wh_cmd.!_wh_tok”
   endif
   errclear
   if _wh_wild then
     continue
     listfile !_wh_tok,6 >prntlf
   elseif finfo(_wh_tok,'exists') then
     # write to same output file as listfile uses above
     echo ![fqualify(_wh_tok)] >prntlf
   else
     setvar hpcierr -1
   endif
   if hpcierr = 0 then
     xeq !hpfile !_wh_tok entry=process_listf <prntlf
   endif
 endwhile
 deletevar _wh_@
 return
  . . . continued. . .
```

Where (cont)

```
 …
elseif "!entry" = "process_udcs" then
   # input redirected from the output of showcatalog
   setvar _wh_udcf rtrim(input())
   setvar _wh_eof finfo(hpstdin,”eof”) -1
   while setvar(_wh_eof,_wh_eof-1) >= 0 do
      if lft(setvar(_wh_rec,rtrim(input())),1) = " " then
         # a UDC command name line
         if pmatch(ups(_wh_cmd),setvar(_wh_tok,word(_wh_rec))) then
            # display: UDC_command_name    UDC_level    UDC_filename
            echo !_wh_tok ![rpt(" ",26-len(_wh_tok))] &
                     ![setvar(_wh_tok2,word(_wh_rec,,-1))+rpt(" ",7-len(_wh_tok2))] &
                     UDC in !_wh_udcf
         endif
      else
         # a UDC filename line
         setvar _wh_udcf _wh_rec
      endif
   endwhile
   return
```

Where (cont)

```
...
elseif "!entry" = "process_listf" then
   # input redirected from the output of listfile,6 or a simple filename
   setvar _wh_eof finfo(hpstdin,'eof')
   while setvar(_wh_eof,_wh_eof-1) >= 0 do
      setvar _wh_fc ""
      if setvar(_wh_fc, finfo(setvar(_wh_tok,ltrim(rtrim(input()))),'fmtfcode')) = ””
         setvar _wh_fc 'script’
      elseif _wh_fc <> 'NMPRG' and _wh_fc <> 'PROG' then
         setvar _wh_fc ""
      endif
      if _wh_fc <> "" and finfo(_wh_tok,'eof') > 0 then
         setvar _wh_lnk “”
         if _wh_fc = “script” and finfo(_wh_tok,'filetype') = 'SYMLINK' then
            setvar _wh_fc 'symlink’
            # get target of the symlink
            file lf7tmp;msg
            continue
            listfile !_wh_tok,7 >*lf7tmp
            if hpcierr = 0 then
               # discard first 4 records
               input _wh_lnk <*lf7tmp
               input _wh_lnk <*lf7tmp
               input _wh_lnk <*lf7tmp
               input _wh_lnk <*lf7tmp
               input _wh_lnk <*lf7tmp
               setvar _wh_lnk "-->" + word(_wh_lnk,,-1)
            endif
         endif
```

Where (cont)

```
…
        # display: qualified_filename file_code or "script" and link if any
        echo !_wh_tok ![rpt(" ",max(0,26-len(_wh_tok)))] !_wh_fc &
                ![rpt(" ",7-len(_wh_fc))] !_wh_lnk
        endif
   endwhile
   return
   endif
```

- :where @sh@

```
SHOWME                   USER    UDC in SYS52801.UDC.SYS
SH                       SYSTEM  UDC in HPPXUDC.PUB.SYS
SH.PUB.VANCE             NMPRG
SHOWVOL.PUB.VANCE        script
BASHELP.PUB.SYS          PROG
HSHELL.PUB.SYS           script
PUSH.SCRIPTS.SYS         script
RSH.HPBIN.SYS            NMPRG
SH.HPBIN.SYS             NMPRG
/bin/csh                 NMPRG
/bin/ksh                 symlink  --> /SYS/HPBIN/SH
/bin/remsh               symlink  --> /ENM/PUB/REMSH
/bin/rsh                 symlink  --> /SYS/HPBIN/RSH
/bin/sh                  symlink  --> /SYS/HPBIN/SH
```

Stream UDC - overview

```
STREAM
ANYPARM streamparms = ![“ ”]
OPTION nohelp, recursion
. . .
if main entry point then
   # initialize …
   - if “jobq=“ not specified then read job file for job “card”
   - if still no “jobq=“ then read config file matching “[jobname,]user.acct”
   - stream job in HPSYSJQ (default) or derived job queue
   - clean up
else
   # alternate entries
   separate entry name from remaining arguments
   . . .
   if entry is read_jobcard then  read job file looking for “:JOB”,
concatenate
      continuation lines (&) and remove user.acct passwords
      . . .
   elseif entry is read_config then
         read config file, match on “[jobname,]user.acct”
      . . .
   endif
```

Stream UDC - “main”

```
# comments …
if "!streamparms" = "" or pos("entry=","!streamparms") = 0
then
   # main entry point of UDC
   setvar _str_jobfile word("!streamparms")       # extract 1st arg
   . . .
   # extract remaining stream parameters
   setvar _str_parms ups( &
            repl(rht("!streamparms",-delimpos("!streamparms"))," ",""))
   if setvar(_str_pos, pos(“;JOBQ=“,_str_parms)) > 0 then
      setvar _str_jobq  word(_str_parms,,2,,_str_pos+5)
   endif
   if _str_jobq = “” then
      # no jobq=name in stream command so look at JOB “card”
      STREAM _str_jobcard entry=read_jobcard <!_str_jobfile
      if setvar(_str_pos,pos(";JOBQ=",_str_jobcard)) > 0 then
         setvar _str_jobq word(_str_jobcard,,2,,_str_pos+5)
      endif
   endif
```

Stream UDC - “main” (cont)

```
if _str_jobq = '' and finfo(_str_config_file,'exists') then
   # No jobq=name specified so far so use the config file.
   STREAM ![word(_str_jobcard,";")]  _str_jobq  entry=read_config &
                <!_str_config_file
   if _str_jobq <> '' then
       # found a match in config file, append jobq name to stream command line
       setvar _str_parms _str_parms + ";jobq=!_str_jobq"
   endif
endif
. . .
# now finally stream the job.
if _str_jobq = '' then
    echo Job file "!_str_jobfile" streamed in default "HPSYSJQ" job queue.
else
    echo Job file "!_str_jobfile" streamed in "!_str_jobq" job queue.
endif
option norecursion
continue
stream !_str_jobfile !_str_parms
. . .
```

Stream UDC - “read_jobcard”

```
else
   # alternate entry points for UDC.
   setvar _str_entry word("!streamparms",,-1)
   # remove entry=name from parm line
   setvar _str_entry_parms
lft('!streamparms',pos('entry=','!streamparms')-1)

   if _str_entry = "read_jobcard" then
     # Arg 1 is the *name* of the var to hold all of the JOB card right of "JOB".
     # Input redirected to the target job file being streamed
     # Read file until JOB card is found.  Return, via arg1, this record,
     # including continuation lines, but less the "JOB" token itself.  Remove
     # all passwords, if any. Skip leading comments in job file.
    setvar _str_arg1 word(_str_entry_parms)
    while str(setvar(!_str_arg1,ups(input())),2,4) <> "JOB " do
    endwhile
    # remove line numbers, if appropriate
    if setvar(_str_numbered, numeric(rht(!_str_arg1,8))) then
       setvar !_str_arg1 lft(!_str_arg1,len(!_str_arg1)-8)
    endif
    ...
```

Stream UDC - “read_jobcard” (cont)

```
…
# concatenate continuation (&) lines
while rht(setvar(!_str_arg1,rtrim(!_str_arg1)),1) = '&' do
   # remove & and read next input record
   setvar !_str_arg1 lft(!_str_arg1,len(!_str_arg1)-1)+ltrim(rht(input(), -2))
   if _str_numbered then
      setvar !_str_arg1 lft(!_str_arg1,len(!_str_arg1)-8
   endif
endwhile
# remove passwords, if any
while setvar(_str_pos,pos('/',!_str_arg1)) > 0 do
   setvar !_str_arg1   repl(!_str_arg1,"/"+word(!_str_arg1,'.,;',,,_str_pos+1),"")
endwhile
# return, upshifted, all args right of "JOB", and strip all blanks.
setvar !_str_arg1 ups(repl(xword(!_str_arg1)," ",""))
return
```

Stream UDC - “read_config”

```
elseif _str_entry = "read_config" then
    # Arg 1 is the "[jobname,]user.acct" name from the job card.
    # Arg 2 is the *name* of the var to return the jobQ name if the acct name
    # Input redirected to the jobQ config file.
    setvar _str_arg1 word(_str_entry_parms," ")
    setvar _str_arg2 word(_str_entry_parms," ",2)
    setvar _str_eof finfo (hpstdin, “eof”)
    …
    # read config file and find [jobname,]user.acct match (wildcards are ok)
    while setvar(_str_eof ,_str_eof-1)  >=  0
and &
            (setvar(_str_rec,ltrim(rtrim(input()))) = “”                or &
             lft(_str_rec,1) = '#'                                       or &
             not pmatch(ups(word(_str_rec,,-2)),_str_ua)               or &
             (pos(',',_str_rec) > 0 and lft(_str_rec,2) <> '@,'  and &
              not pmatch(ups(word(_str_rec)),_str_jname)) )              do
    endwhile
    if _str_eof >= 0 then
       # [jobname,]user.acct match, return jobq name
       setvar !_str_arg2 word(_str_rec,,-1)
    endif
return
```