

HP Technology Forum 2006

DCL Programming Hands-on Session

David J Dachtera

djesys@earthlink.net

DJE Systems - <http://www.djesys.com/>



GET CONNECTED
People. Training. Technology.

This presentation is intended to be displayed or printed in the “Notes View” so it reads like a text book.

If you are viewing this as a “Slide View” .PDF (Adobe Acrobat file), download the .PPT (PowerPoint presentation) from:

http://www.djesys.com/vms/support/session_1066.ppt



Agenda - Introduction

Basic DCL Concepts

Commands

Verbs

Symbols

IF-THEN

IF-THEN-ENDIF

IF-THEN-ELSE-ENDIF

Labels, GOTO

Agenda - Introduction, Cont'd

GOSUB-RETURN

SUBROUTINE ENDSUBROUTINE

Common Lexical Functions

F\$CVTIME

F\$GETDVI

F\$GETJPI

F\$GETQUI

F\$GETSYI

PARAMETERS

Logical Names

Agenda - Introduction, Cont'd

Batch jobs that reSUBMIT themselves

Daily Jobs

Weekly Jobs

Question & Answer

- Break -

Agenda - Intermediate

SUBROUTINE - ENDSUBROUTINE

CALL subroutine [p1[p2[...]]]

Why CALL instead of GOSUB?

More Lexical Functions

F\$SEARCH

F\$TRNLNM

F\$ENVIRONMENT

F\$PARSE

F\$ELEMENT

Agenda - Intermediate, Cont'd

F\$SEARCHing for files

File I/O

Process Permanent Files (PPFs)

File Read Loops

Hands-on Exercise, Part 1

Using F\$ELEMENT to parse input strings

F\$ELEMENT loops

Symbol Substitution

Using Apostrophes (`'symbol'`, `'''symbol'''`)

Using Ampersand (&)

Agenda - Advanced

Logical Name Table Search Order

Tips, tricks and kinks

F\$TYPE()

Tips, tricks and kinks

F\$PARSE()

Tips, tricks and kinks

Loops using F\$CONTEXT(), F\$PID()

Select processes by name, node, etc.

Agenda - Advanced, Cont'd

Using F\$CSID() and F\$GETSYI()

Get/display info. about cluster nodes

The PIPE command

Usage Information

Techniques

Reading SYS\$PIPE in a loop

Hands-on exercise, Part 2

Getting command output into a symbol

Symbol substitution in “image data”

DCL - Programming?

DCL as a programming language?

Certainly!

DCL has many powerful features!

DCL Command Elements

\$ verb parameter_1 parameter_2

DCL Commands consist of a verb and one or more parameters.

DCL Verbs

Internal commands

ASSIGN, CALL, DEFINE, GOSUB, GOTO, IF, RETURN, SET, STOP, others...

External commands

APPEND, BACKUP, COPY, DELETE, PRINT, RENAME, SET, SUBMIT, others...

DCL Verbs, Cont'd

“Foreign” Commands

\$ symbol = value

Examples:

```
$ DIR ::= DIRECTORY/SIZE=ALL/DATE
```

```
$ ZIP ::= $ZIP/VMS
```

More Foreign Commands

The DCL\$PATH Logical Name (V6.2 +)

Behaves similar to the DOS or UN*X “path”:
.COM and .EXE files can be sought by way
of DCL\$PATH

```
$ DEFINE DCL$PATH MYDISK:[MYDIR.PROGS]
```

DCL\$PATH can even be a search list:

```
$ DEFINE DCL$PATH -  
    MYDISK:[MYDIR.COM],MYDISK:[MYDIR.EXE]
```

DCL\$PATH Caveat

Specifying an asterisk (“*”) at the DCL prompt, or an invalid specification which results in DCL seeing an asterisk or other wildcard specification, can produce undesirable results:

```
$ *
```

```
$ dirdisk:*.txt
```

```
%DCL-W-NOLBLS, label ignored - use only within command  
procedures
```

```
·  
·  
·
```

DCL\$PATH Caveat

Determine what might be found via a wildcard specification:

```
$ DIR DCL$PATH:*.COM;
```

```
$ DIR DCL$PATH:*.EXE;
```

DCL\$PATH Caveat

Avoid wildcard problems:

Place a “\$.EXE” program and a “\$.COM” in the DCL\$PATH path. Each should just exit without doing anything.

URL:

[http://www.djesys.com/freeware/vms/make_\\$.dcl](http://www.djesys.com/freeware/vms/make_$.dcl)

Download, RENAME to .COM and invoke it.

\$ @make_\$.com

Command Qualifiers

\$ command/qualifier

\$ command/qualifier=value

\$ command/qualifier=(value,value)

\$ command/qualifier=keyword=value

\$ command/qualifier=-

(keyword=value,keyword=(value,value))

Non-positional Qualifiers

Apply to the entire command, no matter where they appear.

```
$ command param1/qual param2
```

Example:

```
$ COPY A.DAT A.NEW/LOG
```

```
$ DELETE/LOG C.TMP;
```

Positional Qualifiers

Apply only to the object they qualify.

```
$ command param1/qual=value1 -  
    param2/qual=value2
```

Examples:

```
$ PRINT/COPIES=2 RPT1.LIS,          RPT2.LIS
```

```
$ PRINT RPT1.LIS/COPIES=1,-  
    RPT2.LIS/COPIES=3
```

Common Qualifiers

Many commands support a set of common qualifiers:

`/BACKUP /BEFORE /CREATED /EXCLUDE /EXPIRED
/INCLUDE /MODIFIED /OUTPUT /PAGE /SINCE`

See the on-line HELP for specifics.

DCL Statement Elements

\$ vbl = value

DCL statements are typically assignments where a variable receives a value.

Assignment Statements

```
$ vbl = F$lexical_function( params )
```

Examples:

```
$ FSP = F$SEARCH("*.TXT")
```

```
$ DFLT = F$ENVIRONMENT ("DEFAULT")
```

```
$ NODE = F$GETSYI("NODENAME")
```

Assignment Statements

```
$ vbl = string_expression
```

Examples:

```
$ A = "String 1 " + "String 2"
```

```
$ B = A - "String " - "String "
```

```
$ C = 'B'
```

Maximum string length = 255 bytes (up to V7.3-1)
4095 bytes (V7.3-2 +)

Assignment Statements

Character replacement within a string

```
$ vbl[begin,length] := string_expression
```

The colon (":") is **REQUIRED!**

Example:

```
$ A = "String 1"
```

```
$ A[3,3] := "ike"
```

```
$ SHOW SYMBOL A
```

```
A = "Strike 1"
```

Assignment Statements

`$ vbl = numeric_expression`

Examples:

`$ A = 1`

`$ B = A + 1`

`$ C = B + A + %X7F25`

`$ D = %O3776`

Assignment Statements

```
$ vbl[start_bit,bit_count]=numeric_exp
```

Examples:

```
$ ESC[0,8]=%X1B
```

```
$ CR[0,8]=13
```

```
$ LF[0,8]=10
```

```
$ FF[0,8]=12
```

```
$ CRLF[0,8]=13
```

```
$ CRLF[8,8]=10
```

Assignment Statements

```
$ vbl = boolean_expression
```

Examples:

```
$ MANIA = ("TRUE" .EQS. "FALSE")
```

```
$ TRUE = (1 .EQ. 1)
```

```
$ FALSE = (1 .EQ. 0)
```

```
$ YES = 1
```

```
$ NO = 0
```

Assignment Statements

Local Assignment:

```
$ vbl = value
```

Global Assignment:

```
$ vbl == value
```

Assignment Statements

Quoted String:

```
$ vbl = "quoted string"
```

Case is preserved.

Examples:

```
$ PROMPT = "Press RETURN to continue "
```

```
$ INVRSP = "% Invalid response!"
```

Assignment Statements

Unquoted string:

```
$ vbl := unquoted string
```

Case is **NOT** preserved, becomes uppercase.
Leading/trailing spaces are trimmed off.

Examples:

```
$ SAY := Write Sys$Output
```

```
$ SYSMAN := $SYSMAN ! Comment
```

Foreign Commands

```
$ vbl := $filespec[ param[ param[ ...]]]
```

“filespec” defaults to SYS\$SYSTEM:.EXE

Displaying Symbol Values

Use the `SHOW SYMBOL` command to display the value of a symbol:

```
$ A = 15
$ SHOW SYMBOL A
  A = 15    Hex = 0000000F    Octal = 00000000017

$ B := Hewlett Packard
$ SHOW SYMBOL B
  B = "HEWLETT PACKARD"

$ B := "Hewlett" Packard
$ SHOW SYMBOL B
  B = "Hewlett PACKARD"
```

Displaying Symbol Values

Use the SHOW SYMBOL command to display the values of symbols (wildcarded):

```
$ SHOW SYMBOL $*  
$A = "X"  
$RESTART == "FALSE"  
$SEVERITY == "1"  
$STATUS == "%X00030001"
```

Deleting Symbols

Use the DELETE/SYMBOL command to delete symbols:

```
$ DELETE/SYMBOL A
```

```
$ DELETE/SYMBOL/GLOBAL X
```

Conditional Expressions

\$ IF condition THEN statement

Variations:

\$ IF condition THEN \$ statement

\$ IF condition THEN -

\$ statement

Conditional Expressions

\$ IF condition

\$ THEN

\$ statement(s)

\$ ENDIF

Conditional Expressions

\$ IF condition

\$ THEN

\$ IF condition

\$ THEN

\$ statement(s)

\$ ENDIF

\$ ENDIF

Conditional Expressions

```
$ IF condition
```

```
$ THEN
```

```
$     IF condition
```

```
$     THEN
```

```
$         statement(s)
```

```
$     ENDIF
```

```
$     statement(s)
```

```
$ ENDIF
```

Conditional Expressions

\$ IF condition

\$ THEN

\$ statement(s)

\$ IF condition

\$ THEN

\$ statement(s)

\$ ENDIF

\$ ENDIF

Conditional Expressions

```
$ IF condition
```

```
$ THEN statement(s)
```

```
$     IF condition
```

```
$     THEN
```

```
$         statement(s)
```

```
$     ENDIF
```

```
$ ENDIF
```

This may not work in pre-V6 VMS!

Conditional Expressions

\$ IF condition

\$ THEN

\$ statement(s)

\$ ELSE

\$ statement(s)

\$ ENDIF

Labels, GOTO

```
$ GOTO label_1
```

```
.
```

```
.
```

```
.
```

```
$label_1:
```

GOSUB, RETURN

```
$ GOSUB label_1
```

```
.  
. .  
. .
```

```
$label_1:
```

```
$ statement(s)
```

```
$ RETURN
```

SUBROUTINE - ENDSUB...

```
$ CALL label_1[ param[ param[ ...]]
```

```
.  
. .  
. . .
```

```
$label_1: SUBROUTINE
```

```
$ statement(s)
```

```
$ END SUBROUTINE
```

External Procedures

Invoking other DCL procedures:

```
$ @filespec
```

where “filespec” is ddcu:[dir]filename.ext

ddcu:	default = current default device
[dir]	default = current default directory
filename	no default
.ext	default = .COM

Parameters

\$ @procedure_name p1 p2 p3 ... p8

Notes:

- Only eight(8) parameters are passed from the command line, P1 through P8
- Parameters with embedded spaces must be quoted strings.
- Parameters are separated by a space.

Parameters, Cont'd

\$ @procedure_name p1 p2 p3 ... p8

Notes, Cont'd:

- Reference parameters via the variable names P1 through P8.
- No built-in “shift” function. If you need it, write it as a GOSUB.

Parameters, Cont'd

Example: SHIFT Subroutine:

\$SHIFT:

\$ P1 = P2

\$ P2 = P3

.

.

.

\$ P7 = P8

\$ P8 :=

\$ RETURN

Parameters, Cont'd

Use the SHIFT subroutine:

```
$AGAIN:
```

```
$ IF P1 .EQS. "" THEN EXIT
```

```
.
```

```
.
```

```
.
```

```
$ GOSUB SHIFT
```

```
$ GOTO AGAIN
```

Parameters and Symbol Tables

Each procedure depth has its own local symbol table

Local symbols at lesser depth are available to procedures at greater depths

```
$ symb = value
```

```
$ @second_proc
```

```
  $ syma = symb
```

```
  $ write sys$output syma
```

```
  $ exit
```

Parameters and Symbol Tables

Accessing the calling proc.'s parameters...

```
DJAS01::DDACHTERA$ type one.com
$ @two.com 1 2 3 4 5 6 7 8
$ exit
```

```
DJAS01::DDACHTERA$ type two.com
$ set noon
$ show sym p%
$ del/sym p1
$ del/sym p2
$ del/sym p3
$ del/sym p4
$ del/sym p5
$ del/sym p6
$ del/sym p7
$ del/sym p8
$ show sym p%
$ show sym p1
$ write sys$output "P1 = """, p1, """"
$ exit
```

Parameters and Symbol Tables

Accessing the calling proc.'s parameters...

```
DJAS01::DDACHTERA$ set ver
DJAS01::DDACHTERA$ @one a b c d e f g h
$ @two.com 1 2 3 4 5 6 7 8
$ set noon
$ show sym p%
  P1 = "1"

      . . .
  P8 = "8"
$ del/sym p1

      . . .
$ del/sym p8
$ show sym p%
%DCL-W-UNDSYM, undefined symbol - check validity and spelling
$ show sym p1
  P1 = "A"
$ write sys$output "P1 = """, p1, """"
P1 = "A"
$ exit
$ exit
DJAS01::DDACHTERA$
```

Data for Programs (“Image Data”)

Sometimes, data for a program is included in the DCL procedure. This is referred to as “image data”.

Example:

```
$ CREATE MYFILE.TXT  
This is the first line  
This is the second line  
This is the third line  
$ EOD
```

\$ EOD signals end of file, just as CTRL+Z does from your keyboard.

Logical Names

Created using ASSIGN and DEFINE.

```
$ ASSIGN DUA0:[MY_DIR] MY_DIR
```

```
$ DEFINE MY_DIR DUA0:[MY_DIR]
```

Deleted using DEASSIGN.

```
$ DEASSIGN MY_DIR
```

Logical Names

Specifying a logical name table:

\$ ASSIGN/TABLE=table_name

\$ DEFINE/TABLE=table_name

Examples:

\$ DEFINE/TABLE=LNMPROCESS

\$ DEFINE/PROCESS

\$ DEFINE/TABLE=LNMJOB

\$ DEFINE/JOB

\$ DEFINE/TABLE=LNMGROUP

! These require

\$ DEFINE/GROUP

! GRPNAM privilege.

\$ DEFINE/TABLE=LNMSYSTEM

! These require

\$ DEFINE/SYSTEM

! SYSNAM privilege.

Logical Names

Search lists

```
$ DEFINE Inm string_1,string_2
```

Each item of a search list has its own “index” value:

string_1 - index 0

string_2 - index 1

Logical Names

Cluster-wide logical name table (V7.2 and later):

```
$ ASSIGN/TABLE=LNMSYSCLUSTER
```

```
$ DEFINE/TABLE=LNMSYSCLUSTER
```

Requires SYSNAM privilege.

Examples:

```
$ DEFINE/TABLE=LNMSYSCLUSTER
```

```
$ DEFINE/TABLE=LNMSYSCLUSTER_TABLE
```

DEFINE/CLUSTER and **ASSIGN/CLUSTER**
appear in V8.2.

Logical Names

Specifying a translation mode:

\$ ASSIGN/USER

\$ DEFINE/USER

Logical names are DEASSIGNed at next image run-down
Does not require privilege

\$ DEFINE/SUPERVISOR

\$ ASSIGN/SUPERVISOR

/SUPERVISOR is the default
Does not require privilege

\$ ASSIGN/EXECUTIVE

\$ DEFINE/EXECUTIVE

Requires CMEXEC privilege

There is no /KERNEL qualifier. Kernel mode logical names must be created by privileged programs (requires CMKRNL privilege).

Logical Names

Name Attributes

CONFINE

The logical name is not copied into a subprocess.

Relevant only to “private” logical name tables.

If applied to a logical name table, all logical names in that table inherit this attribute.

NO_ALIAS

The logical name cannot be duplicated in the same table in a less privileged (“outer”) access mode.

Note:

`/NAME_ATTRIBUTES` is a non-positional qualifier.

Logical Names

Translation Attributes

CONCEALED

Translation is not displayed unless specifically requested. Useful for “rooted” logical names.

TERMINAL

Translation is not performed beyond the current level.

Was used in earlier VAX machines to save machine cycles.

Note:

`/TRANSLATION_ATTRIBUTES` is a positional qualifier.

Logical Names

“Rooted” Logical Names -

Specifying Translation Attributes

```
$ DEFINE/TRANSLATION_ATTRIBUTES=-  
    (CONCEALED) Inm string
```

```
$ DEFINE Inm string/TRANSLATION_ATTRIBUTES=-  
    (CONCEALED)
```

Logical Names

“Rooted” Logical Names, Cont’d -

Example:

```
$ DEFINE/TRANS=(CONC) SRC_DIR DKA0:[SRC.]
```

```
$ DIRECTORY SRC_DIR:[000000]
```

```
$ DIRECTORY SRC_DIR:[MKISOFS]
```

```
$ DEFINE SRC_AP SRC_DIR:[AP]
```

```
$ DIRECTORY SRC_AP
```

```
$ DEFINE SRC_GL SRC_DIR:[GL]
```

```
$ DIRECTORY SRC_GL
```

Common Lexical Functions

```
$ vbl = F$TRNLNM( -  
    Inm[, table] [, index] [, mode] [, case] [,item] )
```

Examples:

```
$ HOME = F$TRNLNM( "SYS$LOGIN" )
```

```
$ TERMINAL = F$TRNLNM( "SYS$COMMAND" )
```

```
$ MY_VALUE = F$TRNLNM( "MY_LOGICAL_NAME" )
```

Common Lexical Functions

F\$TRNLNM() Items

ACCESS_MODE
CLUSTERWIDE
CONCEALED
CONFINE
CRELOG
LENGTH
MAX_INDEX
NO_ALIAS
TABLE
TABLE_NAME
TERMINAL
VALUE (default)

Common Lexical Functions

```
$ vbl = F$GETSYI( item[, nodename][, csid] )
```

Examples:

```
$ NODE = F$GETSYI( "NODENAME" )
```

```
$ FGP = F$GETSYI( "FREE_GBLPAGES" )
```

```
$ FGS = F$GETSYI( "FREE_GBLSECTS" )
```

Common Lexical Functions

\$ vbl = F\$CVTIME(string[, keyword[, keyword]])

“string” = Absolute time expression

“keyword” = (1st instance) is one of “ABSOLUTE”,
“COMPARISION”, “DELTA”

“keyword” = (2nd instance) is one of “DATE”,
“DATETIME”, “DAY”, “MONTH”, “YEAR”, “HOUR”,
“MINUTE”, “SECOND”, “HUNDREDTH”, “WEEKDAY”

Common Lexical Functions

F\$CVTIME(), Continued...

Defaults:

```
$ vbl = F$CVTIME(string, -  
"COMPARISON", -  
"DATETIME" )
```

Common Lexical Functions

F\$CVTIME(), Continued...

Date Formats:

Comparison

YYYY-MM-DD HH:MM:SS.CC

Absolute

DD-MMM-YYYY HH:MM:SS.CC

Delta

+/-DDDDDD HH:MM:SS.CC

Common Lexical Functions

```
$ vbl = F$GETDVI( dev_name, keyword )
```

“dev_name” is a valid device name

“keyword” is a quoted string

Examples:

```
$ FBLK = F$GETDVI( "DUA0", "FREEBLOCKS" )
```

```
$ MNTD = F$GETDVI( "DKA500", "MNT" )
```

```
$ DVNM := DUA0:
```

```
$ VLNM := VOLNAM
```

```
$ VNAM = F$GETDVI( DVNM, VLNM )
```

Common Lexical Functions

\$ vbl = F\$QETQUI(-
function,-
item,-
value,-
keyword(s))

See the on-line help for descriptions.
More during the Advanced section...

Common Lexical Functions

\$ VBL = F\$GETJPI(pid, keyword)

Examples:

\$ USN = F\$GETJPI(0, "USERNAME")

\$ MOD = F\$GETJPI(0, "MODE")

Recurring Batch Jobs

Jobs can reSUBMIT themselves - use the F\$GETQUI lexical function to obtain the needed information:

```
$ vbl = F$GETQUI( "DISPLAY_JOB", -  
    item,, "THIS JOB" )
```

Useful items:

QUEUE_NAME, FILE_SPECIFICATION,
AFTER_TIME, others.

Daily Batch Jobs

Daily Jobs

Get the /AFTER time:

```
$ AFTER = F$GETQUI( "DISPLAY_JOB",-  
  "AFTER_TIME",,-  
  "THIS_JOB")
```

Add one day to it:

```
$ NEXT = F$CVTIME( ""AFTER'+1-",-  
  "ABSOLUTE", )
```

Weekly Batch Jobs

Weekly Jobs

Get the /AFTER time:

```
$ AFTER = F$GETQUI( "DISPLAY_JOB",-  
  "AFTER_TIME",,-  
  "THIS_JOB")
```

Add seven days to it:

```
$ NEXT = F$CVTIME( ""AFTER'+7-",-  
  "ABSOLUTE", )
```

SUBMIT/AFTER

ReSUBMIT the job for the next run:

```
$ SUBMIT/AFTER=""NEXT"
```

or

```
$ SUBMIT/AFTER=&NEXT
```

Select Tasks by Day

Get the current day name,
look for tasks for that day.

```
$ TODAY = F$CVTIME( ,, "WEEKDAY" )  
$ PRC_NAME := [.'TODAY']WEEKLY.COM  
$ FSP = F$SEARCH( PRC_NAME )  
$ IF FSP .NES. "" THEN -  
$ @ &FSP
```

Example Path

```
[.SUNDAY]WEEKLY.COM
```

Select Monthly Tasks by Day

Get the current day number,
look for tasks for that day.

```
$ TODAY = F$CVTIME( ,, "DAY" )  
$ PRC_NAME := [.'TODAY']MONTHLY.COM  
$ FSP = F$SEARCH( PRC_NAME )  
$ IF FSP .NES. "" THEN -  
$ @ &FSP
```

Example Path

```
[.01]MONTHLY.COM
```

Select Last Day of the Month

Get the current day number +1, see if it's the first of the month.

```
$ TOMORROW = F$CVTIME( "+1-",, "DAY" )  
$ IF TOMORROW .EQ. 1 THEN -  
$ statement
```

Select Yearly Tasks by Date

Get the current day and month numbers,
look for tasks for that day.

```
$ TODAY = F$CVTIME( ,, "MONTH" ) + -  
          F$CVTIME( ,, "DAY" )      ! String values!  
$ PRC_NAME := [.'TODAY']YEARLY.COM  
$ FSP = F$SEARCH( PRC_NAME )  
$ IF FSP .NES. "" THEN -  
$ @ &FSP
```

Example Paths:

```
[.0101]YEARLY.COM  
[.0731]YEARLY.COM
```



Q / A

Speak now or forever
hold your peas.

Break Time !

We'll continue in a few minutes!

Time for a quick break!

Agenda - Intermediate

SUBROUTINE - ENDSUBROUTINE

CALL subroutine [p1[p2[...]]]

Why CALL instead of GOSUB?

More Lexical Functions

F\$SEARCH

F\$TRNLNM

F\$ENVIRONMENT

F\$PARSE

F\$ELEMENT

Agenda - Intermediate, Cont'd

F\$SEARCHing for files

File I/O

Process Permanent Files (PPFs)

File Read Loops

Using F\$ELEMENT to parse input strings

F\$ELEMENT loops

Symbol Substitution

Using Apostrophes (`'symbol'`, `'''symbol'''`)

Using Ampersand (&)

More Internal Subroutines

```
$ CALL subroutine_name[ p1[ p2[ ...]]]  
.  
.  
.  
$subroutine_name: SUBROUTINE  
$ statement(s)  
$ EXIT  
$ ENDSUBROUTINE
```

Why CALL?

The CALL statement allows parameters to be passed on the command line.

SUBROUTINES act like another procedure depth.
(Can be useful when you don't want local symbols to remain when the subroutine completes.)

Searching for Files

F\$SEARCH(string_expression)

ddcu:[dir]FILE.TXT

ddcu:[*]FILE.TXT

ddcu:[dir]*.DAT

ddcu:[dir]*.*

Use for finding files with/without wildcards.

File Search Loops

```
$ SV_FSP :=  
$LOOP_1:  
$ FSP = F$SEARCH( P1 )  
$ IF FSP .EQS. "" THEN GOTO EXIT_LOOP_1  
$ IF SV_FSP .EQS. "" THEN SV_FSP = FSP  
$ IF FSP .EQS. SV_FSP THEN GOTO EXIT_LOOP_1  
$ statement(s)  
$ SV_FSP = FSP  
$ GOTO LOOP_1  
$EXIT_LOOP_1:
```

To avoid locked loops, check that the filespec. Returned by F\$SEARCH() is not the same as the previous iteration.

Multiple F\$SEARCH Streams

To have more than one F\$SEARCH() stream, specify a context identifier.

Examples:

```
$ vb11 = F$SEARCH( SRC1, 111111 )
```

```
$ vb12 = F$SEARCH( SRC2, 121212 )
```

File I/O Statements

OPEN - Make a file available

READ - Get data from a file

WRITE - Output data to a file

CLOSE - Finish using a file

File I/O - OPEN

```
$ OPEN logical_name filespec
```

```
$ OPEN
```

```
  /ERROR=label
```

```
  /READ
```

```
  /WRITE
```

```
  /SHARE={READ | WRITE}
```

File I/O - READ

```
$ READ logical_name symbol_name
```

```
$ READ
```

```
  /DELETE
```

```
  /END_OF_FILE
```

```
  /ERROR
```

```
  /INDEX
```

```
  /KEY
```

```
  /MATCH
```

```
  /NOLOCK
```

```
  /PROMPT
```

```
  /TIME_OUT ! Terminals only
```

File I/O - WRITE

```
$ WRITE logical_name symbol_name
```

```
$ WRITE
```

```
  /ERROR
```

```
  /SYMBOL ! Use for long strings
```

```
  /UPDATE
```

File I/O - CLOSE

```
$ CLOSE logical_name
```

```
$ CLOSE
```

```
  /ERROR
```

```
  /LOG
```

Process Permanent Files (PPFs)

Four PPFs:

SYS\$INPUT (UN*X equivalent: stdin)
SYS\$OUTPUT (UN*X equivalent: stdout)
SYS\$error (UN*X equivalent: stderr)
SYS\$COMMAND (no UN*X equivalent)

Starting with OpenVMS V7.2, the PIPE command adds:

SYS\$PIPE (no UN*X equivalent)

Process Permanent Files (PPFs)

SY\$INPUT (UN*X equivalent: stdin)

Points to your terminal or the current command procedure, unless you redirect it:

```
$ DEFINE SY$INPUT filespec  
$ DEFINE/USER SY$INPUT filespec
```

Process Permanent Files (PPFs)

SY\$OUTPUT (UN*X equivalent: stdout)

When interactive, points to your terminal, unless you redirect it:

```
$ DEFINE SY$OUTPUT filespec  
$ DEFINE/USER SY$OUTPUT filespec
```

In batch, points to the job log unless you redirect it.

Process Permanent Files (PPFs)

SY\$ERROR (UN*X equivalent: stderr)

When interactive, points to your terminal, unless you redirect it:

```
$ DEFINE SY$ERROR filespec  
$ DEFINE/USER SY$ERROR filespec
```

In batch, points to the job log unless you redirect it.

Process Permanent Files (PPFs)

SY\$COMMAND (No UN*X equivalent)

When interactive, points to your terminal, unless you redirect it:

```
$ DEFINE SY$COMMAND filespec  
$ DEFINE/USER SY$COMMAND filespec
```

In batch, points to the current procedure unless you redirect it.

Process Permanent Files (PPFs)

SYS\$PIPE (UN*X equivalent: stdin)

Present only in a PIPE line.

In a DCL procedure, always point to the output of the previous PIPE-line segment, while SYS\$INPUT may point to the procedure.

More reliable than depending on SYS\$INPUT to point to the output of the previous PIPE-line segment.

File I/O - READ Loops

```
$ OPEN/READ INFLE MYFILE.DAT
$READ_LOOP:
$ READ/END=EOF_INFLE INFLE P9
$ statement(s)
$ GOTO READ_LOOP
$EOF_INFLE:
$ CLOSE INFLE
```

Hands-on Exercise

Hands-on Exercise: DIRECTORY in DCL

Elements:

F\$SEARCH()

F\$FILE_ATTRIBUTES()

F\$FAO()

WRITE SYS\$OUTPUT

Hands-on Exercise

Hands-on Exercise: DIRECTORY in DCL

Element: F\$SEARCH(fsp[, stm_id])

fsp: Wildcarded File Specification

* *.*
 ,

stm_id: not used

Hands-on Exercise

Hands-on Exercise: DIRECTORY in DCL

Elements: F\$FILE_ATTRIBUTES(fsp,attr)

fsp: Filespec returned by F\$SEARCH()

attr: We'll use these:

EOF – Count of blocks used

ALQ – Allocation quantity

CDT – File Creation Date

Hands-on Exercise

Hands-on Exercise: DIRECTORY in DCL

Elements: F\$FAO(mask, variable(s))

mask: We'll use these directives

!n< and !> - Enclose an expression

!nAS – ASCII string

!nSL – Signed Longword

n is an optional field size

Hands-on Exercise

Hands-on Exercise: DIRECTORY in DCL

Elements: WRITE SYS\$OUTPUT exp[,...]

We'll use:

```
WRITE SYS$OUTPUT F$FAO( ... )
```

Hands-on Exercise

Hands-on Exercise: DIRECTORY in DCL

1. Build a loop using F\$SEARCH() to find all the files in the current directory.
2. For each file, use F\$FILE() to get the EOF size, the ALQ size and the creation date (CDT), each into a separate variable, and use WRITE SYS\$OUTPUT and F\$FAO() to display the file specification and the file information.
3. Exit the loop when no more files are found.

Parse - F\$ELEMENT()

```
$ vbl = F$ELEMENT( index, delim, string )  
    index - an integer value  
    delim - the delimiter character  
    string - the string to parse
```

F\$ELEMENT() returns the delimiter character when no more elements exist.

Example:

```
$ ELEM = F$ELEMENT( 0, ",", P1 )
```

String Parsing Loops

```
$ CNTR = 0
$LOOP_1:
$ ELEM = F$ELEM( CNTR, "\", " P1 )
$ CNTR = CNTR + 1
$ IF ELEM .EQS. "" THEN GOTO LOOP_1
$ IF ELEM .EQS. "\", " THEN GOTO EXIT_LOOP_1
$ statement(s)
$ GOTO LOOP_1
$EXIT_LOOP_1:
```

Symbol Substitution

Two forms of symbol substitution:

`&symbol_name`

`'symbol_name'` or `''symbol_name''`

“Apostrophe” Substitution

Use the apostrophe to replace part of a string (or command)

```
$ CMD := DIRECTORY/SIZE=ALL/DATE
```

```
$ 'CMD'
```

```
$ CMD := DIRECTORY/SIZE=ALL/DATE
```

```
$ WRITE SYS$OUTPUT "$ ' 'CMD' "
```

Ampersand Substitution

Use the ampersand to insert an entire string as a single entity.

```
$ ME = "David J Dachtera"  
$ DEFINE MY_NAME &ME  
$ SHOW LOGICAL MY_NAME  
    "MY_NAME" = "David J Dachtera"  
(LNM$PROCESS_TABLE)
```

Note that case and formatting (spaces) are preserved.

Symbol Substitution

Order of Substitution:

1. Apostrophe(')
2. Ampersand(&)

Symbol Substitution

Command line length limits:

Before symbol substitution: 255 bytes

4095 bytes (V7.3-2 and later)

After symbol substitution: 1024 bytes

8192 bytes (V7.3-2 and later)

(Thanx to Alex Daniels for pointing out V7.3-2 new features.)

Symbol Substitution - Examples

Apostrophe Substitution

```
$ type apost.com
$ text = "This is an example of apostrophe
substitution"
$ write sys$output "'text'"
```

```
$ set verify
$ @apost.com
$ text = "This is an example of apostrophe
substitution"
$ write sys$output "This is an example of
apostrophe substitution"
This is an example of apostrophe substitution
```

Symbol Substitution - Examples

Ampersand Substitution

```
$ type ampers.com
$ text = "This is an example of ampersand
substitution"
$ define lnm &text
$ show logical lnm
```

```
$ @ampers.com
$ text = "This is an example of ampersand
substitution"
$ define lnm &text
$ show logical lnm
  "LNM" = "This is an example of ampersand
substitution" (LNM$PROCESS_TABLE)
```

Symbol Substitution - Examples

Ampersand Substitution

In LOGIN.COM:

```
$ HELP_PAGES == 64  
$ HE*LP ::= HELP/PAGE=SAVE=&HELP_PAGES
```

HELP will then use /PAGE=SAVE, and up to 64 pages will be retained to allow you to page up using the Prev key to view previously displayed information, even on a real VT terminal.

Note that the `HELP` symbol will still contain the “`&HELP_PAGES`” notation, and the “`HELP_PAGES`” symbol will be evaluated at “run-time”. The value of the `HELP_PAGES` symbol can be changed anytime.

Symbol “Scope”

Determine symbol scope for the current procedure depth:

```
$ SET SYMBOL/SCOPE=(keyword(s))  
  - [NO]LOCAL  
  - [NO]GLOBAL
```

Can help prevent problems due to symbols defined locally at another procedure depth or globally.

Symbol “Scope”

\$ SET SYMBOL/SCOPE=(keyword(s))

Other Qualifiers: /ALL, /VERB and /GENERAL

/VERB applies only to the first “token” on a command line.

/GENERAL applies to all other “tokens” on a command line.

/ALL applies to both.

String Operations

Concatenation:

```
$ vb1 = string1 + string2
```

Example:

```
$ PROC = F$ENVIRONMENT( "PROCEDURE" )  
$ DEVC = F$PARSE( PROC,,, "DEVICE" )  
$ DRCT = F$PARSE( PROC,,, "DIRECTORY" )  
$ FLOC = DEVC + DRCT
```

String Operations

Reduction:

```
$ vbl = string - substring
```

Example:

```
$ DVN = F$GETDVI( "SYS$DISK", "ALLDEVNAM" )  
$ DNM = DVN - "_" - ":"  
$ SHOW SYMBOL DNM  
DNM = "DJVS01$DKA0"
```

String Operations

Substring Replacement:

```
$ vbl[start,length] = string
```

Example:

```
$ nam := hydrichlor  
$ nam[4,1] := o  
$ show symbol nam  
NAM = "HYDROCHLOR"
```

String Operations

Binary Assignment:

```
$ vbl[start_bit,bit_count] = integer
```

Examples:

```
$ CR[0,8] = 13  
$ LF[0,8] = 10  
$ ESC[0,8]= 27  
$ CSI[0,8]= 155
```

DCL “Arrays”

DCL does not support arrays in the usual sense (subscripted variables). However, you can use a counter within a loop to create a list of variables:

```
$ CNTR = 0
$LOOP:
$ CNTR = CNTR + 1
$ FIELD_'CNTR' = vbl
$ IF CNTR .LE. 12 THEN -
$ GOTO LOOP
```

Integer Operations

DCL supports the four basic arithmetic operations:

+ Add

- Subtract

* Multiply

/ Divide

Boolean Operations

DCL supports assignment of boolean values:

```
$ vbl = (condition)
```

Examples:

```
$ TRUE = (1 .EQ. 1)
```

```
$ FALSE = (1 .EQ. 0)
```

Logical Operations

DCL supports logical AND, OR and NOT

```
$ vbl = (int1 .AND. int2)
```

```
$ vbl = (int1 .OR. int2)
```

```
$ vbl = (.NOT. int3)
```

Examples:

```
$ STATUS = &$STATUS
```

```
$ SEVERITY = (STATUS .AND. 7)
```

```
$ FAILURE = (.NOT. (SEVERITY .AND. 1))
```

```
$ EXIT_STATUS = (STATUS .OR. %X10000000)
```

“Result Codes”

DCL commands and VMS programs return their completion status by way of two “permanent” symbols:

\$STATUS

32-bit status of the most recent command

Bit 28 (%X10000000) is the “quiet” bit

\$SEVERITY

three lowest-order bits of \$STATUS

$\$SEVERITY = (\$STATUS .AND. 7)$

“Result Codes”

Examples:

```
$ EXIT %X10000012 ! "Quiet" bit set  
    No Message is displayed
```

```
$ EXIT %X12  
%SYSTEM-E-BADPARAM, bad parameter value
```

```
$ show symbol $S*  
$SEVERITY == "2"  
$STATUS == "%X00000012"
```

Note that these are STRING values!

“Result Codes”

Some DCL commands do not change the values of \$STATUS and \$SEVERITY:

\$ CONTINUE
\$ IF
\$ THEN
\$ ELSE
\$ ENDIF

\$ EOD
\$ GOTO
\$ RECALL
\$ SHOW SYMBOL
\$ STOP[/IDENT]
\$ WAIT

Others...

Error Trapping

Using the “ON” statement, you can set the level of error trapping:

```
$ ON WARNING THEN statement
```

```
$ ON ERROR THEN statement
```

```
$ ON SEVERE_ERROR THEN statement
```

```
$ ON CONTROL_Y THEN statement
```

The most recent “ON” statement determines the action taken.

Error Trapping

“ON” statement order of precedence:

```
$ ON WARNING THEN statement  
WARNING or greater
```

```
$ ON ERROR THEN statement  
Error or greater (WARNING action becomes CONTINUE)
```

```
$ ON SEVERE_ERROR THEN statement  
Severe_Error or greater (WARNING and ERROR actions  
become CONTINUE)
```

Error Trapping

Turn error trapping off or on:

\$ SET NOON

No "ON" action is taken.

\$ SET ON

The current "ON" action is taken in response to an event.

Handling Errors

```
$ SET NOON
$ statement
$ STATUS = &$STATUS
$ SEVERITY = (STATUS .AND. 7)
$ IF SEVERITY .EQ. 0 THEN -
$ GOSUB ANNOUNCE_WARNING
$ IF SEVERITY .EQ. 2 THEN -
$ GOSUB ANNOUNCE_ERROR
$ IF SEVERITY .EQ. 4 THEN -
$ GOSUB ANNOUNCE_FATALERROR
```

Lexical - F\$TRNLNM

Use to translate logical names.

```
$ vbl = F$TRNLNM( -  
    logical_name,-  
    table_name,-  
    index,-  
    mode,-  
    case,-  
    item )
```

Does NOT support wildcard look-ups!

Lexical - F\$ENVIRONMENT

Get information about the process environment.

```
$ vbl = F$ENVIRONMENT( keyword )
```

Some useful keywords:

CAPTIVE	“TRUE” or “FALSE”
DEFAULT	Current default ddcu:[dir]
MESSAGE	Qualifier string
PROCEDURE	Fully qualified filespec.
Others...	

Lexical - F\$ENVIRONMENT

A useful example:

```
$ SET NOON
$ DFLT = F$ENVIRONMENT( "DEFAULT" )
$ MSG = F$ENVIRONMENT( "MESSAGE" )
$ SET DEFAULT ddcu:[dir]
$ SET MESSAGE/NOFACI/NOSEVE/NOIDE/NOTEXT
$ statement(s)
$ SET MESSAGE'MSG'
$ SET DEFAULT &DFLT
```

Lexical - F\$PARSE

Use to verify or extract portions of a file specification.

```
$ vbl = F$PARSE( -  
    filespec,-  
    default_spec,-  
    related_spec,-  
    field,-  
    parse_type)
```

Lexical - F\$PARSE

A useful example:

```
$ SHOW DEFAULT
```

```
MYDISK:[MYDIR]
```

```
$ DFSP = F$ENVIRONMENT( "DEFAULT" ) + ".COM"
```

```
$ FSP = F$PARSE( "LOGIN", DFSP )
```

```
$ SHOW SYMBOL FSP
```

```
"FSP" = "MYDISK:[MYDIR]LOGIN.COM;"
```

Lexical - F\$PARSE

Another useful example:

```
$ PROC = F$ENVIRONMENT( "PROCEDURE" )
$ DEVC = F$PARSE( PROC,,, "DEVICE" )
$ DRCT = F$PARSE( PROC,,, "DIRECTORY" )
$ DFLT = F$ENVIRONMENT( "DEFAULT" )
$ FLOC = DEVC + DRCT
$ SET DEFAULT &FLOC
$ statement(s)
$ SET DEFAULT &DFLT
```

Lexical - F\$PARSE

Getting the parent directory:

```
$ FLSP := ddcu:[dir]
$ DEVC = F$PARSE( FLSP,,, "DEVICE" )
$ DRCT = F$PARSE( FLSP,,, "DIRECTORY" )
$ DRCT = DRCT - "]"[" - "]" + ".-]"
$ PRNT = F$PARSE( DEVC + DRCT ) - ".;"
```

Lexical - F\$PARSE

Verify that a path exists:

```
$ FLSP := ddcu:[dir]
$ FLOC = F$PARSE( FLSP )
$ IF FLOC .EQS. "" THEN -
$ WRITE SYS$OUTPUT "% Path not found"
```

Lexical - F\$GETQUI

Get information about queues and jobs on them.

```
$ vbl = F$GETQUI( -  
    function,-  
    item,-  
    object_identifier,-  
    flags )
```

Can be complicated, is definitely useful.

Lexical - F\$GETQUI

Functions:

CANCEL_OPERATION

DISPLAY_ENTRY

DISPLAY_FILE

DISPLAY_FORM

DISPLAY_JOB

DISPLAY_MANAGER

DISPLAY_QUEUE

TRANSLATE_QUEUE

Lexical - F\$GETQUI

Some useful items:

AFTER_TIME

FILE_SPECIFICATION

ENTRY_NUMBER

JOB_NAME

QUEUE_NAME

QUEUE_PAUSED

QUEUE_STOPPED

There's LOTS more item codes!

Lexical - F\$GETQUI

Typical usage:

1. Use DISPLAY_QUEUE to establish a queue context (object="*")
2. Use DISPLAY_JOB to display jobs on the queue (object="*").
3. Loop back to #2 until no more jobs.
4. Loop back to #1 until a null queue name is returned.

Lexical - F\$GETQUI

To retrieve multiple items about a queue or a job, use the FREEZE_CONTEXT flag on all but the last F\$GETQUI for that item.

Example:

```
$ QN = F$GETQUI(
  "DISPLAY_QUEUE", "QUEUE_NAME", -
  "*", "FREEZE_CONTEXT" )
$ NN = F$GETQUI( "DISPLAY_QUEUE", -
  "SCSNODE_NAME", "*" , )
```

Lexical - F\$GETQUI

Symbols can be useful for shortening statements using F\$GETQUI:

Example:

```
$ DSPQ := DISPLAY_QUEUE
```

```
$ QUNM := QUEUE_NAME
```

```
$ FZCT := FREEZE_CONTEXT
```

```
$ SCNN := SCSNODE_NAME
```

```
$ QN = F$GETQUI( DSPQ, QUNM, "*" , FZCT )
```

```
$ NN = F$GETQUI( DSPQ, SCNN, "*" , )
```

F\$GETQUI - Loop

F\$GETQUI () Loop Example:

```
$ DSPQ := DISPLAY_QUEUE
$ DSPJ := DISPLAY_JOB
$ QUNM := QUEUE_NAME
$ JBNM := JOB_NAME
$ FZCT := FREEZE_CONTEXT
$ ALJB := ALL_JOBS
$ JNXS := "JOB_INACCESSIBLE"
$ SAY := WRITE SYS$OUTPUT
$! Continued on the next slide...
```

F\$GETQUI - Loop

```
$ TEMP = F$GETQUI( "" )
$QLOOP:
$ QNAM = F$GETQUI( DSPQ, QUNM, "*" )
$ IF QNAM .EQS. "" THEN EXIT
$ SAY ""
$ SAY "QUEUE: ", QNAM
$JLOOP:
$ NOXS = F$GETQUI( DSPJ, JNXS, , ALJB )
$ IF NOXS .EQS. "TRUE" THEN GOTO JLOOP
$ IF NOXS .EQS. "" THEN GOTO QLOOP
$ JNAM = F$GETQUI( DSPJ, JBNM, , FZCT )
$ SAY "      JOB: ", JNAM
$ GOTO JLOOP
```

F\$GETQUI - Caveat

VMS provides only ONE queue context per process.

Using SHOW QUEUE in between F\$GETQUI() invocations will destroy the current queue context.

Lexical - F\$CVTIME

Most useful for adding and subtracting days, hours, minutes and/or seconds to/from a date.

Examples:

```
$ NEXT_WEEK = F$CVTIME( "+7-", "ABSOLUTE", )
$ MONTH_END = ( F$CVTIME( "+1-", , "DAY" ) .EQ. 1 )
$ YEAR_END   = ( MONTH_END .AND. -
                  ( F$CVTIME( "+1-", , "MONTH" ) .EQ. 1 ) )
$ NOW = F$CVTIME( , , "TIME" )
$ LATER = F$CVTIME( , , "TIME" )
$ ELAPSED_TIME = -
  F$CVTIME( "'LATER'-'NOW', , "TIME" )
```

Lexical - F\$DELTA_TIME

New for OpenVMS V7.3-2 and later.

Useful for subtracting dates in absolute format.

Returns the difference between the two values as a delta-time expression:

Examples:

```
$ STRT := 15-JUL-2003 16:26:35.77
$ ENDT := 15-JUL-2003 16:26:41.39
$ WRITE SYS$OUTPUT F$DELTA_TIME( STRT, ENDT )
0 00:00:05.62
```

Lexical - F\$EXTRACT

Use to extract substrings.

```
$ vbl = F$EXTRACT( -  
    offset,-      ! Zero relative!  
    length,-  
    string )
```

Note:

The offset is “zero-relative”; i.e., starts at zero(0).

Lexical - F\$GETDVI

Use to get information about devices.

```
$ vbl = F$GETDVI( "ddcu:", item )
```

Some useful items:

ALLDEVNAM

FREEBLOCKS

LOGVOLNAM

MAXBLOCK

TT_ACCPORNAM

Many others...

Lexical - F\$EDIT

Use to modify strings.

```
$ vbl = F$EDIT( string, keyword(s) )
```

Keywords:

COLLAPSE

COMPRESS

LOWERCASE

TRIM

UNCOMMENT

UPCASE

Lexical - F\$GETJPI

Use to get information about your process or process tree.

```
$ vbl = F$GETJPI( pid, item )
```

To get info. about the current process, specify PID as null (“”) or zero(0).

Example:

```
$ MODE = F$GETJPI( 0, "MODE" )
```

Lexical - F\$GETJPI

Some useful items:

IMAGNAME

MASTER_PID

MODE

PID

PRCNAM

USERNAME

WSSIZE

Lexical - F\$GETJPI

A note of caution:

The AUTHPRIV and CURPRIV items can return strings which are too long to manipulate in V7.3-1 and earlier.

Lexical - F\$GETSYI

Use to get information about the system.

```
$ vbl = F$GETSYI( item[,nodename][,cluster_id] )
```

Can be used to retrieve the value of any system parameter, as well as values associated with some other keywords (see HELP or the DCL Dictionary).

Some useful items:

CLUSTER_MEMBER

HW_NAME

CLUSTER_FTIME

NODENAME

CLUSTER_NODES

F\$CONTEXT and F\$PID

Use to locate selected processes.

Use F\$CONTEXT to set up selection criteria.

Use F\$PID to locate selected processes.

F\$CONTEXT and F\$PID

Use F\$CONTEXT to set up process selection criteria.

```
$ TMP = F$CONTEXT( "PROCESS", -  
                  CTX, "MODE", "INTERACTIVE", "EQL" )  
$ TMP = F$CONTEXT( "PROCESS", -  
                  CTX, "NODENAME", "*", "EQL" )
```

Selection criteria are cumulative.

F\$CONTEXT and F\$PID

Use F\$PID to locate selected processes using the context symbol set up by F\$CONTEXT():

```
$LOOP:
```

```
$ PID = F$PID( CTX )
```

```
$ IF PID .EQS. "" THEN GOTO EXIT_LOOP
```

```
$ statement(s)
```

```
$ GOTO LOOP
```

```
$EXIT_LOOP:
```

```
$ IF F$TYPE( CTX ) .EQS. "PROCESS_CONTEXT" THEN -
```

```
$ TMP = F$CONTEXT( "PROCESS", CTX, "CANCEL" )
```

Other Lexical Functions

Lexicals

A set of functions that return information about character strings and attributes of the current process.

Additional information available:

F\$CONTEXT	F\$CSID	F\$CVSI	F\$CVTIME	F\$CVUI	F\$DEVICE	
F\$DIRECTORY		F\$EDIT	F\$ELEMENT	F\$ENVIRONMENT		
F\$EXTRACT						
F\$FAO	F\$FILE_ATTRIBUTES		F\$GETDVI	F\$GETJPI	F\$GETQUI	F\$GETSYI
F\$IDENTIFIER		F\$INTEGER	F\$LENGTH	F\$LOCATE	F\$MESSAGE	F\$MODE
F\$PARSE	F\$PID	F\$PRIVILEGE		F\$PROCESS	F\$SEARCH	F\$SETPRV
F\$STRING	F\$TIME	F\$TRNLNM	F\$TYPE	F\$USER	F\$VERIFY	

Q & A

Speak now or forever hold your peas.

Break Time !

We'll continue in a few minutes!

Time for a quick break!

Agenda - Advanced

Logical Name Table Search Order

Tips, tricks and kinks

F\$TYPE()

Tips, tricks and kinks

F\$PARSE()

Tips, tricks and kinks

Loops using F\$CONTEXT(), F\$PID()

Select processes by name, node, etc.

Agenda - Advanced, Cont'd

Using F\$CSID() and F\$GETSYI()

Get/display info. about cluster nodes

The PIPE command

Usage Information

Techniques

Reading SYS\$PIPE in a loop

Getting command output into a symbol

Symbol substitution in “image data”

Logical Name Table Search Order

```
$ show logical/table=lnm$system_directory
```

```
(LNM$SYSTEM_DIRECTORY) [kernel]
[shareable,directory]
[Protection=(RWC,RWC,R,R)]
[Owner=[SYSTEM]]
```

```
.
.
.
```

```
"LNM$FILE_DEV" [super] = "LNM$PROCESS"
    = "LNM$JOB"
    = "LNM$GROUP"
    = "LNM$SYSTEM"
    = "DECW$LOGICAL_NAMES"
```

```
"LNM$FILE_DEV" [exec] = "LNM$PROCESS"
    = "LNM$JOB"
    = "LNM$GROUP"
    = "LNM$SYSTEM"
    = "DECW$LOGICAL_NAMES"
```

Logical Name Table Search Order

Modifying the search order:

If no CMEXEC privilege:

```
$ DEFINE/TABLE=LNMPROCESS_DIRECTORY LNMPFILE_DEV -  
LNMPROCESS , LNMPJOB , LNMPGROUP , LNM_APPL , LNMPSYSTEM
```

With CMEXEC privilege:

```
$ DEFINE/TABLE=LNMPROCESS_DIRECTORY/EXEC -  
LNMPFILE_DEV -  
LNMPROCESS , LNMPJOB , LNMPGROUP , LNM_APPL , LNMPSYSTEM
```

Logical Name Table Search Order

Lexical Functions Caveat:

F\$TRNLNM() uses the LNM\$FILE_DEV search list.

F\$TRNLNM() should be used for all new development.

F\$LOGICAL() uses a hard-coded search list.

F\$LOGICAL() is deprecated.

Lexical Function - F\$TYPE()

Used to get the datatype of a symbol's contents or the symbol type.

```
$ vbl = F$TYPE( symbol_name )
```

Returns:

“STRING”

Symbol contains an ASCII character string (non-numeric)

“INTEGER”

Symbol contains a numeric value (string or longword)

“PROCESS_CONTEXT”

Symbol was established by F\$CONTEXT()

“” (null string)

Symbol was not found in the environment

Lexical Function - F\$TYPE() Examples

```
$ a1 = "1234"  
$ a2 = "01B7"  
$ a3 = 1234  
$ a4 = %x01b7  
$ show symbol a%  
  A1 = "1234"  
  A2 = "01B7"  
  A3 = 1234      Hex = 000004D2      Octal = 00000002322  
  A4 = 439      Hex = 000001B7      Octal = 00000000667  
$ say := write sys$output  
$ say f$type( a1 )  
INTEGER  
$ say f$type( a2 )  
STRING  
$ say f$type( a3 )  
INTEGER  
$ say f$type( a4 )  
INTEGER
```

F\$CONTEXT(), F\$PID() Loops

Sample loop to find processes:

```
$ ctx :=
$ IF F$EXTR( 0, 1, P1 ) .EQS. "%"
$ THEN
$ PID = F$FAO( "!XL", &P1 )
$ ELSE
$ if p1 .eqs. "" then p1 := *
$ TMP = F$CONTEXT( "PROCESS", CTX, "PRCNAM", P1, "EQL" )
$ if p2 .eqs. "" then p2 = f$getsyi( "nodename" )
$ TMP = F$CONTEXT( "PROCESS", CTX, "NODENAME", P2, "EQL" )
$ PID = F$PID( CTX )
$ IF PID .EQS. "" THEN EXIT %X8EA !Process not found
$ ENDIF
$RELOOP:
$ GOSUB OUTPUT_PSTAT
$ IF F$TYPE( CTX ) .NES. "PROCESS_CONTEXT" THEN GOTO EXIT
$ PID = F$PID( CTX )
$ IF PID .NES. "" THEN -
$ GOTO RELOOP
$EXIT:
```

Using F\$CSID() and F\$GETSYI()

Sample Loop to get cluster node info:

```
$ CONTEXT = ""
$START:
$ id = F$CSID( CONTEXT )
$ IF id .EQS. "" THEN EXIT
$ nodename = F$GETSYI ("NODENAME",,id)
$ hdwe_name = F$GETSYI("HW_NAME",,id)
$ arch_name = F$GETSYI("ARCH_NAME",,id)
$! ARCH_NAME with ID works on V7.2 and later only.
$ soft_type = F$GETSYI("NODE_SWTYPE",,id)
$ soft_vers = F$GETSYI("NODE_SWVERS",,id)
$ syst_idnt = F$GETSYI("NODE_SYSTEMID",,id)
$ gosub op_node_info
$ GOTO START
```

PIPE

PIPE command

Introduced in OpenVMS V7.2

Early PIPE has issues:

- mailbox quotas
- synchronization.

O.k. in V7.3 and later

PIPE

PIPE command

Pipeline syntax is essentially the same as UN*X,
except for the PIPE verb:

```
$ PIPE/qualifier(s) command | command  
...
```

PIPE

```
$ PIPE/qualifier(s) command | command
```

...

- SYS\$OUTPUT of one command becomes SYS\$INPUT of the next, except when invoking a DCL proc.
- Sometimes need to distinguish between SYS\$INPUT and SYS\$PIPE
- SYS\$COMMAND is not changed

PIPE

PIPE command

- Not all programs and commands expect input from SYS\$INPUT (COPY, SEARCH, PRINT, etc.)

PIPE

PIPE command

- Redirection is available for SYS\$INPUT, SYS\$OUTPUT and SYS\$ERROR
- Can only use output redirection for the last element in a pipeline
- No append operator (“>>”) for output redirection. Use “APPEND SYS\$PIPE output_file” as the last element of the pipeline.

PIPE

PIPE command

Multiple commands can be specified in a subshell:

```
$ PIPE -
```

```
(command ; command ; ...) | -  
command
```

- Semi-colon (“;”) must be surrounded by white space.

PIPE

PIPE command

Conditional operators:

`&&`

Continue if previous command returns a success status

`||`

Continue if previous command returns a failure status

PIPE

PIPE command

Other operators:

&

Commands up to "&" execute asynchronously in a subprocess. Equivalent to SPAWN/NOWAIT.

PIPE

PIPE command

TEE?

The HELP for PIPE includes an example of a TEE.COM that can be used at the user's discretion.

```
$ PIPE -  
command | -  
@TEE filespec | -  
command ...
```

PIPE

```
$ ! TEE.COM -      command procedure to
$ !                display/log data flowing
$ !                through a pipeline
$ ! Usage: @TEE log-file
$ !
$ OPEN/WRITE  tee_file 'P1'
$ LOOP:
$   READ/END_OF_FILE=EXIT  SYS$PIPE LINE
$   WRITE SYS$OUTPUT LINE
$   WRITE tee_file LINE
$   GOTO LOOP
$ EXIT:
$   CLOSE tee_file
$   EXIT
```

PIPE

```
$! WYE.COM -          command procedure to display
$!                  and log data flowing through
$!                  a pipeline
$!   Usage: @WYE log-file
$!
$ OPEN/WRITE  tee_file 'P1'
$ LOOP:
$  READ/END_OF_FILE=EXIT  SYS$PIPE LINE
$  WRITE SYS$OUTPUT LINE
$  WRITE tee_file LINE
$  WRITE SYS$COMMAND LINE
$  GOTO LOOP
$ EXIT:
$  CLOSE tee_file
$  EXIT
```

Write two copies of the pipeline data:

```
$ PIPE -
    command | -
    (OPEN/WRITE SYS$COMMAND filespec ; -
    @WYE filespec ; CLOSE SYS$COMMAND) | -
    command
```

PIPE

PIPE command

Sub-process performance

Can be improved by using these qualifiers:

`/NOLOGICAL_NAMES`

`/NOSYMBOLS`

PIPE

PIPE command

Restrictions:

PIPEs cannot be nested. However, if a .COM file appears as a PIPE element, it can invoke another PIPE.

Don't use "<" or ">" in path specifications. Use "[" and "]" instead.

No "append output" operator (">>")

No "2>&1"

SYS\$OUTPUT and SYS\$ERROR are the same unless SYS\$ERROR is redirected ("2>").

Complex PIPEs may require an increase in the user's PRCLM quota (subprocess limit).

PIPE

PIPE command

Some commands make no sense in PIPEs:

SUBROUTINE - ENDSUBROUTINE

EXIT, RETURN

GOTO or GOSUB

IF-THEN[-ELSE]-ENDIF

PIPE

PIPE command

Image verification is “off” by default
(SET VERIFY=IMAGE ; command)

Hands-on Exercise

Incorporate PIPE into the DIRECTORY in DCL exercise.

New Elements:

PIPE

READ, /END_OF_FILE

F\$PARSE()

Hands-on Exercise

Incorporate PIPE into the DIRECTORY in DCL exercise.

New Element: PIPE cmd | cmd

Hands-on Exercise

Incorporate PIPE into the DIRECTORY in DCL exercise.

New Element: READ/END=|bl Inm sym

Hands-on Exercise

Incorporate PIPE into the DIRECTORY in DCL exercise.

New Element: F\$PARSE(fsp,,, item)

fsp – Filespec read from SYS\$PIPE

item – We'll use these:

NAME

TYPE

VERSION

Hands-on Exercise

Incorporate PIPE into the DIRECTORY in DCL exercise.

Modify the code from the previous exercise to READ the filespec from SYS\$PIPE instead of using F\$SEARCH(). EXIT the proc. at end of file.

Use F\$PARSE() to get the name, extension and version into separate variables. WRITE that out along with the file information.

Hands-on Exercise

Incorporate PIPE into the DIRECTORY in DCL exercise.

3. Write a “shell” procedure to use PIPE:

Use DIRECTORY/NOHEAD/NOTRAIL to generate a list of files to SYS\$OUTPUT.

Invoke the exercise proc. to read file specifications from the pipeline and display the attributes for each file.

Extra Bonus:

Let the procedure detect whether it is being used in a pipeline or not.

Detect the Presence of SYS\$PIPE:

DCL Procedures can change their behavior when used in a PIPEline:

MORE.COM:

```
$ ipt := sys$input
$ if f$trnlrm( "sys$pipe" ) .nes. "" then -
$ ipt := sys$pipe
$ if p1 .eqs. "" then p1 = ipt
$ if f$type( more_pages ) .eqs. "" then -
$ more_pages = 64
$ type/page=save=&more_pages &p1
$ exit
```

PIPE: Command Output -> Symbol

Get the current count of interactive logins into a symbol.

```
$ PIPE -  
    SET LOGINS | -  
    (READ SYS$PIPE P9 ; DEF/JOB P9 &P9)  
$ P9 = F$TRNLNM( "P9" )  
$ LOGINS = F$INTEGER( F$ELEM( 2, "=", P9 ) )
```

PIPE: Symbols in Image Data

Use symbols to provide input to a program:

```
$ USNM = F$GETJPI( 0, "USERNAME" )
$ PIPE -
(WRITE SYS$OUTPUT "SET ENV/CLU" ; -
WRITE SYS$OUTPUT "DO SHOW USER/FULL ", USNM) | -
RUN SYS$SYSTEM:SYSMAN
```

PIPE: File List for ZIP from DIRECTORY

Use the DIRECTORY command to selectively provide a list of files to be archived using ZIP for OpenVMS:

```
$ PIPE -  
DIRECTORY/NOHEAD/NOTRAIL/MODIFIED/BEFORE=date | -  
ZIP/LEVEL=8/VMS archive_name/BATCH=SYS$PIPE:
```

Q & A

Speak now or forever hold your peas.



Thank You!

Congratulations!

You survived!





Thank You!

Remember to fill out the evaluation forms!



HP Technology Forum 2006

Thank you!



GET CONNECTED
People. Training. Technology.