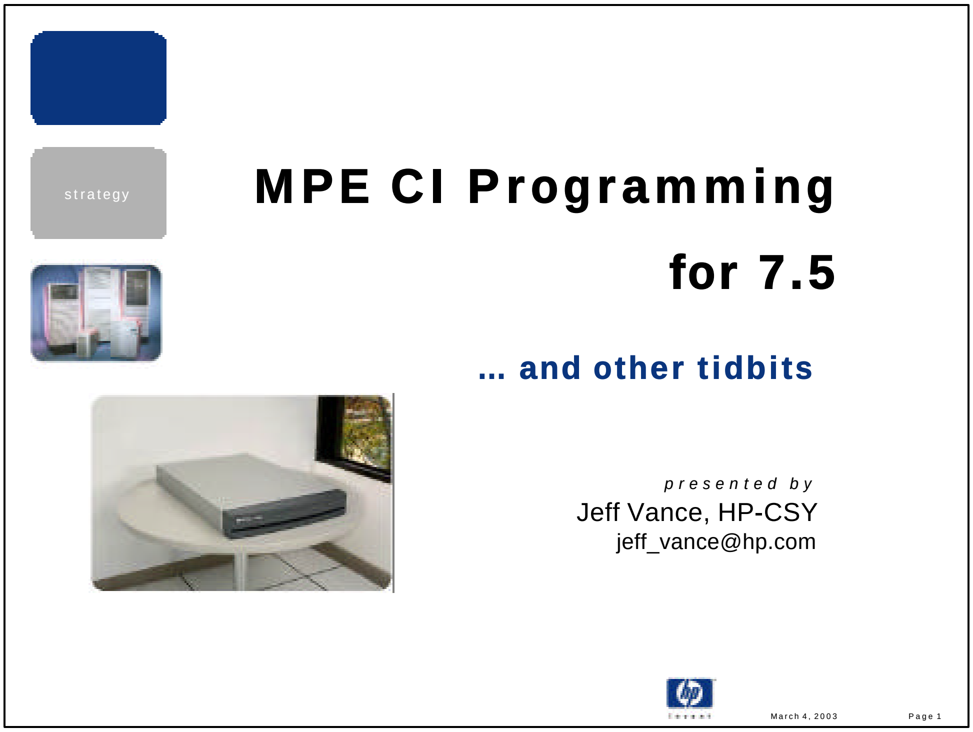


strategy

MPE CI Programming for 7.5

… and other tidbits

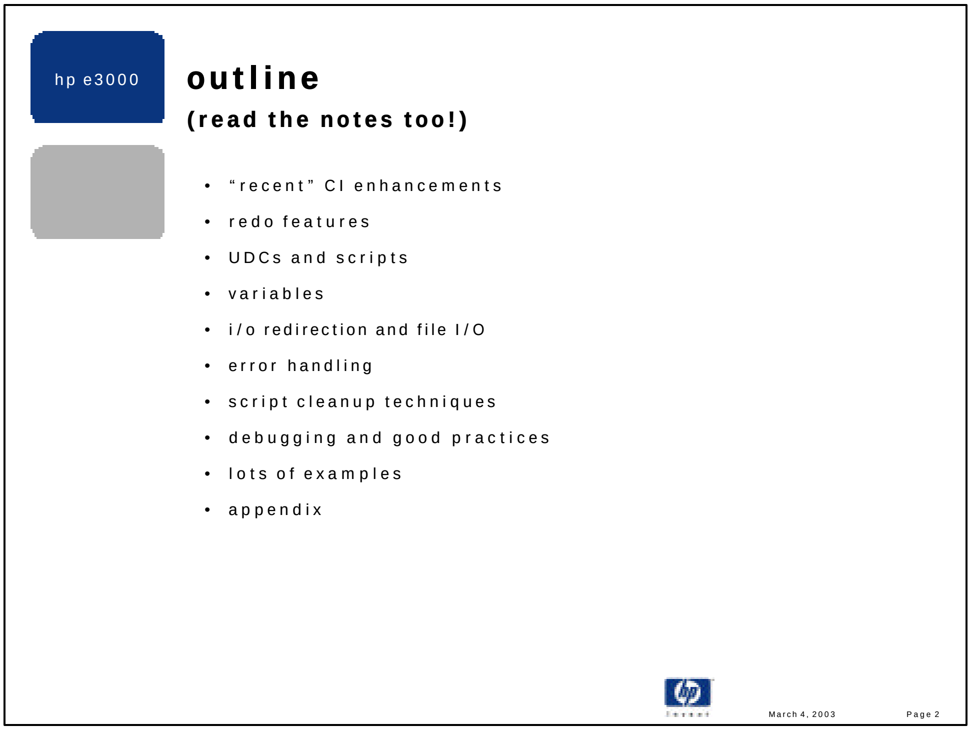
presented by

Jeff Vance, HP-CSY
jeff_vance@hp.com

invent

March 4, 2003

Page 1



hp e3000

outline

(read the notes too!)

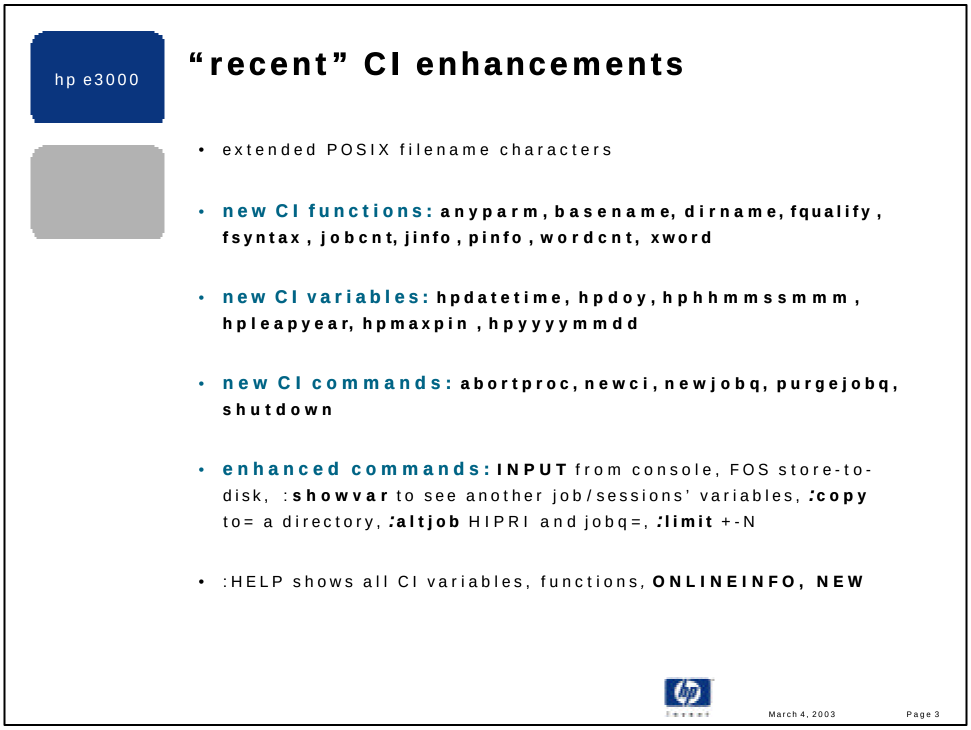
- “recent” CI enhancements
- redo features
- UDCs and scripts
- variables
- i/o redirection and file I/O
- error handling
- script cleanup techniques
- debugging and good practices
- lots of examples
- appendix

March 4, 2003

Page 2

•This version of my “CI Programming” talk focuses on comparisons of command files vs. UDCs, I/O redirection for reading files, and examples. There is much information in the appendix, including longer examples.

•The notes are an integral part of these slides. Please read the notes, as they contain many more details than are presented in the slides alone.



hp e3000

“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**

March 4, 2003 Page 3

•Above enhancements begin with MPE/iX release 6.0 and span up to release 7.5.

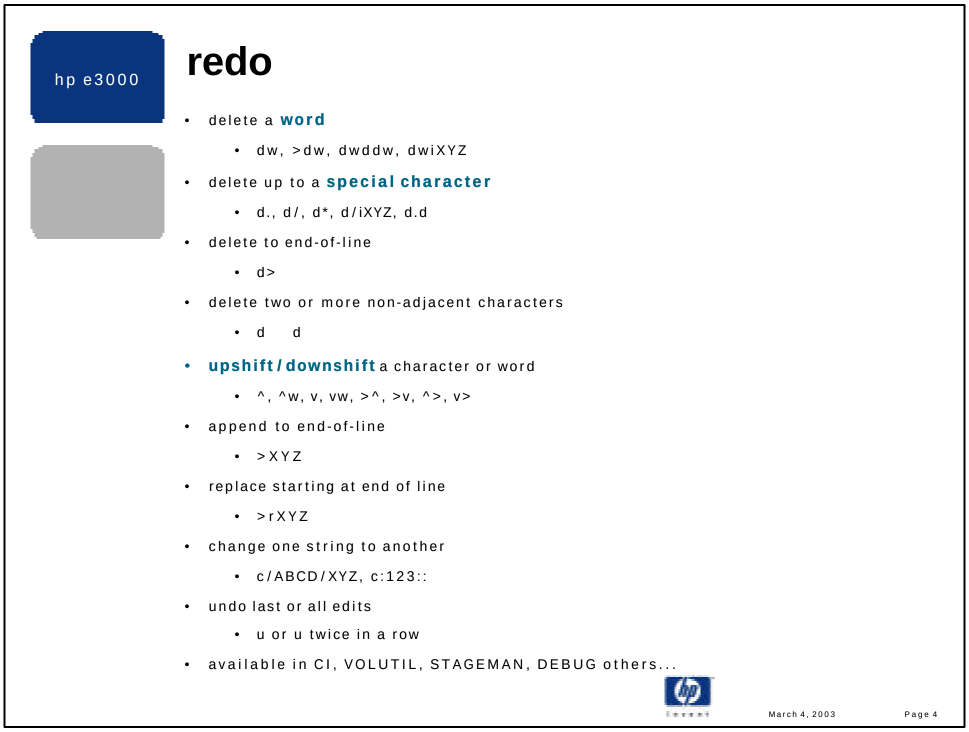
•Extended POSIX filename characters are: “**~\$%^*+|{}:**” in addition to “_-.” that were originally supported.

•Enter HELP on each of these commands to ensure you are current on their usage. For example, did you know the system manager can display the user variables from another job or session? Are you aware that you can wait until one or more jobs terminate via the enhanced PAUSE command? Did you remember that the OPTION command is not just in a UDC/script header, but can also be used as a CI command? Have you tried using the INPUT command to send a prompt to the system console and wait for an operator reply? Did you know you can abort a single process without killing an entire job or session? If you ever run low on processes, did you know the NEWCI command will save you one process per logon? Do you know an easy way to determine the maximum number of processes supported on one of your systems (answer: HPMAXPIN variable). In 7.0 Express 1, you can use the PINFO CI function to retrieve detailed information on an individual process (PIN) or thread.

• HELP NEW - shows all features of the CI that are (relatively) new.

• HELP ONLINEINFO - shows current URLs where information on the e3000 is available online.

• Also, HELP VARIABLES, HELP EXPRESSIONS, HELP OPERATORS, and HELP FUNCTIONS all provide useful information.



hp e3000

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...

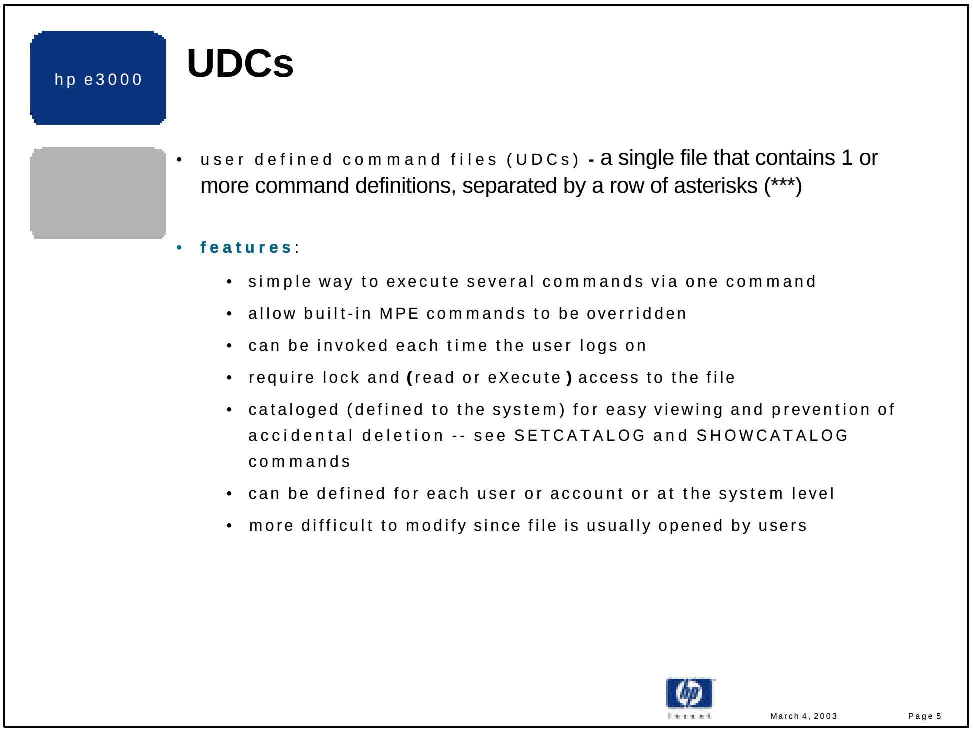
March 4, 2003 Page 4

•REDO was enhanced in late 5.5 to operate on “words”. A word is defined as any set of characters delimited by a: space, comma, semicolon, equal sign, left or right parentheses, left or right brackets, single quote or double quote. A “word” in redo is the same as the default word definition used by the DELIMPOS, WORD and XWORD CI functions.

•REDO deals with words as it does with characters. Words can be deleted (dw), upshifted (^w) and downshifted (vw).

•Words can be operated on from the end of the line: >dw - deletes the last word, >^w - upshifts the last word, >vw - downshifts the last word.

•Upshifting and downshifting can be useful when editing POSIX file names, or entering procedure names in the debugger -- times when character case matters.



hp e3000

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

March 4, 2003 Page 5

UDCs were the only way to group commands together and execute them as a single command on classic MPE V systems and earlier. Today, we can still use UDCs, and we can use command files (or “scripts) for the same basic purpose. However, there are important differences between UDCs and scripts that users should consider. The similarities and differences of UDC compared to scripts are discussed in the next few slides.

Similarities

•UDCs and scripts reside in standard MPE ASCII files

•they both support parameters with optional default values

•they both require read or execute access

•they both support the options: HELP, NOHELP, LIST, NOLIST, BREAK, NOBREAK, PROGRAM, NOPROGRAM

Differences

1. Cataloging:

One or more UDCs are collected into a single file. This file can be assigned (or cataloged) to a particular user, an account or the entire system. Multiple UDC files can be cataloged to the same or to different users and/or accounts simultaneously. The SHOWCATALOG and SETCATALOG commands provide this cataloging service. Once a UDC file is cataloged it is opened by the user process and cannot be deleted or modified until after the file has been un-cataloged (and closed). However, the POSIX shell’s “mv” command does allow an open UDC file to be replaced. The changes are immediate to users just logging on, but are not seen by current users unless the re-logon, or re-setcatalog.

The benefits of UDC cataloging are:

•many UDCs can reside in the same physical file,

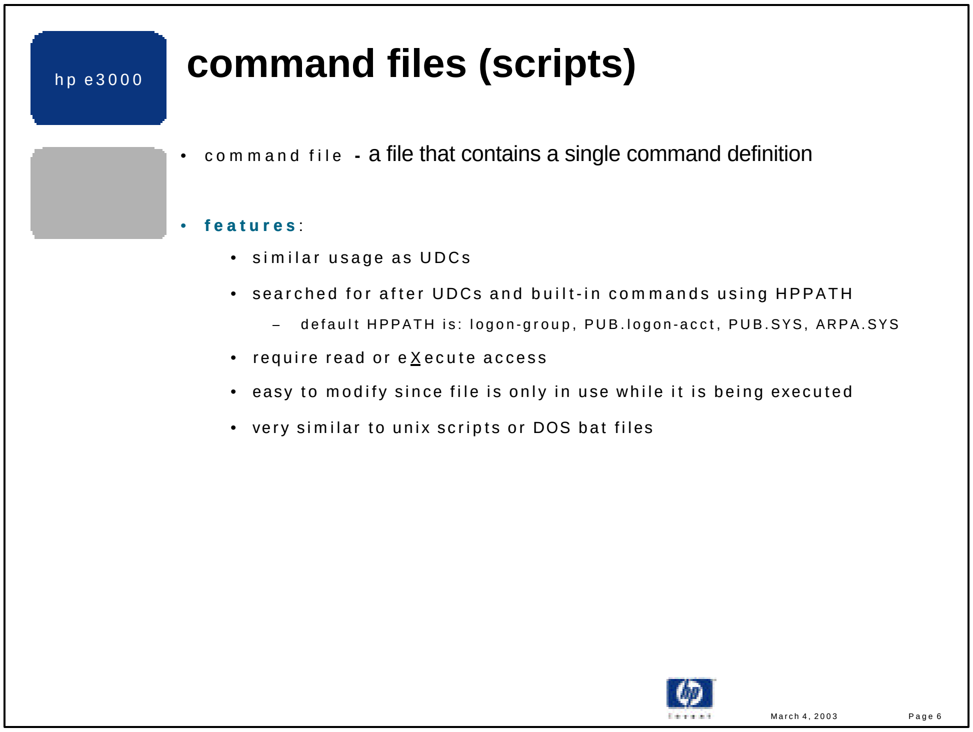
•the UDC file cannot be accidentally purged or modified, since the file is open,

•visibility as to which UDCs are available to which users on the system.

The disadvantages of this cataloging approach are:

•cumbersome to modify individual UDCs defined in the UDC file,

•overhead to catalog the UDC file at logon time.



hp e3000

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 HPPATH 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

March 4, 2003 Page 6

Command files (scripts) are single files that contain the commands to be executed. These files can reside anywhere on a system; however, typically they are located in groups or directories referenced in the HPPATH variable. Like UDCs, scripts are invoked via their name, however, since a script is a file, it can be entered as a qualified filename or as an unqualified filename. Most commonly, script names are entered as unqualified names (just the base name), and thus the HPPATH variable is used to complete ("qualify") the name based on successive group/directory names defined in HPPATH. UDC names can be up to 16 characters long, and thus are longer than standard MPE filenames; however, POSIX script names can be longer than UDC names.

2. Command override mechanism:

A UDC name can be the same name as a built-in MPE command. The CI resolves a user entered command name by checking for a UDC prior to searching for a built-in CI command. Thus, a UDC can hide a built-in CI command. For example, a UDC can be named RUN, hence overriding the :RUN command.

A script cannot override a built-in CI command. For example, if a command file named RUN.PUB.SYS exists and the user enters ":run ….", the built-in :RUN command will be executed, not the script. Typically, command file names are different from UDC and built-in command names. The :XEQ command is provided to execute scripts with the same name as built-in commands or UDCs.

Note: after a user has logged on, UDCs are searched for in the following order:

•user level UDCs, starting at the first user file shown by :SHOWCATALOG

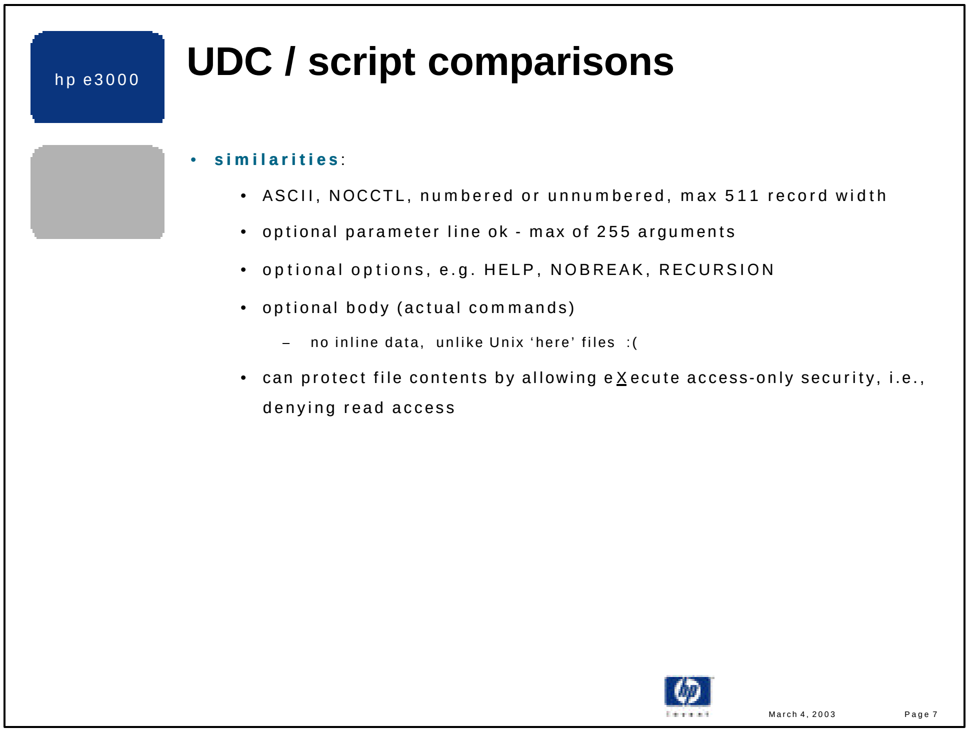
•account level UDCs,. starting at the first account file listed by :SHOWCATALOG

•system level UDCs, starting at the first system file displayed by :SHOWCATALOG.

Multiple files at the same level (user, account, system) are searched for (and executed, if found) based upon the order the files are cataloged.

Note: OPTION RECURSION causes the UDC commands within the option recursion UDC to be searched for starting at the first file cataloged at the user level, regardless of the level of the executing UDC.

Note: the UDC search order is different at logon time.



hp e3000

UDC / script comparisons

- **similarities**:
  - ASCII, NOCCTL, numbered or unnumbered, max 511 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

March 4, 2003 Page 7

3. Logon execution:

UDCs support the OPTION LOGON option. A single UDC at each level (user, account and system) can be executed at logon. Even if there are several UDCs at a given level with OPTION LOGON defined, only one UDC (the first) per level will be executed at logon time -- the remaining OPTION LOGON UDCs at that level are ignored at logon. The order that UDCs are executed during logon is the opposite of the execution order after logon. Namely, system level UDC are invoked first, followed by account UDCs, followed last by user level logon UDCs. This order allows system managers to control access to their system and to administer other security related policies via a system level logon UDC. Users cannot override a system level logon UDC, at logon time nor during normal command usage.

Scripts do not support OPTION LOGON. However, it is not uncommon for an OPTION LOGON UDC to simply invoke a script to do the real work. For example:

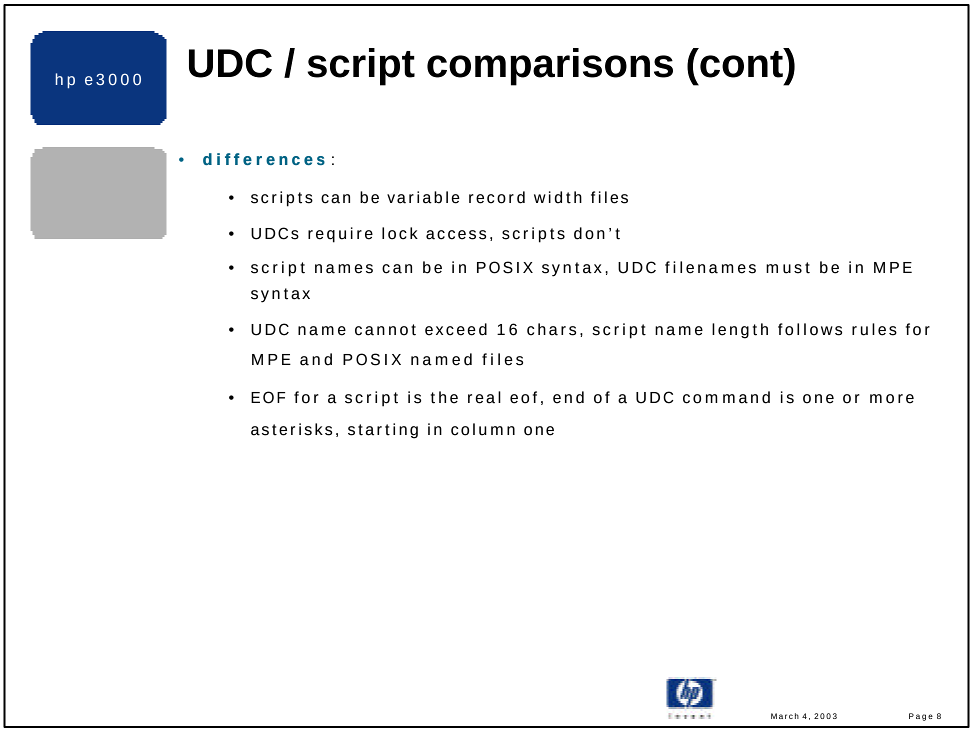
```
MYLOGONUDC                                   file: LOGIN
OPTION LOGON                                 # my logon script
# invoke login script         setvar hppath hppath+",scripts.sys,hpbin.sys"
xeq login                                    setvar hpredosize 100
***                                          if hpinteractive then ...
```

4. Command name:

A UDC name can be from 1 to 16 character long and consist solely of alphanumeric characters, with the first character being a letter.

Note: A UDC filename can be an MPE syntax symbolic link pointing to a POSIX named UDC file, if for some reason the actual UDC file needed to reside in the HFS. For example,

```
:newlink udclk,  /usr/local/udcs/system.udc
:setcatalog udclk ; append
```



hp e3000

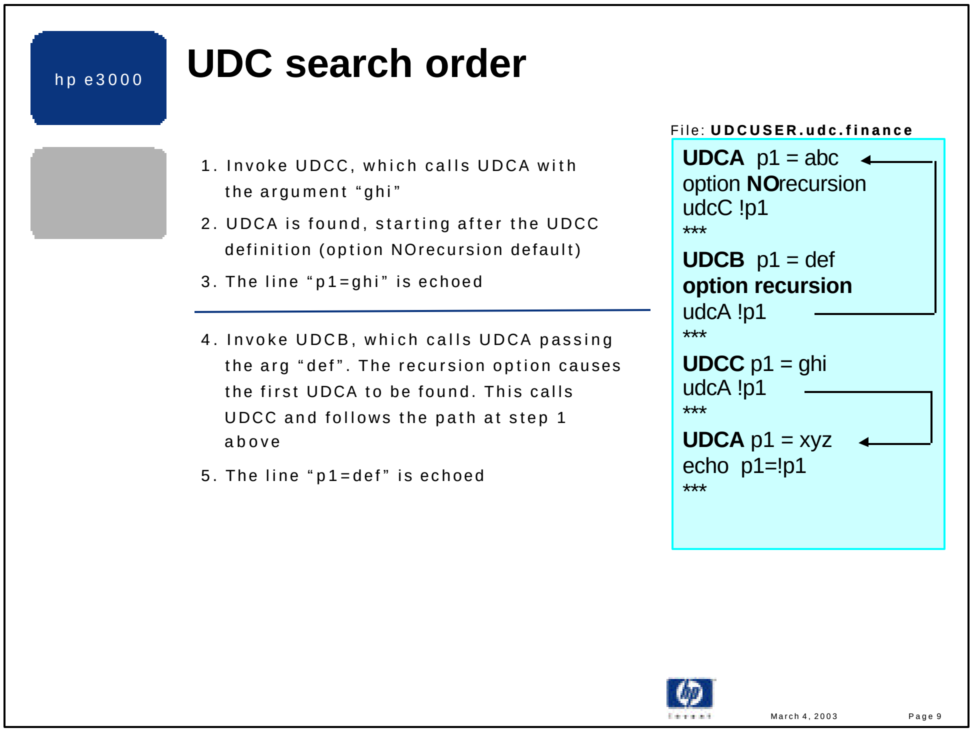
UDC / script comparisons (cont)

- **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

March 4, 2003 Page 8

A script name follows the same rules as all filenames. These rules differ depending on the syntax specified. MPE syntax filenames must be from 1 to 8 alphanumeric characters, with the first character being a letter. If the MPE name is qualified it can contain a lockword, group and account names, each having the same restrictions. POSIX syntax script names follow the rules for any POSIX-named file: 1 to 255 characters long, beginning with any valid character except a dash (-), case sensitive and several special characters are supported. Like MPE names, POSIX names can be qualified or unqualified. Unqualified (base) names are completed by pre-pending POSIX elements from the HPPATH variable to the base name.

Note: a POSIX named script cannot be qualified via HPPATH unless HPPATH contains directory names in POSIX syntax.



hp e3000

UDC search order

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

File: **UDCUSER.udc.finance**

```
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
***
```

March 4, 2003 Page 9

•In the example above, :HELP UDCA, only finds the first definition of the UDC.

•OPTION RECURSION is necessary in UDCs that support multiple entry points, otherwise a UDC would not be able to invoke itself recursively - as required by entry points (which are discussed elsewhere).

•OPTION RECURSION causes the UDC search to start completely over -- all the way back to the first user level UDC cataloged. From this point all user, account and system level UDCs are checked in order to resolve the command name.

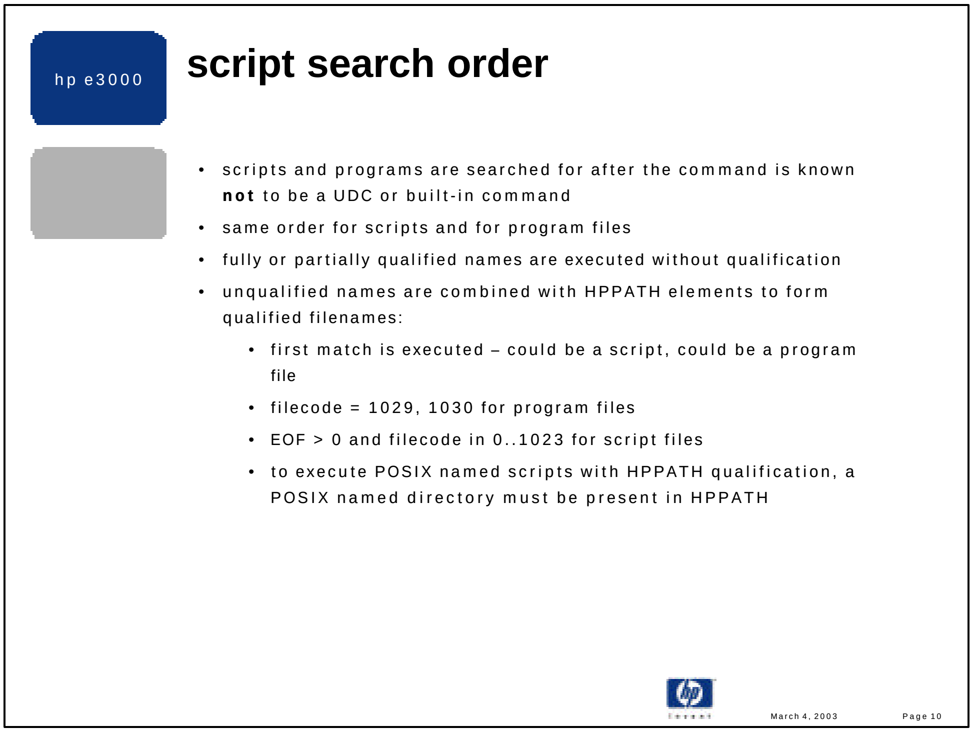
•OPTION NORECURSION is the UDC default and causes resolution of the next command to commence just **after** the current UDC. Thus, in the example above, when udcC is processing the command "udcA", it tries to resolve that command name by first checking if it it a UDC defined somewhere "below" the definition of udcC. However, when udcB calls udcA with OPTION RECURSION set, the search goes back to the first UDC catalogued. In this example, that is the udcA, at the beginning of the file. The RECURSION scope is local to the current UDC and is not inherited by successive UDCs. So, when the first udcA calls udcC which calls udcA, this invocation of udcA is **not** recursive, and thus executes the last udcA defined in the file.

•OPTION RECURSION and NORECURSION are also CI built-in commands and can appear anywhere in the UDC body.

•Scripts are recursive by definition and OPTION NORECURSION has no meaning.

•Like OPTION RECURSION, none of the UDC/script options are inherited when one UDC invokes another UDC in a nested fashion, **except** for OPTION NOBREAK. Once a UDC or script is encountered with OPTION NOBREAK specified,. all other UDCs/scripts that are called by the NOBREAK UDC/script are treated as if OPTION NOBREAK were specified -- regardless of how BREAK is defined in the called UDC/scripts. OPTION NOBREAK is somewhat common in conjunction with OPTION LOGON UDCs.

•OPTION NOPROGRAM is new to MPE/iX (MPE XL). This option indicates that the UDC or script is **not** allowed to be executed from within a program (via calling the HPCICOMMAND intrinsic). Once an OPTION NOPROGRAM UDC is encountered all UDC searching stops. The command in question may still be resolved to be a built-in command, or a script or program file, but further UDC searching ceases. For example, define a UDC named LISTF, with OPTION NOPROGRAM, which simply does ECHO LISTF!. If :LISTF is executed from the CI it will execute the LISTF UDC. If :LISTF is executed from VOLUTIL (which calls the HPCICOMMAND intrinsic to execute all non-VOLUTIL commands), the UDC is found, **but**, since it is OPTION NOPROGRAM, the UDC is not executed and no other UDCs are processed. :LISTF is found to be a known CI command, and the real :LISTF command is executed. In the slide example above, assume the first UDCA haa OPTION NOPROGRAM defined. If UDCB is executed from VOLUTIL, the first UDCA would be located, and since it is OPTION NOPROGRAM, it would not be executed. Also, the second UDCA in the file would not be executed either, because all UDC processing stops when the NOPROGRAM UDCA was found.



hp e3000

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

March 4, 2003 Page 10

•HPPATH can contain POSIX names, e.g.. "/bin, /usr/bin/local" etc., mixed with or not mixed with traditional MPE group and group.account names.

•Typically script names should be chosen to not collide with UDC names nor with built-in command names.

•Qualifying a script name that is also a UDC or built-in command name does **not** work. For example, suppose you have a script named ABORTIO, which is also the name of a CI command,and this script resides in the XEQ.SYS group. If you enter:

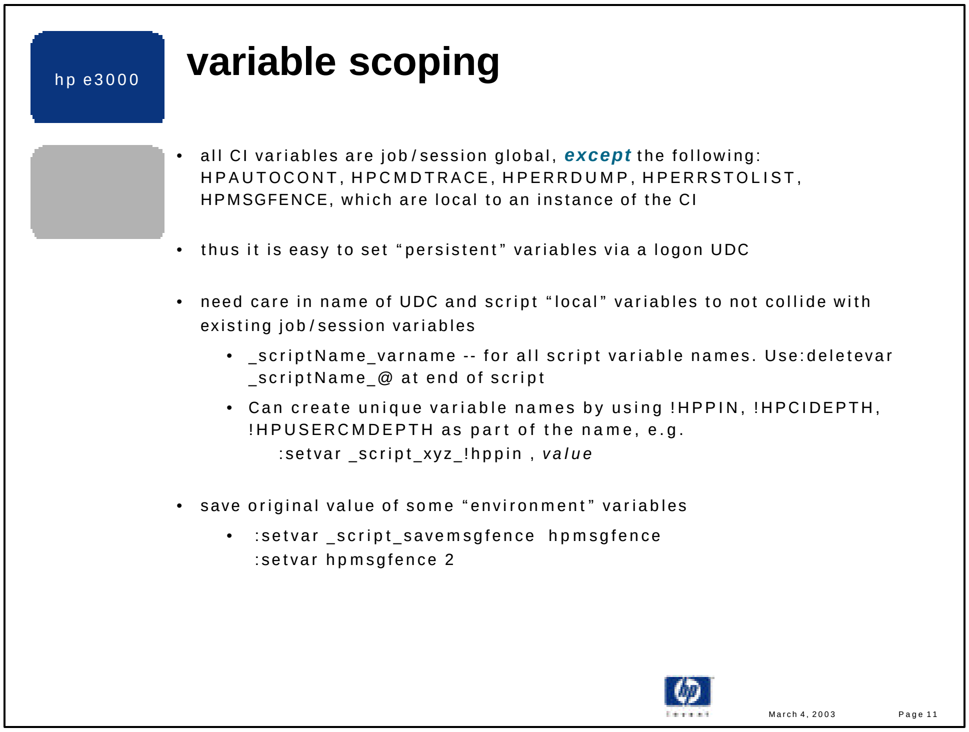
```
:abortio.xeq.sys 17,20
```

you will see this CI error:

```
ABORTIO has exactly one parameter, the device number. (CIERR 3027)
```

Why? The CI is really executing the built-in ABORTIO command and passing the arguments: ".xeq.sys", "17", "20". The CI has "strange" name parsing rules for reasons of MPE V compatibility, and decides the command name ends on the first non-alpha character -- "." in this case. Thus, the command name is "ABORTIO" and the first parameter is ".xeq.sys". The ABORTIO command only expects a single LDEV number and thus reports the above error.
The remedy is to use the XEQ command which expects its first parameter to be the name of a script or program file.

```
:xeq abortio 17, 20                    or
:xeq abortio.xeq.sys 17, 20            works fine.
```



hp e3000

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

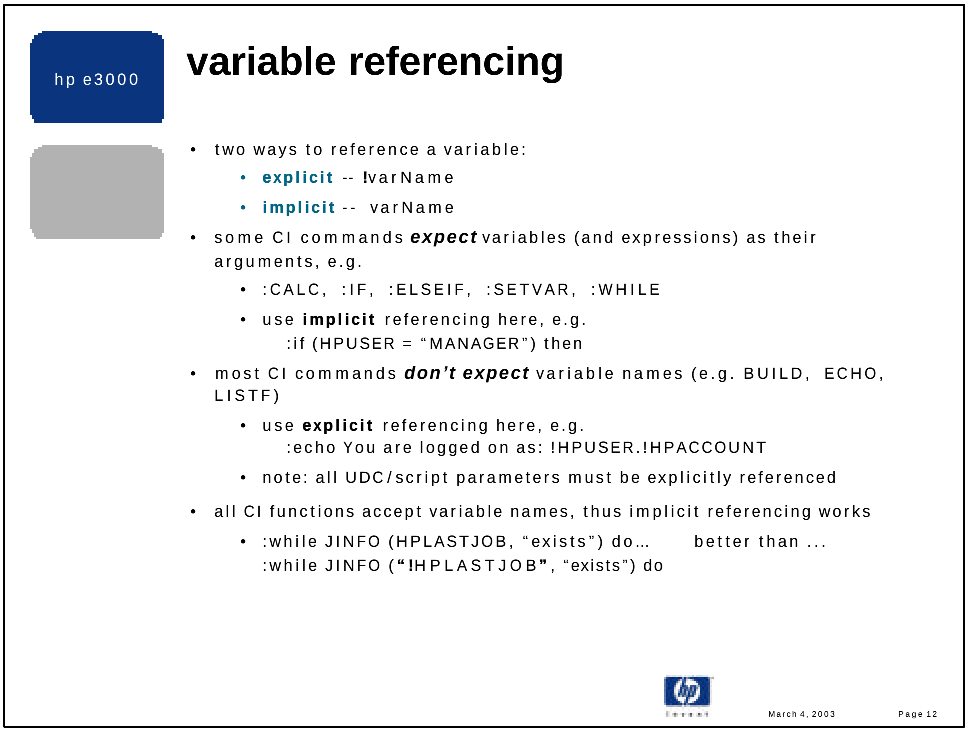
  -
    ```
    :setvar _script_savemsgfence  hpmsgfence
    :setvar hpmsgfence 2
    ```

March 4, 2003 Page 11

•The variables that are not job/session global reside in a local CI data structure, and thus are unique to each CI. If you run a child CI program it can have a different value for these variables, and any settings you do in that CI are not reflected when you exit back to the root CI.

•Since (almost) all CI variables are scoped global to the job or session environment, you can set/create variables in logon UDC, scripts etc. and these variables are available to the job or session. User variables are not automatically deleted when a script or UDC exits.

•Since (almost) all CI variables are scoped global to the job or session environment, you may need care in choosing a unique variable name. If you have a variable named XYZ defined from the CI, and you execute a script that sets XYZ and then deletes it before exiting, your CI set XYZ variable is gone. For this reason, it is generally important to use script variable names that have a decent chance of being unique to that script. A convention I use is to prefix all script variable names with the name of the script. For example, if my script is named CH and I need a counter variable named "j", I will name it _CH_J in my script.



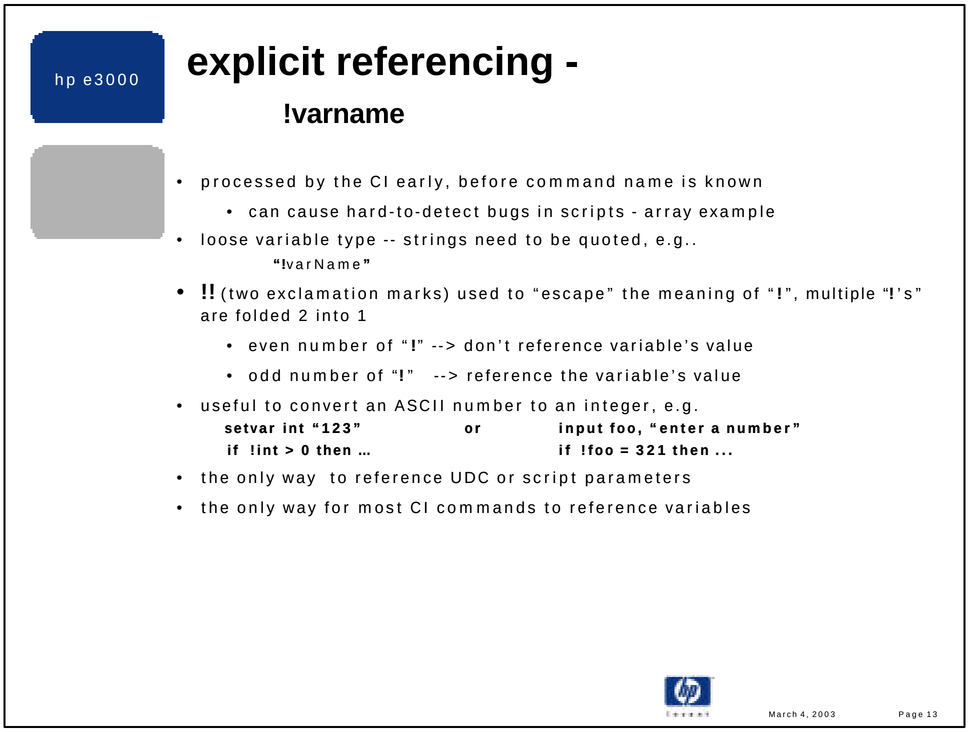
hp e3000

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

March 4, 2003 Page 12

•I see many people confused on when to put an exclamation mark in front of a variable name and when you don't need to. Since it **almost** always works to code as !varname or "!varname" this becomes the standard practice. Some users find the rules to be ambiguous so they opt to use !varname. Although, I think this is unnecessary and less "attractive", it works fine **most** of the time. There are, however, situations when using !varname results in difficult-to-diagnose programming bugs, which are shown in the next slide.



hp e3000

explicit referencing -

!varname

- processed by the CI early, before command name is known
  - can cause hard-to-detect bugs in scripts - array example
- loose 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

March 4, 2003 Page 13

•Bang folding: echo !!!!varname writes !!varname to $stdlist
echo !!!varname writes ! followed by varname's **value** to $stdlist

•Number conversion example:

```
:setvar x "123"
:if !x > 0 then …
:input x, "enter a value"          # user enters 123
:if x = 123 then                   # ERROR, x is a string!
:if !x = 123 then                  # OK
```

•Here is an example when using an explicit variable reference is incorrect (assume an "array" of variables: name1, name2, name3, etc.):

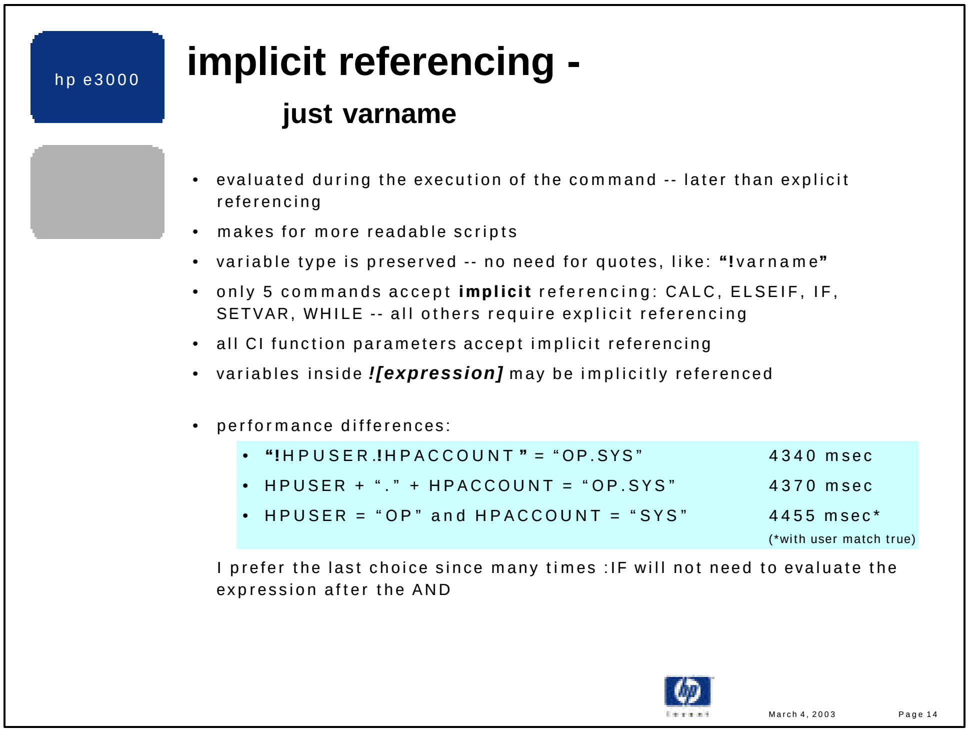
```
1) setvar j 0
2) while setvar(j,j+1) <= limit and name!j <> "EXIT"  do …
```

Line 2 will always be reference the N-1th element in the "array", and, worse, will try to reference 'name0' the first time.

Solution: 2) while setvar(j,j+1) <= limit and name**![j - 1]** <> "EXIT" do … -- or --

```
while setvar(j,j+1) <= limit do
   if name!j <> "EXIT" then ...
```

•The biggest issue with regards to using !varname is that varname's type is lost. Sometimes, this is exactly what is desired, as shown in the number conversion examples. Other times, the type was never intended to be lost, and therefore the user needs to surround !varname with quotes to preserve its string type. It is this second usage that I am trying to reduce because, in my opinion, readability and maintainability suffer.



hp e3000

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

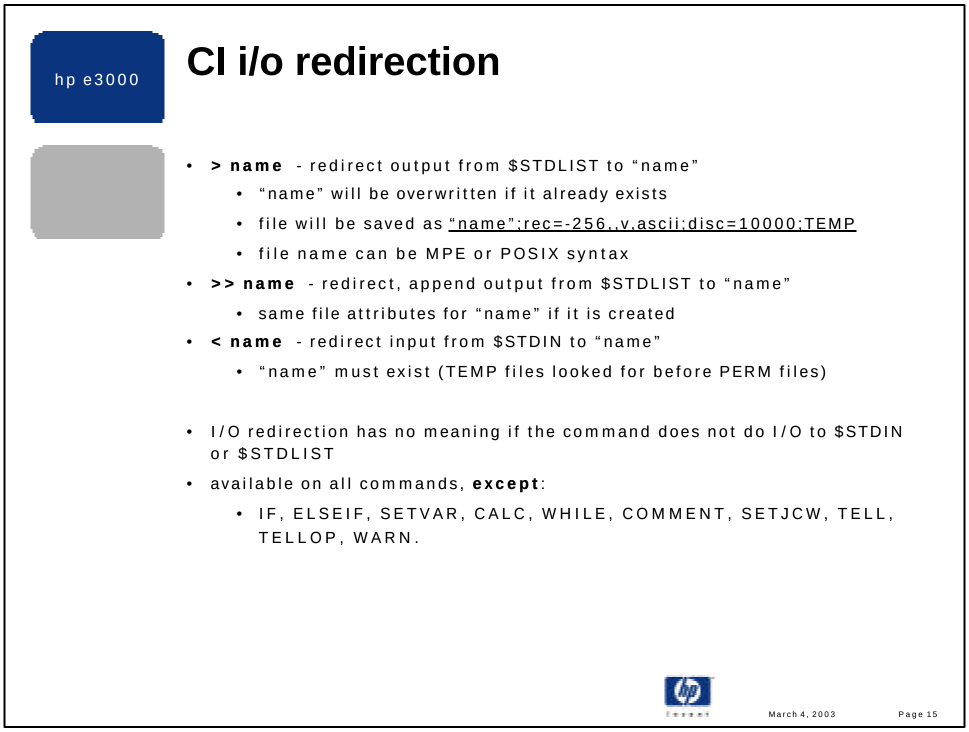
- performance differences:

• **“!**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

March 4, 2003 Page 14

•I prefer to use implicit referencing whenever possible. It makes scripts easier to read (closer to conventional programming), avoids problems of early explicit referencing shown on the previous page, and preserves the variable’s type. So my recommendation is that in the five commands listed above, and for all function arguments, and inside ![expressions] use implicit referencing as your first choice.



hp e3000

CI i/o redirection

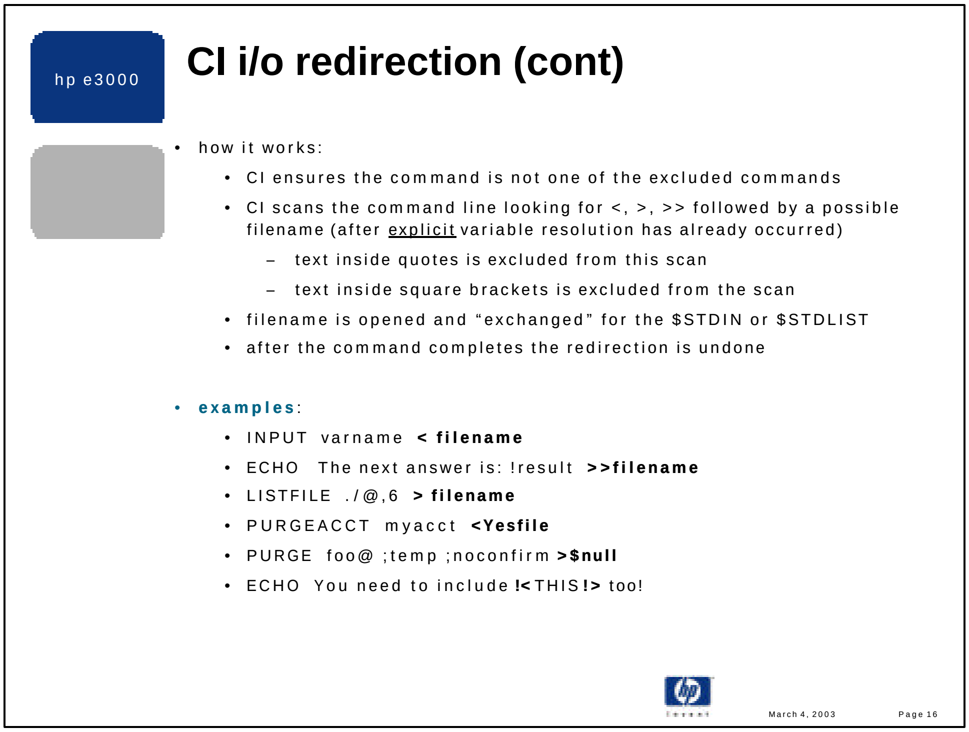
- **> name** - redirect output from $STDLIST to “name”
  - “name” will be overwritten if it already exists
  - file will be saved as “name”;rec=-256,,v,ascii;disc=10000;TEMP
  - file name can be MPE or POSIX syntax
- **>> name** - redirect, append output from $STDLIST to “name”
  - same file attributes 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.

March 4, 2003 Page 15

•I/O redirection in the CI works similarly to the same feature in DOS and Unix systems. Of course, there are some exceptions: on MPE the file created by output redirection is a TEMP, variable record width file. The motivation for these choices is that we didn’t want to mistakenly overwrite a permanent file if the “>” or “>>” symbols on a command line were not really intended for redirection. We decided to make the default record with be variable so that the file created and also be read more easily by the CI, since trailing spaces (found in fixed ASCII files) would not need to be stripped. All of the I/O redirection defaults can be overridden via a file equation.

•There are 10 CI command that do not accept I/O redirection. Five of these are commands that introduce an expression as one of their parameters. Since expressions can contain “<“, “>” it was decided to disable I/O redirection on these command. The remaining commands are excluded because we were conservative and careful when I/O redirection was introduced in MPE XL Release 2.1. We did not want to break existing scripts, UDCs, or JCL that might have “>” or “<“ in one of these commands, causing the CI would to remove the symbol and following name, and write to a file.



hp e3000

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 <u>explicit</u> variable resolution has already occurred)
    - – text inside quotes is excluded from this scan
    - – text inside square brackets is excluded from the 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!

March 4, 2003 Page 16

•The CI first replaces all **explicit** variable referencing by the variable’s value. Next, all **![expression**] references are evaluated and replaced by the result. Then, the CI deals with processing any I/O redirection it encounters on the command line. This order allows a target redirection filename to be contained in a variable or ![expression]. Also, by this time in the command processing, the CI has determined the command name and thus can check the exclusion list to make sure I/O redirection is permitted for the command being executed

•If an I/O redirection symbol is found but the token immediately right of it is not a legal filename, the CI assume I/O redirection was not intended. E.g.:

:echo abc >123 does not create a file named “123” but instead echo's:
abc >123

•Also, if the I/O redirection symbol appear inside a quoted string or inside square brackets, it is not interpreted an I/O redirection. E.g.:

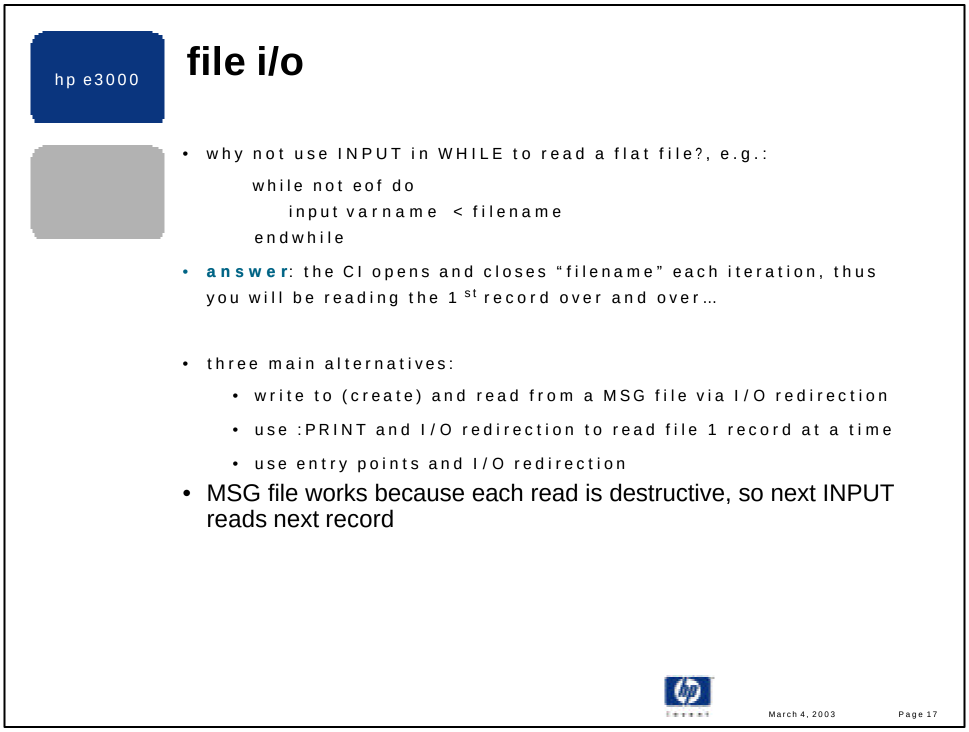
:echo abc “>xyz” does not create a file named “XYZ” but instead echo's:
abc “>xyz”

And,

:echo abc [>def] does not create a file named “DEF” but instead echo's:
abc [>def]

The reason that square brackets are excluded is to support selection equations which are contained by square brackets and allow relational operators, such as “<“ and “>”.

•To tell the CI to ignore I/O redirection in commands that it would otherwise accept I/O redirection you need to place a “!” in front of the I/O redirection token. This “escapes” the special meaning of the I/O redirection symbol and is consistent with the use of multiple exclamation marks in front or potential variable names.



hp e3000

file i/o

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

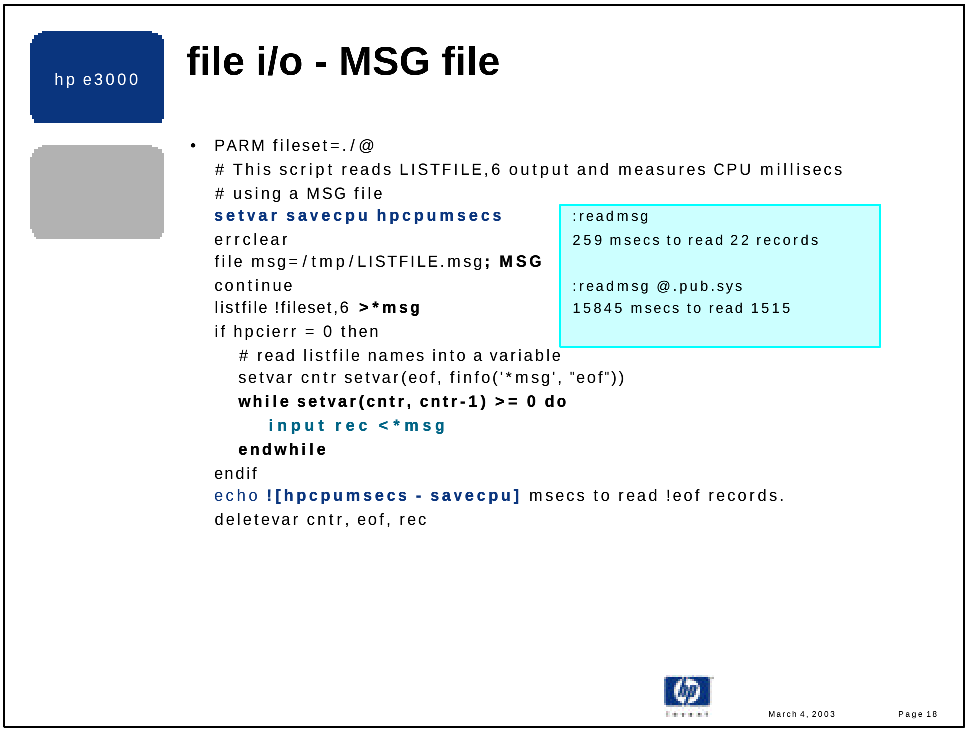
```
while not eof do
   input varname  < filename
endwhile
```

- **answer**: the CI opens and closes “filename” each iteration, thus you will be reading the 1 st record over and over…

- 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 file works because each read is destructive, so next INPUT reads next record

March 4, 2003 Page 17

• INPUT <flat_file in the WHILE loop fails because the CI opens the redirected file for each iteration in the loop. Thus, an open is done for each record in the file. Not only is this expensive, it also means that the file’s record pointer (current record) is reset to the **beginning** of the file each time INPUT is executed. Therefore, INPUT from a flat file in a WHILE loop always reads (and re-reads!) the first record of the file.



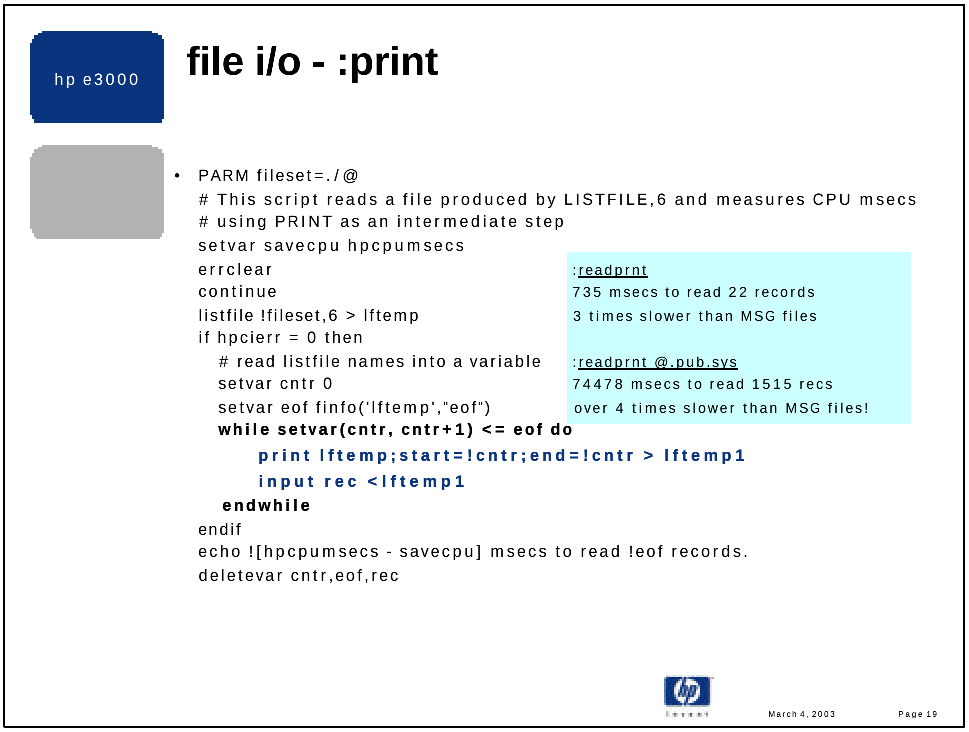
hp e3000

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
```

- •Each read of a MSG file is destructive so it works with INPUT in a while loop.
- •Example shows using POSIX names to keep temporary files.
- •Shows setting two variables in one CI command line.
- •Shows how to measure the performance of a script or UDC.



hp e3000

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
```

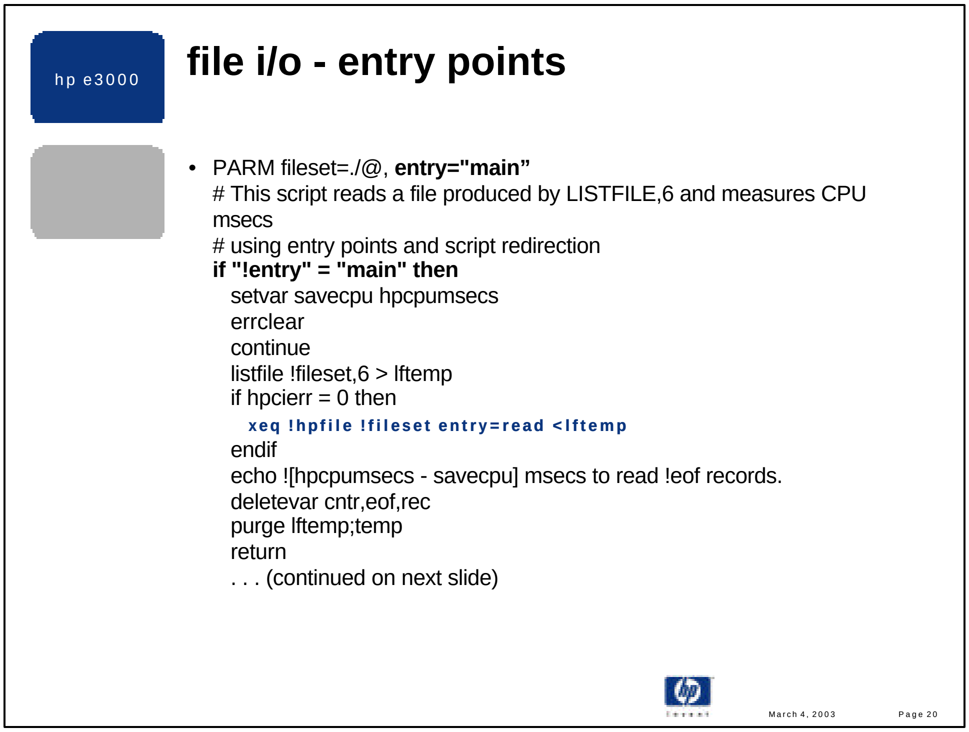
```
:readprnt
735 msecs to read 22 records
3 times slower than MSG files

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

March 4, 2003 Page 19

•The PRINT method is the least efficient of the three choices presented. This technique requires two opens and closes for each record in the file: one open for PRINT, one open for the output redirection, one close for PRINT and another close to redirect output back to $STDLIST.

•The PRINT technique is also not any easier to code than the MSG file method, so why use it?

•Perhaps the data is already in a file and the file is not large (or performance is unimportant).
In this case, using PRINT may be appropriate since the script is intuitive and easy to write, and may be better (faster) than copying the existing data to a MSG file first.



hp e3000

file i/o - entry points

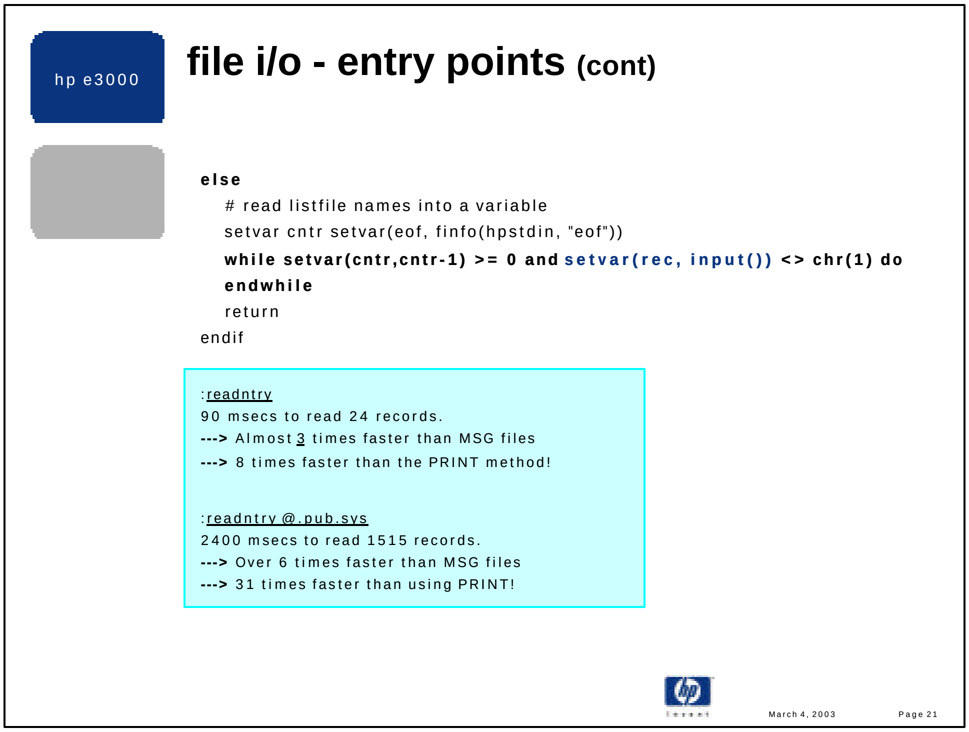
- PARM fileset=./@, **entry="main"**

```
# 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)
```

March 4, 2003 Page 20

•The choices of “entry” for the name of the entry control parameter and “main” for the default value of the entry control parameter value are arbitrary but self-documenting.

•All initialization should be done in only the “main” entry portion of the script, rather than earlier in the script. This is more efficient (and perhaps the only correct way) since the initialization code is invoked only once.



hp e3000

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
```

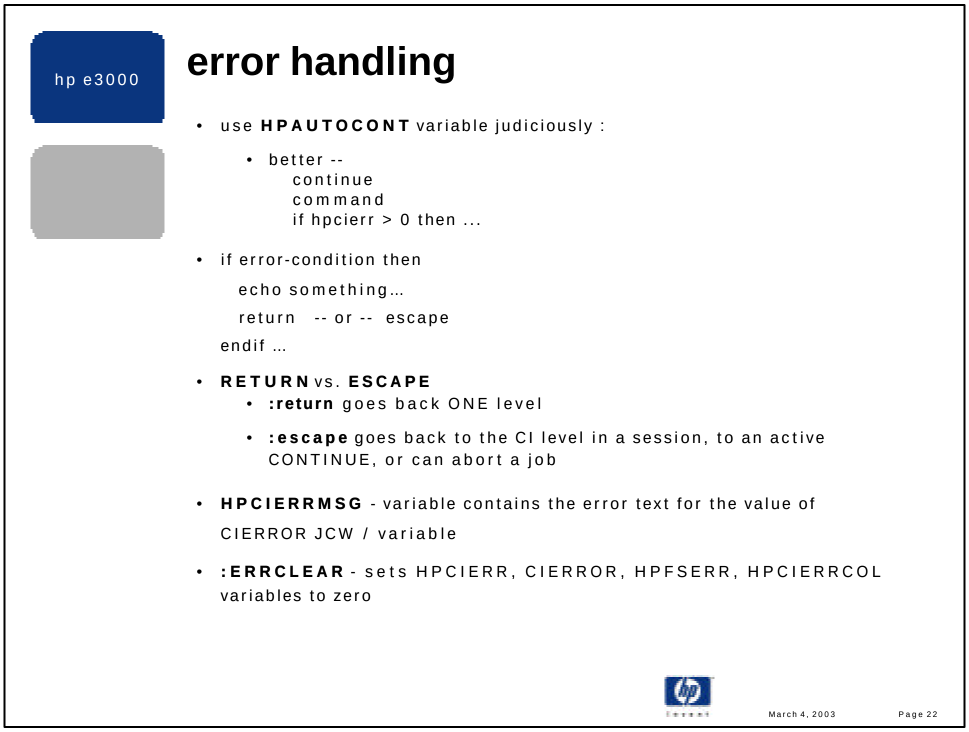
```
:readntry
90 msecs to read 24 records.
---> Almost 3 times faster than MSG files
---> 8 times faster than the PRINT method!


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

March 4, 2003 Page 21

•Use the HPSTDIN variable to get the name of the redirected input file, so there is less hard coding of the temporary file names.

•This example doesn’t do anything with the contents of the file. Each record is placed in the variable REC, one record overwriting the previous.

•The *input()* function is used here, rather than the INPUT command in the other examples. A motivation for doing this is to eliminate the body of the WHILE loop, which increases performance. The silly test for “<> chr(1)” is done to satisfy the requirements for an expression and is expected to always be true.



hp e3000

error handling

- use **HPAUTOCONT** variable judiciously :
  - better --

```
continue
command
if hpcierr > 0 then ...
```

- if error-condition 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

March 4, 2003 Page 22

•HPAUTOCONT = true is sometimes useful, but can be a dangerous practice. It allows every command to behave as if it is proceeded by a :CONTINUE command. This may be desired for some of the commands in a job or script, but not necessarily **all** of the commands. I find it safer and more reliable to leave HPAUTOCONT set to false (default) and to use an explicit **:continue** in front of each command that I want to test for success or failure. This allows me to control the behavior of the script, e.g., I can do some cleanup if an error occurs, and at the same time, it permits the script to abort if an unexpected failure arises.

•I think that scripts are more maintainable and easier to read if the error checking portion reports the trouble and then simply exits. This is preferred to using constructs such as:

```
if <error> then
   report problem…          # Don't handle errors this way if possible!
else
   execute more code…

if <error> then
   report error              # Do handle errors this way if possible!
   return
endif
```

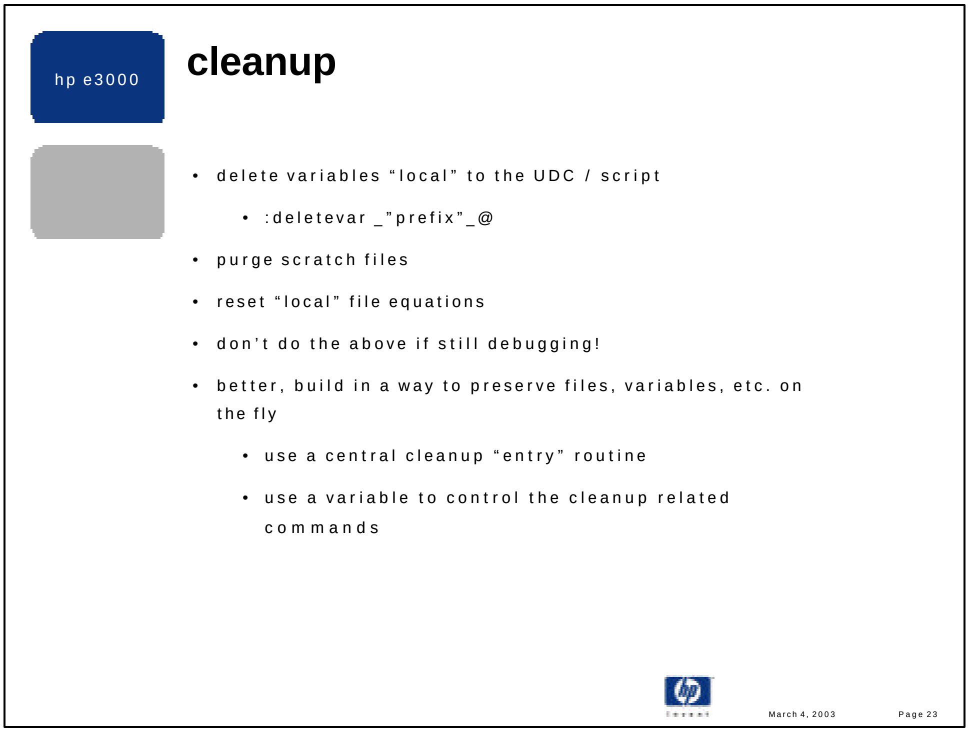
•RETURN causes execution to resume in the calling environment. RETURN is useful as a method of exiting an alternate entry in a script or UDC. RETURN does not set CI error related variables and cannot directly cause the calling environment to abort. Returning from a script closes the file; however that is the only cleanup done automatically by the system. Scratch files, file equations, variable, etc., in general, should be cleaned up prior to exiting a script or UDC.

•ESCAPE causes execution to resume at the main CI level for sessions, and at the calling environment **if** a :continue proceeded the invoking command. If the calling environment is a job and the invoking command was not "protected" by a :continue then ESCAPE will abort the job. Additionally, ESCAPE can set CIERROR and HPCIERR to an error number, but the default is to not alter these variables. ESCAPE mimics to some degree the TRY/ RECOVER / ESCAPE construct provided by Pascal, which is used by a large portion of the MPE/iX operating system. ESCAPE is useful when a script or UDC needs to duplicate the CI's error handling. This duplication can be further improved by exploiting the HPFSERR and HPCIERRCOL predefined variables, which provide the associated file system error (if any), and the column position of the offending command line parameter, where the CI would locate the caret ('^") in an error message.

•The HPCIERRMSG string variable contains the error/warning message associated to the current value of the CIERROR variable. Note that message inserts (like the offending filename) are not, and thus, some messages will contain "!" as insert place holders.

•The ERRCLEAR command is useful in the initialization part of scripts to set all error related predefined CI variables to zero. It is over twice as fast compared to setting all four variables individually. It is approx 25% faster than setting only HPCIERR and CIERROR to zero separately. It is slightly slower (7%) than setting only CIERROR to zero.

•HPCIERR is signed -- CI warnings are negative, CI errors are positive. CIERROR contains the absolute value of HPCIERR -- thus there are no CI warnings with the same absolute value as a CI error. The CI keeps HPCIERR and CIERROR in sync, but users can change their values independent of each other.



hp e3000

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, 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

March 4, 2003 Page 23

Some cleanup examples:

•Using a cleanup “entry” routine -

```
…
elseif  “!entry” = “cleanup” then
     # do all script cleanup here
     if finfo(_foo_file,”exists”) then
         purge !_foo_file
     endif
     if _foo_used_feq then
         reset !_foo_feq
     endif
     echo  End of ![basename(hpfile)] ...
     deletevar _foo_@
     escape 0
endif
```

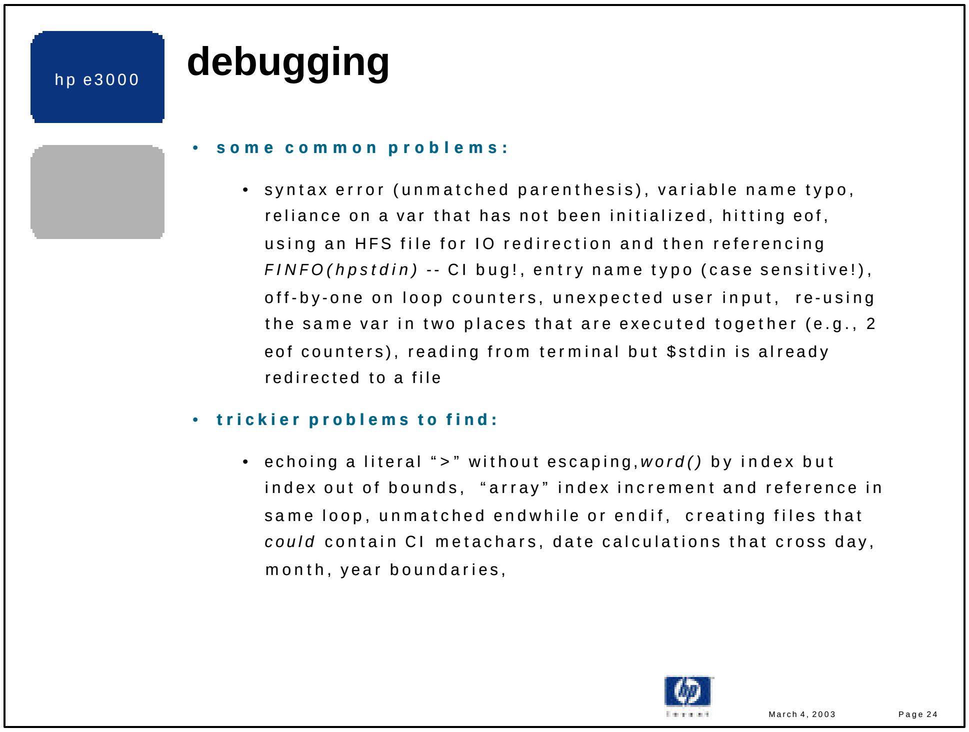
•Allowing variables and files to be saved or deleted on the fly -

```
…
elseif  “!entry” = “cleanup” then
     if bound(_foo_debug) then
         escape
     endif
     # do all script cleanup here
     if finfo(_foo_file, “exists”) then …
```

or

```
…
!_foo_del  reset !_foo_feq
!_foo_del  purge !_foo_file
!_foo_del  deletevar _foo_@

… somewhere  _foo_del is set as:
setvar _foo_del “#”    -- or --   setvar _foo_del “”
```

•Scripts and UDCs cannot trap break or other signals (like pending process aborts) thus it is not possible to clean up correctly in all



hp e3000

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 IO 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 (e.g., 2 eof counters), reading from terminal but $stdin is already redirected to a file
- **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,

March 4, 2003 Page 24

•insert echo/showvar statements, revealing a variable's value and/or a location in the script.

•don't initially delete variables and scratch files.

•turn on command tracing (HPCMDTRACE) within suspect sections of the script -- implies omitting OPTION NOHELP too.

•check OPTION RECURSION setting in UDCs with entry points.

•force an unexpected condition by hard-coding the rare value – may be the only way to test certain code paths.

•steal working fragments from other scripts.

•add your own tracing into complex scripts, via a "hidden" command line parm or a special variable.

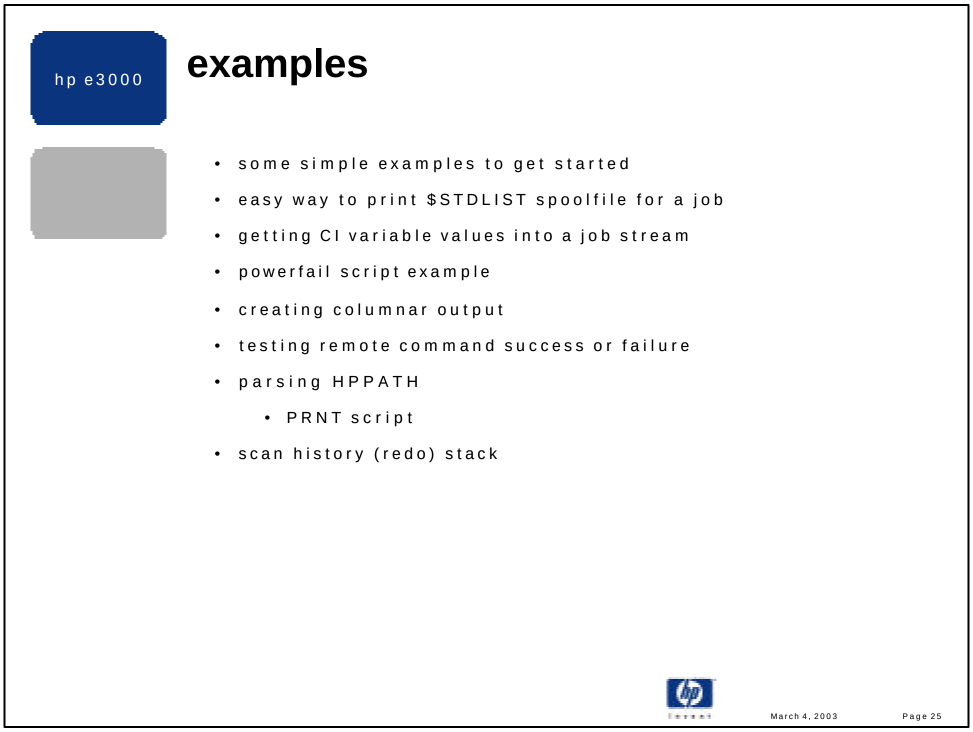
•use HPLEAPYEAR, HPDATETIME for date calculations, e.g. :

```
    setvar tmp hpdatetime        # reference the predefined var only once
    setvar tmpdate lft(tmp,8)    # just the yyyymmdd part
    setvar tmptime str(tmp,9,6)  # just the hhmmss part
```

Don't do below for three reasons:

```
    setvar tmpdate "20!hpyear"+"!hpmonth"+"!hpdate"
```

1) may need leading zeros in the string date,
2) use HPYYYY (4 digit string) instead of HPYEAR (2 digit integer),
*3) what happens if the month changes after HPMONTH is referenced?

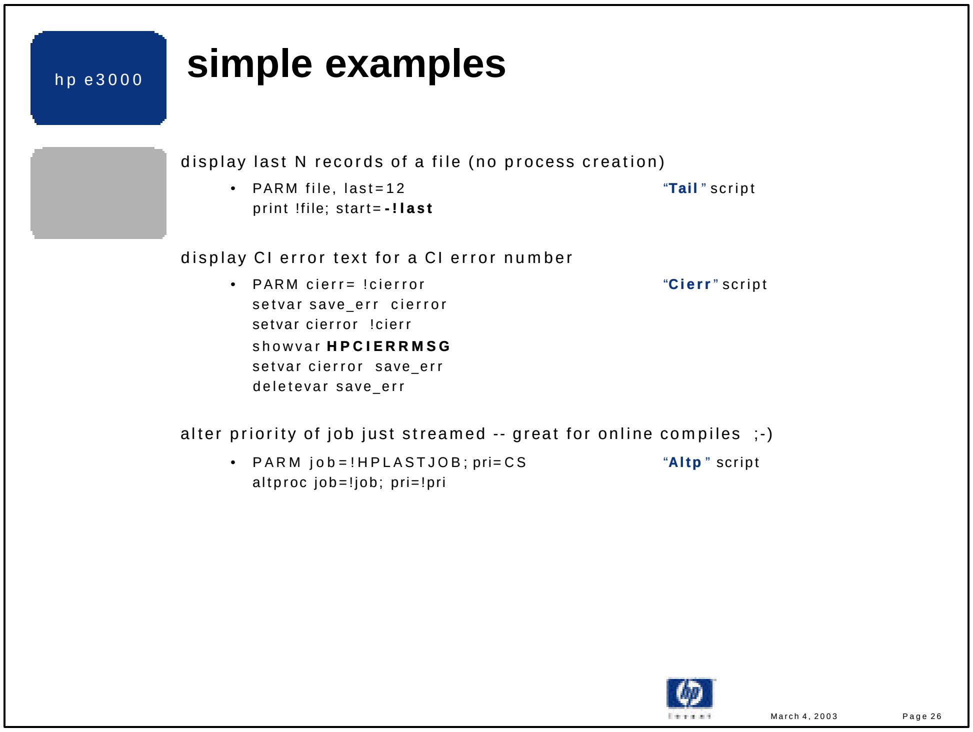


hp e3000

examples

- some simple examples to get started
- easy way to print $STDLIST spoolfile for a job
- getting CI variable values into a job stream
- powerfail script example
- creating columnar output
- testing remote command success or failure
- parsing HPPATH
  - PRNT script
- scan history (redo) stack

March 4, 2003 Page 25



hp e3000

simple examples

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

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

display CI error text for a CI error number

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

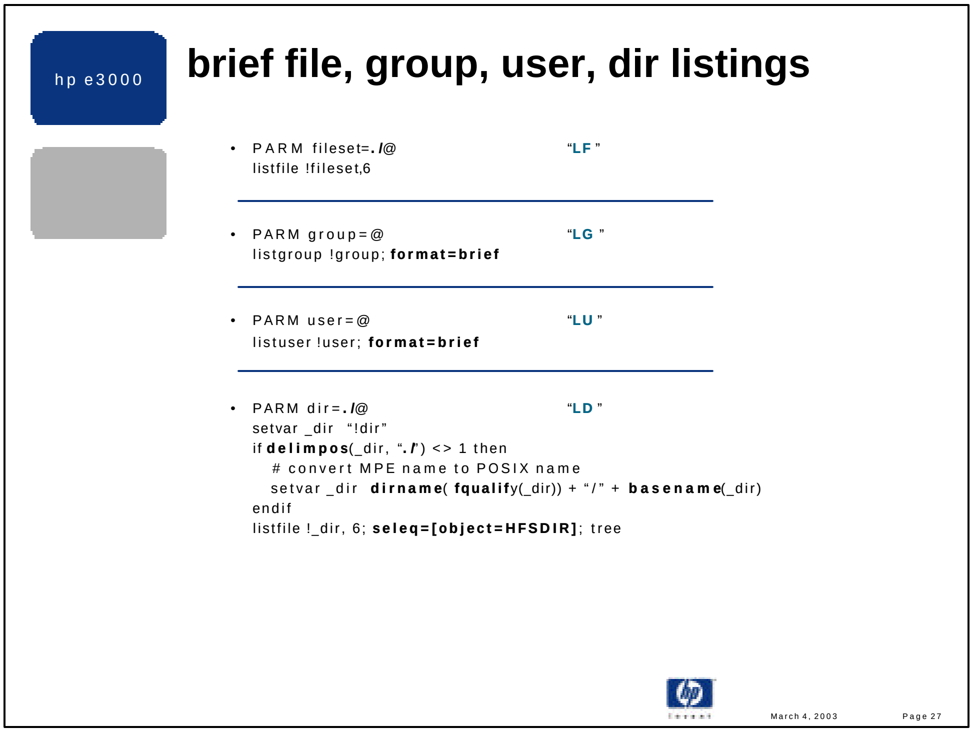
alter priority of job just streamed -- great for online compiles ;-)

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

March 4, 2003   Page 26

•The tail script has no process create overhead, unlike the POSIX tail.hpbin.sys program.

•The HPCIERRMSG CI variable contains the error text for the error defined by the current value of the CIERROR variable (JCW). Note that message inserts values, that would normally be displayed by the CI in processing an error, are not inserted via HPCIERRMSG.



hp e3000

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
```

March 4, 2003 Page 27

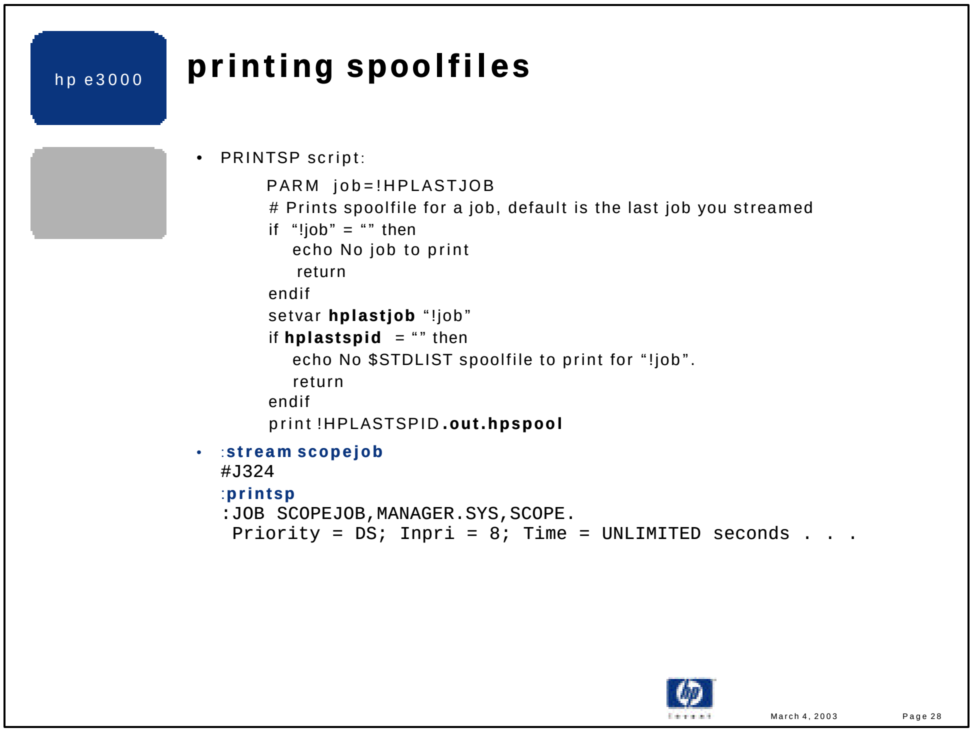
•The last example (LD) shows the BASENAME, DELIMPOS, DIRNAME and FQUALIFY functions being used.

•*DELIMPOS(_dir, “./”) <> 1* tests if the directory name in _dir starts with a dot or slash, and thus is a POSIX named directory. The FSYNTAX function could have been used for this purpose too.

•An MPE name can be converted to a POSIX name easily:

> •DIRNAME returns the directory portion, in POSIX syntax, of a filename, but does not qualify the name.
>
> •FQUALIFY qualifies the name in _dir. Now, DIRNAME will return the absolute path of the name in _dir1, less the file portion of the name.
>
> •BASENAME returns just the base (file) portion of the name in _dir. When appended to the result of DIRNAME(…) the result is a fully qualified, POSIX name.

•LISTFILE will search for just POSIX (HFS) named directories (seleq=[object=hfsdir]), and the TREE option tells LISTFILE to search recursively, following all sub-directories.



hp e3000

printing 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 seconds . . .
```

March 4, 2003 Page 28

•The default value for the parameter "JOB" is the job number of the job most recently streamed by you (HPLASTJOB variable).

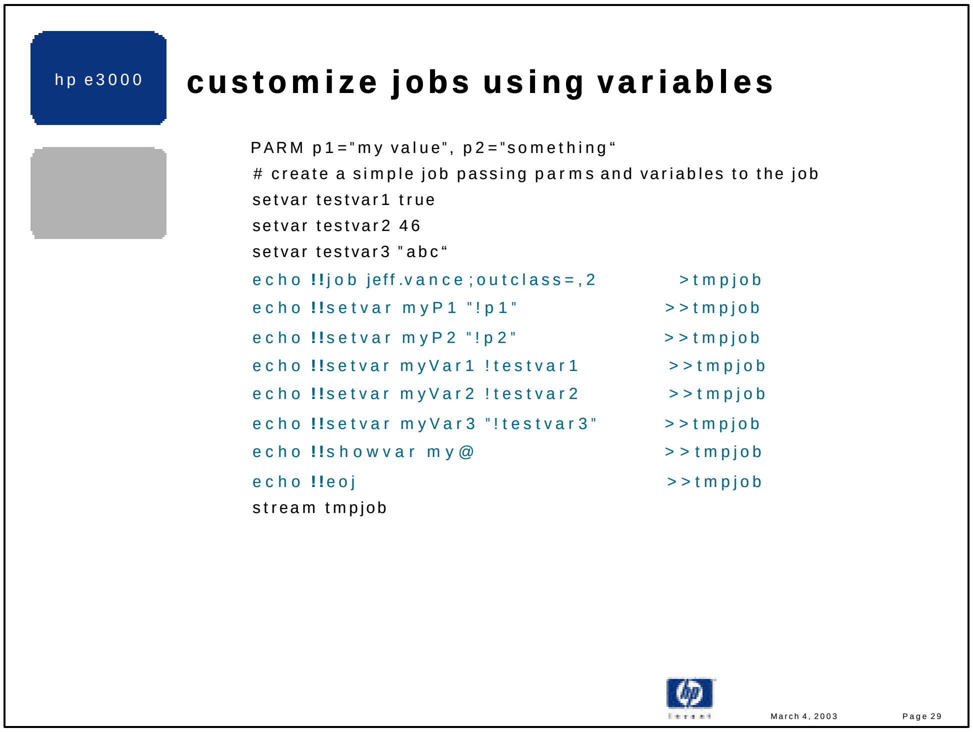
•If you have not streamed a job (or HPLASTJOB is empty for some other reason) the script reports an error and exits.

•The HPLASTSPID variable contains the spoolfile number (Onnnn) for the $STDLIST spoolfile for the job referenced in the HPLASTJOB variable. HPLASTSPID is not a qualified MPE filename, so the ".out.hpspool" suffix needs to be appended.

•All output spoolfiles live in @.OUT.HPSPOOL.

•Could be improved by saving the value of HPLASTJOB before setting it to the JOB parameter, and then reinstating this saved value before the script ends.

•Could check for the existence of "!hplastspid.out.hpspool" before trying to print it.



hp e3000

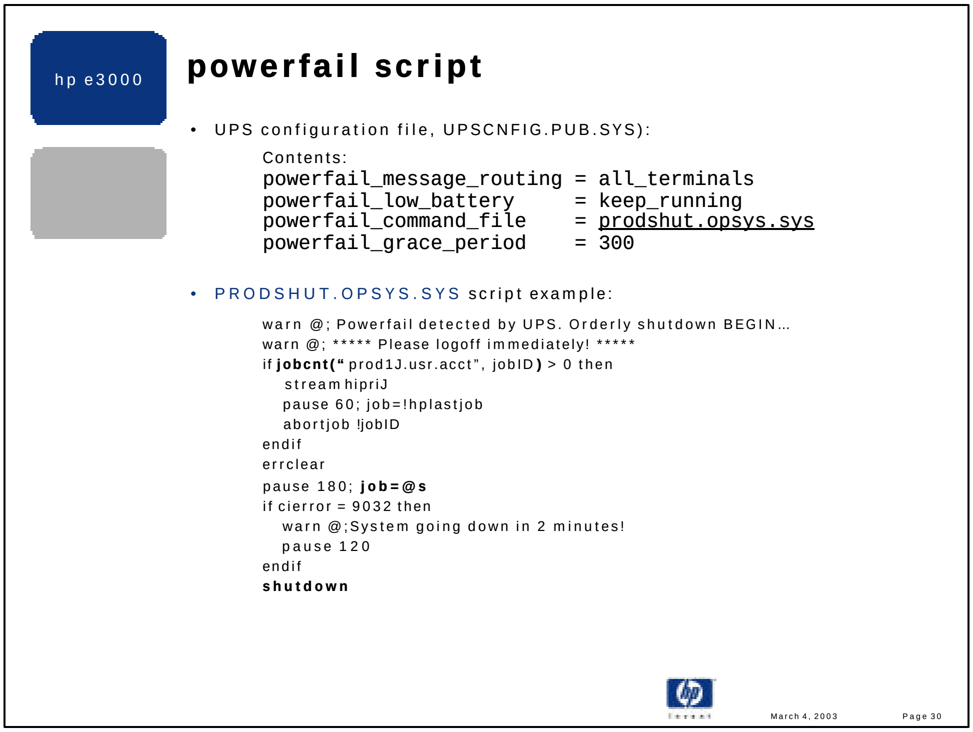
customize jobs using variables

```
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
```

March 4, 2003 Page 29

•Simple script creates a simple job stream. To stream the job you need to execute this script.

•Easy method to pass parameters, variables, even user input to a job stream. Need to be careful on use of quotes. Remember all parameter references need to be explicitly referenced, thus the type (string, boolean, integer) of the parm is lost.



hp e3000

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
```

March 4, 2003 Page 30

UPSMON accepts a default configuration file named UPSCNFIG.PUB.SYS. This file can be overridden via UPSUTIL’s NEWCONFIG command, which prompts for a simple configuration file (flat ASCII, 32 - 128 bytes wide, numbered or unnumbered). The UPS config file consists of the following (each occupying its own unique record): the fully qualified MPE file name (it’s own name) must be the first record. The remaining contents (records) are optional and in the form: <u>config keyword</u> = <u>value</u>. Below is the configuration file syntax:

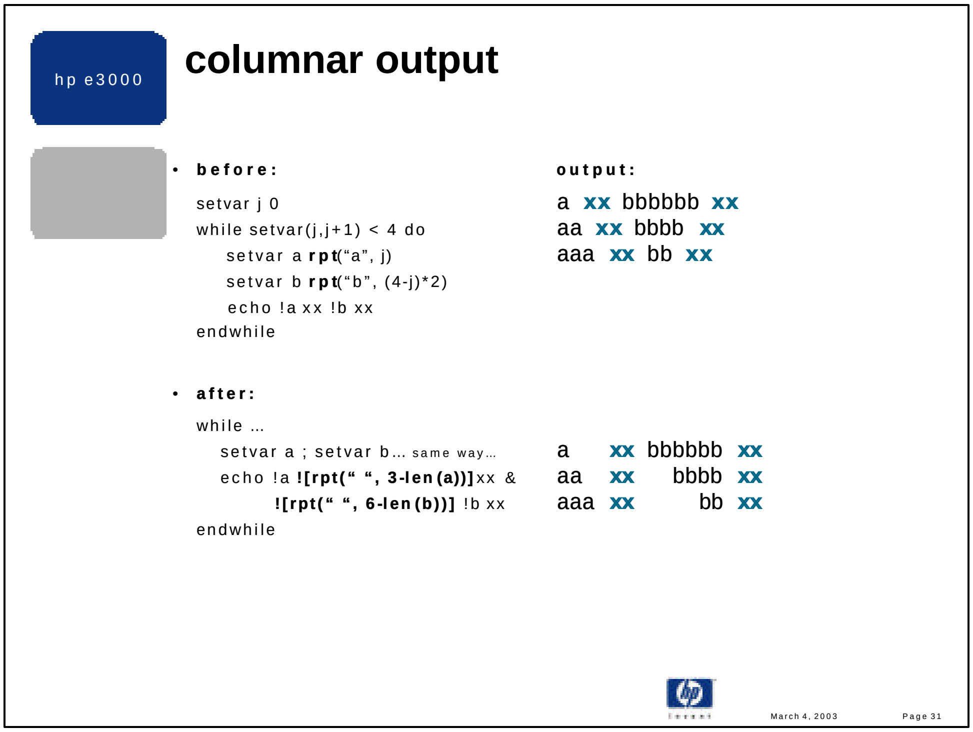
```
Config_file_name
powerfail_message_routing = <all_terminals | console_only>
powerfail_command_file    = <MPE filename> [;parm1 parm2 … parmN] *
powerfail_grace_period    = <number of seconds, 0 .. 1800> *
powerfail_low_battery     = <system_abort | keep_running (provides ~120 extra sec)> *
```

* mainline for 7.5 and patch for 7.0

The powerfail_grace_period specifies the number of seconds to wait, after detecting a powerfail, prior to invoking the script named in the powerfail_command_file setting. After “powerfail_grace_period” seconds expires the script is executed. This script can perform needed system cleanup, but caution is necessary since the system is running on batteries at this point. The most important consideration is to ensure that all disk writes are consistent.

If the power remains off at some point the battery will run out. The power_fail_low_battery setting allows you to squeeze approx 2 more minutes from your shutdown script before the system bellies up. The default is not to play Russian Roulette with your data, thus the system is aborted at the 2 minute warning. However, a knowledgeable, risk adverse system manager may specify “keep_running” to gain more time for their cleanup script to complete. The risk is that if the script fails to complete in the remaining ~2 minutes the system will fail and disk states may be corrupted.

Note the new SHUTDOWN CI command, which also supports a RESTART option to allow the system to be restarted after the shut down is complete. OP capability is required to issue SHUTDOWN from the CI.



hp e3000

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
```

March 4, 2003 Page 31

•The "align" script demonstrates using the *rpt()* function to pad output with a variable number of spaces, based on the length of the data fitting in a field.

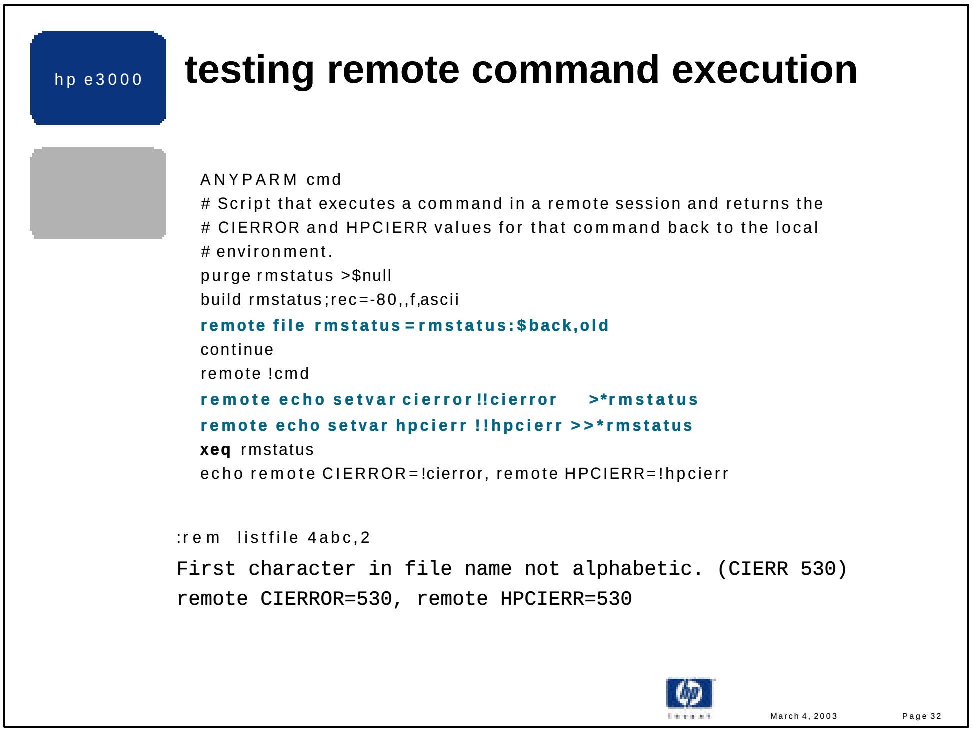
•Variable A is set from 1 to 3 "a"s. Variable B is set from 6 to 2 "b"s, in this example.

•The **before** script shows staggered columns – the "xx"s should form a column.

•When ![rpt(" "…)] is used just ***after*** echoing a value, the proceeding value becomes left justified.

•When ![rpt(" "…)] is used just ***before*** echoing a value, the proceeding value becomes right justified.

•The **after** script shows a left justified variable followed by a right justified variable.



hp e3000

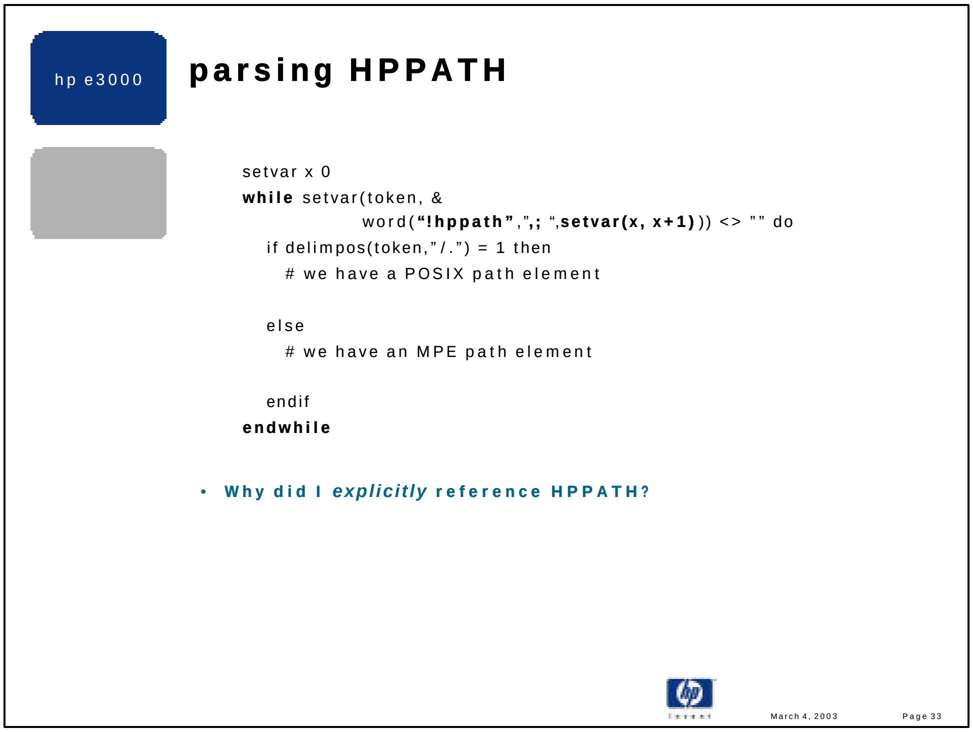
testing remote command execution

```
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 >$null
build rmstatus;rec=-80,,f,ascii
remote file rmstatus=rmstatus:$back,old
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
```

March 4, 2003 Page 32

- Available on Jazz at: http://jazz.external.hp.com/src/scripts/index.html
- Thanks to Craig Fairchild for this method.
- Creates a “status” file which is filled in on the remote system, using remote variable values, but the file lives on the local system.
- The contents of this “status” file are two commands which get executed on the local system. These commands set the CIERROR and HPCIERR variables to their respective values on the remote system.
- The double bangs (!!) prevent the local values of CIERROR and HPCIERR from being referenced. Instead, “!CIERROR” and “!HPCIERR” are executed (and thus referenced) on the remote system.
- Once the “status” file/script is executed on the local system (where it lives – no dscopy needed) the local CIERROR and HPCIERR are set to the corresponding values from the remote CI.



hp e3000

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 did I *explicitly* reference HPPATH?**

March 4, 2003 Page 33

•HPPATH defaults to “!!HPGROUP, PUB, PUB.SYS, ARPA.SYS”. Users can add their own path elements, which can be contained in one or more CI variables. HPPATH elements can name an MPE group, an MPE group.account, an absolute POSIX named directory, or a relative POSIX named directory. HPPATH elements are separated by a comma, semicolon or a space.

•The 2nd argument to *word()* is the delimiter list, which is passed as the same delimiters used in HPPATH.

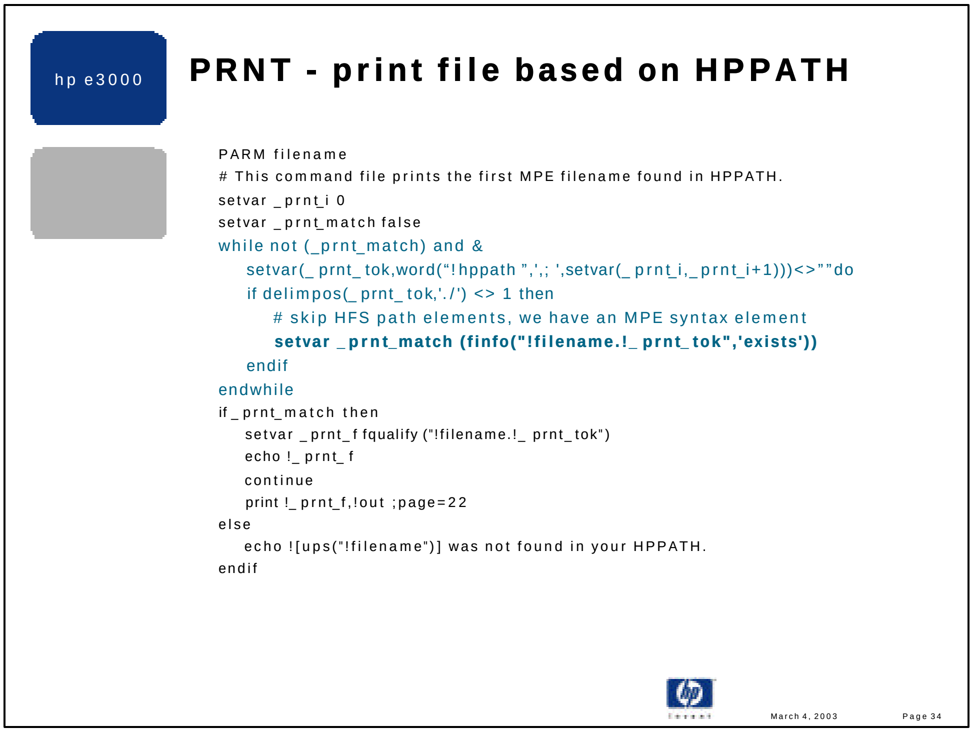
•The 3rd argument to *word()* is the token number to extract. In this case, the token number is incremented by 1, starting at 0. Thus the 1st token is parsed out, then the 2nd, then the 3rd, etc.

•The loops stops once the “next” token in HPPATH is empty, which usually means you reached the end of HPPATH. Using an index in *word()*, which is the 5th argument, is trickier since this index value cannot be incremented beyond the end of the string being parsed. If that occurs then *word()* reports an error and stops. So usually is is easier to extract the Nth word rather than a word starting a position N.

•The most important aspect in this otherwise simple loop is to **explicitly** reference HPPATH, which resolves variables that are contained in the HPPATH value, like !HPGROUP, for instance. Failure to do this will cause the *delimpos()* statement to test the variable name (actually the “!”) rather than its value.

•The *delimpos()* function returns the string index where the first of the supplied delimiters characters is found in the passed string. If the extracted token begins with a “.” or a “/” then a 1 is returned and we know this path element is a POSIX directory name. Otherwise we assume and MPE group[.acct] name.

•Being able to parse HPPATH correctly allows you to create scripts that locate unqualified commands(files), or scripts that act upon a file you find by applying HPPATH elements to a base filename. One possible action would be to print the file, and this example is shown next.



hp e3000

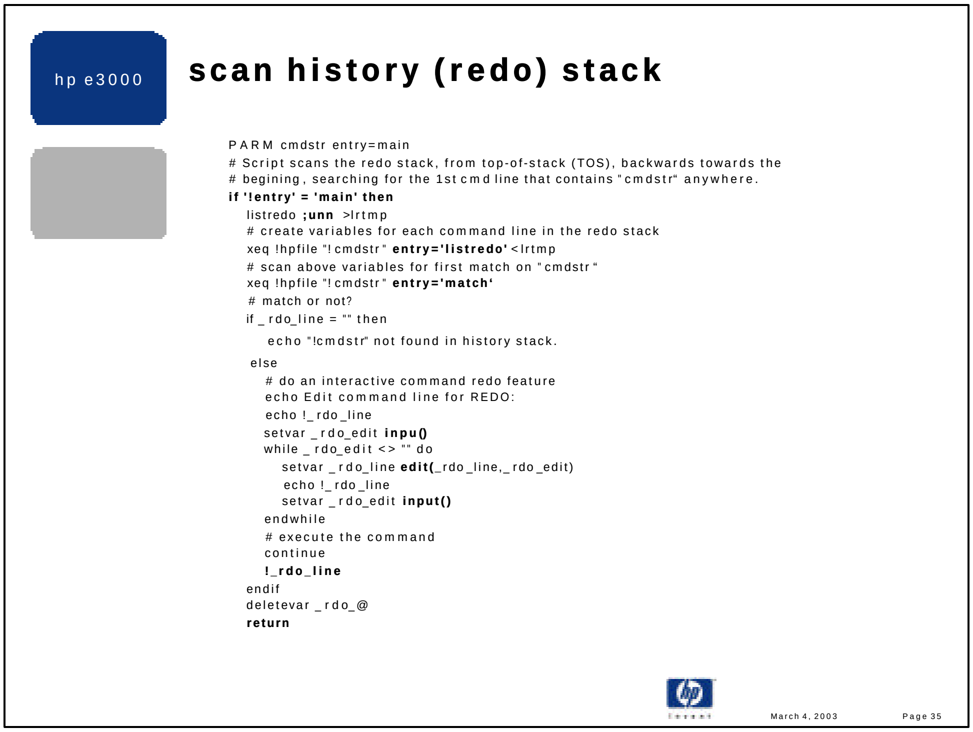
PRNT - print file based on HPPATH

```
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
```

March 4, 2003 Page 34

- •Available on Jazz at: http://jazz.external.hp.com/src/scripts/index.html
- •The WHILE loops parse HPPATH, appends each MPE group[.acct] element to the filename parameter, and tests to see of the resulting filename exists.
- •If the file exists the WHILE loop stops, the qualified filename is displayed, and the file is printed to $STDLIST with 22 lines per "page". Otherwise, an error is reported.
- •Could be enhanced to work with POSIX named "filename" arguments and POSIX path elements.
- •I use this script all the time !
- •Example:

```
:echo !hppath
UDCS,PUB,PUB.SYS,ARPA.SYS,scripts.sys,hpbin.sys
:prnt rem
REM.UDCS.VANCE
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 >$null
. . .
```



hp e3000

scan history (redo) stack

```
PARM cmdstr entry=main
# Script scans the redo stack, from top-of-stack (TOS), backwards towards the
# begining, 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 inpu()
      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
```

March 4, 2003 Page 35

•Available on Jazz at: http://jazz.external.hp.com/src/scripts/index.html

•The REDO and DO commands let you re-execute (and edit) a command from the CI's history (redo) stack. The size of this stack is defined by the HPREDOSIZE variable, and the default is 20, meaning the most recent 20 commands are kept in this area. Commands issued in jobs or from UDCs or scripts are not posted to the history stack. The REDO, DO, and ERRCLEAR commands are also not posted.

•The LISTREDO command displays the contents of the history stack, starting at the least recent command and ending at the most recent command. However, when you REDO/DO using a string, the CI searches the history stack beginning at the most recent command (top-of-stack), looking backwards towards the beginning of the stack. The first entry found **anchored in column 1** with the string parameter supplied to DO/REDO is processed. Thus, if you want to re-execute the most recent command which **ends** with "foo", the CI's DO/REDO mechanism falls short. But, the RDO script here works!

•Thanks to Ray Shahan, who provided me a script that scans the history stack and got me interested in this script.

•The "RDO" script uses two alternate entry points ("listredo" and "match") and a "main" entry.

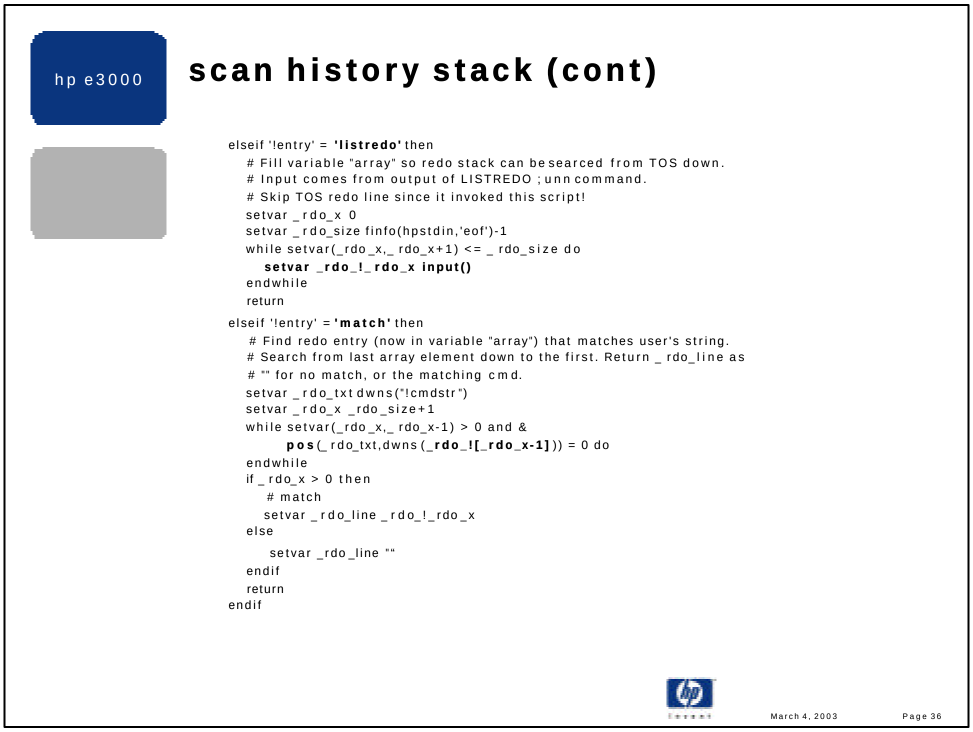
•The UNN option to the LISTREDO command suppresses command line numbers and thus eliminates some parsing.

•There is a subtle but very important difference in the use of the **INPUT** command vs. the *input()* function:

> •The INPUT command does not modify the named variable (if it already exists) when the user provides no input, such as just hitting <return>
>
> •The *input()* function always returns what the user inputs, and thus, in this usage, causes the variable to always be set, even if set to a null (empty) value.
>
> •These differences are important in the WHILE loop above. If the INPUT command was used instead of the input function, when the user just pressed <return> to stop editing the command line, the _rdo_edit variable would not be modified. Thus, _rdo_edit would contain the last value it had, which could have been the last edit applied to the command image. This most likely would result in an infinite while loop. Use the *input()* function or add ;DEFAULT="" to the INPUT command to solve this null user input issue.
>
> •:help inputfn show the *input()* function.

•The *edit()* function accepts a redo-like edit string and applies it to the first parameter. The result, after applying the edit, is functionally returned.

•The edited command is executed in this script by explicitly referencing the variable that contains the command image.



hp e3000

scan history stack (cont)

```
elseif '!entry' = 'listredo' then
   # Fill variable "array" so redo stack can be searced 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
```

March 4, 2003 Page 36

•The “listredo” entry reads input from the TEMP file that contains the output from the LISTREDO command. The HPSTDIN predefined variable contains the name of the file used to capture LISTREDO’s output. The TOS command in the history stack is omitted since this command will always be the command that invokes this script.

•Since we want to scan the history file from TOS back towards the beginning, but the order output by LISTREDO is oldest to newest, we collect each history stack image into a CI variable “array”:

•setvar _rdo_!_rdo_x *a-value* creates a numbered CI variable which can be easily indexed.

•The “match” entry scans each “array” element in reverse order from how it was created. This gives us our TOS -> beginning scan direction, which is the same direction used by the CI’s REDO and DO commands.

•The *pos()* function provides the functionality to match the desired string anywhere in the command image. Of course we downshift the user’s matching string and each command image before applying the *pos()* function. The case shift is done temporarily so that the original image is visible for editing and re-execution.

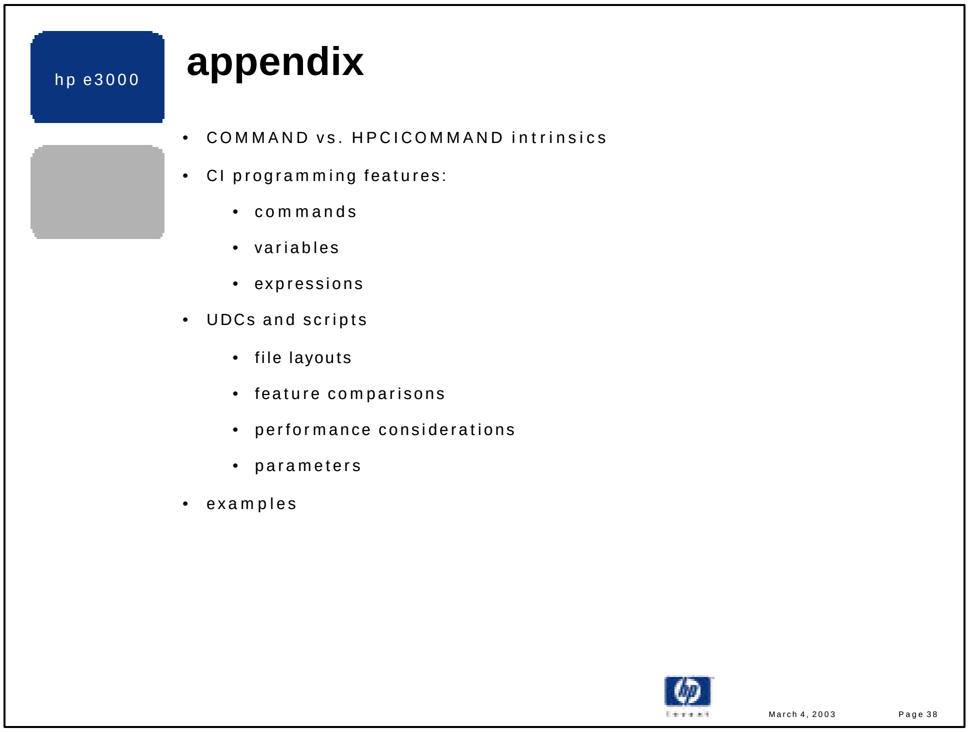
•If the DELETEVAR in the *main* entry were commented out you would see the following CI vars created by this script. Note the format of an “array”.

```
:showvar _rdo_@
_RDO_X = 2
_RDO_SIZE = 20
_RDO_1 = lf p@.udcs
_RDO_2 = cg udcs
_RDO_3 = diff prnt prnt.scripts.sys
_RDO_4 = diff "prnt prnt.scripts.sys“
_RDO_5 = hped prnt
_RDO_6 = diff "prnt prnt.scripts.sys“
_RDO_7 = copy prnt,.scripts.sys
_RDO_8 = purge prnt
_RDO_9 = lf
_RDO_10 = prnt tok
_RDO_11 = prnt rem
_RDO_12 = shiwvar hppath
_RDO_13 = showvar hppath
_RDO_14 = prnt prnt
_RDO_15 = prnt rem
_RDO_16 = echo !HPPATH
_RDO_17 = lr
_RDO_18 = rdo tok
_RDO_19 = hped rdo
_RDO_20 = lr
_RDO_TXT = udcs
```

hp e3000

scan history stack (cont)

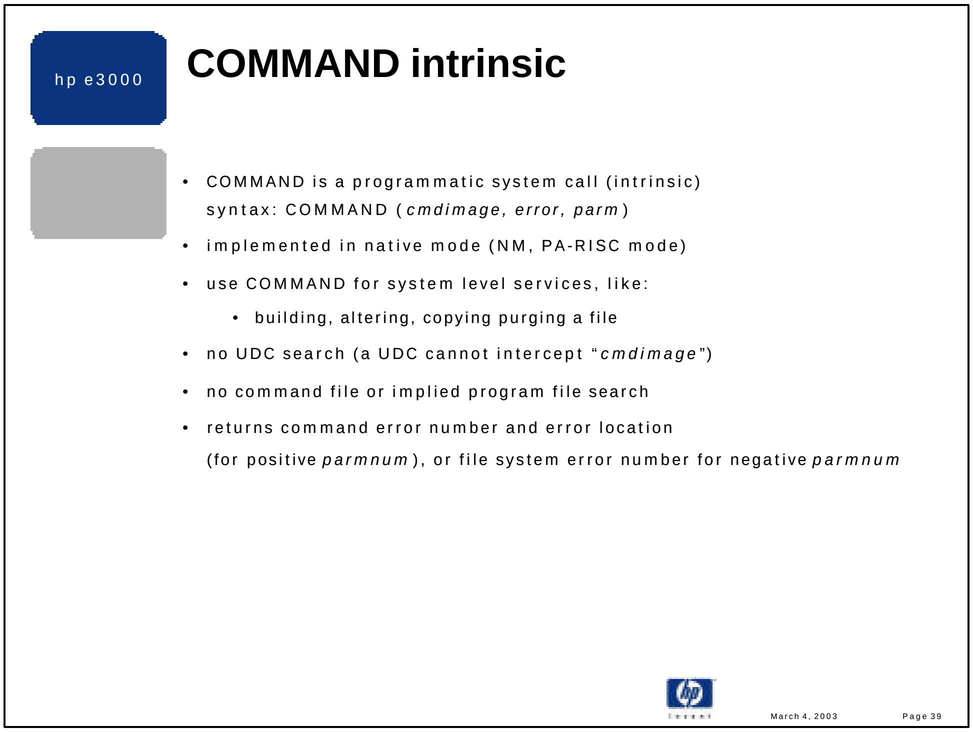
```
:listredo
       1) listf,6
       2) Showtime
       3) run editor
       4) run edit.pub.sys
       5) hpedit rem
       6) listredo ;unn
       7) showjob
       8) me
       9) spme
      10) showproc 0
      11) listredo

: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. …
FRI, FEB 28, 2003,  5:21 PM
```



hp e3000

appendix

- COMMAND vs. HPCICOMMAND intrinsics
- CI programming features:
  - commands
  - variables
  - expressions
- UDCs and scripts
  - file layouts
  - feature comparisons
  - performance considerations
  - parameters
- examples



hp e3000

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*

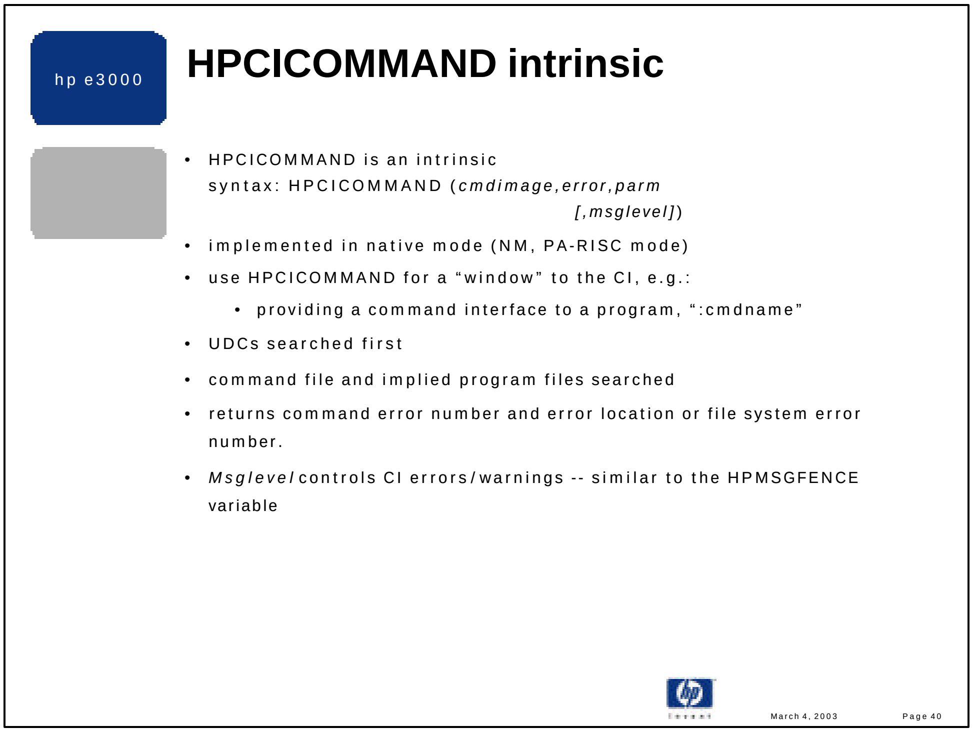
March 4, 2003 Page 39

COMMAND is a user-callable system level API that executes the command passed in as the *cmdimage* argument. *Cmdimage* can name any built-in MPE command including the XEQ command, which directly executes scripts and program files. *Cmdimage* cannot name a UDC or imply a script or program filename. *Cmdimage* must be terminated with an ASCII carriage return (#13) and cannot exceed 512 bytes, including the CR.

It is recommended to call the COMMAND intrinsic to obtain a **system service**, such as creating a file, etc. Other intrinsics may provide the same function, yet it is sometimes easier to call COMMAND since the programmer is likely familiar with the interactive CI command that provides the desired service. COMMAND is recommend over HPCICOMMAND in this case since the *cmdimage* passed to COMMAND cannot be intercepted by a UDC. For example, to create a new file one could call COMMAND passing the string: "build filename". The built-in MPE BUILD command will be executed, even if there exists a UDC named "BUILD" -- which may do anything, and may not actually create the file at all.

The *error* argument returns zero, or a CI error number in case of a command execution error. This is the same error number reported if *cmdimage* is executed interactively, and is the value of the predefined CIERROR JCW/variable. If *cmdimage* executes with an error or warning there is no indication of this fact, other than the *error* return value. Specifically, there is no error message reported to $STDLIST, and the CIERROR and HPCIERR CI variables are not modified. In fact COMMAND operates by locally setting the HPMSGFENCE variable to 2, thus suppressing all CI error and warning messages. This is verifiable by executing SHOWVAR programmatically via the EDITOR, e.g.:

```
:showvar hpmsgfence   (= 0)
:editor
/:showvar hpmsgfence  (= 2)  Note: a leading ":" causes editor to call COMMAND with the
                             string following the ":".  This is common for many programs.
```



hp e3000

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

March 4, 2003 Page 40

HPCICOMMAND is a user-callable system level API that executes the command passed in as the *cmdimage* argument. *Cmdimage* is identical to that passed to the COMMAND intrinsic, except that it can name UDCs, scripts and program filenames, in addition to most of the built-in MPE commands. Due to implementation constraints the following built-in commands cannot be executed via COMMAND or HPCICOMMAND:

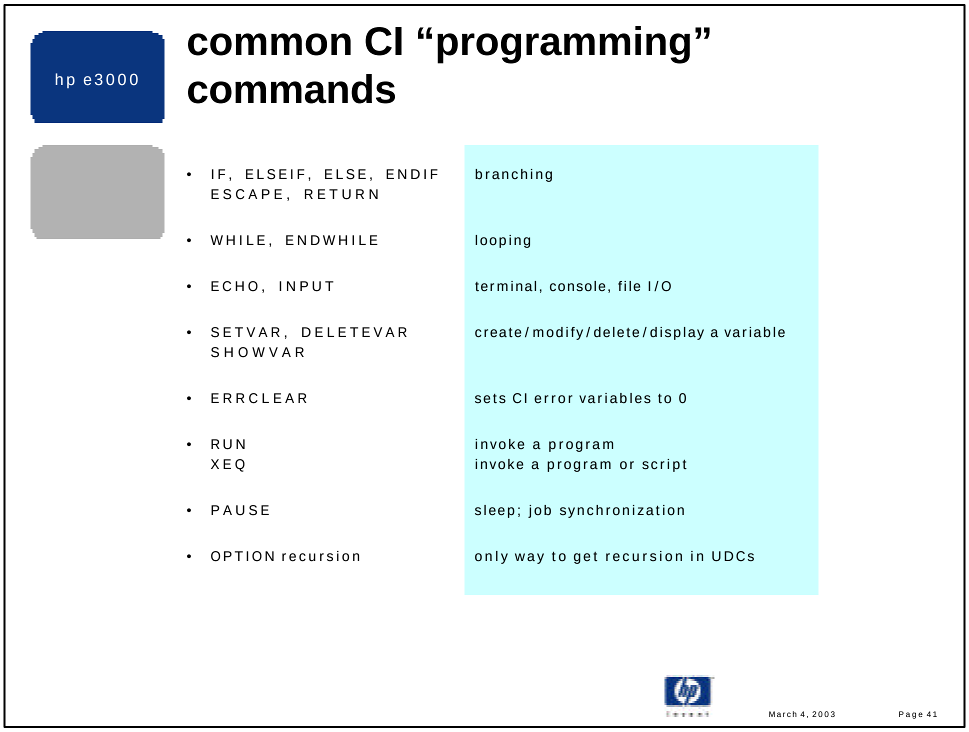
> ABORT, BYE, CHGROUP, DATA, DISMOUNT, DO, EOD, EOJ, EXIT, HELLO, IMF, IMFMGR, JOB, LISTREDO, MOUNT, NRJE, REDO, RESUME, RJE, SETCATALOG, VSUSER.

However, the remaining 245 CI commands can all be executed programmatically via COMMAND or HPCICOMMAND.

It is recommended to call the HPCICOMMAND intrinsic as a simple way for a program to **provide a “window” to the CI**. It is common for MPE programs to accept a leading colon (“:”) to indicate that what follows is a CI command to execute, and not a command recognized by the program. A nice feature of HPCICOMMAND is that it executes UDCs, which makes the “window” to the CI more natural and powerful for the end user.

The *error* argument and *parm* arguments work the same as in COMMAND, except HPCICOMMAND will set the CIERROR and HPCIERR CI variables to 0, or an error number if the passed in command fails.

The optional *msglevel* parameter is unique to HPCICOMMAND and controls the HPMSGFENCE setting described in the COMMAND notes. By default *msglevel* is passed as 0, meaning that all CI errors and warning messages are written to $STDLIST, just as if *cmdimage* was executed interactively. *Msglevel* can be set to any legal HPMSGFENCE value and causes HPCICOMMAND to control error, warning and some diagnostic output identically to how the CI interprets HPMSGFENCE. Entering HELP hpmsgfence will show the details.



hp e3000

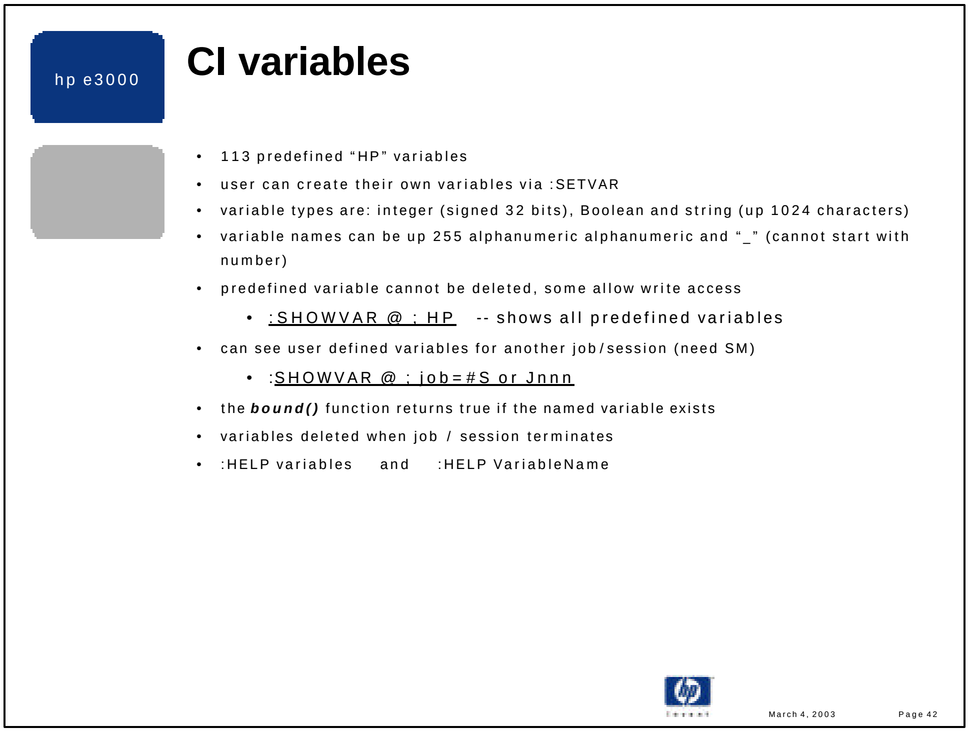
common CI "programming" commands

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

March 4, 2003 Page 41

•The CI supports commands that provide the basic requirements of a programming language: storage, branching/looping and I/O. The CI expands on these necessities by providing a rich set of predefined variables and functions, many of which are described later.

•There are 270 CI commands as of release 7.5, but the 18 commands above are common in most scripts and UDCs that have any level of complexity, such that they are considered a "program".



hp e3000

CI variables

- 113 predefined “HP” variables
- user can create 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
  - <u>:SHOWVAR @ ; HP</u> -- shows all predefined variables
- can see user defined variables for another job/session (need SM)
  - :<u>SHOWVAR @ ; job=#S or Jnnn</u>
- the ***bound()*** function returns true if the named variable exists
- variables deleted when job / session terminates
- :HELP variables and :HELP VariableName

March 4, 2003 Page 42

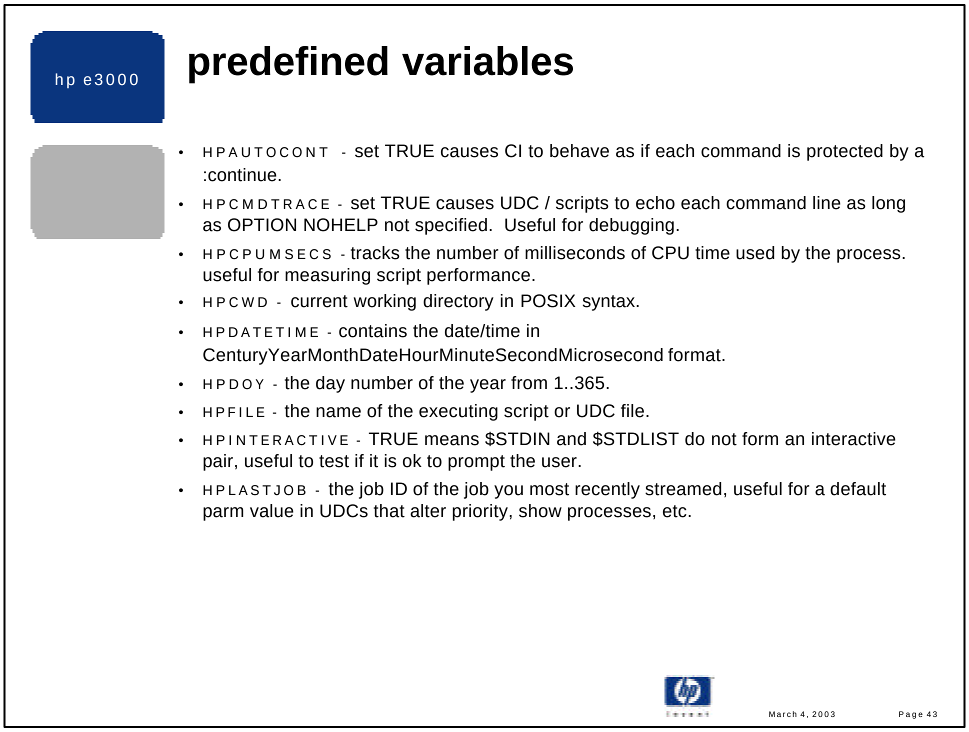
•CI variables can be strings (up to 1024 bytes in length), 32 bit signed integers or boolean TRUE/FALSE. There is not support for 64 bit integers or unsigned 32 bit numbers.

•See the slides on variable arrays for a method to determine the maximum number of CI variables that can be defined. This maximum is a function of the length of the variable’s name and the length of its value. The longer your variable names and/or their values the fewer variables can be stored by the CI. A typical range is 8,000 to 9,000 user variables can be defined.

•A summary of all of the predefined variables is available by entering HELP VARIABLES. The details for a specific variable can bee seen by entering <u>HELP varname</u>. For example, if you have trouble remembering the new values for the HPMSGFENCE variable, enter HELP HPMSGFENCE and see:

```
               HPMSGFENCE  A variable used by the CI that controls the output for all
               CI errors, warnings and skipped commands.  Skipped commands refer
               to commands that are not executed by the CI because they follow a
               conditional expression that evaluated FALSE.

               HPMSGFENCE is divided into 2 fields, 3 bits each in size.
               The low order field (bits 29..31) controls the output of  CI error and
warning messages as:
               0  =  display all CI errors and warning
               1  =  show only errors, warning are suppressed
               2  =  suppress all CI errors and warning messages.
         The next field (bits 26..28) controls the output of  skipped commands and the
                "*** EXPRESSION FALSE: ...", "*** EXPRESSION TRUE: ...", and "***
RESUME
               EXECUTION OF COMMANDS" messages:
               0  =  show all skipped commands and the above "***…” messages
               1  =  show only the "***..." messages, suppress commands that are
skipped.  Integer value is 8.
               2  =  suppress the skipped commands and the "***...” messages.  Integer
value is 16.
               Etc…
```

•HPMAXPIN is new to 7.0 and returns the maximum number of processes supported by your system



hp e3000

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.

March 4, 2003 Page 43

•I rarely use HPAUTOCONT. I prefer to be explicit when I am anticipating that the next command may fail. Also, there is slight extra overhead with HPAUTOCONT. Lastly, its original value should be saved and re-instated before the script ends.

•HPCMDTRACE is often useful, despite being overly verbose . There is a simple example that toggles the HPCMDTRACE value in the “Examples” section of this presentation.

•My CI prompt contains HPCWD, e.g.. :setvar hpprompt “!!hpcwd: “

•I use HPCPUMSECS to measure script performance as follows:

- save its value at script entry
- save its value near the script end
- calculate the time in the script as: end_value - start_value.

•Express 1 of 6.0 added 5 new variables related to the date and time.

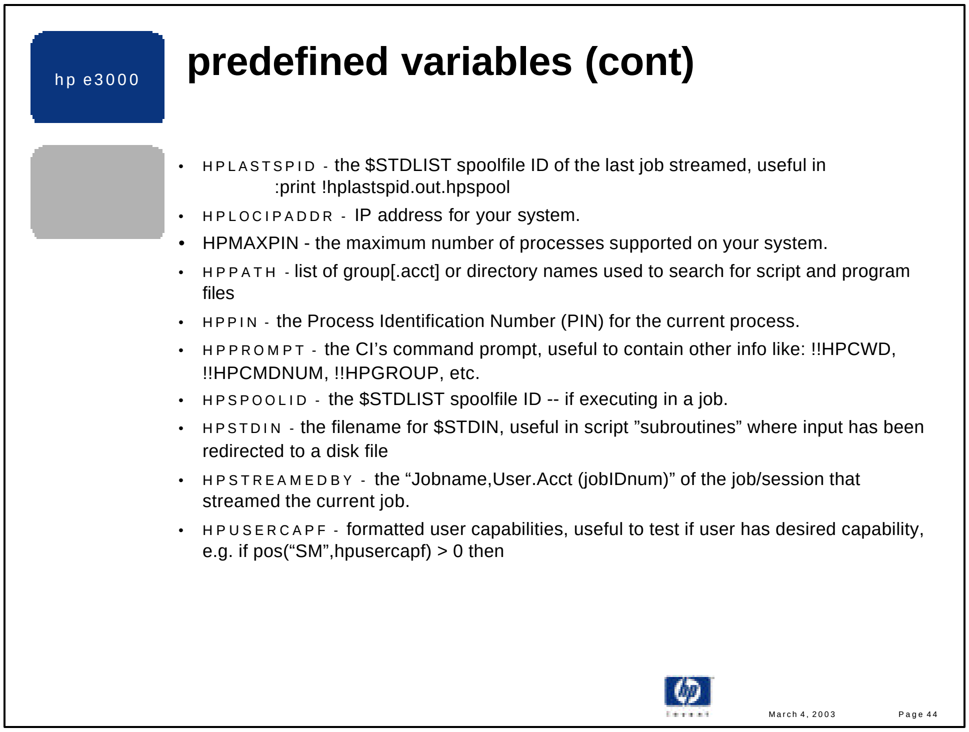
- HPDATETIME - is a string that contains “YYYYMMDDHHMMSSMMM”. The value of this variable is that the date and time are retrieved autonomously, thus you are guaranteed that the time portion of the variable is not early the next day. Note: currently the microseconds field has only tenths of a second resolution due to restrictions on the CLOCK intrinsic call.
- HPDOY - an integer variable containing
- HPHHMMSSMMM - current time in hour, minutes, seconds, micro-seconds.
- HPLEAPYEAR - a boolean variable that is true when the current year is a leap year.
- HPYYYYMMDD - a string variable that contains the year, month and date as an autonomous value.

•HPFILE reduces the need to hard-code the filename of your script, e.g.:

```
if user-selected-help then
   echo ![hpfile] -- Syntax: …
   …
```

•HPLASTJOB can be modified which is useful when referencing the HPLASTPSID variable. E.g..

```
:setvar hplastjob “#J12”
:print !hplastspid.out..hpspool
```



hp e3000

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

March 4, 2003 Page 44

•My HPPATH contains “HPBIN.SYS” so I can run the POSIX programs more easily.

•HPREMIPADDR and HPREMPORT are useful for determining how a user is connecting to your system.

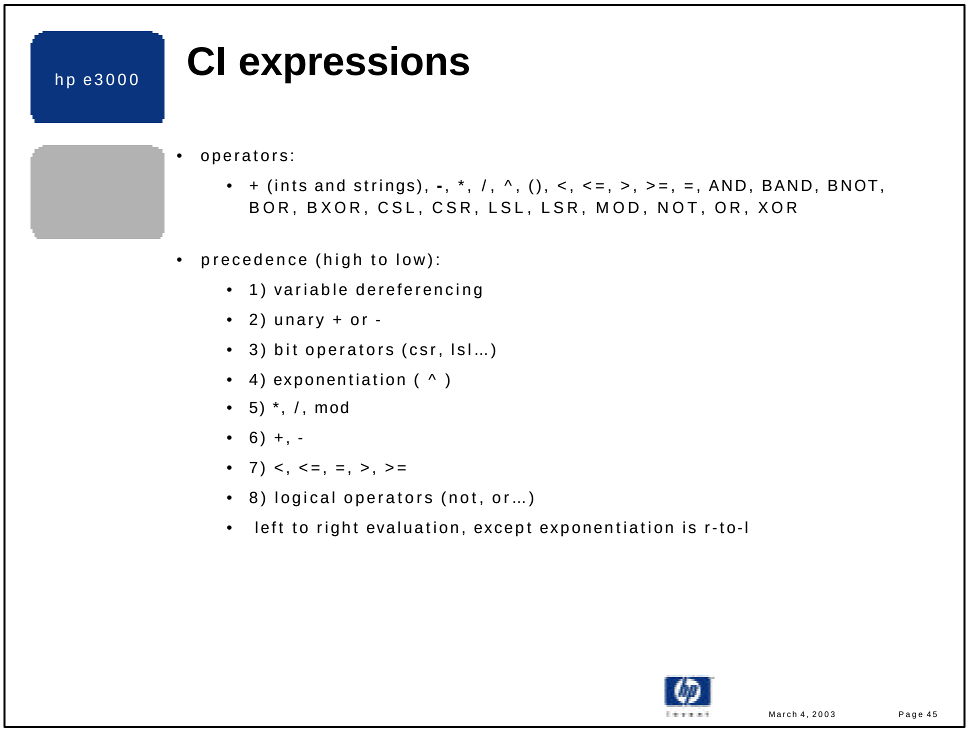
•HPSTREAMEDBY shows the same information as seen at the beginning of the $STDLIST output of a job

•A common way to see if the user has sufficient capabilities is:

```
:if pos(“SM”,hpusercapf) > 0 then        # has SM cap
or
:setvar has_SM  (pos(“SM”,hpusercapf) > 0)
```

•A geek way to see if a user has SM capability is:

```
:if hpusercap < 0 then        # has SM cap, since sign bit (bit 0) is set
or
:if odd(hpusercap lsr 31) then  …        # more “geeky”
```



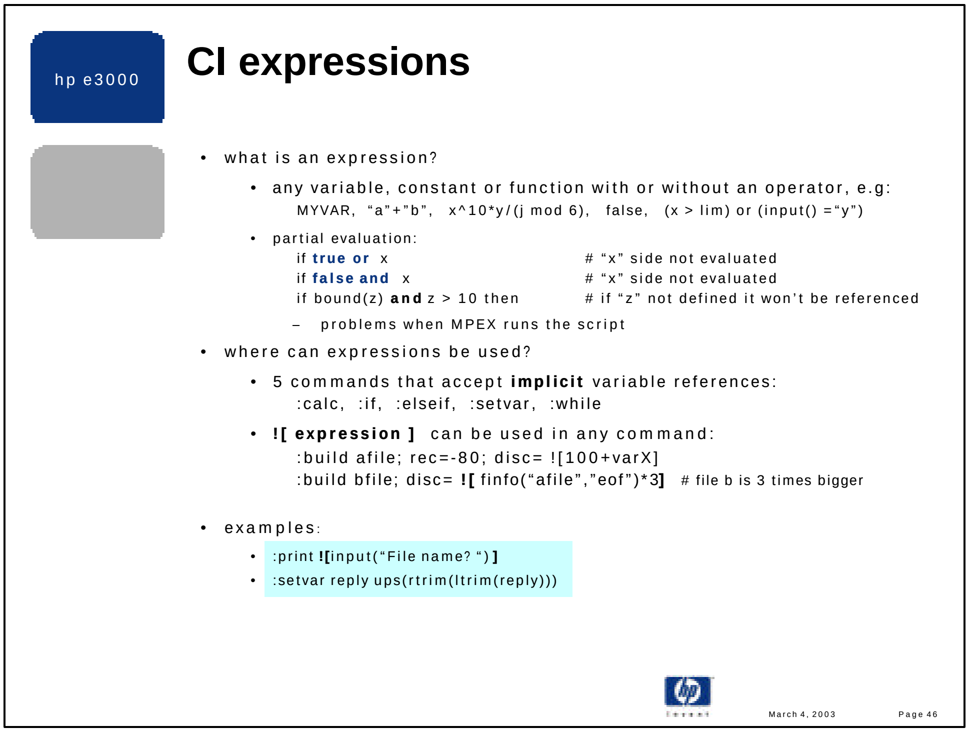
hp e3000

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

March 4, 2003 Page 45

•:HELP operators and :HELP band, etc. provides additional information.



hp e3000

CI expressions

- what is an expression?
  - 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")
  - partial evaluation:

```
if true or  x                        # "x" side not evaluated
if false and  x                      # "x" side not evaluated
if bound(z) and z > 10 then          # if "z" not defined it won't be referenced
```

    – problems when MPEX runs the script
- where can expressions be used?
  - 5 commands that accept **implicit** variable references:
    :calc, :if, :elseif, :setvar, :while
  - **![expression]** can be used in any command:
    :build afile; rec=-80; disc= ![100+varX]
    :build bfile; disc= **![** finfo("afile","eof")*3**]** # file b is 3 times bigger
- examples:
  - :print **![**input("File name? ")**]**
  - :setvar reply ups(rtrim(ltrim(reply)))

March 4, 2003 Page 46

•Expressions are expected naturally in five CI command (CALC, IF, ELSEIF, SETVAR and WHILE), but they must be forced to be evaluated in the remaining CI commands. This forcing is done by enclosing the expression inside square brackets with a leading "!".

•A powerful feature of CI expression evaluation is what is called "partial evaluation". Most programming languages support this concept, which is, performing the minimum level of evaluation needed to determine if a boolean expression is true or false. Not only does this allow the CI to evaluate expressions more efficiently, it is necessary for some compound expressions. For example, consider the following expression:

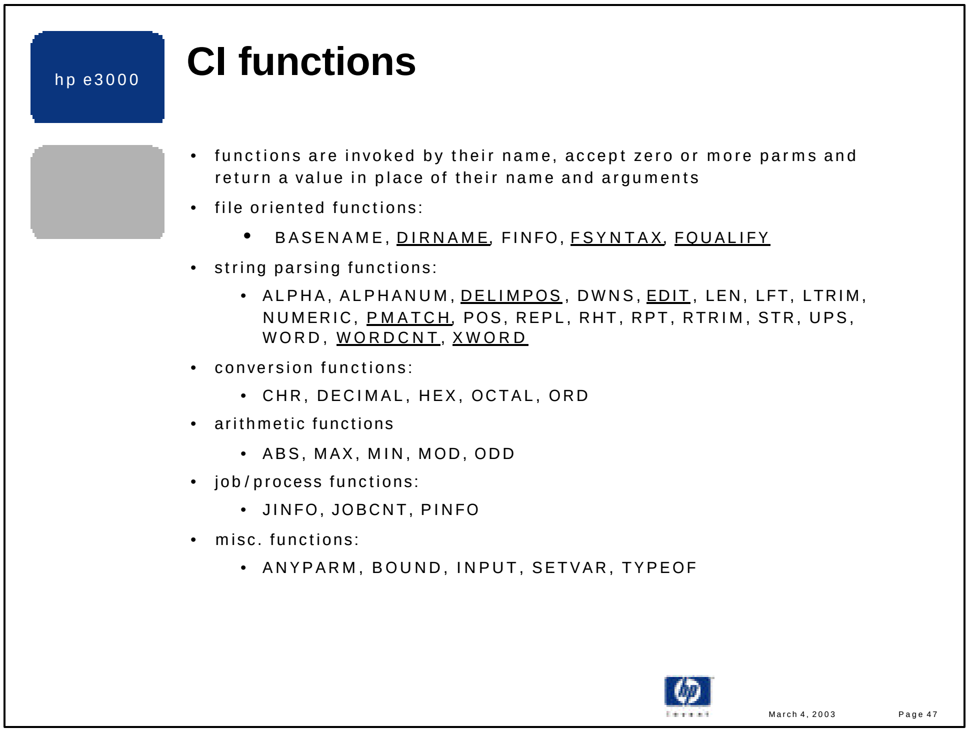
    if FALSE and lft(input("OK to continue?"),1) = "y" then …

If the CI had to evaluate the entire expression then the user would see the prompt and be required to enter input. Clearly this is not desirable since the expression will be FALSE regardless of the user input, and the user should not be bothered with the prompt. To my knowledge, MPEX still does not support partial evaluation with respect to the existence or not of variables. That is, in a statement like:

    if TRUE or lft(varA, 1) = "" then …

MPEX evaluates the expression and enforces that "varA" exists, even though the TRUE clause could halt the left-to-right evaluation. The above expression produces no errors in the CI (regardless of varA's existence), thus CI scripts written to exploit partial evaluation may not work correctly in an MPEX environment.

•As will be seen in the examples at the end of this presentation, some expressions can be long and complex. The motivation for writing expressions this way is purely performance, and sometimes hinders support of the script.



hp e3000

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
- job/process functions:
  - JINFO, JOBCNT, PINFO
- misc. functions:
  - ANYPARM, BOUND, INPUT, SETVAR, TYPEOF

March 4, 2003 Page 47

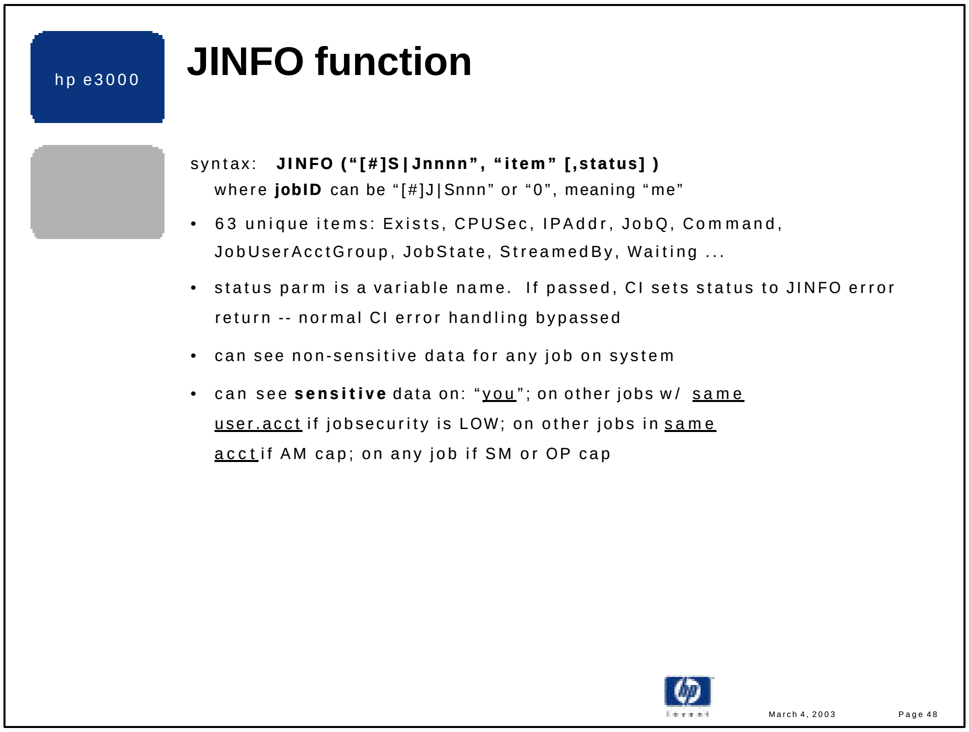
•The CI currently supports 56 functions, over twice as many functions as in the base release of 5.0. However, the CI only supports predefined functions -- user written functions are unavailable.

•Help is available for all functions by entering HELP functionName. A summary of the CI functions can be seen by entering HELP FUNCTIONS. To get function help on a function that has the same name as a CI command, enter HELP function_nameFN, e.g.. HELP setvarFN or HELP inputFN.

•The arguments to a function can be a literal constant, the name of a variable, or another function. When a variable is used as a function argument, its value will be used as the argument value. However, five functions accept a variable name but do not evaluate the variable (i.e. they don’t use its value): JINFO, PINFO, SETVAR, WORD and XWORD.

•Functions can be nested, that is, function A can invoke function B to obtain the value for one of function A’s parameters. The only nesting limit is defined by the size of the CI’s internal buffer that holds the command line -- currently 511 bytes. There is an exception to nesting -- the ANYPARM function is special. Since anyparm() ignores all delimiters, including all but the last right parentheses, it cannot be nested inside other functions, nor can other functions be nested within anyparm’s argument.

•DIRNAME(“f.g.a”)		“/A/G”	
•FSYNTAX(“f.g.a”)		“MPE”	
FSYNTAX(“./a[c-g]”)		“POSIX;WILD”	
•FQUALIFY(“f”)		“F.GRP.ACCT”	or “/CWD/F”
FQUALIFY(“./f”)		“/CWD/f”	
•DELIMPOS(“a,b;c d”)	2	useful when delimiter is a set of two or more characters	
•EDIT(“ab;cd,ef”,”dw”)		“;cd ef”	full REDO programmatic editing
•PMATCH(“ab”,”abc”)		FALSE	easy way to add pattern matching
PMATCH(“ab@”,”abc”)	TRUE		
•WORDCNT(“a b,c=,d”)	5	test if a variable contains the expected number tokens	
(value of ‘c’ is null, but counts as a token -- consistent with word and xword)			
•XWORD(“Hi there, Fred”)	“there, Fred”		



hp e3000

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

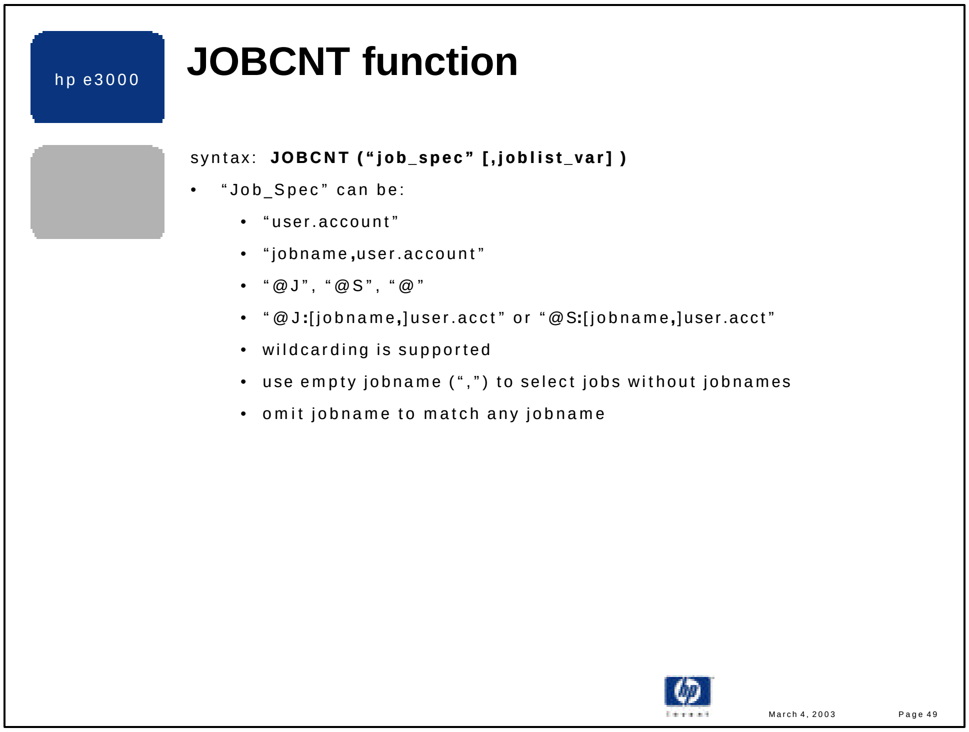
- :help JINFO provides all of the items, security rules and some examples.
- if **JINFO** (***HPLASTJOB***, "EXISTS") then …
  # you know the job exists, at least right now!
- if **JINFO** ("S543", "IPADDR") <> "" then
  # Session 543 is connected via the network
- if **JINFO** (***target_job***, "FMTPRIORITY") = "DQ" then
  # 'target_job' is currently in the DQ dispatcher queue
- setvar ***state*** **JINFO** (***HPLASTJOB***, "STATE", ***status***)

```
while status = 0 and state = "WAIT" do …
   setvar state JINFO (HPLASTJOB, "STATE", status)
endwhile
```

- if JOBCNT("@J", ***list***) > 0 then

```
while  JINFO (word(list), "EXISTS") do
     setvar list xword(list)
     ….
```

- while **JINFO**(*hplastjob*, "EXECUTING") do ...



hp e3000

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

March 4, 2003 Page 49

The JOBCNT function returns the number of job/sessions that match the “job_spec”, regardless of the state of the matching job/sessions. In other words, JOBCNT does not filter based on whether the job is waiting, scheduled, executing, etc. The function return is valid only for the moment it is returned, as a system’s job/session count can continually fluctuate.

The “job_spec” parameter allows just jobs or just session to be selected for a given “user.acct” specification. For example, to find only the jobs logged on as MANAGER.SYS use:

```
JOBCNT (“ @J:MANAGER.SYS”)
```

It is possible to retrieve the job/session IDs for the matching jobs by passing the “joblist_var” parameter. This unquoted argument names an existing or new CI string variable. It will be set to a list of matching job/session IDs of the form: J|Snnn, followed by a space, followed by the next ID, etc. For example:

“S123 S445 J9 S567 J10”

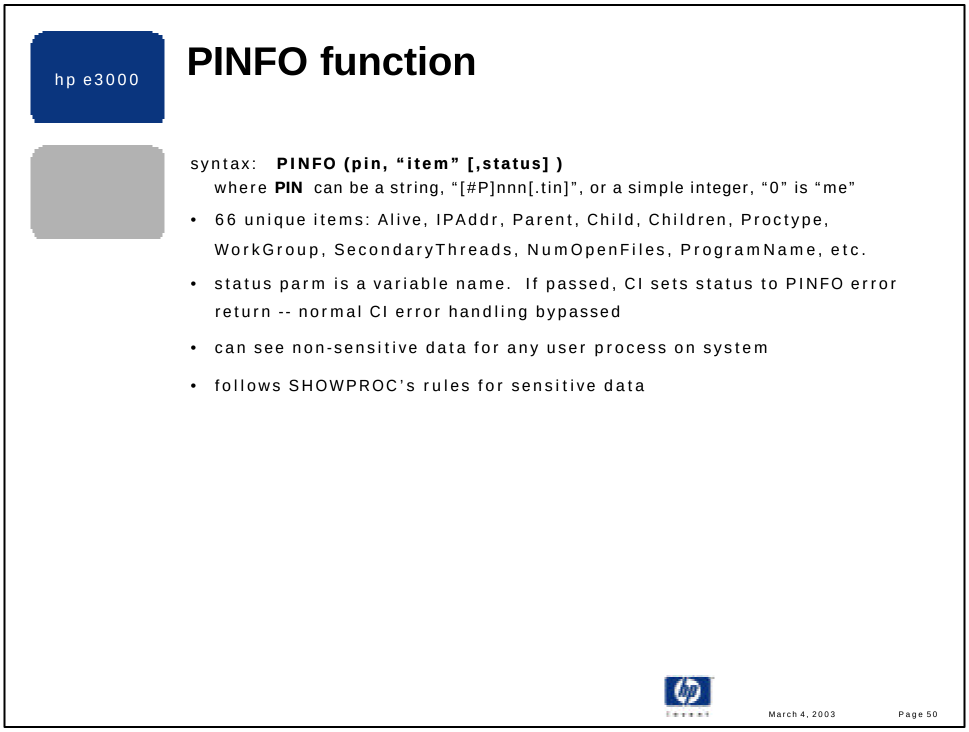
Since CI string variables currently cannot exceed 1024 characters, it is possible that the “joblist_var” passed to JOBCNT cannot contain all of the matching job IDs. This situation is only detected by comparing the number of tokens in the “joblist_var” against the function return. For example:

```
setvar cnt JOBCNT(“@”,jlist)
if cnt <> wordcnt( jlist) then ...   # not all matching jobs in variable
```

Assuming three digit job numbers, approximately 204 matches will fit in the “joblist_var” variable. Possible solutions to this restriction are:

- use separate JOBCNT calls for jobs and sessions
- use separate JOBCNT calls for various target accounts

There are no restrictions on the use of JOBCNT. Any user, regardless of their capabilities, can specify any “job_spec” and retrieve the matching job/session IDs.



hp e3000

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

March 4, 2003 Page 50

• documented in 7.0 Express 1 Communicator or on Jazz at:
http://jazz.external.hp.com/papers/Communicator/7.0/exp1/ci_enhancements.html

•:help PINFO provides all of the items, security rules and some examples.

• if **PINFO** (***HPPIN***, "Info") = "PRINT" then …
# info="PRINT" was specified for your process...

• if **PINFO** (547, "IPADDR") <> "" then
# This process is connected via the network

• if **PINFO** (***target_pin***, "SchedQ") = "DS" then
# 'target_pin' is currently in the DS dispatcher queue

•walk down process tree:

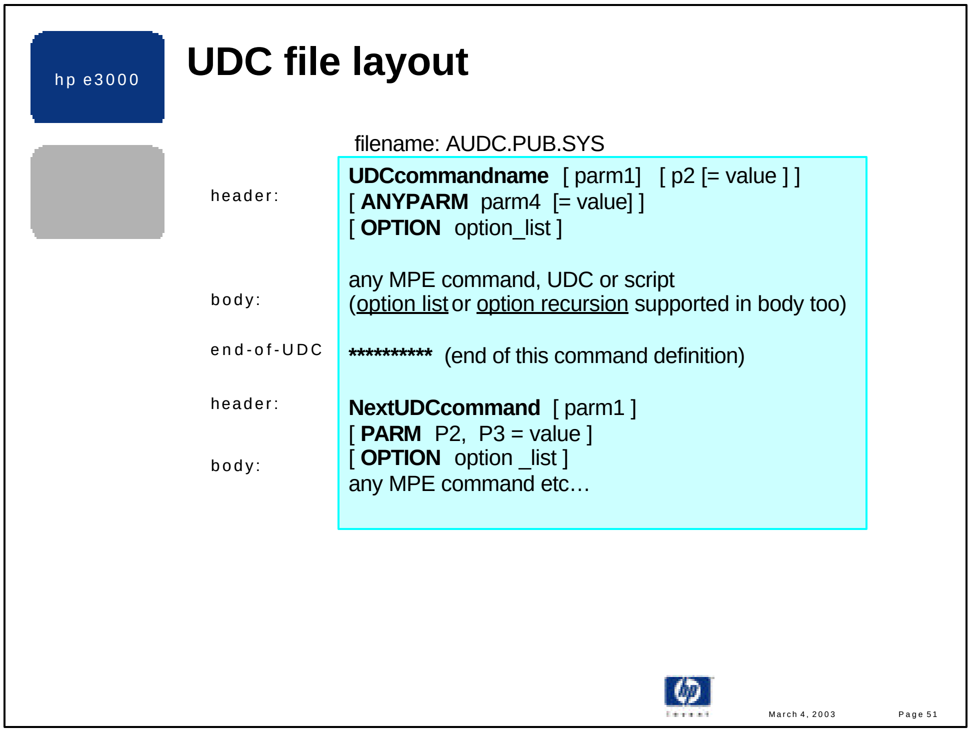
```
setvar p  PINFO ( 0, "jsmainPin")
while p <> 0 do
    setvar p PINFO ( p, "child")
endwhile
```

•walk up process tree:

```
setvar p 0
while PINFO ( p, "proctype") <> "JSMAIN" do
    setvar p PINFO ( p, "parent")
endwhile
```

•find state of each descendant process:

```
setvar kids PINFO ( 0, "children")
setvar kids word(kids,"/",2)    # get rid of count field
setvar k 0
while setvar(p,word(kids,,setvar(k,k+1))) <> "" and PINFO (p, 'alive') do
   echo Pin: !p, state=![PINFO (p, "procState")]
endwhile
```



hp e3000

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…

March 4, 2003 Page 51

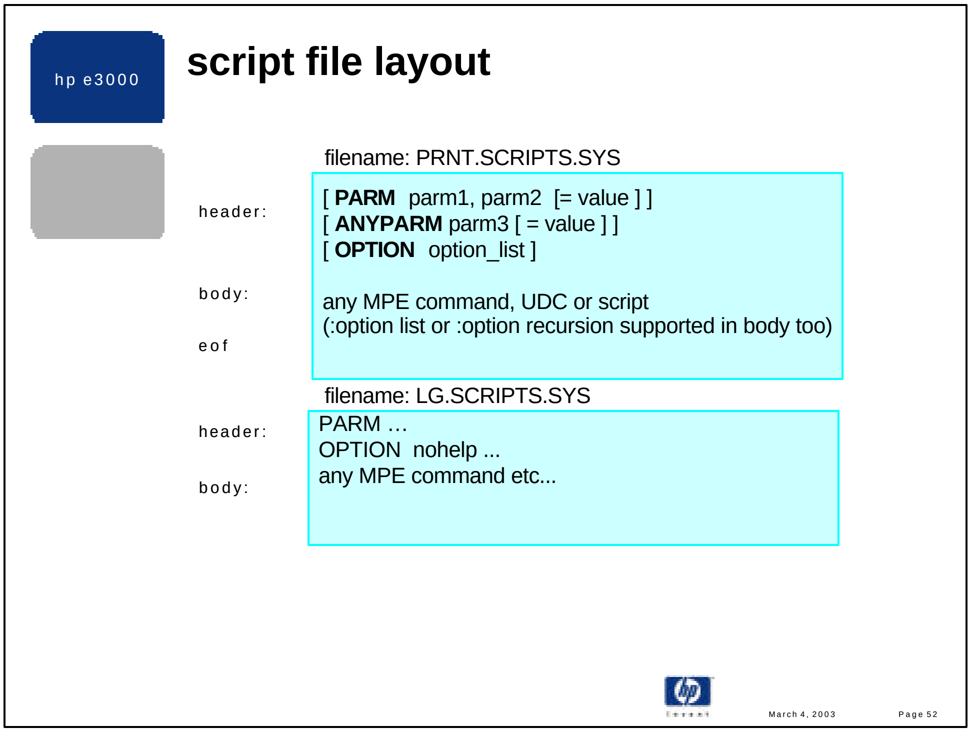
•A UDC file contains one or more individual UDCs, separated by an asterisk in column one (characters right of the asterisk are ignored).

•The **header** consists of the UDC name (required), zero or more parameters, and zero or more UDC options.

- •The parameter line may immediately follow the UDC name, or can begin on the following line introduced with the reserved word PARM.
- •If ANYPARM is specified it must be the last parameter defined.
- •The OPTION line conventionally follows all parameters, though this is not required. Two options (RECURSION and LIST) may appear in the body as well as the option line.
- •The header ends at the first non-PARM, non-ANYPARM, non-OPTION command line.

•The **body** consists of zero or more commands, where the command can be a comment (#), a UDC, a built-in CI command, a command file name or a program file name. The body ends when an asterisk is found in column one. However, a UDC can exit prior to this end point in several ways:

- •an error can cause the UDC to terminate
- •the :RETURN command exits the UDC
- •the :ESCAPE command exits the UDC
- •the :EOJ command in a UDC executing in a job
- •the :BYE command in a UDC executing in a session



hp e3000

script file layout

header:

body:

eof

filename: PRNT.SCRIPTS.SYS

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

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

header:

body:

filename: LG.SCRIPTS.SYS

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

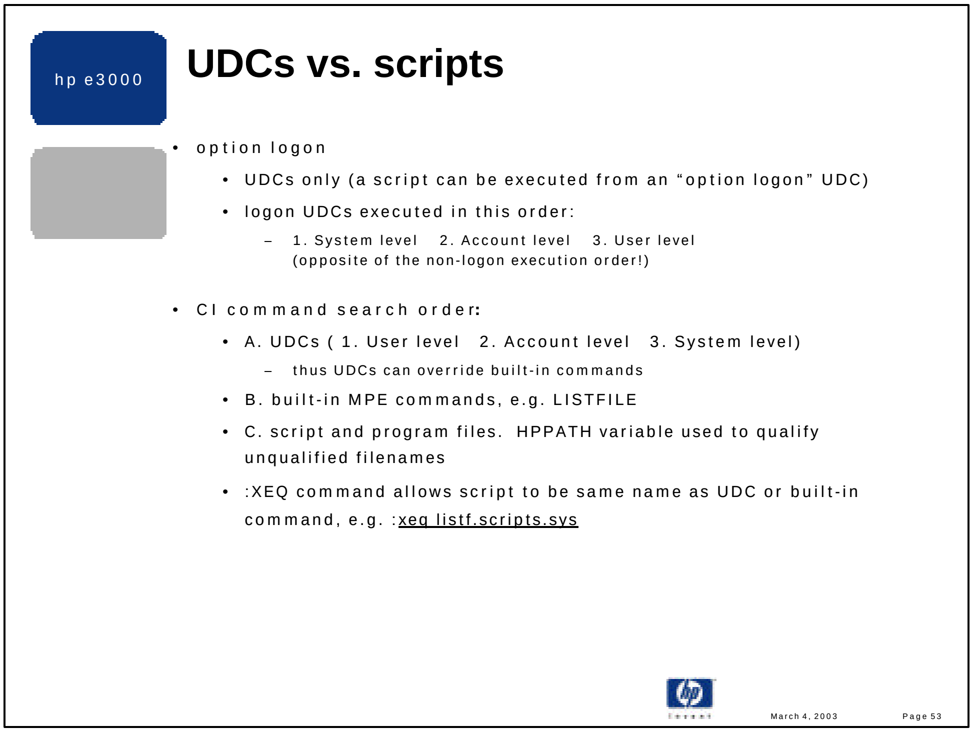
March 4, 2003 Page 52

•A script has the same parts (header, body) as a UDC with a few differences:

- •There is no script name in the header -- the script name is the filename, thus if there are any parameters a PARM or ANYPARM line is required.
- •An asterisk does not terminate a script. Thus a file equation can be reference the name of a script to execute from within a script. For example:

```
file xy z= /bin/scripts/local-xyz
*xyz  parm1  parm2 …
```

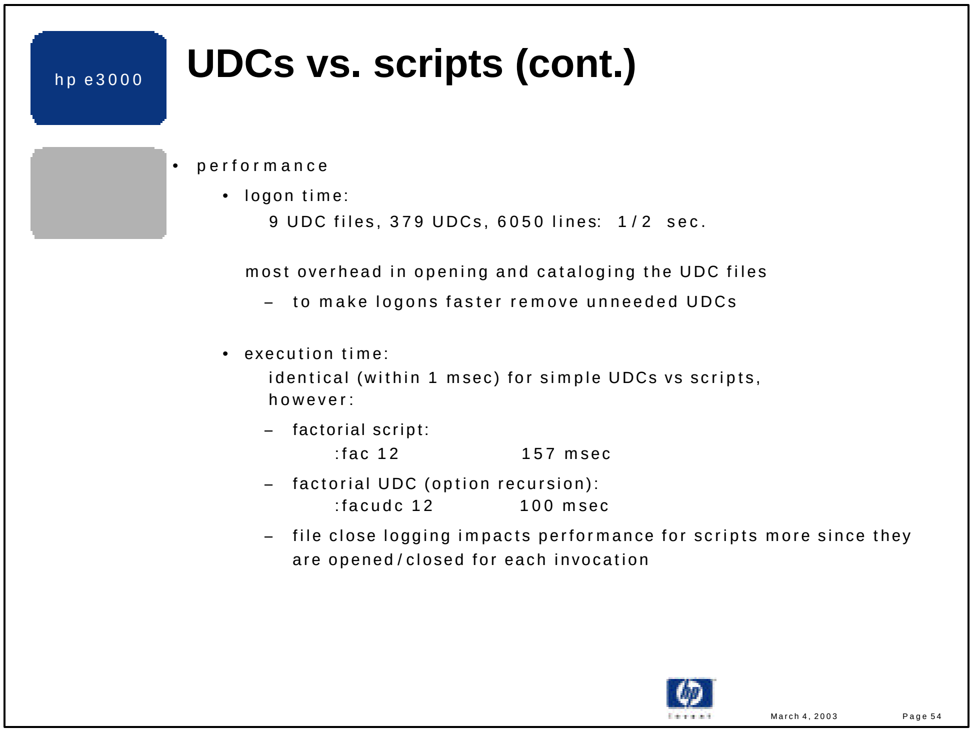
The above "*xyz" works only in a script -- in a UDC, the leading '*' (if it was in column one) would indicate the end of the UDC command.



hp e3000

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



hp e3000

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

March 4, 2003   Page 54

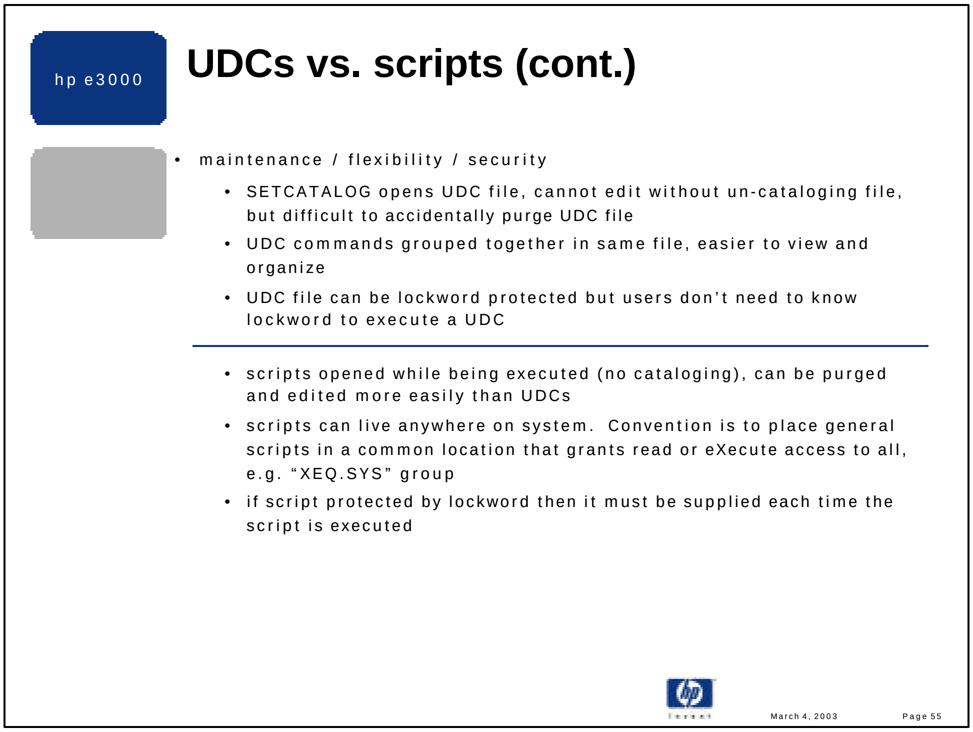
Script FAC:

```
PARM  f
# compute up to 12 factorial.
if typeof(!f) <> 1 or !f <= 0 or !f >= 13 then
   echo Expected an integer between 1 and 12.
   return
endif
if not bound(factor) then
   setvar savecpu hpcpumsecs
   setvar factor 1
   echo !f  factorial used ....
endif
if !f > 1 then
   setvar factor factor * !f
   xeq !hpfile ![!f-1]
else
   # all done, report answer and cpu time
   echo          ![hpcpumsecs-savecpu]  msecs to compute.
   echo Answer is: !factor   ![octal(factor)]   ![hex(factor)]
   deletevar factor
endif
```

UDC FACUDC:

```
 FACUDC  f
  OPTION RECURSION
    if typeof(!f) <> 1 or !f <0 or !f >= 13 then
       echo Expected an integer between 1 and 12.
  return
endif
  if not(bound) factor then
  setvar savecpu hpcpumsecs
    setvar factor 1
       echo !f factorial used ...
endif
 if !f > 1 then
       setvar factor factor*!f
    facudc ![!f-1]
else
        # all done, report answer and cpu time
        echo       ![hpcpumsecs-savecpu] msecs to...
             echo Answer is : !factor   ![octal(factor)] ...
      deletevar factor
endif
```

*****



hp e3000

UDCs vs. scripts (cont.)

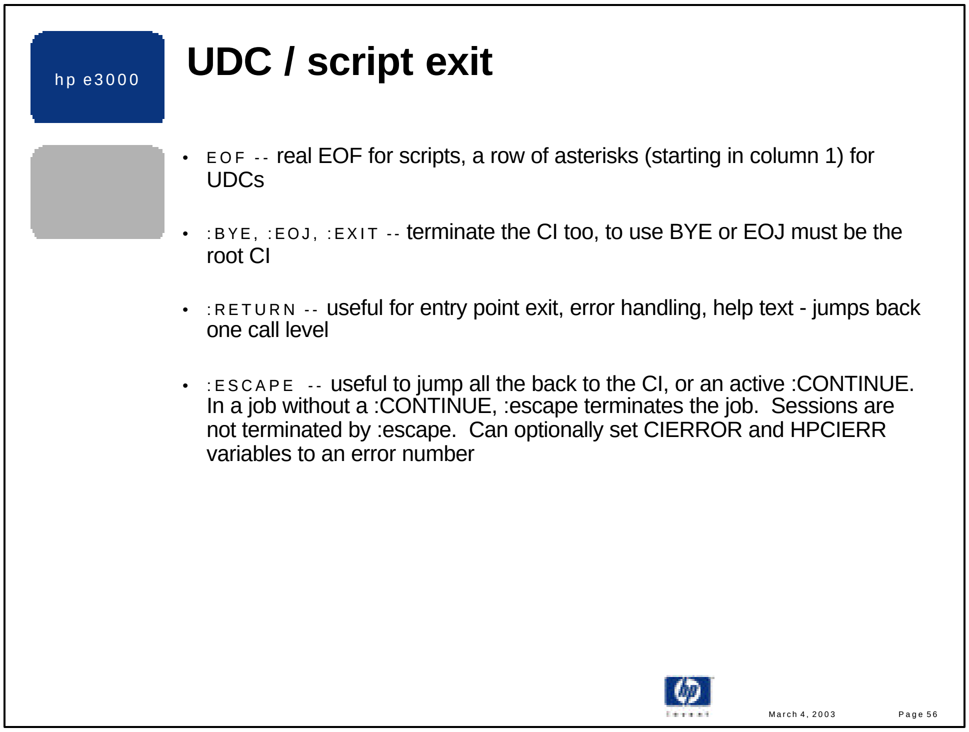
- maintenance / flexibility / security
  - SETCATALOG opens UDC file, cannot edit without un-cataloging file, but difficult to accidentally purge UDC file
  - UDC commands grouped together in same file, easier to view and organize
  - UDC file can be lockword protected but users don't need to know lockword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

March 4, 2003 Page 55

•SETCATALOG user needs to know the lockword, but the the user executing individual UDCs does not ever need to specify a lockword.

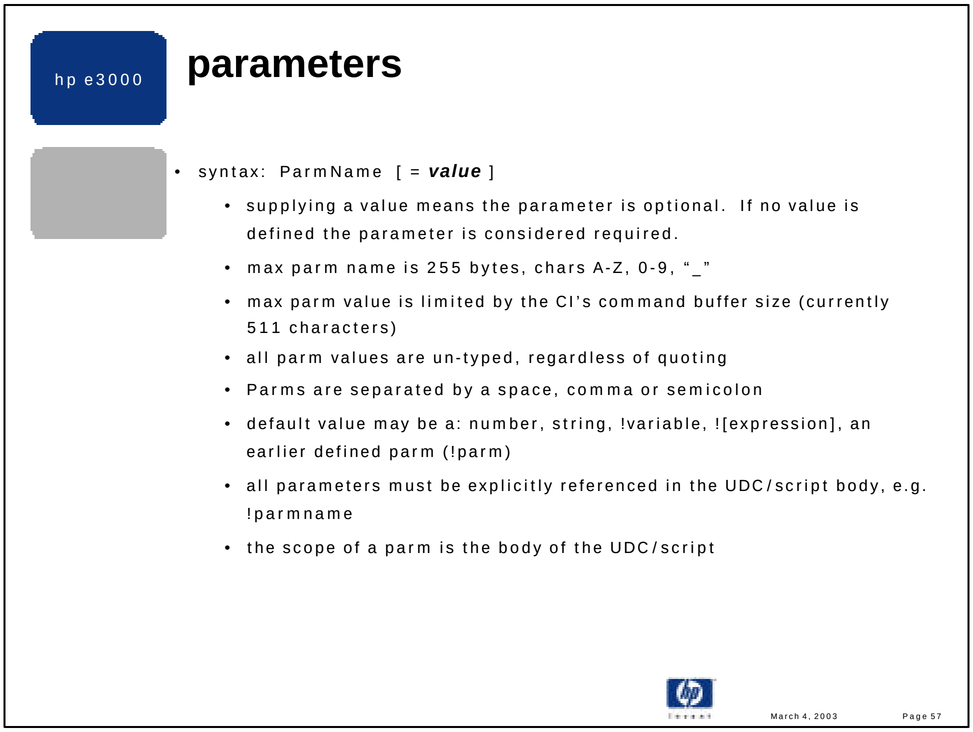
•Note: the POSIX shell's "mv" command allows a new UDC to overwrite an existing UDC file that is being accessed. The result of this is that user that just logon see the new UDC file, while users that were cataloged to the original file see no difference until the re-logon. Once they all re-logon, the old file is purged by the system, since the file open count went to zero.



hp e3000

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



hp e3000

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

March 4, 2003 Page 57

•A parameter and variable can have the same name but this should be avoided to improve support of UDC and scripts

•PARM p1=abc
setvar p1, "xyzzy"
echo P1=!p1 ---> P1=abc
echo P1=![p1] ---> P1=xyzzy

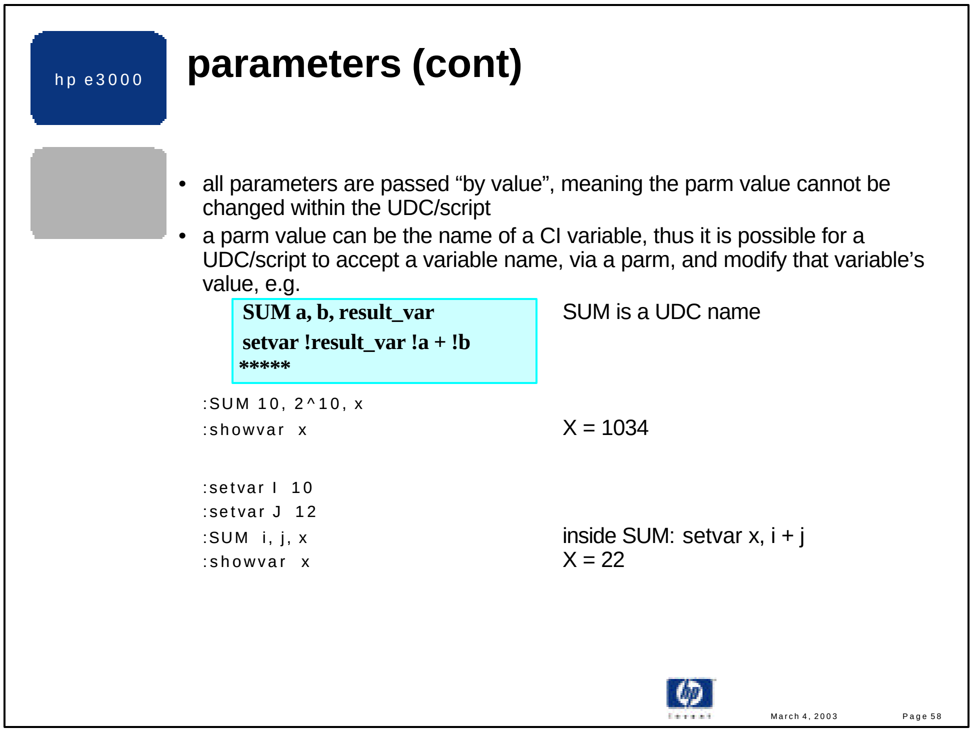
Note: explicit referencing (!x) looks for parameters first, then if no match searches for variables. Implicit referencing (x) does not look for parameters at all, and only searches for a variable name.

•PARM p1, p2=abc, p3="def", p4=1, p5="1", p6=true,p7="false", p8=!p2, p9=![rht(HPJOBNAME,-2)]

•Argument P1 is required. Argument P8 contains the value of P2. Argument P9 defaults to the value of the HPJOBNAME variable -- less the first character.

•Internal to the CI all parameter values are stored as strings, but since parameters must be explicitly referenced (!parmname) their string type is **not** preserved. Thus, to a CI programmer all parameter values are un-typed:

•calc typeof(p2)	= 0 # no meaning since parm p2 was not explicitly referenced (assume no variable named P2)
•calc typeof(!p2)	= 0 # no meaning (assume no variable named ABC)
•calc typeof("!p2")	= 2 # string, regardless of p2's value since value was quoted
•calc typeof(!p4)	= 1 # integer
•calc typeof(!p5)	= 1 # integer, quotes around default value don't matter
•calc typeof(!p6)	= 3 # boolean
•calc typeof("!p6")	= 2 # string since I quoted it!
•calc typeof(!p7)	= 3 # boolean



hp e3000

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                        SUM is a UDC name
setvar !result_var !a + !b
*****

: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
```

March 4, 2003 Page 58

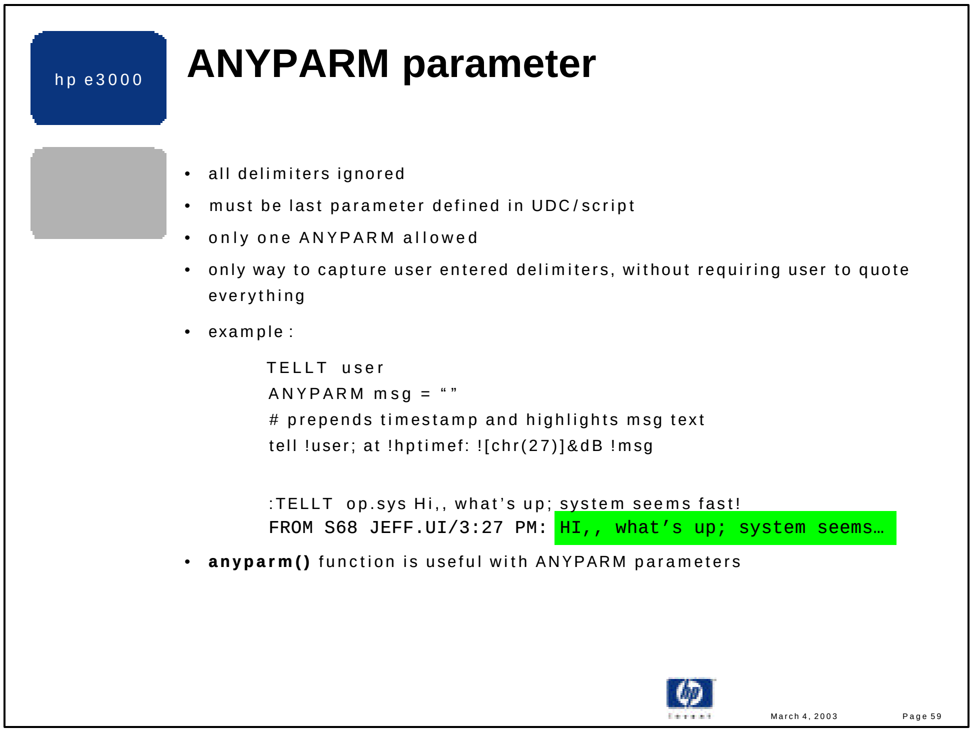
•Note: inside the SUM UDC the parameters A and B cannot be changed. For example, if

:setvar a,a+1

appeared inside SUM, it would try to create a CI variable named A, but would fail since a job/session global variable named A does not exist and thus cannot be referenced. If instead,

:setvar a,!a+1

appeared inside the SUM UDC, this would create a new CI variable named A with a value equal to the value of the parameter A+1. Neither example alters the parameter’s value.



hp e3000

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…
```

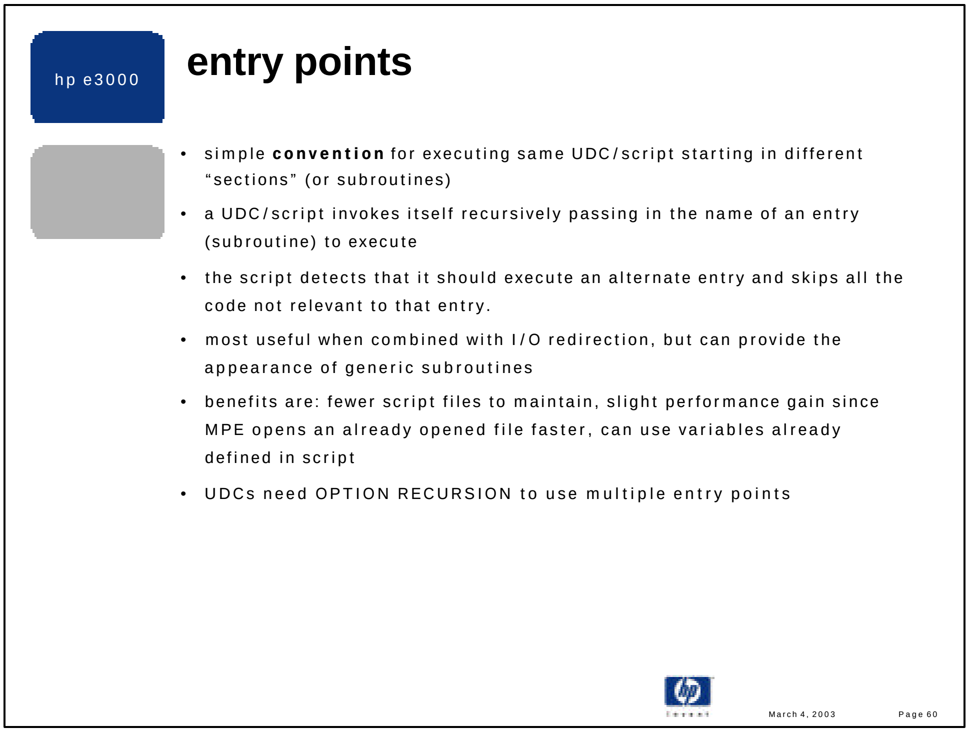
- **anyparm()** function is useful with ANYPARM parameters

March 4, 2003 Page 59

•A few examples using ANYPARM and the anyparm function are shown in other parts of this talk, with respect to capturing an INFO= string.

•The only way to get an ANYPARM parameter value to default to “” (empty string) is as follows:

```
ANYPARM  p = ![“”]          # correct
ANYPARM  p = “”             # wrong - default value is literally the two quote marks
```



hp e3000

entry points

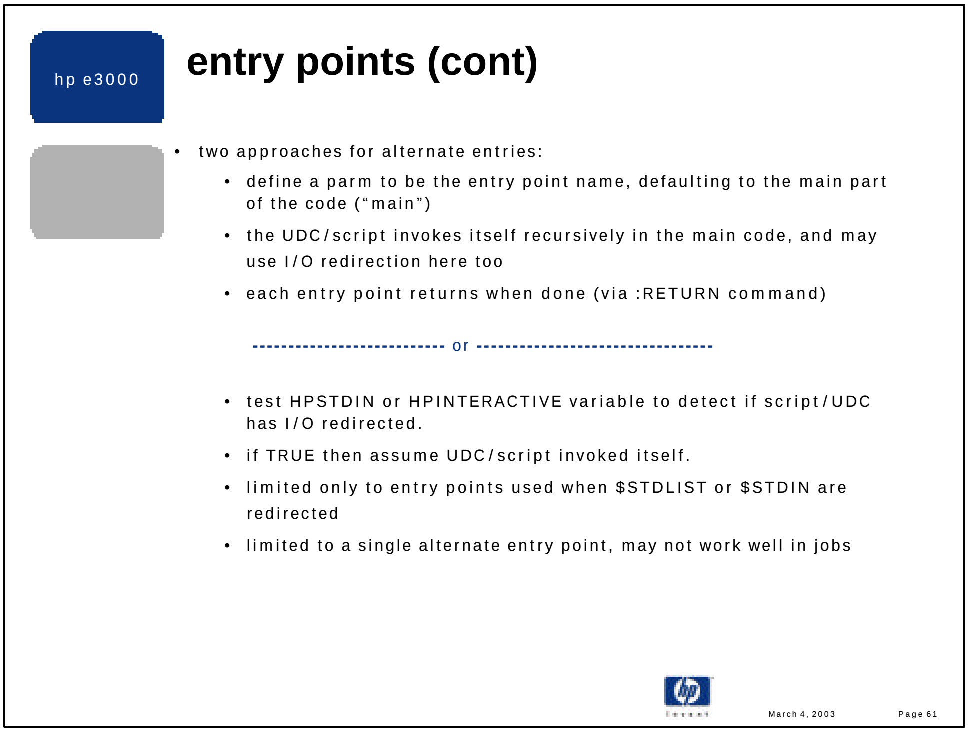
- simple **convention** for executing same UDC/script starting in different “sections” (or subroutines)
- a UDC/script invokes itself recursively passing in the name of an entry (subroutine) to execute
- the script detects that it should execute an alternate entry and skips all the code not relevant 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

March 4, 2003 Page 60

•There is no limit to the number of entry points, and there is no required order: all entry points can appear in the beginning of the script, the end or both.

•An entry point is just a programming convention implemented by adding another parameter to the PARM line, and passing the desired entry point name to the script/UDC when it is invoked. This extra parameter is never explicitly provided by the user.

•By definition, all scripts and UDCs using alternate entries are recursive.



hp e3000

entry points (cont)

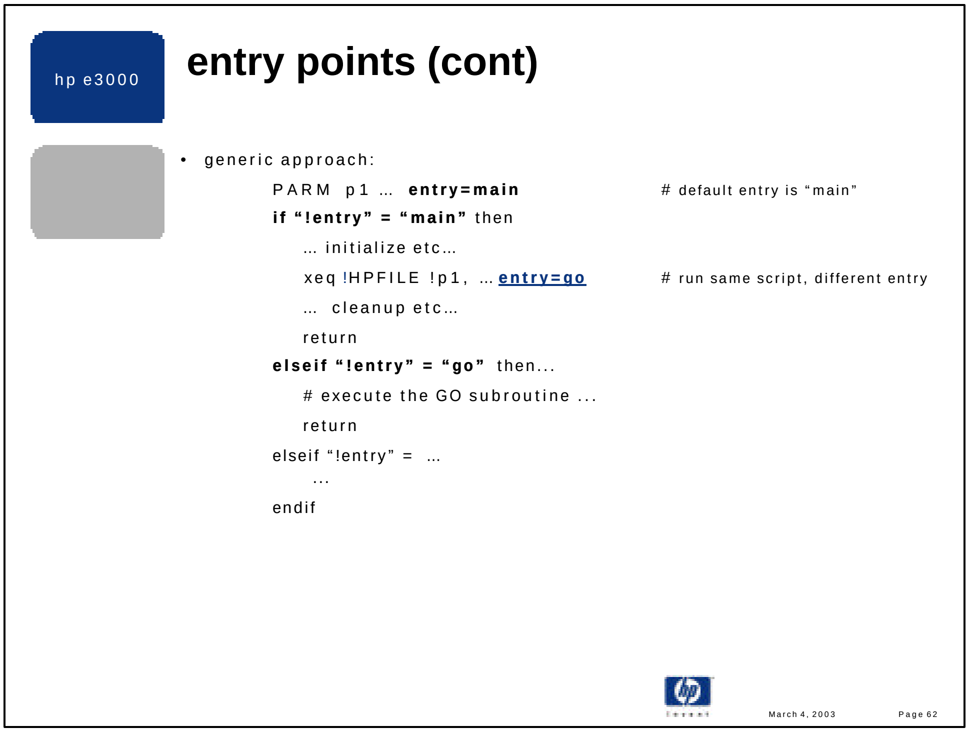
- two approaches for alternate entries:
  - define a parm to be the entry point name, defaulting to the main part of the code (“main”)
  - the UDC/script invokes itself recursively in the main code, and may use I/O redirection here too
  - each entry point returns when done (via :RETURN command)

---------------------------- 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

March 4, 2003 Page 61

•My preference is the first approach since it is the most flexible method. In fact, I usually structure my scripts to be able to work with multiple alternate entry points, even if I need only a single alternate entry at the time the script is being first written.



hp e3000

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
```

March 4, 2003 Page 62

•This shows a script structured so that it can accept multiple alternate entry points.

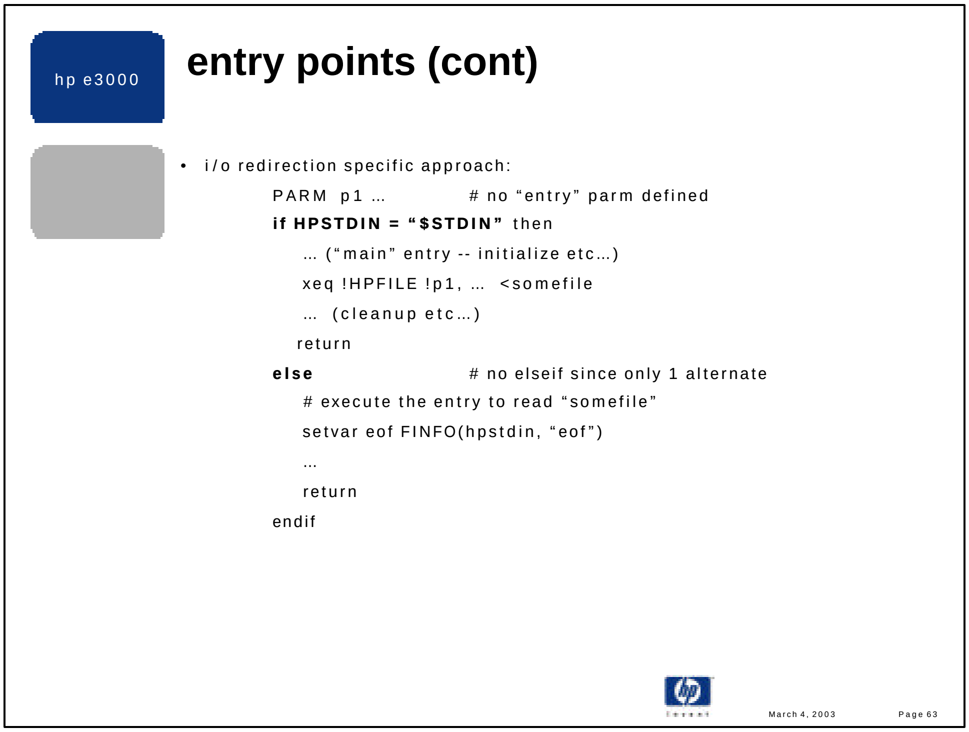
•There should be little or no code before the if “!entry” = “main” line.

•Notice that RETURN is used to exit the main and all alternate entries. This is not required since the CI will drop out of the entry block of code, reach the eof and naturally return back to where the script called itself. However, performance is improved using RETURN in the manner shown above.

•ANYPARM scripts with entries use a slightly different structure and require more parsing:

```
ANYPARM  p1 = ![‘’]
if “!p1” = “” or pos(“entry=“,”p1”) = 0 then
    # main entry for script …..
    xeq !hpfile some parm value  entry =do_this
    return
else
    # parse out entry name and execute entry subroutine, entry name is last word
    setvar _entry word(“!p1”,,-1)
    # remove “entry=name” from parm line
    setvar _parm lft(“!p1”,pos(‘entry=‘,”!p1”)-1)
    # case on entry name
    if _entry = “do_this” then …
        return
    elseif _entry = ...
    endif
endif
```

•UDCs with entries need to specify OPTION RECURSION so that the UDC can invoke itself with the alternate entry name. OPTION RECURSION can be in the UDC header or a separate CI command.



hp e3000

entry points (cont)

- i/o redirection specific approach:

```
PARM  p1 …            # no “entry” parm defined
if HPSTDIN = “$STDIN” then
    … (“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
```

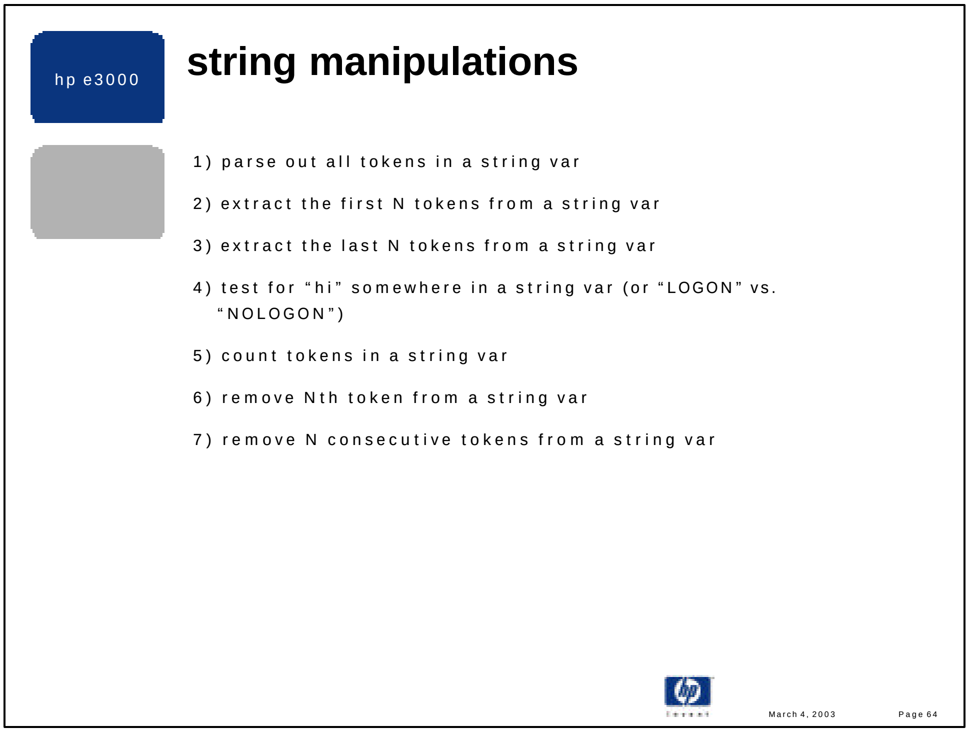
March 4, 2003 Page 63

•Note: the HPSTDIN = “$STDLIST” test above could be replaced with:
if HPINTERACTIVE then…

•This approach to alternate entry points works fine for its limited uses. It does not handle multiple alternate entries and requires I/O redirection for the single alternate entry.

•In a job you must use the HPSTDLIST test since HPINTERACTIVE is always FALSE.

•If the script itself is run with I/O redirected then both tests (HPSTDIN and HPINTERACTIVE will be inaccurate, and the generic approach must be used.



hp e3000

string manipulations

1) parse out all tokens in a string var

2) extract the first N tokens from a string var

3) extract the last N tokens from a string var

4) test for “hi” somewhere in a string var (or “LOGON” vs. “NOLOGON”)

5) count tokens in a string var

6) remove Nth token from a string var

7) remove N consecutive tokens from a string var

March 4, 2003 Page 64

setvar x "ab c;de,,fg;hij=k lmn,op=qr”

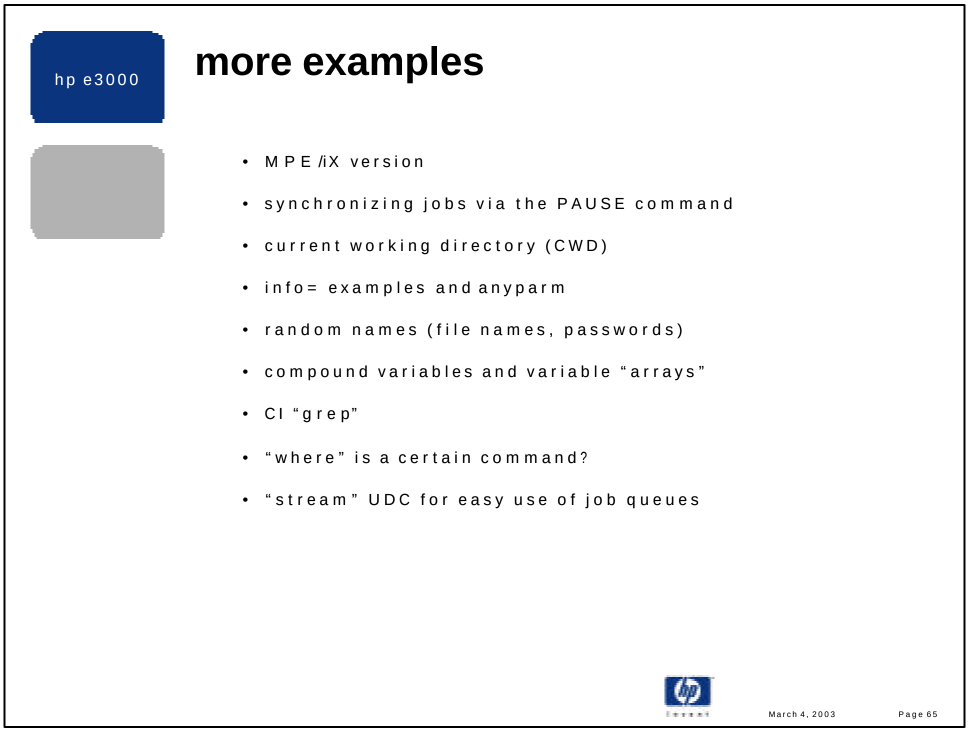
```
1) setvar j 0                                  -or-
                          setvar j 0
   while j <= len(x) do                                                               while setvar(j,j+1)
<= wordcnt(x) do
      setvar tok word(x, , , j, j+1)                        setvar tok word(x, , j)
   endwhile                                                                           endwhile
        2136 msec for 500 iterations                         2298 msec
   -or-

   # below fails on a null token, but otherwise is simple:
   setvar j 0
   while setvar(tok, word(x, ,setvar(j,j+1))) <> “” do
   endwhile
           1686 msec
```

```
2) setvar toks  lft(x, delimpos(x, , N)-1)     -or-        setvar j 0  and  setvar toks ‘’
   # note the var toks includes the delimiters             while setvar(j,j+1) <= N do
     between the individual tokens                                      setvar toks  toks + word(x,,j) + “ “

                                                           endwhile

                                                           # note toks may not contain the original delimiters
```

```
3) setvar toks rht(x, -delimpos(x, , -N)-1)    -or-        setvar j 0  and  setvar toks ‘’
   # same notes as for 2)                                                            while setvar(j,j+1)
<= N do

                                                             setvar toks word(x,,-j) + “ “ + toks

                                                           endwhile
```

4) pos(“hi”,x) is potentially wrong. What if you want only “hi” and not “high”?
if word(x, , , ,pos(“hi”,x)) = “hi” then …

```
5) setvar cnt wordcnt(x)
```

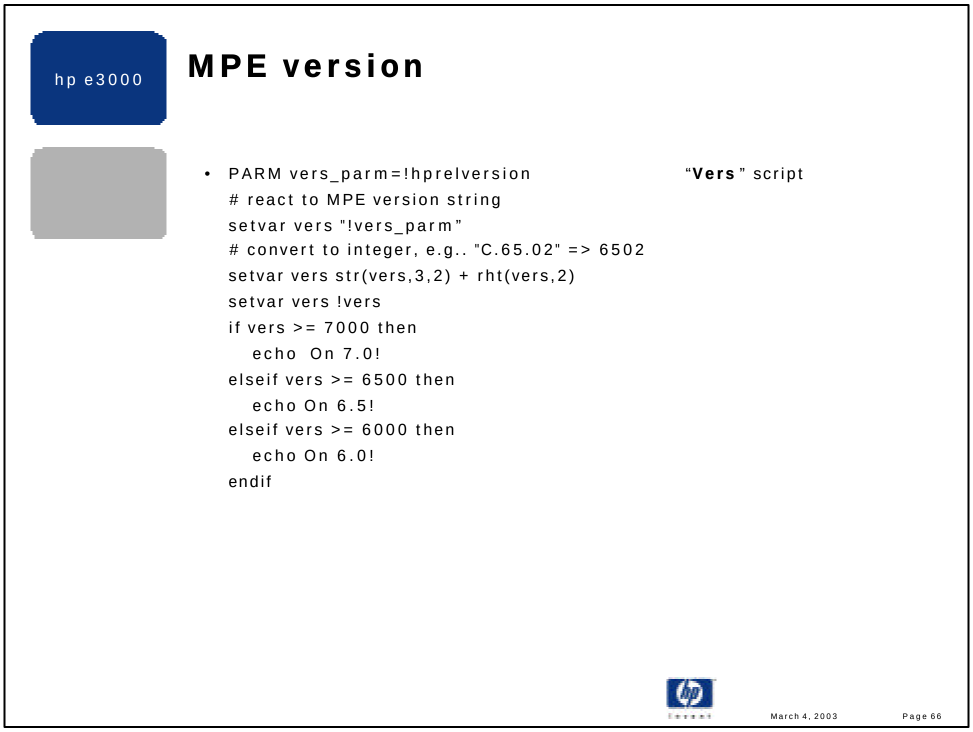


hp e3000

more examples

- MPE/iX version
- synchronizing jobs via the PAUSE command
- current working directory (CWD)
- info= examples and anyparm
- random names (file names, passwords)
- compound variables and variable “arrays”
- CI “grep”
- “where” is a certain command?
- “stream” UDC for easy use of job queues

March 4, 2003 Page 65



hp e3000

MPE version

“**Vers**” script

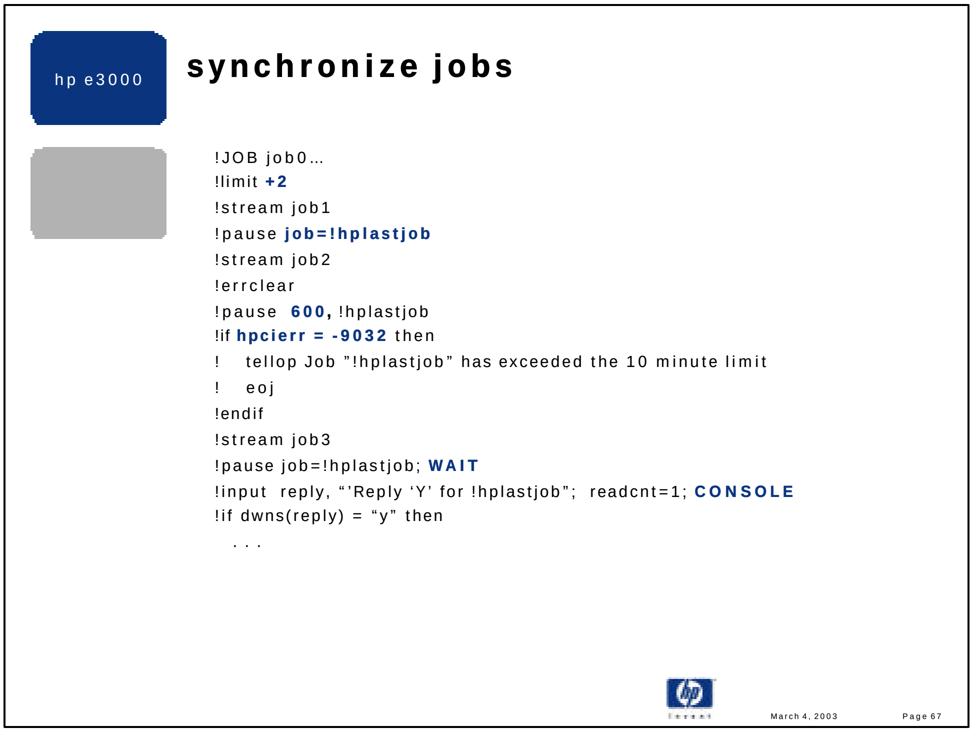
```
• PARM vers_parm=!hprelversion
  # react to MPE version string
  setvar vers "!vers_parm”
  # convert to integer, e.g.. “C.65.02” => 6502
  setvar vers str(vers,3,2) + rht(vers,2)
  setvar vers !vers
  if vers >= 7000 then
     echo  On 7.0!
  elseif vers >= 6500 then
     echo On 6.5!
  elseif vers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

March 4, 2003 Page 66

•The CI does not support a direct mechanism to let the programmer know if a certain command, function, variable or other new feature is present on the system at hand. The *bound()* function lets you test for the existence of any variable prior to referencing it. It is trickier to test for the existence of a function prior to invoking it. Thus, it may be necessary to test the MPE OS version prior to using a new feature. However, the CI version variables only reflect what the version strings displayed by the :SHOWME command. Thus, as you are aware, the version granularity is sometimes lacking.

```
•:showvar @vers@
HPOSVERSION = C.70.00
HPRELVERSION = C.70.01
HPVERSION = X.70.11
```

```
• :vers
 On 7.0!

  :vers C.65.01
  On 6.5!
```



hp e3000

synchronize jobs

```
!JOB job0…
!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
  . . .
```

March 4, 2003 Page 67

•The job limit is increased by 2.

•The 1st pause sleeps until the job just streamed (job1) completes.

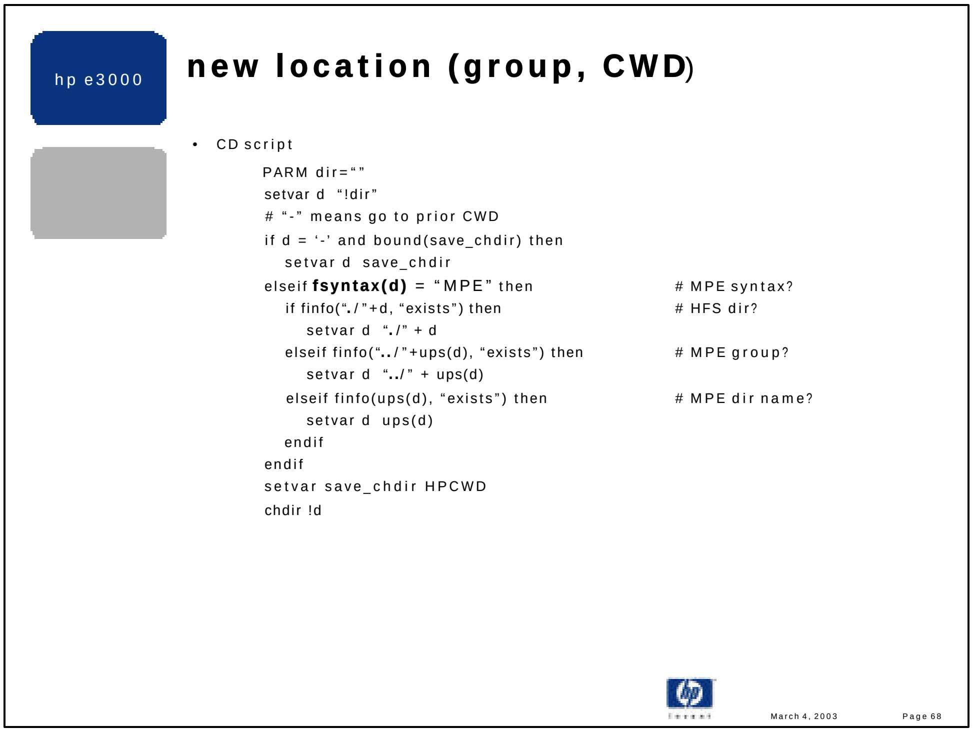
•The 2nd pause sleeps until job just streamed (job2) completes or 10 minutes, whichever happens first. CIERR 9032 is reported if the pause expires and the job is still alive.

•The 3rd pause sleeps while job3 is introduced or waiting. As soon as job3 starts executing (or terminates, if it is a short lived job) the pause expires.

•The INPUT command displays a message to the system console and waits for a reply. INPUT will only accept a 1 character response from the operator, in this example. Syntax:

```
INPUT [NAME=]varname
      [[;PROMPT=]prompt ] [[;WAIT=]seconds  ]
      [[;READCNT=]chars ] [[;DEFAULT=]default_str ]
      [;CONSOLE ]
```

•HPCIERR shows positive CI errors and negative CI warnings. CIERROR = abs(hpcierr)



hp e3000

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
```

March 4, 2003 Page 68

•The HPCWD variable contains your current working directory in POSIX syntax. Your current directory is the same as your logon group until you explicitly change it via the CHDIR CI command.

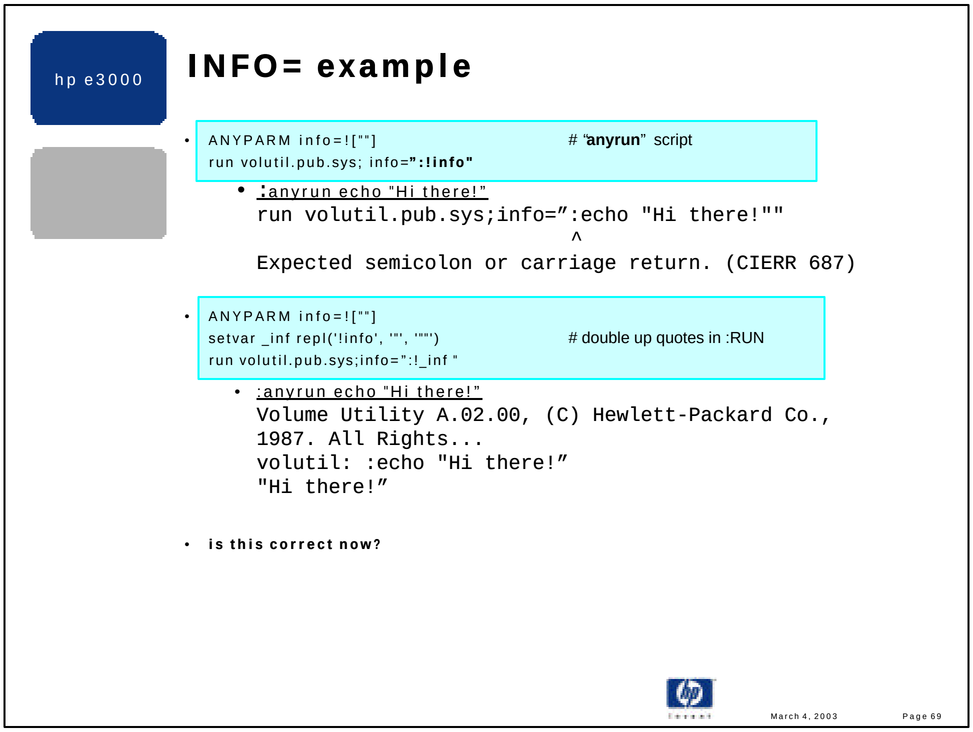
•CD script hierarchy is: 1) dirname as is, 2) ./+dirname, 3) group name (“../”+dirname) 4) uppercase MPE dirname

Note: the CHGROUP command also changes your CWD; whereas, the CHDIR command does not alter your logon group. CHGROUP has security implications since it can give you GU (group user) file access. There are **no** security implications with CHDIR.

•cd - changes your current directory to the previous directory you’ve CD’d to.

• CD examples:

```
(assume CWD = /SYS/PUB)
:cd ../NET                              # CWD=NET.SYS
:cd -                                   # CWD=PUB.SYS
:cd /TELESUP/PRVXL                      # CWD=PRIVXL.TELESUP
:cd                                     # CWD=PUB.SYS
:cd foo                                 # CWD=/SYS/PUB/foo
:cd ..                                  # CWD=PUB.SYS
:cd net                                 # CWD=NET.SYS
```



hp e3000

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)
    ```

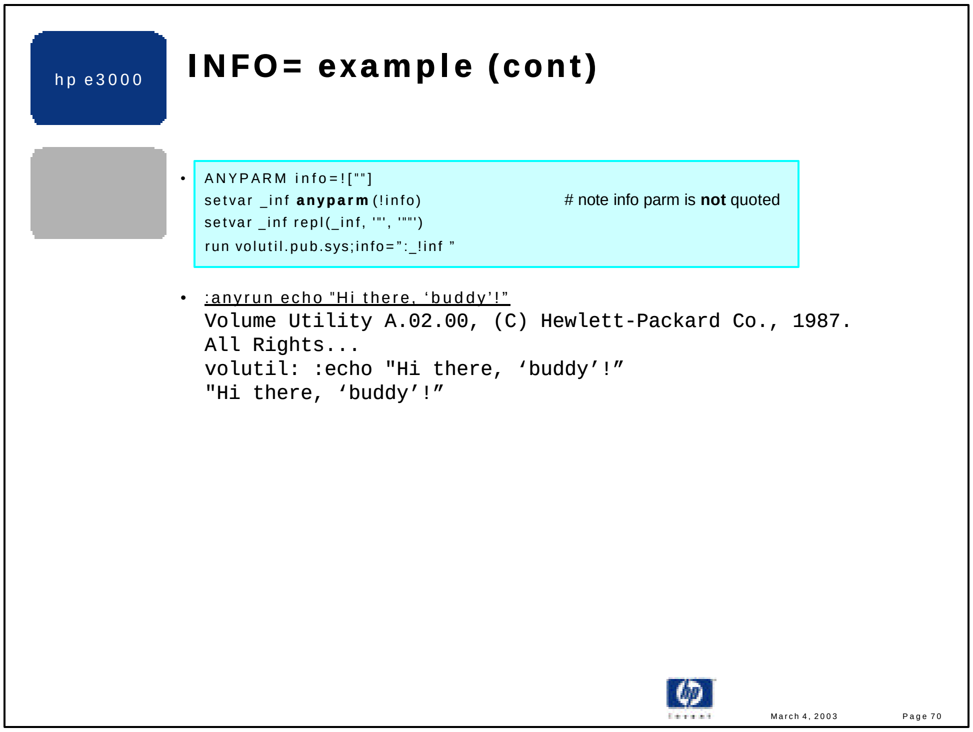
-
  ```
  ANYPARM info=![""]
  setvar _inf repl('!info', '"', '""')        # double up quotes in :RUN
  run volutil.pub.sys;info=":!_inf "
  ```
  - :anyrun echo "Hi there!”
    ```
    Volume Utility A.02.00, (C) Hewlett-Packard Co.,
    1987. All Rights...
    volutil: :echo "Hi there!”
    "Hi there!”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

March 4, 2003 Page 69

•Shows how to set an ANYPARM parameter to null, which is not intuitive! If an ANYPARM parameter is defaulted to “” the quotes are accepted literally as its default value.

•This example does **not** handle both kinds of quotes in the info= string.

•This example does **not** handle single quote mark in the REPL function call.



hp e3000

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 "
  ```

- <u>:anyrun echo "Hi there, ‘buddy’!”</u>

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

March 4, 2003 Page 70

•Do not quote the parameter being passed to the *anyparm()* function.

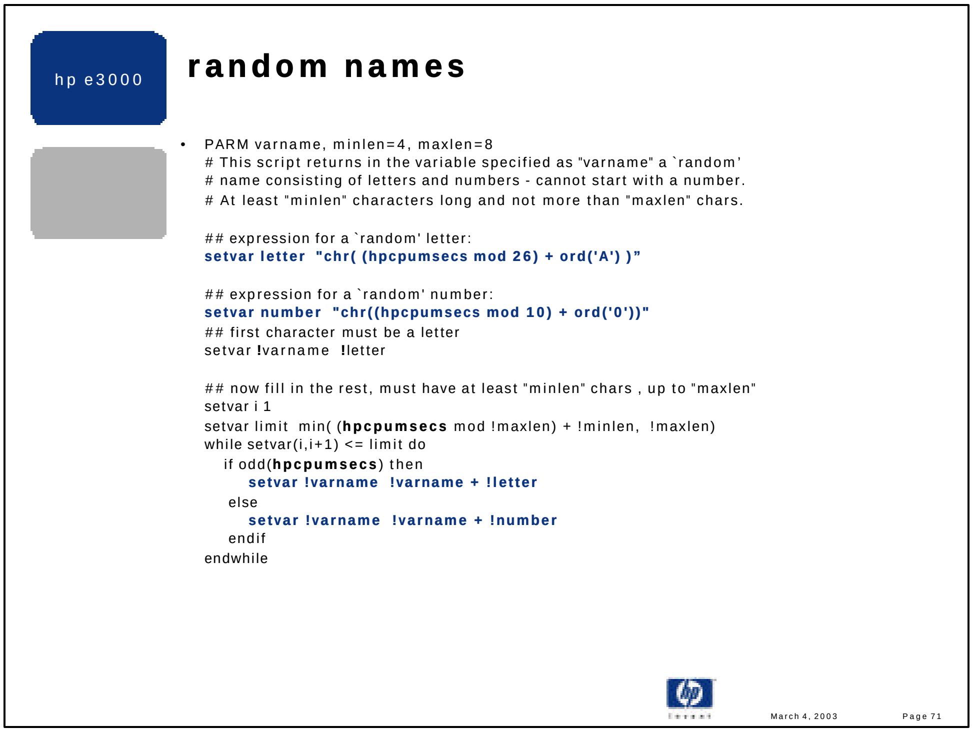
•Note: the *anyparm()* function has some special considerations:

•it cannot be nested inside other functions, e.g.
lft(anyparm(!parm), x) is **NOT** supported

•it cannot be combined with other expressions, e.g.
anyparm(!parm) + chr(x) is **NOT** supported.

•*anyparm()* must be the only function in the expression

•:<u>help anyparm</u> has more details; :<u>help functions</u> shows all of the CI functions.



hp e3000

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
```

March 4, 2003 Page 71

•Script on jazz at: **http://jazz.external.hp.com/src/scripts/randname.txt**

•This example shows a script returning a value via a passed in variable.

•Shows using HPCPUMSECS to get a sort of pseudo random number.

•Breaking down the line: **setvar letter "chr((hpcpumsecs mod 26) + ord('A'))"**

- •HPCPUMSECS returns some large number
- •mod 26 returns a number in the range of 0..25
- •ord("A") is 65 and is the decimal number of the letter "A" (uppercase)
- •chr(0..25 + 65) is chr(65..90), which is one of the letters A..Z

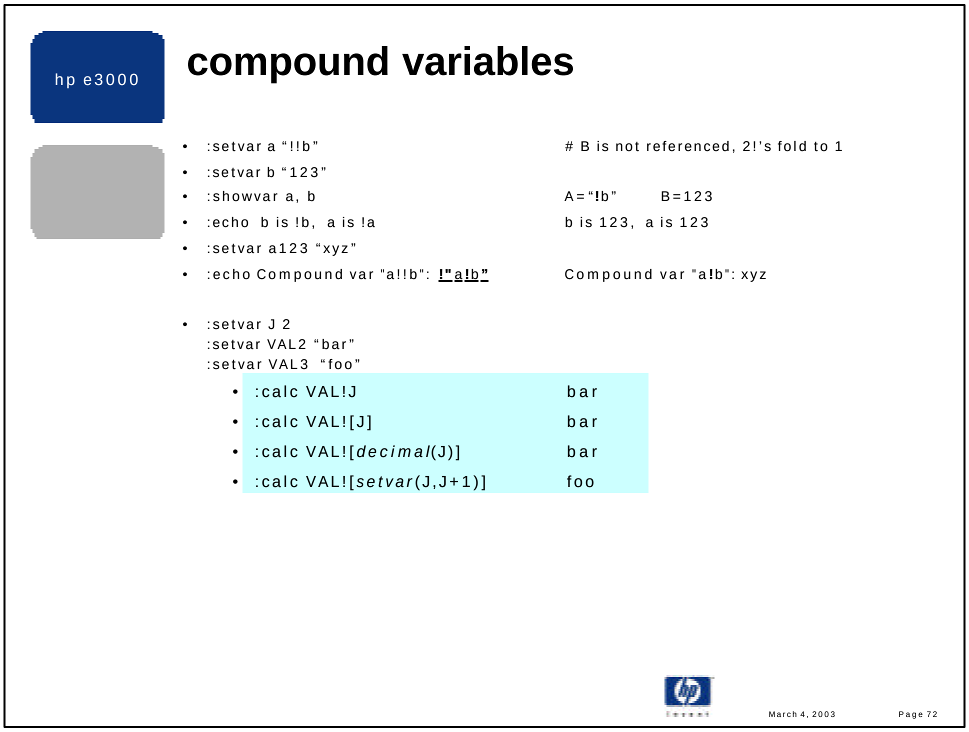
•The same logic applies to the "number" line above.

•The LIMIT line is evaluated as (using the parameter default values):

- (hpcpumsecs mod 8) is a number in the range of 0..7
- + minlen makes the number in the range 4..11
- min(4..11, 8) returns a pseudo random number in the range of 4..8, which is exactly what is desired.

•The WHILE loop iterates "limit-1" times, filling in the 2nd through "limit" characters in the name. If the HPCPUMSECS value is odd at this moment we append to the name a "random" letter, else a "random" number is appended.

•It would be nice to have a pseudo random number and name generator in the CI core, IMO!



hp e3000

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
```

March 4, 2003   Page 72

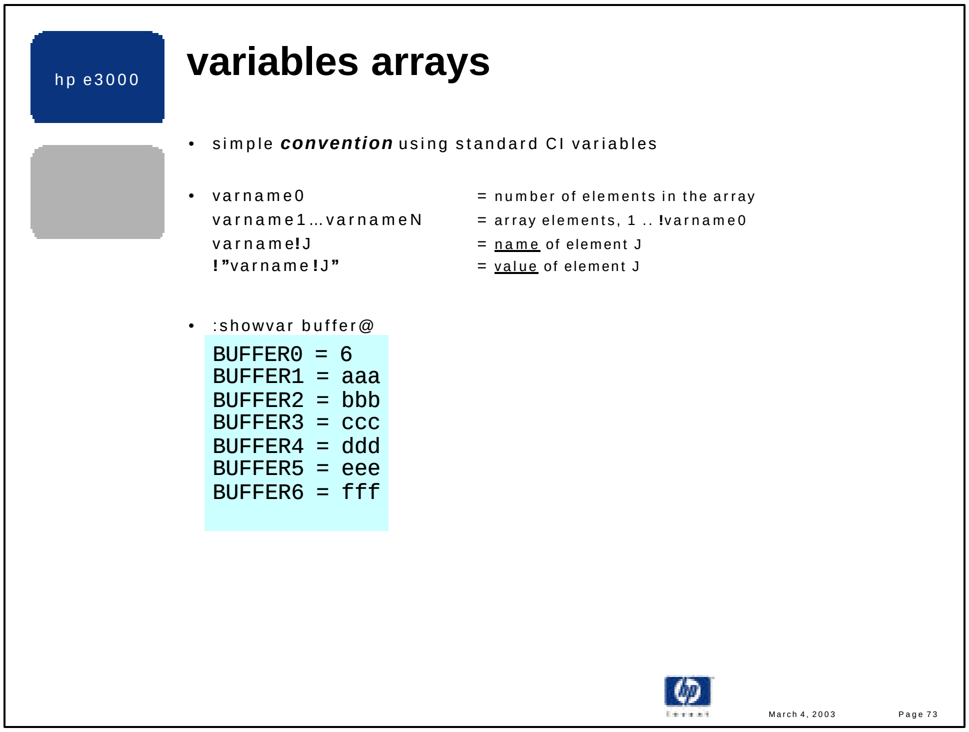
•The CI allows two or more variable names to be concatenated to form a new variable name, and to reference the value of this derived variable.

•A common application of compound names is variable arrays, discussed next.

•The value of a variable can reference another variable, e.g..

```
:setvar color “red”
:setvar bg  “!!color”
:showvar bg
BG = !color
:echo 123 !bg 456
123 red 456
```

•Explicit variable referencing resolves all levels of recursion; thus :showvar X and :echo !X will not produce the same result when X is set to “!!name” of another variable.



hp e3000

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
```

March 4, 2003 Page 73

•CI does not formally support arrays, but this simple convention works well. The technique also support heterogeneous arrays.

•Max number of CI variables depends on the length of the variable name and the size of its value.

•In 7.5 an approximate **maximum** number of user variables is **10,800** unique variables. This is derived as follows:

```
deletevar @
setvar z  0
while true do
   setvar z z+1
   setvar zz 0
   while setvar(zz,zz+1) <= 26 do
      setvar ![chr(ord("A")+zz-1)]!z  true                # A1, B1, C1… Z1 followed by
   endwhile
                                           # A2, B2, C2…Z2 etc.
endwhile
```

Executing this script fills the variable table, evident by the CI error reported below:

```
Symbol table full: addition failed.  To continue, delete some variables, or start a
new session. (CIERR 8122)
```

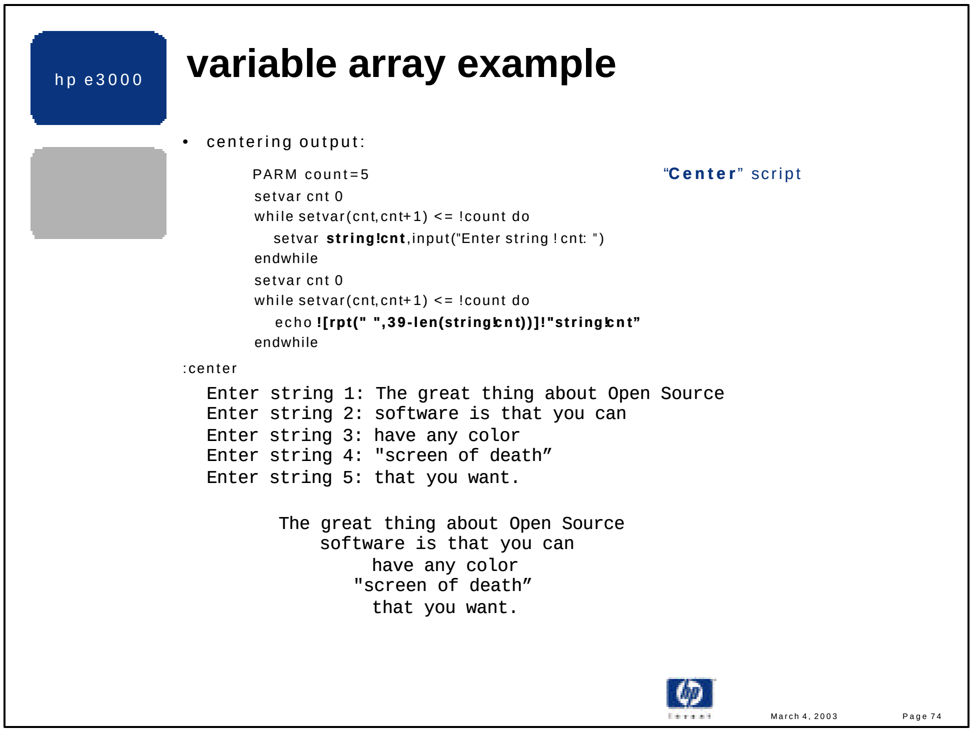
:calc ((z-1)*26)+zz+2 (the +2 is for the two local vars z, zz)

```
10804,  $2A34,  %25064
```

•An approximate more **typical** maximum number of user variables on 7.5 is: **8,347** unique variables, derived as:

```
deletevar @
setvar  z 0
setvar name '!![rpt(chr((hpcpumsecs mod 26)+ord("A")),(hpcpumsecs mod 14)+2)]'
  # var names begin w/ A-Z, from 2..15 chars long
setvar value '!![rpt(chr((hpcpumsecs mod 26)+ord("A")),(hpcpumsecs mod 60)+1)]'
  # var values begin w/ A-Z, from 1 to 60 chars long
while true do
    setvar !name![setvar(z,z+1)]  "!value"
endwhile
```

```
:calc z+3                          # + 3 for local variables: z, name, and value
 8347,  $209B,  %20233
```



hp e3000

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.
```

March 4, 2003 Page 74

•The “center” script shows generically the following:

•how to create a CI variable “array”

•how to access a variable “array”

•the **!”literal!name1”** construct, which allows compound variable names to be referenced. If literal = FOO, name1 = FUM and FOOFUM = 23 then
!”literal!name1” = !”FOO!name1” = !”FOOFUM” = !FOOFUM = 23

•**![rpt(“ “, fieldWidth - lenOfVar)]** puts the correct number of blanks before echoing the field’s value.

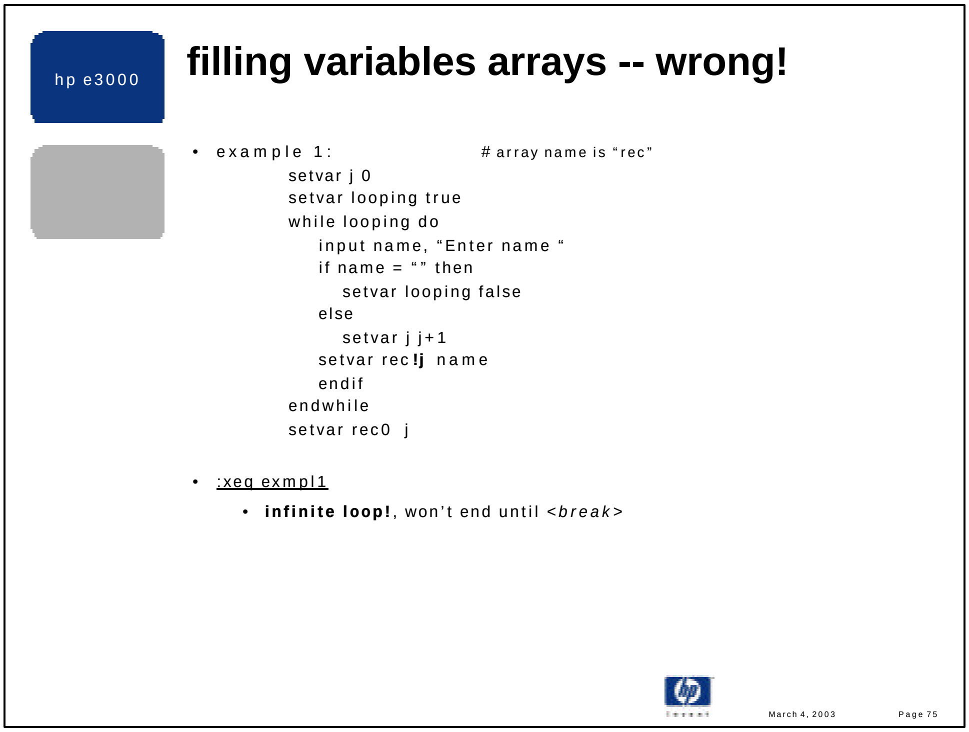
•Specifically, the “count” parameter is the number of elements in the “array”.

•**string!cnt,** where cnt is from 1..5, defines each element in the “array”.

•**!”string!cnt”** references the value of each element in the “array”.

•The rpt() function places the correct number of spaces before each line is echoed.

(The Open Source quote comes from Gavin Scott, Allegro Consultants, June ‘01 from the HP3000-L list.)



hp e3000

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*>

March 4, 2003 Page 75

•The previous “center” example had the size (or number of elements) of the array defined and thus hard-coded. This example is more general, in that, the size of the “array” is determined based on user input. In this case, when the user just presses <*return*>, meaning no more input, that defines the size of the array. These arrays can be very dynamic, limited only by the maximum number of variables supported by the CI – defined elsewhere in these slides.

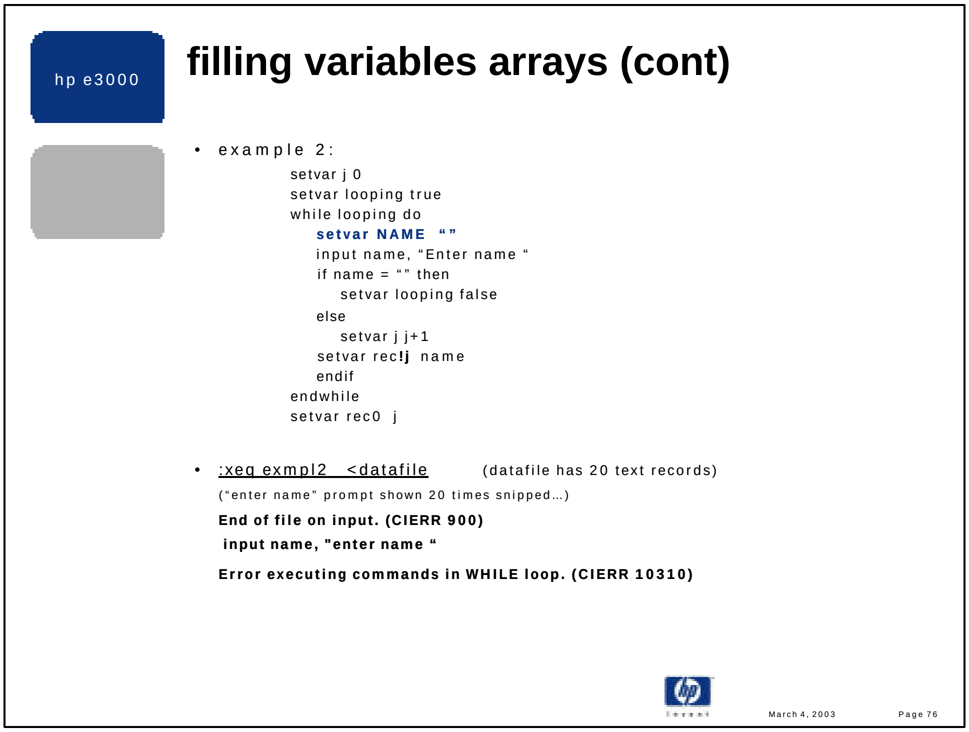
•To fix the infinite loop bug the variable “name” needs to be cleared or deleted inside the while loop. Recall that the INPUT command does not change the value of the variable if it times out or if the input value is null (“”). Thus we need to :deletevar name each iteration, or set it to “”, or use the input() function. Recall that the input() function returns an empty string, “”, if it times out or if the user just presses <*return*>.

•Syntax:

```
INPUT  [NAME=]varname
                  [[;PROMPT=]prompt ] [[;WAIT=]seconds  ]
                  [[;READCNT=]chars ] [[;DEFAULT=]default_str ]
                  [;CONSOLE ]
```

•The variable, , will always be created by INPUT if it does not yet exist. Varname's value is typically the exact value entered as a response by the user; however, if the user enters no response (either by just pressing the enter key, or via the INPUT read expiring) varname's value is determined as follows:

- if a DEFAULT= value is provided that becomes the value for varname.
- if no DEFAULT= is specified and varname already exists it is not changed.
- if no DEFAULT= is specified and varname does not exist it is created with a value of "" (empty string).



hp e3000

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
```

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

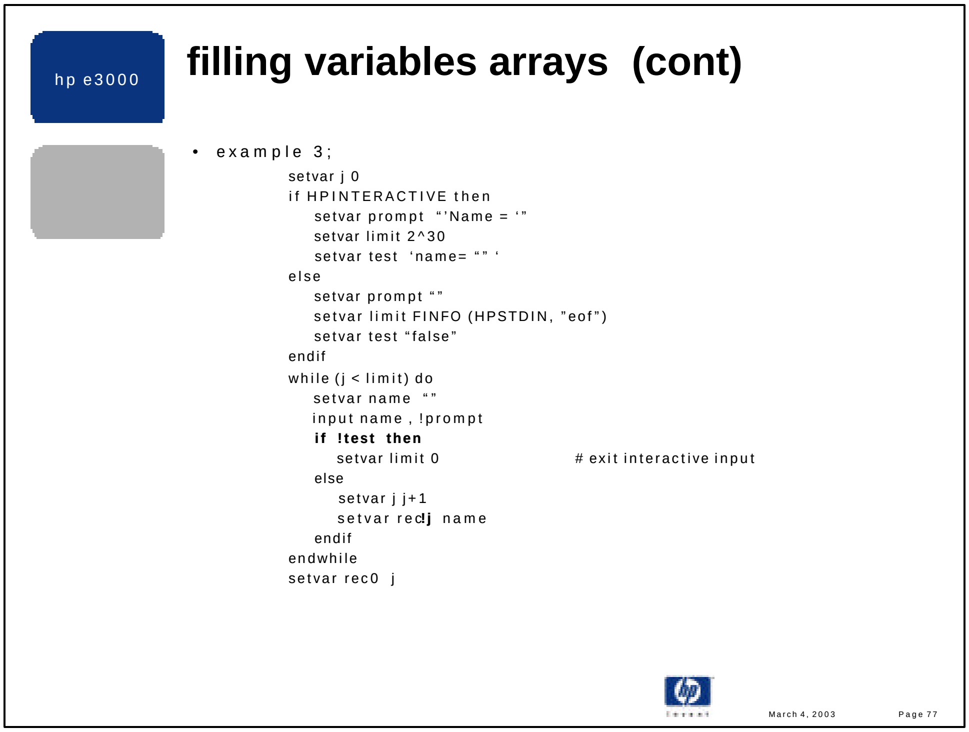
```
(“enter name” prompt shown 20 times snipped…)
End of file on input. (CIERR 900)
 input name, "enter name “
Error executing commands in WHILE loop. (CIERR 10310)
```

•Script as written works fine interactively!

•Works correctly if a line in DATAFILE is empty (but it must be variable width file)

•Otherwise, if datafile is fixed ASCII, you will see the “Enter name” prompt 20 times (no crlf) and get eof error on INPUT, as shown in the slide.

•The next slide shows how to modify this script to work correctly when $STDIN is redirected and still function as expected when invoked interactively.



hp e3000

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
```

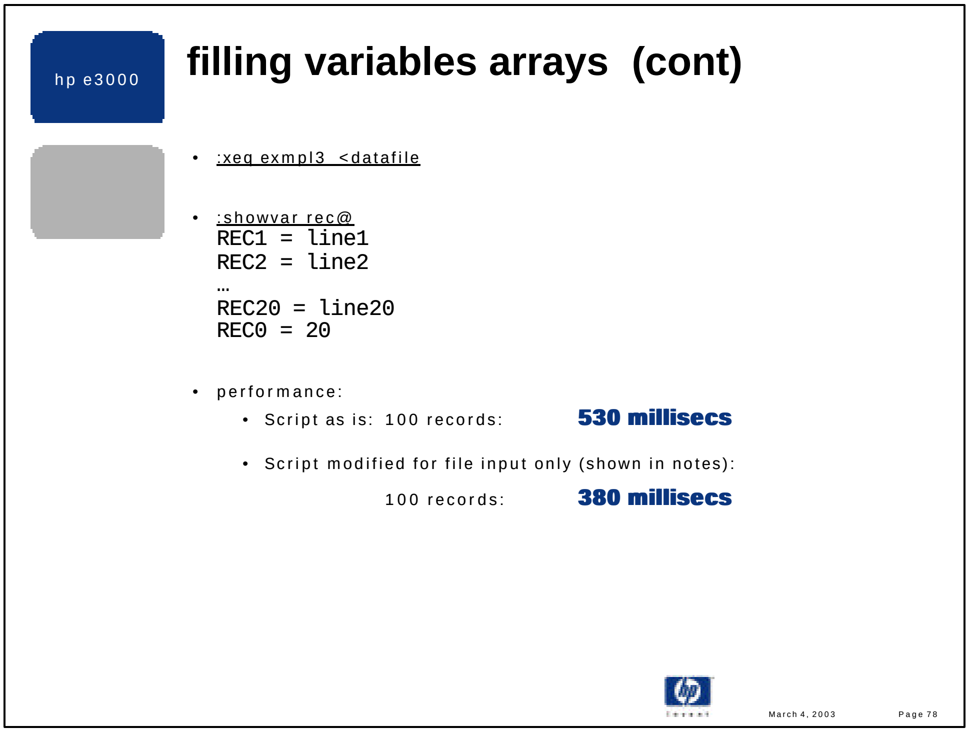
March 4, 2003 Page 77

•Don’t want blank lines in datafile to stop while loop, so we don’t test for “” in the redirected case.

•Each variable, **rec!j,** is 80 bytes long -- no blanks were stripped. This may be fine, or you can use the rtrim() function to remove the trailing spaces.

•Shows how you can make a ***dynamic*** CI command line, e.g.. **if !test then** ...

•Shows finfo(), HPINTERACTIVE and HPSTDIN.



hp e3000

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):

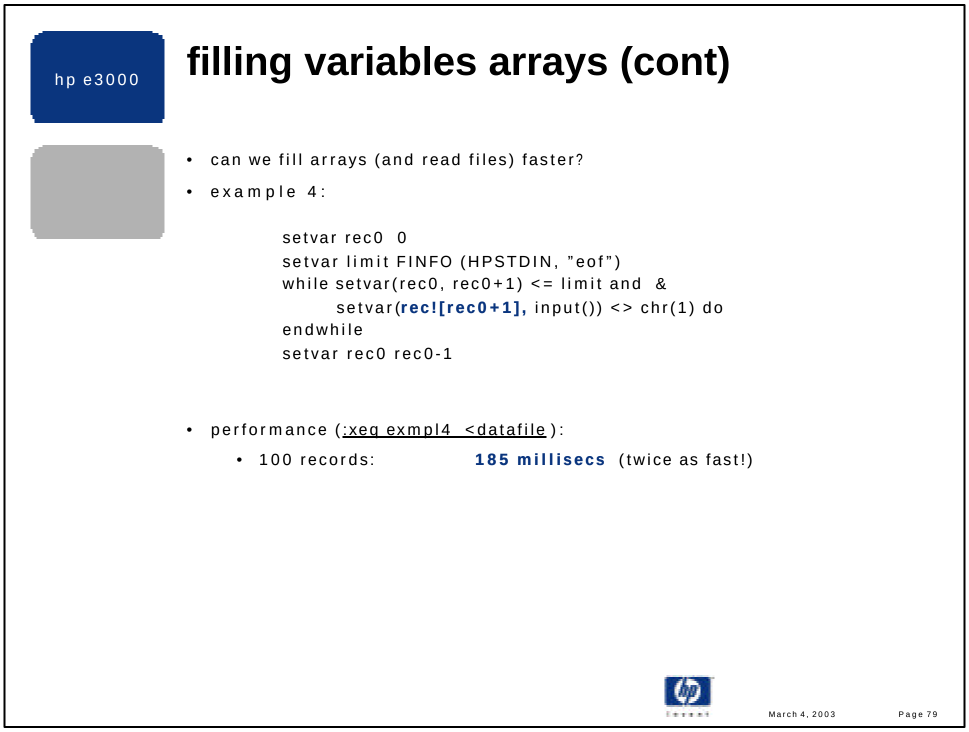
    100 records: **380 millisecs**

March 4, 2003 Page 78

•The script as written works correctly for both interactive and redirected environments; however, the most common usage is when input is redirected to a file. The next slide shows the script optimized for file input.

•Here is the modified version of example 3 to handle only file input:

```
setvar j 0
setvar limit FINFO (HPSTDIN, ”eof”)
while (j < limit) do
   setvar name  “”
   input name , !prompt
   setvar j j+1
   setvar rec!j  name
endwhile
setvar rec0  j
```



hp e3000

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!)

March 4, 2003 Page 79

• Is **rec0** being incremented TWICE in the while loop?

• No. Explicit referencing, ![rec0+1], is performed by the CI before the command name is even known to be “WHILE”. Thus, the command actually processed by the WHILE CI code is:

setvar(rec0, rec0+1) <= limit and setvar(**!rec1**, input()) <> chr(1)

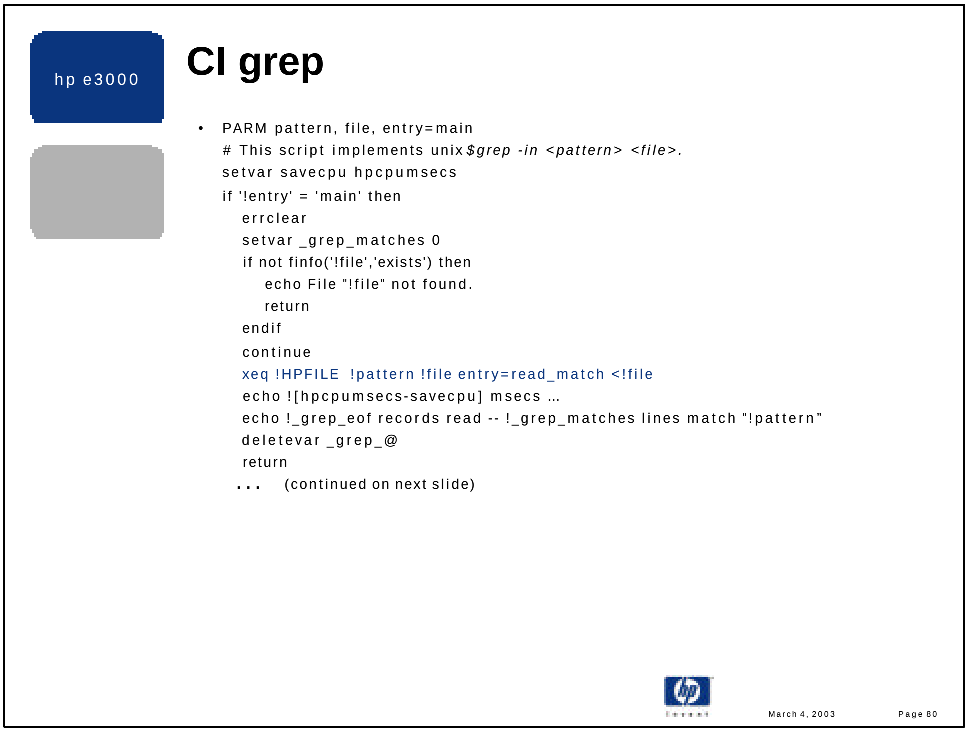
Note: if rec**!**[rec0+1] was replaced with rec**!**rec0, as I originally wrote the test script, then the loop counter and array high water mark (rec0) would be overwritten by the first record in the input file.

• This version of the script is twice as fast with just a little thought.

• Shows the *input()* function.

• Shows empty WHILE body.

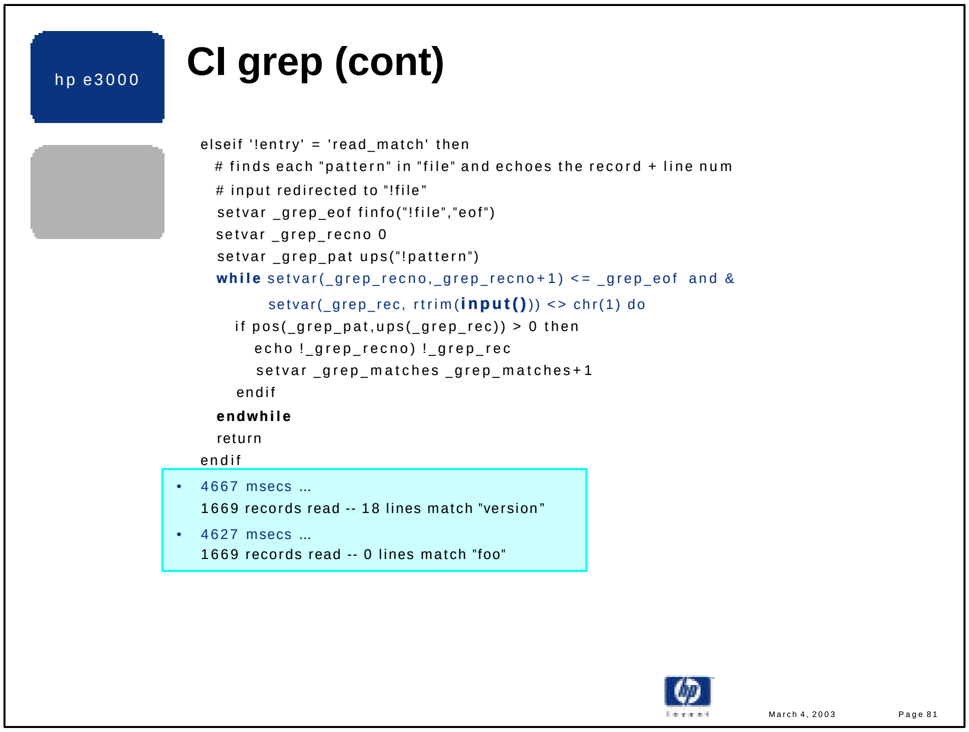
• The test against chr(1) is arbitrary but needed to have an empty while body.



hp e3000

CI grep

```
• PARM pattern, file, entry=main
  # This script implements unix $grep -in <pattern> <file>.
  setvar savecpu hpcpumsecs
  if '!entry' = 'main' then
     errclear
     setvar _grep_matches 0
     if not finfo('!file','exists') then
        echo File "!file" not found.
        return
     endif
     continue
     xeq !HPFILE  !pattern !file entry=read_match <!file
     echo ![hpcpumsecs-savecpu] msecs …
     echo !_grep_eof records read -- !_grep_matches lines match "!pattern"
     deletevar _grep_@
     return
   . . .    (continued on next slide)
```



hp e3000

CI grep (cont)

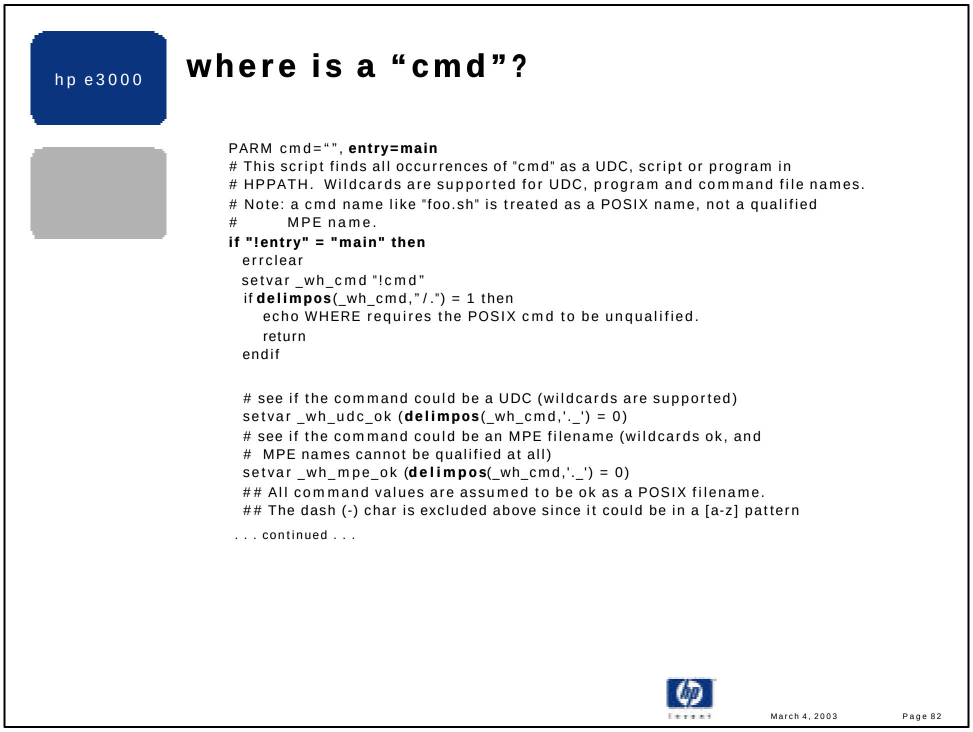
```
elseif '!entry' = 'read_match' then
  # finds each "pattern" in "file" and echoes the record + line num
  # input redirected to "!file"
  setvar _grep_eof finfo("!file","eof")
  setvar _grep_recno 0
  setvar _grep_pat ups("!pattern")
  while setvar(_grep_recno,_grep_recno+1) <= _grep_eof  and &
         setvar(_grep_rec, rtrim(input())) <> chr(1) do
     if pos(_grep_pat,ups(_grep_rec)) > 0 then
        echo !_grep_recno) !_grep_rec
        setvar _grep_matches _grep_matches+1
     endif
  endwhile
  return
endif
```

- 4667 msecs …
  1669 records read -- 18 lines match "version"
- 4627 msecs …
  1669 records read -- 0 lines match "foo"

March 4, 2003 Page 81

•It takes approximately 4.6 seconds to read, upshift and find a string literal in a 1669 record ascii file, and approximately 123 seconds to so the same in a 45,149 record file.

•xeq grep.hpbin.sys “-in pattern file” is much faster for large files! The GREP program in HPBIN.SYS does not support CI or shell wildcarding. If you need to grep a pattern on a set of files start grep from the shell.



hp e3000

where is a “cmd”?

```
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 . . .

March 4, 2003 Page 82

The *where* script combines many CI programming ideas: multiple entry points are used with input redirection, two forms of file I/O are used, several newer CI function are called, output is aligned in columns, and several more complex CI expressions are encountered. Plus, this script has proven valuable to me and others in CSY numerous times. The next few slide notes will go over some of the more salient points of the *where* script.

where can be found on Jazz at: **http://jazz.external.hp.com/src/scripts/where.txt**

•the PARM line allows the “cmd” argument to default to “”, in which case a usage statement is displayed. The by-convention “entry=main” argument is used to handle alternate entry points, with the default entry being named “main”. The user of *where* will never specify this parameter.

•the ERRCLEAR command is invoked to set CIERROR, HPCIERROR, FSERROR, and HPCIERRCOL predefined variables to 0.

•the delimpos() function is invoked several times and is better than using pos() when two or more characters are being checked. For instance, it is more efficient to code:

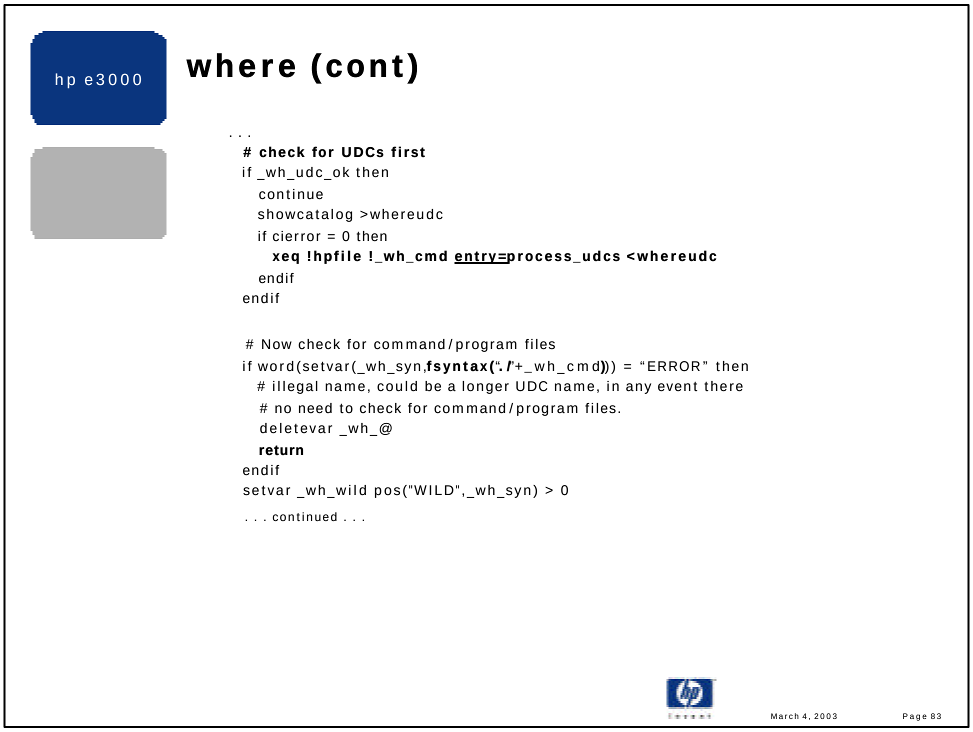
```
if delimpos(var,”abc”) > 0
```

which tests if an “a” or “b” or “c” appears in *var*, than to code:

```
if pos(“a”,var) > 0 or pos(“b”,var) > 0 or pos(“c”,var) > 0
```

•intentionally, there are separate tests to see if the “cmd” parameter could potentially be a UDC and/or a MPE named file. Currently, these tests are identical; however, over time the rules may change and this script will be easier to maintain in that event.

•all values of “cmd”, at this point, are assumed to be a legal POSIX filename. Later, the fsyntax() function will be called to ensure that “cmd” is a legal filename.



hp e3000

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 . . .
```

March 4, 2003 Page 83

•now, assuming “cmd” could be a UDC name, the SHOWCATALOG command is executed with output redirected to a TEMP file named “whereudc”.

•If SHOWCATALOG worked without error, the *where* script invokes itself recursively, via the XEQ command, to display relevant UDC information. The predefined HPFILE variable contains the fully qualified name of the current script, and is used here in case the next author decides to use a different filename. This allows the script filename to not be hard-coded into the script.

•the XEQ command invokes, via HPFILE, the script again, passing the same “cmd” value as the first argument. An alternate entry point is passed as the second parameter, via the by-convention usage of “entry=“. Input to *where* is redirected from the file that the SHOWCATALOG command created.

•the fsyntax() function is called after processing UDCs since a UDC name can be longer than a valid MPE filename. Also, the *where* script expects that all names, even POSIX command names, to be passed in unqualified. There are not explicit checks for qualified MPE names (f.g.a) since it is ambiguous if a name such as “foo.sh” is the name of a shell script, or a partially qualified MPE name. Since the user of this script is not expected to use the “MPE-escaped” syntax for POSIX names, a “./” is prepended to the “cmd” name that is parsed by fsyntax().

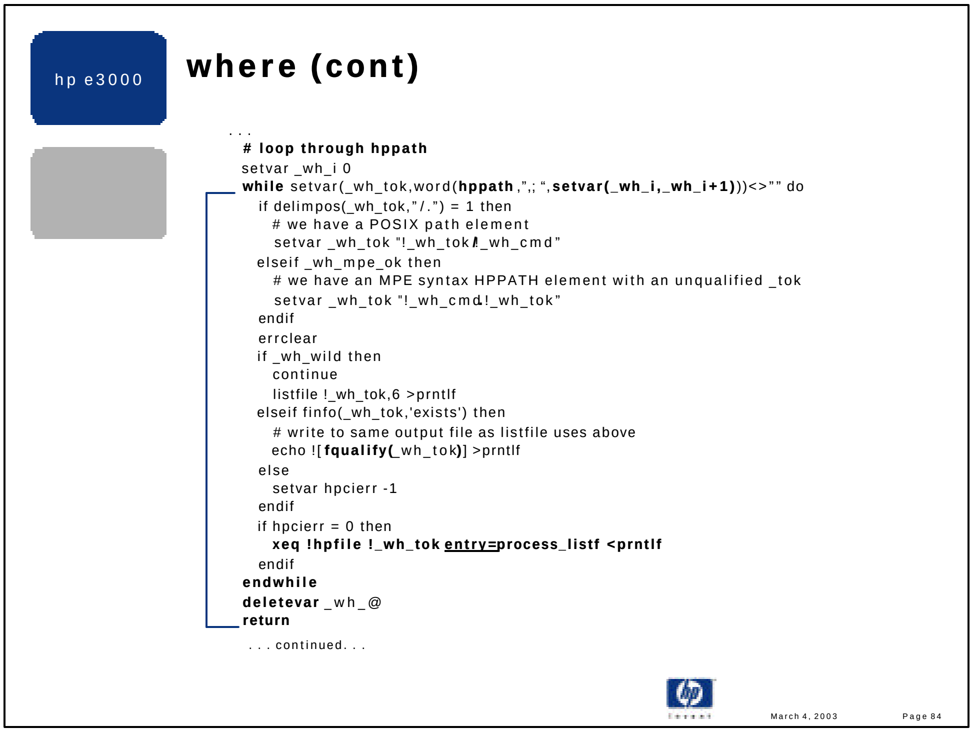
•if there is a syntax error the script exits via the RETURN command.

•a variable is set to true if there are any wildcard characters in the “cmd” value. In general, if an expression evaluates to a boolean (true or false) it can be used to directly set the value of a variable. For example:

```
setvar x (a > b)
```

is more efficient than:

```
if (a > b) then
   setvar x true
else
   setvar x false
endif
```



hp e3000

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. . .
```

March 4, 2003 Page 84

•this slide shows the end of the "main" entry code in the *where* script.

•here is the loop that parses each element in HPPATH, tests to see if a file exists based on the "cmd" value and the extracted element from HPPATH, and invokes an entry "subroutine" to display the filename and other file attributes.

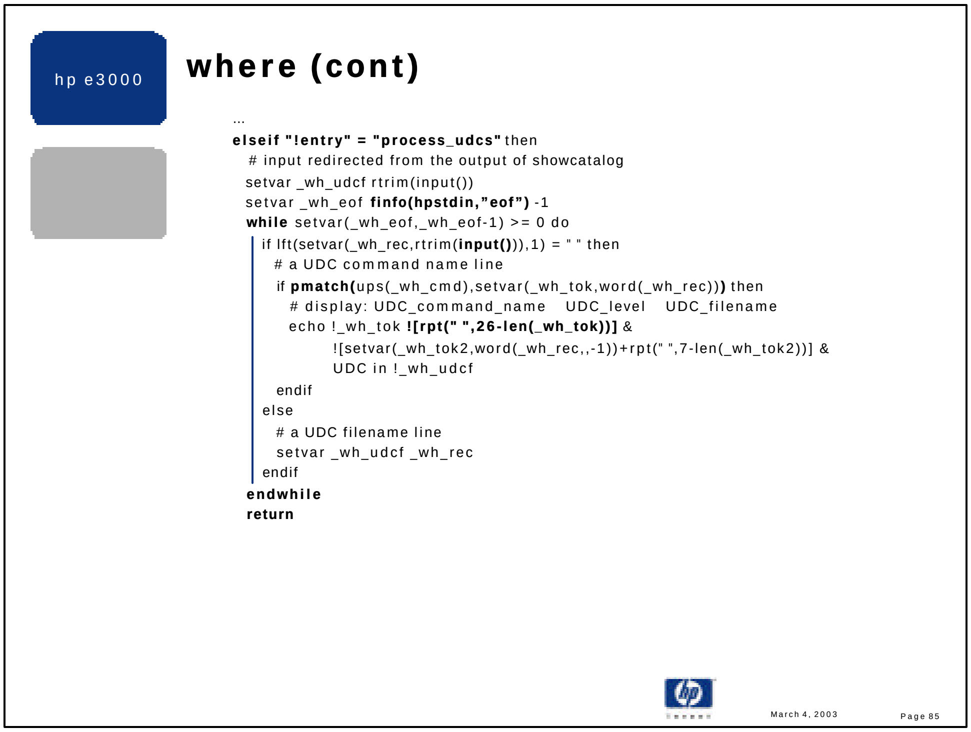
•the word() function extracts a token from HPPATH based on the defined delimiters of a comma, semicolon or a space. The word counter/index (_wh_i) is incremented inside the argument to word(), which is not necessary, but more convenient and slightly more efficient.

•the delimpos() function is used to see if the extracted HPPATH element is an MPE name or a POSIX name. POSIX elements are prepended to the "cmd" value and MPE path elements are appended to "cmd".

•if the "cmd" value was wildcarded, e.g. "grep@", then the LISTFILE command lists the full filenames to disk. Otherwise, the non-wildcard name is qualified by calling the **fqualify**() function, and written to the same output file used by LISTFILE. This allows a single entry routine to do all of the formatted output for a file.

•XEQ and HPFILE are used again to invoke the script recursively, this time passing the "process_listf" entry name, and redirecting input to a file that contains the equivalent of a LISTFILE,6 output.

•regardless of success or failure, all _wh_@ variables are deleted and control returns to the invoker of the script. In this script the two TEMP files are not purged and the file equation, seen later, is not reset. For scripts with more complex cleanup, I often use an alternate entry point specifically for doing all of the cleanup. This entry is invoked in place of executing a simple RETURN.



hp e3000

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
```

March 4, 2003 Page 85

•this is the “**process_udcs**” entry routine. It is invoked with input redirected to the output of a simple SHOWCATALOG command.

•it primes the variable _wh_udcf by reading the first record of the input file, which, in this case, is the name of the first cataloged UDC file.

•setting a counter to the “EOF” value of the input file and decrementing it to zero is a common method of processing the entire file. The HPSTDIN predefined variable contains the name of the $STDIN input file. In this case, it is the name of the file input was redirected to (which is the name of the file the SHOWCATALOG output was redirected to). HPSTDIN is used so that the I/O file name is not hard-coded throughout the script -- only where it is first created.

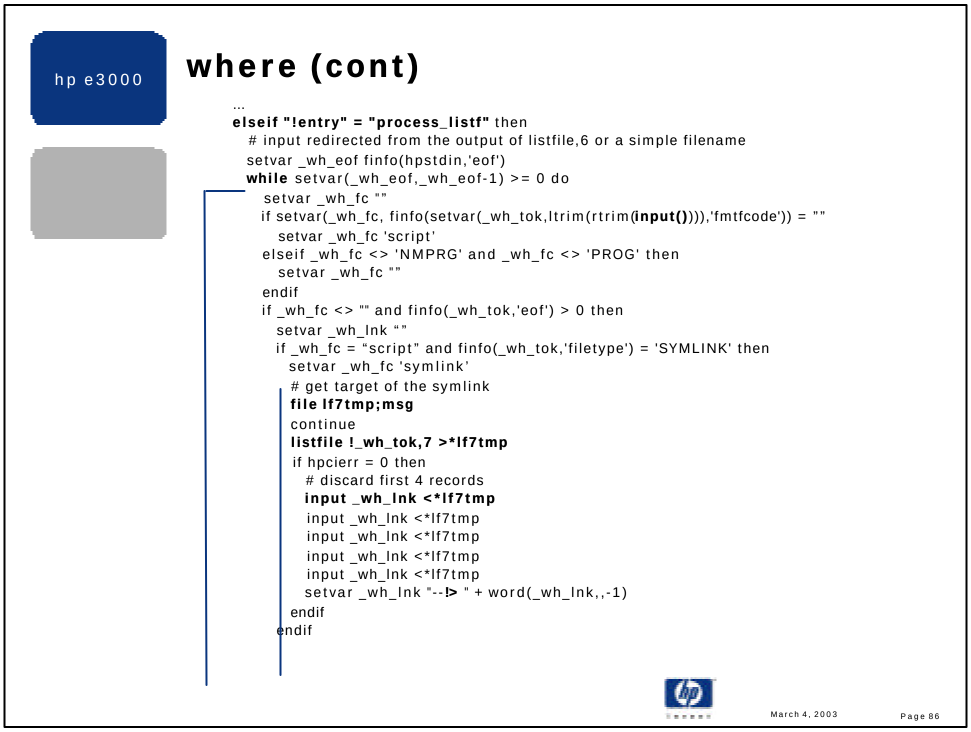
•the while loop decrements the eof counter, reads a record from the input file, trims trailing spaces from the record, decides if the record is a UDC filename (leftmost byte <> “ “) or a UDC command name record.

•if the record is a UDC command name that matches the “cmd” parameter value, a line of output is generated, containing: the UDC command name, the UDC level (user, account or system), and the UDC filename.

•All output is “tabularized” via the rpt() function by prepending or appending the appropriate number of spaces before or after the echoed value.

•The pmatch() function is an easy way to add pattern matching power to your scripts. HELP pmatch provides more information. Since “cmd” could also be the name of a POSIX file, its value is not permanently upshifted. Local upshifting is needed since all UDC names reported by SHOWCATLOG are in uppercase.

•the entry routine exits, via RETURN, back to its caller, which is the “main” entry code. The RETURN command closes the file (*where*) and resets I/O redirection back to its state prior to the invocation of the entry point -- in this case input is back to the terminal $STDIN.



hp e3000

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
```

March 4, 2003 Page 86

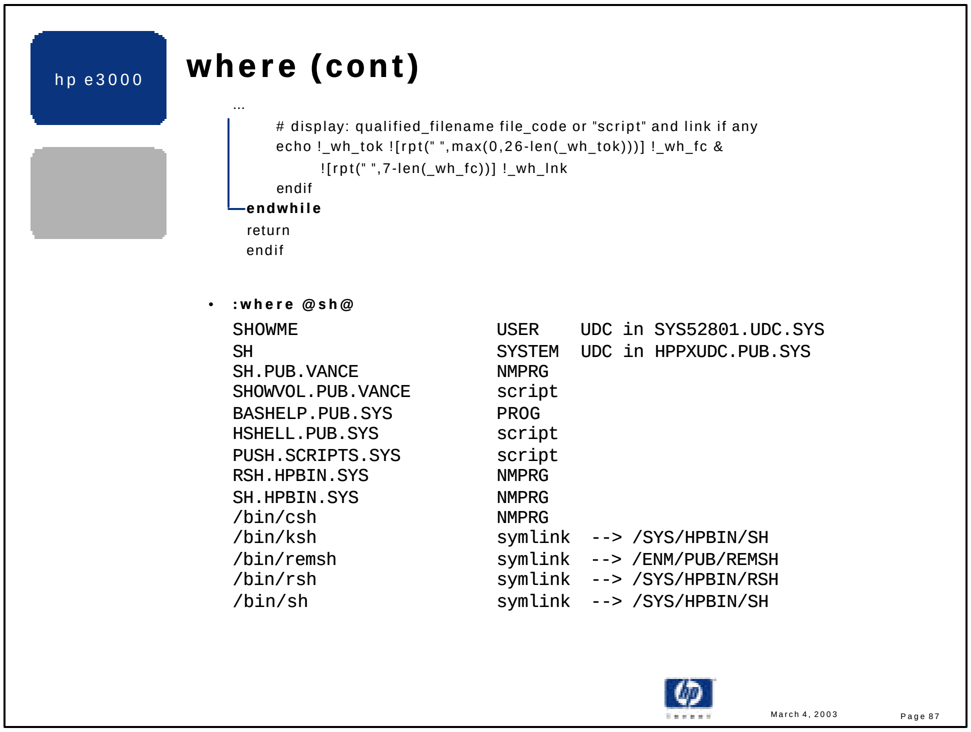
•this is the “**process_listf**” entry routine. It is invoked with input redirected to the output of a LISTFILE,6 command.

•the while loop reads each record in the input file, tests to ensure the file could be a legitimate script or program file, and symbolic links are resolved.

•the input() function reads each filename in the input file, after which, trailing and leading blanks are trimmed. The _wh_tok variable is set to this trimmed value. The finfo() function is called, passed this same trimmed name, to obtain the formatted file code, which is stored in the _wh_fc variable. If the file code is blank (“”) it is arbitrarily set to “script”. All of this is done is a single command line.

•if the EOF is positive and the file code is “script” then the script tests to see if the name might be the name of a symbolic link.

•if FINFO returns “symlink” for the file type then the target of the link is retrieved. This is done using a small MSG file and I/O redirection, as follows: 1) a LISTFILE,7 is written to the MSG file, 2) if the LISTFILE is successful the MSG file is read (all reads are destructive), 3) the first four records in the MSG file can be discarded, done by reading them and ignoring the input, 4) the last word/token in the fifth record contains the name of the target of the symlink, which is extracted, and has “-->” prepended to enhance the final output. The “-->” strings need a “!” to escape the meaning of “>”, which if not done, causes the following ECHO statement to perform output redirection.



hp e3000

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
```

March 4, 2003 Page 87

•this concludes the “process_listf” entry and the *where* script.

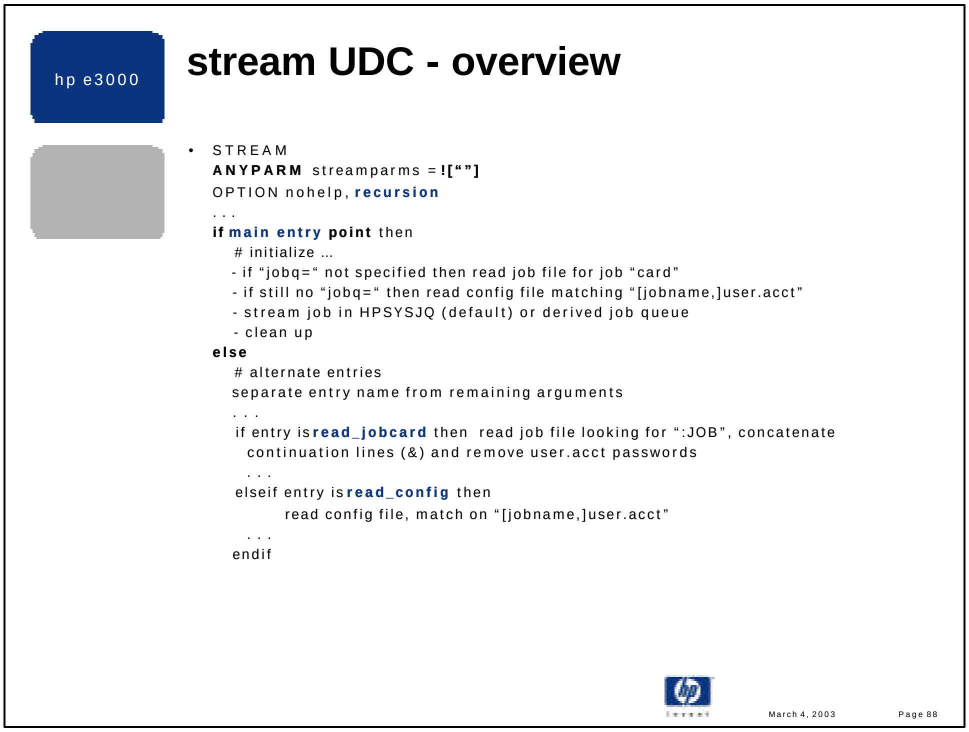
•the ECHO command displays qualified (MPE or POSIX) filename, the file code (which can be set to a non-MPE value of “script”, and symbolic link info, if pertinent. Note again that the rpt() function is used to left justify the file code string and any symlink display.

•as should be done for all entry routines, RETURN exits back to the “main” entry, where cleanup is done.

•Example:

```
HPPATH = !HPGROUP,PUB,PUB.SYS,ARPA.SYS,scripts.sys,hpbin.sys,/bin
: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
```

March 4, 2003

Page 88

•http://jazz.external.hp.com/src/scripts/stream.txt

•Shows entry points used with UDC. ANYPARM requires more parsing and a convention for the entry specification. In my example, the entry is always specified as “entry=name” and is the last argument in the command line.

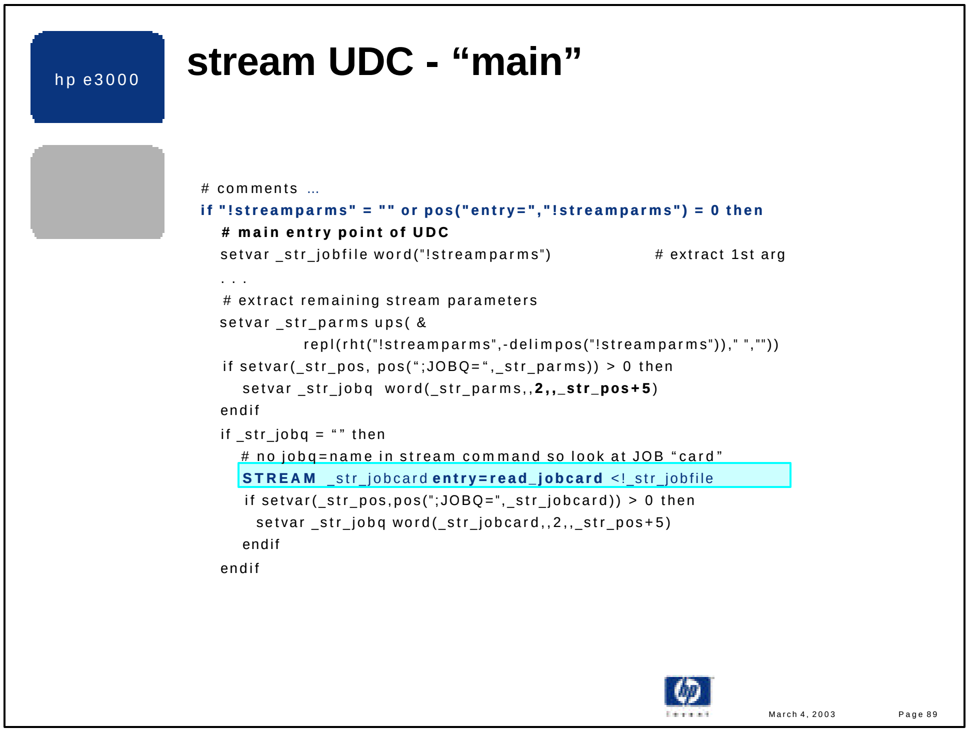
•Shows how to default an ANYPARM value to nothing, ![“”]. Quotes by themselves don’t work, and, in fact, cause the value to default to the quote marks literally.

•OPTION NOHELP chosen since this UDC overrides a built-in CI command. If a user enters “help stream” they will not see the contents of this UDC; instead, they will see the HELP text for the real STREAM command.

•OPTION RECURSION is specified, since there are several recursive calls to the STREAM UDC, as a way to process the various entry points. OPTION NORECURSION will be executed prior to invoking the real :stream command.

•Sample job queue configuration file:

```
          # (All comments appear at the end of this file for search performance
reasons)
          j@,usr1.acct   jobqJ
          usr1.acct                jobq1
          @.acct                   jobq2
          @.@                                     mySysDefq
          ...
```



hp e3000

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
```

March 4, 2003 Page 89

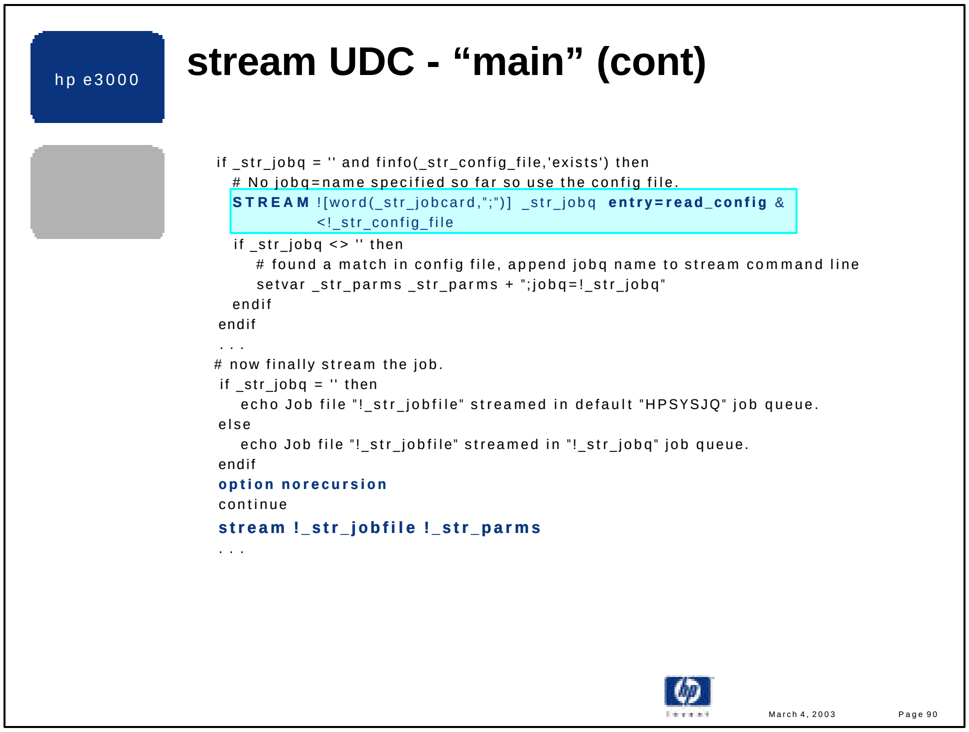
•The main entry point is detected by the absence of all parameters or by the lack of the “entry=“ keyword.

•The first parameter extracted is the name of the file to be streamed.

•The remaining parameters are captured in the variable _str_parms, after the command line has been upshifted and all blanks have been removed.

•If the “;JOBQ=“ keywords is found in the command line the queue name is extracted. You might wonder why the second word (instead of the default of 1), and why at a position that indexes the “=“ rather than the character immediately right of the “=“? Using **word(_str_parms,,,,_str_pos+6)** works in all cases, including a null (empty) jobq value. However, it fails when ;jobq= with no value is the last token on the command line. It fails in this case since the index (_str_pos+6) is beyond the end of the _str_parms string length. Extracting the **second** word starting at the “=“ works in all cases.

•If “jobq=“ is not present in the command line, the STREAM UDC invokes itself (highlighted in blue) using an alternate entry point, with $STDIN redirected to the file being streamed. This method allows the stream file to be read efficiently by the UDC.



hp e3000

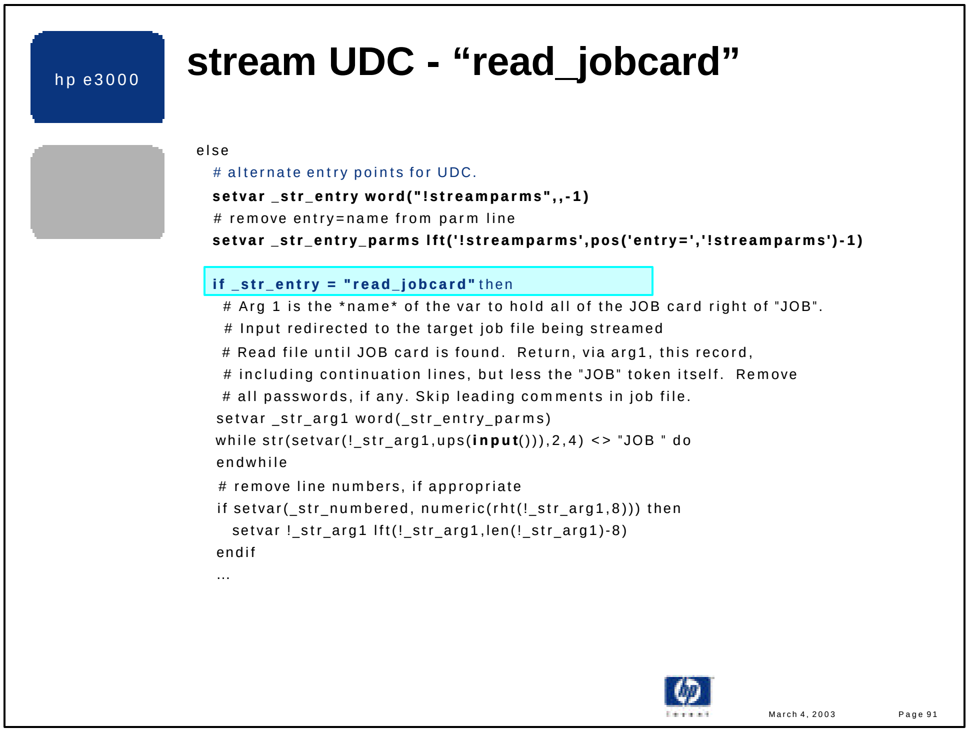
stream UDC - “main” (cont)

```
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
 . . .
```

March 4, 2003 Page 90

•If “jobq=“ is not found in the job “card” and if the simple configuration file exists, the STREAM UDC is again invoked recursively to read the config file looking for a match. The config file has two fields: the first field is a [jobname,]user.acct name, the second field is the corresponding job queue name. Wildcards are supported in the first field. The code that processes the config file is shown later.

•Finally, the real STREAM CI command is invoked with an appended jobq=name if appropriate. To execute the real STREAM command, OPTION NORECURSION is specified; otherwise the STREAM UDC would be invoked (and in this case we would have an infinite loop -- eventually stopped by a CI limit that disallows UDC nesting beyond 100 levels.



hp e3000

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
    ...
```

March 4, 2003 Page 91

•The next few slides detail the two alternate entry points for the STREAM UDC. If the entry is not “main” then it is an alternate entry. The first step is to determine which entry is being called by extracting the entry name. By convention the entry name is the last parameter passed to the UDC, and thus is extracted via word(…,-1).

•Next, the “entry=name” needs to be removed from the parameter line so that the alternate entry routines can freely parse the arguments.

•Now a test can be made for each individual entry name, and each entry point can be coded like a subroutine. All entries have read and write access to all of the variables set by the UDC.

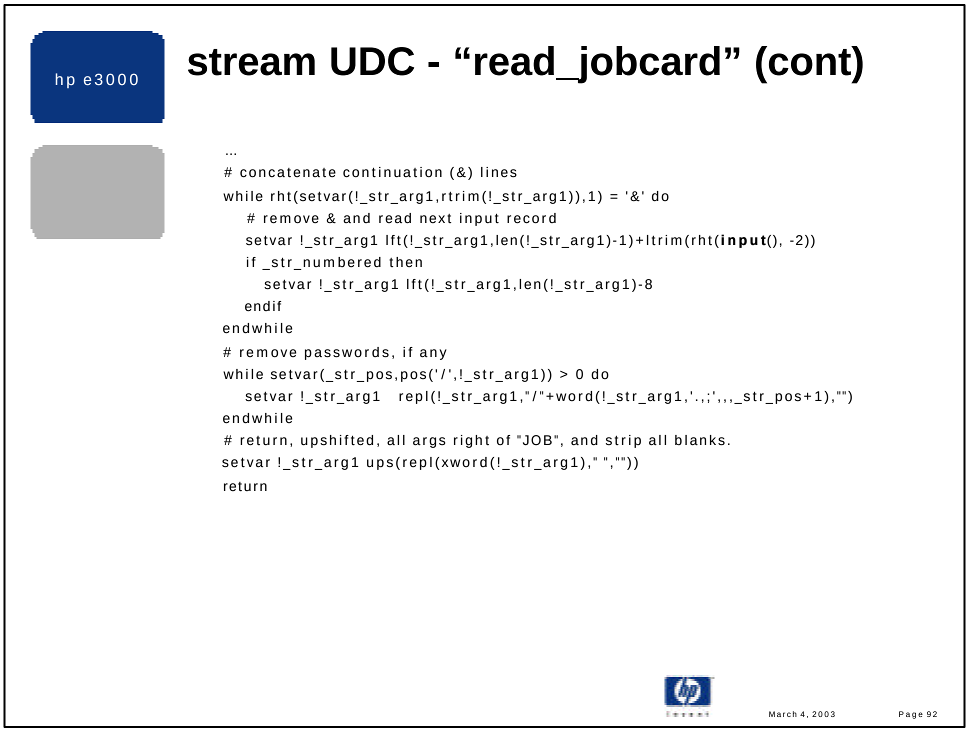
•The “read_jobcard” entry defines the first parameter (arg1) to be the **name** of a CI string variable that will contain the full job “card” line minus the pseudo colon and the word “JOB” (“!JOB “).

•Input has been redirected to the stream job file, which the “main” entry verified exists.

•Since there can be comments preceding the JOB command line, these are skipped by the WHILE loop above. This WHILE loop reads the JOB record, via the input() function, and stops.

•A simple test is made to determine if the stream file is numbered or unnumbered: if the last 8 characters of the JOB card record are numeric then the entire file is considered numbered.

•continued...



hp e3000

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
```

March 4, 2003 Page 92

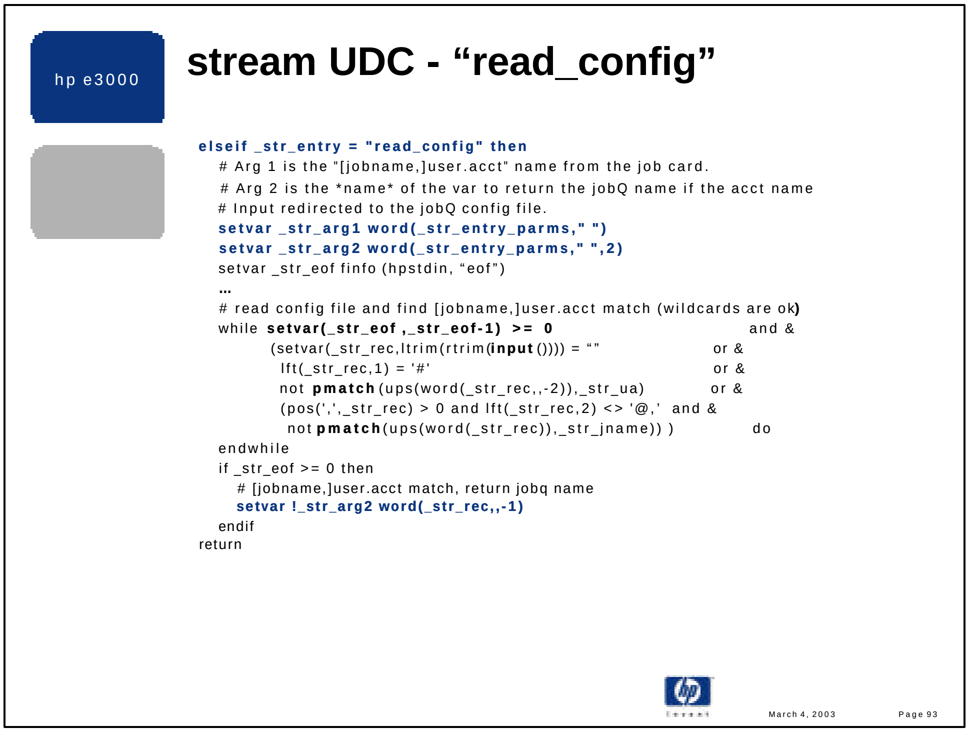
•If the JOB record is continued (ends with an ampersand) then the first WHILE loop above will read the remaining continuation lines.

•Each continuation line is appended to the return variable (arg1) after numbers and leading spaces (ltrim) are removed.

•!_str_arg1 is referenced, rather than simply “_str_arg1” since the contents of _str_arg1 is the **name** of a variable. For instance, in the STREAM UDC arg1 is passed as “_str_jobcard”. After calling the read_jobcard entry the main body of the UDC will test the value of _str_jobcard, looking for a JOBQ parameter. Using !_str_arg1 on the left side of a SETVAR is like using “_str_jobcard”.

•Next, any user, account and/or group passwords, if present, are removed (not blanked over). If a password is found (pos of “/” > 0) then the “/” and the password itself are replaced with “”.

•Finally, the concatenated, password filtered, de-numbered, de-blanked and upshifted JOB record is returned, via arg1, to the caller. The “:JOB “ portion is also removed by the xword function.



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
```

•The “read_config” entry reads the config file (verified by “main” to exist) looking for a user.acct match. This entry defines the first parameter (arg1) to be the string “[jobname,]user.acct” from the JOB record returned by the read_jobcard entry. The second parameter (arg2) is defined to be the name of a CI string variable the will hold “” or the corresponding job queue name.

•These two arguments are easily extracted via word(…,1) and word(….,2). Note that one is the default parameter number for word().

•Input has been redirected to the configuration file.

•The WHILE loop stops if the entire config file has been read or on the first match. Based on this implementation, specific entries (specific [jobname,]user.acct) names should proceed generic, wildcarded names.

•The WHILE loop continues for empty (blank) records and comment (#) lines.

•A match is defined as: user.acct matches and if a jobname is present in the config file (and not simply “@”) the jobnames must match too. Wildcard support is easy with pmatch()!

•Line by line evaluation of this WHILE loop:

  •decrement a counter that initially contains the number of records in the config file. When this counter is negative the file has been completely read.

  •set the variable _str_rec to a record in the config file, after trimming all trailing and leading blanks, test if the result is empty and if so continue the while loop.

  •if the _str_rec record starts with “#” then skip it since it is a comment record.

  •if the second-to-last word in the record (this is the user.acct token -- second-to-last is used rather than first to handle an optional jobname which is terminated by a comma) doesn’t match the user.acct already extracted from the JOB card then continue the loop.

  •if user.acct matches and the config record has a jobname (pos comma > 0) and the jobname is not “@” and the jobname doesn’t match the already extracted jobname from the JOB card then continue the loop.

•The loop ends when either all records have been read without a match, or there is a match. If the loop counter (_str_eof) is >= 0 then there was a match and the corresponding job queue name (last word in the config file record) is returned via arg2.