DCL Programming
Hands-on Session

David J Dachtera
djesys@earthlink.net
DJE Systems - http://www.djesys.com/

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/dclprog.ppt

When published with the Symposium Session notes, this presentation might be converted to .PDF in the slide view only. Go to the URL shown to get the final PowerPoint presentation, then view it the way that works best for you.

Agenda - Introduction

Basic DCL Concepts
- Commands
- Verbs
- Symbols

IF-THEN

IF-THEN-ENDIF

IF-THEN-ELSE-ENDIF

Labels, GOTO

In this presentation, we'll begin by going over some DCL basics: commands, verbs and symbols.

We'll look at conditional statements and conditional statement blocks.

We'll look at logical control and how to pass control from one section of code to another.

Agenda - Introduction, Cont'd

- GOSUB-RETURN
- SUBROUTINE ENDSUBROUTINE
- Common Lexical Functions
  - F$CVTIME
  - F$GETDVI
  - F$GETJPI
  - F$GETQUI
  - F$GETSYI
- PARAMETERS
- Logical Names

Then, we'll look at some of the more modular structures, including internal subroutines, using the GOSUB and RETURN statements.

We'll look at some functions built into DCL that allow you to manipulate dates and times, get information about devices, processes, batch and print queues and even from the system itself.

We'll look at passing parameters to DCL procedures, getting information from logical names and symbols, ...

Agenda - Introduction, Cont'd

Batch jobs that reSUBMIT themselves

Daily Jobs

Weekly Jobs

Question & Answer

- Break -

We'll use what we've learned to build batch jobs which resubmit themselves, and change their behavior based on the day of the week, month or year.

We'll also have question and answer sessions to help everyone understand what we have covered as we go along.

We'll have a short break before continuing into the intermediate material.

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

We'll then get deeper into some of the more advanced functions like complex internal subroutines, some of the more useful lexical functions, ...

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 (&)

...processing wildcarded file specifications, reading and writing disk and process-permanent files, parsing strings and parameters, and using symbol substitution.

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.

In the advanced section, we'll discuss Logical Name Table Search order and how to change it, using ampersand's (&) special characteristics to your advantage – assign the entire content of a string to a logical name, preserving case and spacing.

We'll talk about F$TYPE() and some good ways to use it.

We'll talk about using F$PARSE() to validate and navigate paths.

We'll talk some more about using F$CONTEXT() and F$PID().

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"

We'll see how to use F$CSID() and F$GETSYI() to return information about cluster nodes, with a practical example.

We'll take a look at the PIPE command, and examine in detail its flexibility and intricacies. We'll see how to use PIPE to get the output of a command into a symbol, and explore other PIPE "magic".

DCL - Programming?

DCL as a programming language?

Certainly!

DCL has many powerful features!

Never thought of DCL as a programming language?

Well, that's even what Digital once said - but not any more!

DCL has many powerful features that can help automate many operations that you or your operators may be performing manually.

DCL Command Elements

$ verb parameter_1 parameter_2

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

To begin understanding DCL, first let's review a few basic concepts.

DCL commands always begin with a dollar sign ("$").

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

Some commands can be as simple as:
$ SET VERIFY
or
$ LOGOUT
or
$ EXIT

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...

Commands in DCL are either internal to DCL or are executed by programs which are external to DCL.

Here we see some examples of both internal and external commands.

Notice that SET, STOP and other commands can be either internal or external depending upon the keyword after the verb.

DCL Verbs, Cont'd

"Foreign" Commands

$ symbol = value

Examples:

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

$ ZIP :== $ZIP/VMS

Commands can be added or customized using symbols or "foreign commands".

In the slide, the DIR symbol redefines the behavior of the DIRECTORY command, while the ZIP symbol provides a means to invoke the ZIP program in such a manner that it can accept parameters and qualifiers from the command line.

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]
```

Another way to find “foreign” or external commands is to use the DCL$PATH logical name. DCL$PATH was introduced in OpenVMS V6.2.

DCL$PATH behaves very much like the DOS or UN*X “path” - .COM and .EXE files can be located via the DCL$PATH path.

DCL$PATH can have a single translation or it can be a search list.

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
       .
       .
       .
```

There is an aspect of DCL$PATH of which you need to be aware:

DCL will observe wildcard specifications when seeking a file by way of DCL$PATH. This can produce undesired results.

Even an invalid specification might be interpreted as a wildcard specification. The second example shows what can happen if a space is left out of a DIRECTORY command.

DCL$PATH Caveat

Determine what might be found via a wildcard specification:

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

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

You can determine in advance what might be found via a wildcard specification. Just issue a DIRECTORY command for .COM files and another for .EXE files.

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

Download, RENAME to .COM and invoke it.

$ @make_$.com

To avoid problems, you can place a “$.EXE” program or a “$. COM” procedure in the DCL$PATH path. They should both simply exit without doing anything.

A DCL procedure to create these can be downloaded from the internet at the URL shown in the slide.

Command Qualifiers

```
$ command/qualifier

$ command/qualifier=value
$ command/qualifier=(value,value)

$ command/qualifier=keyword=value
$ command/qualifier=-
      (keyword=value,keyword=(value,value))
```

Command qualifiers specify additional information or alternate behaviors of commands.

Some qualifiers accept a value or a list of values. When specifying a single value, the parentheses can be left out.

Some qualifiers accept a keyword or a list of keywords. Each keyword may accept a value or a list of values.

The HELP for the command will usually indicate whether a list may be specified for qualifier and/or keyword values.

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;
```

Some qualifiers have the same effect no matter where they appear on the command line. These are called non-positional qualifiers.

The slide shows some examples. The /LOG qualifier is usually non-positional.

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
```

Some qualifiers can appear more than once in a command. These are called positional qualifiers. They qualify (or modify) the command element to which they are immediately adjacent.

An example of this is the /COPIES qualifier of the PRINT command. When applied to the PRINT command, it is global to all the files in the print job. When applied to single file specifications in a PRINT job, it modifies only those files which match that file specification.

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.

The VMS run time library UTIL$SHR provides support for a set of common qualifiers that have been made available in many of the more common commands.

You can find these in the HELP for the DIRECTORY command, SEARCH, PRINT and others.

DCL Statement Elements

$ vbl = value

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

Elements of a DCL statement (as opposed to a command) look very much like other programming languages.

DCL statements always begin with a dollar sign (“$”).

Here we have an example of an assignment statement. A variable receives a value. The value can be a literal expression, the name of another symbol, the result of a function, the result of an arithmetic or string operation, etc.

Assignment Statements

$ vbl = F$lexical_function(params)

Examples:

$ FSP = F$SEARCH(“*.TXT”)
$ DFLT = F$ENVIRONMENT (“DEFAULT”)
$ NODE = F$GETSYI(“NODENAME”)

Here we see a variable which receives the value returned by a built-in (“lexical”) function. The built-in function is part of the DCL lexicon.

The slide also shows some examples.

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 +)

Here are some examples of string operations.

The first operation is a string concatenation.

The second operation is string reduction.

The third operation is a symbol substitution.

What’s happening in statement three?

String Length Limitations

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

Maximum string length for SHOW SYMBOL =
- 500 bytes (up to V7.3-1)
- 4087 bytes (V7.3-2 +)

Maximum string length that DCL can manipulate =
- 510 bytes (up to V7.3-1)
- 8173 bytes (V7.3-2 +)

The longest command string that you can pass to DCL before symbol substitution is 255 bytes for V7.3-1 and earlier, 4095 bytes for V7.3-2 and later.

The longest string that SHOW SYMBOL can display without an informational message (%DCL-I-SYMTRUNC, preceding symbol value has been truncated) is 500 bytes for V7.3-1 and earlier, 4087 bytes for V7.3-2 and later. This appears to be an output buffer size limitation.

The longest string that can be manipulated using lexical functions, concatenation or reduction is 510 bytes for V7.3-1 and earlier, 8173 bytes for V7.3-2 and later.

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"
```

We can replace characters within a string. Note that the colon-equal (";=", local symbol) or colon-equal-equal (":==", global symbol) operators must be used for this type of assignment.

How this is done is illustrated by the example. The characters "ing" are replaced with "ike". Thus, "String 1" becomes "Strike 1".

Assignment Statements

$ vbl = numeric_expression

Examples:

$ A = 1
$ B = A +1
$ C = B + A + %X7F25
$ D = %O3776

Here, we see some examples of numeric assignments.

We have an assignment using a literal and other assignments using numeric additions.

Note the use of hexadecimal notation in the third example and octal notation in the fourth.

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

These are examples of assigning values to bits within a string. The result is always a string. This is useful for constructing escape sequences and binary values.

In the fifth and sixth examples, the result is a two byte string containing a carriage-return and a line-feed (in that order).

Assignment Statements

$ vbl = boolean_expression

Examples:

$ MANIA = (“TRUE” .EQS. “FALSE”)
$ TRUE = (1 .EQ. 1)
$ FALSE = (1 .EQ. 0)
$ YES = 1
$ NO = 0

Here we have examples of assignment of a “truth value” or a boolean value.

The last two examples are ordinary numeric literal assignments. They illustrate the defaults for “true” (“yes”) and “false” (“no”).

Assignment Statements

Local Assignment:

$ vbl = value

Global Assignment:

$ vbl == value

Symbols can be either local to the current procedure level (“depth”) and all levels deeper, or global to all procedure levels (“depths”).

Local symbols are available to the current procedure and any that it invokes.

Global symbols are available to the current procedure and any that it invokes, as well as the procedure(s) which invoked the current procedure.

Assignment Statements

Quoted String:

$ vbl = “quoted string”

Case is preserved.

Examples:

$ PROMPT = “Press RETURN to continue “

$ INVRSP = “% Invalid response!”

When a quoted string is assigned to a symbol, the case and contents of the string are preserved intact.

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

In the case of an unquoted string (uses the “colon-equal[-equal]” sequence), all text becomes upper case, and leading and trailing spaces and TABs are trimmed off. If the unquoted string contains an embedded quoted string, the case and content of the quoted portion of the string will be preserved.

Comment delimiters are observed as usual. The comment is not considered part of the unquoted string.

Foreign Commands

$ vbl := $filespec[param[param[…]]]

“filespec” defaults to SYS$SYSTEM:.EXE

Maximum string length is 510 bytes.

A “foreign command” is a special case where a symbol can be interpreted by DCL as verb. The value of the symbol can include qualifiers and/or parameters in addition to the file specification of the executable file.

If necessary, foreign commands can be defined using quoted strings if, for example, the case of an argument or embedded spaces within an argument needs to be preserved.

Using symbol substitution, strings of up to 1024 bytes can be constructed.

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"
```

To display the value of a symbol, use the SHOW SYMBOL command.

SHOW SYMBOL will display the value of the symbol and indicate whether the symbol as displayed is local or global. Local symbol values are displayed with a single equal sign ("="). Global symbol values are displayed with the double equal ("==").

Note that integer symbols are displayed in three radices: Decimal, Hexadecimal and Octal.

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"
```

To display the value of symbols whose names have beginning characters in common, use the SHOW SYMBOL command with a wildcarded symbol name expression.

Conditional Expressions

$ IF condition THEN statement

Variations:

$ IF condition THEN $ statement

$ IF condition THEN -

$ statement

Conditional expressions provide logical control based on conditions you specify.

In this form, the IF-THEN structure can be stated on a single line or it can be continued across two or more lines.

In either case, the “$” after THEN is optional.

Conditional Expressions

```
$ IF condition
$ THEN
$       statement(s)
$ ENDIF
```

Another variation is the IF-THEN[-ELSE]-ENDIF structure.

This variant allows multiple statements (or no statements) to be included in the THEN or ELSE clause.

Although it is not required, it is recommended that the THEN or ELSE statement appear on a line by itself.

Conditional Expressions

```
$ IF condition
$ THEN
$       IF condition
$       THEN
$           statement(s)
$       ENDIF
$ ENDIF
```

The IF-THEN[-ELSE]-ENDIF structure allows IF-THEN[-ELSE]-ENDIF structures to be nested.

Conditional Expressions

```
$ IF condition
$ THEN
$       IF condition
$       THEN
$           statement(s)
$       ENDIF
$       statement(s)
$ ENDIF
```

Other statements can be included either before or after a nested IF-THEN[-ELSE]-ENDIF structure.

Conditional Expressions

```
$ IF condition
$ THEN
$       statement(s)
$       IF condition
$       THEN
$           statement(s)
$       ENDIF
$ ENDIF
```

If one or more statements are included before a nested IF-THEN[-ELSE]-ENDIF structure, it is recommended that the preceding THEN or ELSE statement appear on a line by itself. In some older versions of VMS, this is a requirement.

For current and future versions of VMS, it is recommended that this guideline be observed to prevent your procedures from “breaking” due to a VMS upgrade, or due to being used on an older VMS version.

Conditional Expressions

```
$ IF condition
$ THEN statement(s)
$       IF condition
$       THEN
$           statement(s)
$       ENDIF
$ ENDIF
```

This may not work in pre-V6 VMS!

Here's an example of some code that might not work in some older versions of OpenVMS.

Notice that the THEN clause includes a DCL statement, instead of being on a line by itself.

Conditional Expressions

```
$ IF condition
$ THEN
$       statement(s)
$ ELSE
$       statement(s)
$ ENDIF
```

Here is an example of an IF-THEN-ELSE-ENDIF block.

As we discussed earlier, the THEN and ELSE statements are recommended to be on lines by themselves.

Either the THEN or ELSE portions may contain nested IFs of any kind.

Labels, GOTO

```
$ GOTO label_1
  .
  .
  .
$label_1:
```


The GOTO statement provides for logical control within your procedures.

Combined with IF, GOTO provides for powerful logical control within your procedures.

Labels are defined by including the label name on a line followed immediately by a colon.

Any statement can follow a label on a line; however, this is recommended only for the SUBROUTINE statement. Otherwise, place the label on a line by itself for readability.

GOSUB, RETURN

```
$ GOSUB label_1
  .
  .
  .
$label_1:
$ statement(s)
$ RETURN
```

The GOSUB and RETURN statements let you create internal subroutines.

All symbols local to the current procedure level are available, as are all global symbols.

Combined with IF, GOSUB and RETURN provide for powerful logical control within your procedures.

Labels are defined by including the label name on a line followed immediately by a colon.

SUBROUTINE - ENDSUB...

```
$ CALL label_1[ param[ param[ …]]
  .
  .
  .
$label_1: SUBROUTINE
$ statement(s)
$ END SUBROUTINE
```

Another form of subroutine is enclosed within the SUBROUTINE and ENDSUBROUTINE statements. This form of subroutine is similar to invoking an external procedure.

Use the CALL statement to invoke this form of internal subroutine. Optionally, parameters to be passed to the subroutine can be included on the CALL statement.

We'll discuss this further in the Intermediate portion of the DCL Programming Session.

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

DCL procedures can invoke other DCL procedures. These can be stand-alone procedures or subroutines of the procedure(s) which invoke them (“external” subroutines).

Each procedure invoked creates a new procedure level or “depth”. Symbols local to the current depth are available to all deeper procedures, but not above. Global symbols are available to all procedure depths.

Each procedure “inherits” the VERIFY condition of the one before it. If VERIFY is on, it stays on; if off, it stays off. Each procedure depth has its own “ON” condition and so can change error handling as it needs to using “SET [NO]ON” and/or “ON condition THEN statement”.

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.

This slide shows how to pass parameters when invoking a DCL procedure either interactively or within another DCL procedure.

Only eight(8) parameters can be passed from the command line. These parameters can contain lists of items. We'll discuss that further in the Intermediate portion of this session.

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.

Within a procedure, you reference parameters using the symbol names P1 through P8. These symbols are local to the current procedure level.

There is no built-in "SHIFT" function that can be used to exhaust the list of parameters, as there is in UN*X and DOS. If you need this functionality, write it as a GOSUB.

Parameters, Cont'd

Example: SHIFT Subroutine:

```
$SHIFT:
$ P1 = P2
$ P2 = P3
  .
  .
  .
$ P7 = P8
$ P8 :=
$ RETURN
```


Here's an example SHIFT subroutine.

It SHIFTs the values of the parameters by one position, discarding the first and blanking out the last.

The next slide shows how to implement this.

Parameters, Cont'd

Use the SHIFT subroutine:

```
$AGAIN:
$ IF P1 .EQS. "" THEN EXIT
  .
  .
  .
$ GOSUB SHIFT
$ GOTO AGAIN
```

Here's and example of how to implement the SHIFT subroutine.

The idea is to process all the parameters on the command line, one by one - as P1 - then exit when none remain unprocessed.

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.

Sometimes, data as input to a program is included in a DCL procedure. This is referred to as “image data”, since it is usually data to be read from SYS$INPUT by an executable image (program).

In the example, a simple text file is CREATEd from the image data provided. “$ EOD” signals end-of-file in a DCL procedure, just as CTRL+Z does from your terminal keyboard. In most cases, any other command beginning with a dollar sign (“$”) will also signal end-of-file and attempt to execute the command.

Some programs do not respond to EOF as expected. EDT in line mode is one example.

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 are another kind of variable. These values can be global to a process, a job (processes owned by a parent process), a UIC group, all processes on the system, or any process which has access to the logical name table in which the logical name is defined.

The ASSIGN and DEFINE statements are similar, except for the order of their arguments.

The DEASSIGN statement is used to delete logical names.

Logical Names

Specifying a logical name table:

```
$ ASSIGN/TABLE=table_name
$ DEFINE/TABLE=table_name
```

Examples:

```
$ DEFINE/TABLE=LNM$PROCESS
$ DEFINE/PROCESS
$ DEFINE/TABLE=LNM$JOB
$ DEFINE/JOB
$ DEFINE/TABLE=LNM$GROUP      ! These require
$ DEFINE/GROUP                ! GRPNAM privilege.
$ DEFINE/TABLE=LNM$SYSTEM     ! These require
$ DEFINE/SYSTEM               ! SYSNAM  privilege.
```

Logical names can be created in any logical name table to which the process has access.

The examples here show some of the convenience qualifiers available for certain logical name tables.

Notice that privileges are required to modify certain logical name tables.

Logical Names

Search lists

$ DEFINE lnm 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 can have more than one translation. These are called “search list” logical names.

Each translation of a search list logical name is referred to by its “index”. We’ll discuss that when we look at the F$TRNLNM() Lexical function.

Logical Names

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

$ ASSIGN/TABLE=LNM$SYSCLUSTER
$ DEFINE/TABLE=LNM$SYSCLUSTER
Requires SYSNAM privilege.

Examples:

$ DEFINE/TABLE=LNM$SYSCLUSTER
$ DEFINE/TABLE=LNM$SYSCLUSTER_TABLE

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

Logical names can also be created in a cluster-wide table. This was new in V7.2. Special system functions keep these tables synchronized on each node of the cluster.

The examples here show the convenience name available for the cluster-wide logical name table. LNM$SYSCLUSTER is provided. The translation of LNM$SYSCLUSTER is LNM$SYSCLUSTER_TABLE.

Notice that privileges are required to modify the cluster-wide logical name table are the same as for the system-wide logical name table.

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).

Access modes of logical names specify additional levels of privilege or supercession.

More privileged access modes (called “inner” modes) require privileges in order to replace the current definition of a logical name. However, another translation of a logical name can be created in a less privileged (“outer”) mode. Such a condition creates an “alias” of the more privileged logical name.

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.

The attributes of a logical name determine how the name is treated with respect to subprocesses and less privileged access modes.

If a name is specified with “CONFINE”, it is not copied to the logical name tables associated with processes spawned by other processes.

If a name is specified with “NO_ALIAS”, another translation cannot be specified in a less privileged (“outer”) access mode.

Non-positional qualifiers were discussed earlier in this session.

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.

The translation attributes of a logical name determine how its translation is handled at various times.

If a logical name’s translation is specified as “CONCEALED”, its translation is not displayed unless that is specifically requested.

If a logical name’s translation is specified as “TERMINAL”, no further attempt is made to translate the current translation of the logical name. This is used in some CONCEALED logical names. It was also used in early VAX systems to reduce the machine time requirements for translation of logical names.

Logical Names

"Rooted" Logical Names -

Specifying Translation Attributes

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

Rooted logical names provide a means of specifying a root level from which additional paths can be specified.

The next slide shows examples of rooted logicals.

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
```

This slide illustrates how to DEFINE a rooted logical and how to use rooted logicals to DEFINE other logical names.

Common Lexical Functions

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

Examples:

```
$ HOME = F$TRNLNM( “SYS$LOGIN” )
$ TERMINAL = F$TRNLNM( “SYS$COMMAND” )
$ MY_VALUE = F$TRNLNM( “MY_LOGICAL_NAME” )
```

F$TRNLNM() can be used to retrieve logical name translations and information about the logical name or its translation. Logical name tables, access modes and indices were introduced earlier.

“case” is one of “CASE_BLIND” which is the default, “CASE_SENSITIVE”, “INTERLOCKED” or “NONINTERLOCKED”. “INTERLOCKED” refers to cluster-wide logical name processing and depends on that to have already completed. The default is to ignore cluster-wide logical name processing.

“item” can be used to return info not already known about a logical name. See the next slide.

Common Lexical Functions

F$TRNLNM() Items

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

Some items return a numeric value (LENGTH, MAX_INDEX).

Some items return either “TRUE” or “FALSE” (CLUSTERWIDE, CONCEALED, CONFINE, CRELOG, NO_ALIAS, TABLE, TERMINAL).

Some items return a string value (ACCESS_MODE, TABLE_NAME, VALUE).

Common Lexical Functions

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

Examples:

$ NODE = F$GETSYI(“NODENAME”)
$ FGP = F$GETSYI(“FREE_GBLPAGES”)
$ FGS = F$GETSYI(“FREE_GBLSECTS”)

F$GETSYI() can be used to retrieve information about the running system. In some cases, it can also be used to get information about other members of an OpenVMS cluster, either by node name or by Cluster System ID (V7.2 and later).

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"

F$CVTIME() can be used to develop routines to get and compare elements of the system date/time.

This can be useful for procedures that need to change their behavior based on the day, date and/or time of day.

Common Lexical Functions

F$CVTIME(), Continued…

Defaults:

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

This slide shows the default behavior of F$CVTIME() if no arguments are provided.

The default value for “string” is the current date and time.

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
  - +/-DDDDD HH:MM:SS.CC

This slide illustrates the date/time formats used and returned by F$CVTIME(). Times can be returned in comparison format which is suitable for “IF” statements, absolute format which is suitable for /AFTER, /BEFORE and /SINCE qualifier values, or delta format which is suitable for use in other date expressions to arrive at another date based on the delta expression.

Note, however, that F$CVTIME() cannot calculate the delta between two date/time expressions.

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)

The F$GETDVI() function is useful for retrieving information about system devices and disk/tape volumes.

The examples show some of the information that can be returned by F$GETDVI(). Notice that either literal strings or symbols can be used as arguments to DCL lexical functions.

Common Lexical Functions

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

See the on-line help for descriptions.

More during the Advanced section...

F$GETQUI() is a useful, if rather complex lexical function.

We won't go into great detail about it at this point in the session. Later in this session, we will show how a batch job can use F$GETQUI() to get information about itself. A little later, we'll discuss using F$GETQUI in a loop to get information about queues and the jobs in them.

See also the on-line HELP and the DCL Dictionary for additional information about F$GETQUI().

Common Lexical Functions

$ VBL = F$GETJPI(pid, keyword)

Examples:

$ USN = F$GETJPI(0, “USERNAME”)
$ MOD = F$GETJPI(0, “MODE”)

F$GETJPI() can be used to get information about your own process or about any other process to which you have access. Normal rules of OpenVMS privilege apply: to get information about other processes within your UIC group, you need GROUP privilege; to get information about processes outside of your UIC group, you need WORLD privilege.

For information about the current process, specify the PID argument as a zero(0) as shown in the examples, or as a null string.

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.

Here’s where we start to discuss batch jobs that can reSUBMIT themselves.

Information that the job may need, such as the queue name, the job name, the procedure name, etc. can either be hard-coded or it can be retrieved using the F$GETQUI() lexical function by specifying “THIS_JOB” as the fourth parameter.

See the DCL Dictionary or the on-line HELP for more items that can be retrieved.

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”, )
```

Here’s an illustration of one method for getting some of the information needed to allow a batch job to SUBMIT itself for the next day.

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”, )
```

Here’s an illustration of one method for getting some of the information needed to allow a batch job to SUBMIT itself for next week.

SUBMIT/AFTER

ReSUBMIT the job for the next run:

```
$ SUBMIT/AFTER="'NEXT'"
  or
$ SUBMIT/AFTER=&NEXT
```

Here's an example of how to use the calculated next run time to reSUBMIT the job for the next run.

Two methods of symbol substitution are illustrated.

The first method uses "apostrophe substitution" within a quoted string. This preserves case and embedded spaces.

The second method uses ampersand substitution which also preserves case and embedded spaces.

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

Here’s an illustration of one method for finding tasks to be performed on a specific day.

Notice that the day name is used as the name of the subdirectory where the tasks (procedures) for that day will be found.

The symbol substitution methods used in the example will be discussed in more detail in the Intermediate portion of this session.

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

Here’s an illustration of one method for finding tasks to be performed on a specific day of the month.

In this case, the day number is used as the name of the subdirectory where the tasks (procedures) for that day will be found. Specifically, the tasks for the first day of the month will be run.

The symbol substitution methods used in the example will be discussed in more detail in the Intermediate portion of this session.

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
```

Here's an illustration of one method for determining whether the current day is the last day of the month.

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

Here’s an illustration of one method for finding tasks to be performed annually.

In this case, both the month number and the day number are used as the name of the subdirectory where the tasks (procedures) for that day will be found. Specifically, the tasks for the first day of the first month and the 31st day of the seventh month will be run.

The symbol substitution methods used in the example will be discussed in more detail in the Intermediate portion of this session.

Q / A

Speak now or forever

hold your peas.

Let’s take a moment to discuss questions that have come up in our discussion so far...

Break Time !

We'll continue in a few minutes!

Time for a quick break!

The Intermediate portion of this session will follow immediately after a short 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

Now, we'll get deeper into some of the more advanced functions like complex internal subroutines, some more lexical functions, ...

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 (&)

...processing wildcarded file specifications, reading and writing disk files and process permanent files, parsing strings and parameters, and using symbol substitution.

More Internal Subroutines

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

DCL provides internal SUBROUTINEs that act like external procedures. This allows for easier parameter passing than GOSUB, also.

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.)

The CALL statement allows parameters to be passed to the SUBROUTINE.

SUBROUTINEs act like another procedure depth. Local symbols are local to the subroutine and global symbols are visible to the SUBROUTINE and the code that CALLs it.

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 wild cards.

Here's a deeper look at the F$SEARCH lexical function.

F$SEARCH() is useful for finding files using both absolute and wildcarded file specifications.

If the file you specify is not found, F$SEARCH returns a null string.

If you supply a wildcarded filespec, F$SEARCH returns the next matching filespec on each subsequent invocation. When there are no more matching files, a null string is returned.

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 last iteration.

Here’s an example of a loop which uses F$SEARCH.

Note that if the search specification is not wildcarded, F$SEARCH will return the same string over and over. The example shows how to avoid locked loops by saving the filespec after each invocation and comparing the previous string to the current string. If they match, exit the loop.

Multiple F$SEARCH Streams

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

Examples:

```
$ vbl1 = F$SEARCH( SRC1, 111111 )
$ vbl2 = F$SEARCH( SRC2, 121212 )
```

You can have more than one F$SEARCH stream at a time. Just supply a unique context identifier for each stream.

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

DCL provides four statements for performing file I/O: OPEN, READ, WRITE and CLOSE.

Use OPEN to begin using a file.

Use READ to get data from a file.

Use WRITE to write data to a file or to update existing records.

Use CLOSE to finish using a file and release the associated resources.

File I/O - OPEN

```
$ OPEN logical_name filespec

$ OPEN
   /ERROR=label
   /READ
   /WRITE
   /SHARE={READ|WRITE}
```

The OPEN statement establishes a “channel identifier” which you can use in READ and WRITE statements as well as in the CLOSE statement to finish using the file.

/READ opens the file for reading. The file must exist.

/WRITE opens the file for writing. If not accompanied by /READ, a new file is created.

/SHARE specifies how other I/O streams may use the file.

/ERROR is used to specify a label where control should be transferred when an error occurs.

File I/O - READ

```
$ READ logical_name symbol_name

$ READ
   /DELETE
   /END_OF_FILE
   /ERROR
   /INDEX
   /KEY
   /MATCH
   /NOLOCK
   /PROMPT
   /TIME_OUT ! Terminals only
```

The READ statement retrieves data from a file.

The qualifiers shown provide for labels to receive control in case of error or at end of file, and provide ways to specify a key to match, which index to search, how to match the key value specified (RMS indexed files), a prompt string to use when READing from a terminal, and a TIME_OUT value for a time to wait for input from a terminal.

File I/O - WRITE

```
$ WRITE logical_name symbol_name

$ WRITE
   /ERROR
   /SYMBOL ! Use for long strings
   /UPDATE
```

The WRITE statement is used to write data to a file or to update an existing record in a file.

/ERROR is used to specify a label where control should be transferred when an error occurs.

/UPDATE is used to update an existing record.

/SYMBOL is used to write strings longer than 255 bytes.

File I/O - CLOSE

```
$ CLOSE logical_name

$ CLOSE
   /ERROR
   /LOG
```

The CLOSE statement is used to finish using a file. The buffers are flushed and all associated resources are released.

/ERROR is used to specify a label where control should be transferred when an error occurs.

/LOG is used to avoid an error and a message when closing a file that isn't open.

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)

DCL has access to the Process Permanent Files or I/O streams associated with a process. Some of these have parallels in the UN*X and DOS/Windows worlds.

SYS$INPUT is equivalent to the UN*X standard input (stdin) stream.

SYS$OUTPUT is equivalent to the UN*X standard output (stdout) stream.

SYS$ERROR is equivalent to the UN*X standard error (stderr) stream.

SYS$COMMAND has no UN*X equivalent.

SYS$PIPE is present when using the PIPE command.

Process Permanent Files (PPFs)

SYS$INPUT (UN*X equivalent: stdin)

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

```
$ DEFINE SYS$INPUT filespec
$ DEFINE/USER SYS$INPUT filespec
```

The SYS$INPUT stream, which is equivalent to the UN*X standard input (stdin) stream, usually points to your terminal.

When a DCL procedure is invoked, SYS$INPUT points to the current procedure. Similarly, when a DCL procedure is SUBMITted to batch, SYS$INPUT points to the current procedure.

SYS$INPUT can be redirected using DEFINE (or ASSIGN). If DEFINEd in /USER mode, it will be DEASSIGNed when a program is run and that program terminates. If DEFINEd in /SUPERVISOR mode (the default if no access mode is specified in the DEFINE (or ASSIGN) command) or an “inner” (more privileged) mode, it remains in effect until explicitly DEASSIGNed.

Process Permanent Files (PPFs)

SYS$OUTPUT (UN*X equivalent: stdout)

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

```
$ DEFINE SYS$OUTPUT filespec
$ DEFINE/USER SYS$OUTPUT filespec
```

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

The SYS$OUTPUT stream, which is equivalent to the UN*X standard output (stdout) stream, usually points to your terminal.

When a DCL procedure is invoked using /OUTPUT, SYS$OUTPUT points to the file specified. Similarly, when a DCL procedure is SUBMITted to batch, SYS$OUTPUT points to the batch job's log file (if any).

SYS$OUTPUT can be redirected using DEFINE (or ASSIGN). If DEFINEd in /USER mode, it will be DEASSIGNed when a program is run and that program terminates. If DEFINEd in /SUPERVISOR mode (the default if no access mode is specified in the DEFINE (or ASSIGN) command) or an “inner” (more privileged) mode, it remains in effect until explicitly DEASSIGNed.

Process Permanent Files (PPFs)

SYS$ERROR (UN*X equivalent: stderr)

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

```
$ DEFINE SYS$ERROR filespec
$ DEFINE/USER SYS$ERROR filespec
```

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

The SYS$ERROR stream, which is equivalent to the UN*X standard error (stderr) stream, usually points to your terminal.

When a DCL procedure is invoked using /OUTPUT, SYS$ERROR points to the file specified. Similarly, when a DCL procedure is SUBMITted to batch, SYS$ERROR points to the batch job's log file (if any).

SYS$ERROR can be redirected using DEFINE (or ASSIGN). If DEFINEd in /USER mode, it will be DEASSIGNed when a program is run and that program terminates. If DEFINEd in /SUPERVISOR mode (the default if no access mode is specified in the DEFINE (or ASSIGN) command) or an “inner” (more privileged) mode, it remains in effect until explicitly DEASSIGNed.

Process Permanent Files (PPFs)

SYS$COMMAND (No UN*X equivalent)

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

```
$ DEFINE SYS$COMMAND filespec
$ DEFINE/USER SYS$COMMAND filespec
```

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

The SYS$COMMAND stream has no UN*X equivalent.

For an interactive process, SYS$COMMAND always points to your terminal, unless you explicitly redirect it. For a batch job, SYS$COMMAND always points to the current procedure.

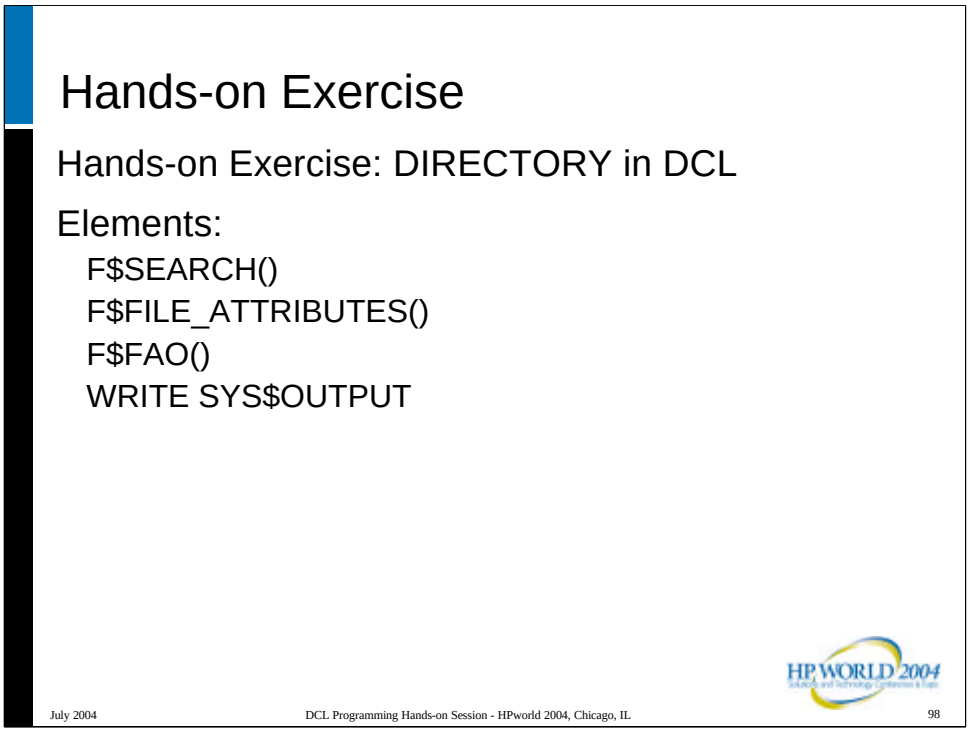
SYS$COMMAND can be redirected using DEFINE (or ASSIGN). If DEFINEd in /USER mode, it will be DEASSIGNed when a program is run and that program terminates. If DEFINEd in /SUPERVISOR mode (the default if no access mode is specified in the DEFINE (or ASSIGN) command) or an “inner” (more privileged) mode, it remains in effect until explicitly DEASSIGNed.

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
```

Here's an example of a loop to read a file and process its records.

The /END_OF_FILE qualifier is used to direct control to the "EOF_INFLE" label at end of file.



Time for a hands-on exercise!

Let’s write a Directory procedure using DCL.

The DCL elements we’ll use are:
o F$SEARCH()
o F$FILE_ATTRIBUTES()
o F$FAO()
o WRITE SYS$OUTPUT

F$FILE_ATTRIBUTES() and F$FAO() are new to our discussion. We’ll only use a small handful of their keywords and directives. We'll go over them briefly, first.

Hands-on Exercise

Hands-on Exercise: DIRECTORY in DCL

Element: F$SEARCH(fsp[, stm_id)

fsp: Wildcarded File Specification

.;*

stm_id: not used

We've already talked about F$SEARCH().

We'll be F$SEARCHing for all the files in the current directory.

We'll have a single F$SEARCH() stream, so we won't be using the stream_id parameter.

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

F$FILE_ATTRIBUTES() is new to our discussion. It returns information about files. It has many keywords.

We'll use these F$FILE_ATTRIBUTES() keywords to get attributes of the files we find using F$SEARCH():

o EOF is the count of blocks actually used by data in the file.

o ALQ is the count of blocks allocated to the file.

o CDT is the creation date of the file.

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

F$FAO() (formatted ASCII Output) is also new to our discussion. We'll use only three of its simpler directives.

The "!n<" and "!>" operators indicate that the string which results from the expression within them should be of a specified length: n bytes. The results will either be padded or truncated as needed.

The "!AS" operator means that an ASCII string is expected in the output.

The "!SL" operator means that a signed longword integer is expected in the output.

Hands-on Exercise

Hands-on Exercise: DIRECTORY in DCL

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

We'll use:
WRITE SYS$OUTPUT F$FAO(...)

We've already discussed using WRITE to output data. In this exercise, we'll WRITE our output data to the SYS$OUTPUT stream.

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.

Here's the description of the design for the procedure we'll write:

First, use F$SEARCH() to find all the files in the current directory.

Second, use F$FILE to get the EOF and ALQ sizes as well as the creation date (CDT) for each file; use WRITE SYS$OUTPUT and F$FAO() to display the data in neat columns.

Exit the loop when there are no more files to display.

No time limit, no wrong answers.

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 )
```

Moving on to some more advanced lexical functions, we start with F$ELEMENT. Use this to parse strings by searching for specific characters, such as comma, pipe symbol, space, etc.

The "index" starts at zero. The "delim" parameter can be any single character.

If the value of "index" points to an element beyond the end of the string, the function returns the delimiter character.

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:
```

Here’s an example of a loop for retrieving all the elements of a string. In the example, elements of the string are delimited by commas.

Note that the index is incremented before the element returned is examined. This is one way to help avoid locked loops. Ignoring null elements might not always be desirable.

When F$ELEMENT returns a comma, control is transferred to the EXIT_LOOP_1 label.

Symbol Substitution

Two forms of symbol substitution:

`&symbol_name`

`'symbol_name'` or **`"''symbol_name'"`**

Let's look at symbol substitution. DCL provides two "passes": one for symbols preceded by an ampersand ("&") and another for symbols preceded by an apostrophe, or two apostrophes when used within a quoted string.

“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'”
```

Symbol substitution with the apostrophe causes the contents of the variable to be added to the command buffer as string data. Individual command elements (separated by “whitespace”) will be treated as individual elements.

When executing a DCL procedure with VERIFY on (SET VERIFY), the results of the substitution will appear on your screen or in the log file.

When using apostrophe outside of a quoted string, use a single leading and trailing apostrophe.

When using apostrophe within a quoted string, use two leading apostrophes and a SINGLE(!) trailing apostrophe.

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 with the ampersand causes the contents of the variable to be added to the command buffer as string data. Individual string elements (separated by “whitespace”) will be treated as part of a single element, as illustrated in the example. Case is preserved.

Ampersand substitution is only used outside of quoted strings.

Symbol Substitution

Order of Substitution:

1. Apostrophe(‘)
2. Ampersand(&)

Symbol substitution occurs in the order shown.

First, symbols preceded by apostrophes are processed. When expanded, the value of a symbol preceded by apostrophe is treated as one or more “words” or tokens.

Second, symbols preceded by an ampersand are processed. The value of a symbol preceded by an ampersand is treated as a single “word” or token.

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.)

The maximum length of a command line before symbol substitution is 255 bytes in OpenVMS up to and including V7.3-1. As of V7.3-2 and later, this increased to 4095 bytes.

DCL has a somewhat larger internal buffer which allows for the results of symbol substitution.

See the HELP topics “=“ and “:=“ for further information. (Steve Hoffman)

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
```

Here is an example of symbol substitution using the apostrophe (“”) operator.

With VERIFY “on”, we can see that when invoked, the contents of the symbol named TEXT will appear within the quotes, either on the screen or in a log file (batch job log, or /OUTPUT when invoked interactively or from another procedure).

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)
```

Here is an example of symbol substitution using the ampersand (“&”) operator.

With VERIFY still turned “on”, we can see that when invoked, the contents of the symbol named TEXT do not appear in the output until the SHOW LOGICAL command is issued.

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.

Controlling symbol scope can help control confusion when a symbol name is used in more than one nested procedure.

When symbol scope is set to NOLOCAL, local symbols from lesser procedure depths are “invisible” to the current procedure depth all “greater” depths.

When symbol scope is set to NOGLOBAL, global symbols are “invisible” to the current procedure depth all “greater” depths.

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.

Other SET SYMBOL/SCOPE qualifiers control how the symbol scoping rules are applied.

/VERB applies to the first “token” (or “word”) on a command line.

/GENERAL applies to all other “tokens” (or “words”) on a command line.

/ALL applies to all of the “tokens” (or “words”) on a command line.

“Words” are delimited by the space (ASCII 32) character for DCL’s purposes.

String Operations

Concatenation:

```
$ vbl = string1 + string2
```

Example:

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


Connecting two or more shorter strings together is known as “concatenating”. The original strings remain unchanged. The target string includes the contents of the original strings as one longer string. No spaces or other characters are inserted or appended.

The example illustrates the use of the F$ENVIRONMENT and F$PARSE lexical functions. Two elements of the procedure file specification are then concatenated together for use later on in the procedure.

String Operations

Reduction:

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

Example:

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

Removing a substring from a longer string is known as string reduction. The first instance of the substring is removed from the longer string, and the result is stored in the target symbol.

The example illustrates stripping the underscore and colon from a device name.

String Operations

Substring Replacement:

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

Example:

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

Substrings within a longer string can be replaced. The starting position and the length of replacement (in bytes) are indicated within the square brackets following the name of the target string. The colon-equal operator (":=")or the colon-equal-equal operator (":==") **MUST** be used for this type of assignment.

The example illustrates replacing a single character near the middle of a string.

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
```

Binary values can also be assigned using the square brackets following the name of the target symbol.

The equal operator (“=“) or equal-equal operator (“==“) **MUST** be used for this type of assignment.

The first value within the brackets is the bit displacement into the target field. The second value is the number of bits to be affected by the assignment.

DCL “Arrays”

DCL does not support arrays in the usual sense. 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
```


While DCL does not support arrays in the sense of subscripted variables as one might find in a 3GL, a counter can be used to sequentially create variables with a number appended to the variable name. This can be done using symbol substitution to create a target variable name, as shown the example.

Integer Operations

DCL supports the four basic arithmetic operations:

+	Add
-	Subtract
*	Multiply
/	Divide

DCL can perform arithmetic operations on integer values. Integers are treated as signed longword (32 bit) values.

Boolean Operations

DCL supports assignment of boolean values:

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

Examples:

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

The “truth” value of conditional expressions can be assigned to variables for use in multiple comparisons within a procedure.

Logical Operations

DCL supports logical AND, OR and NOT

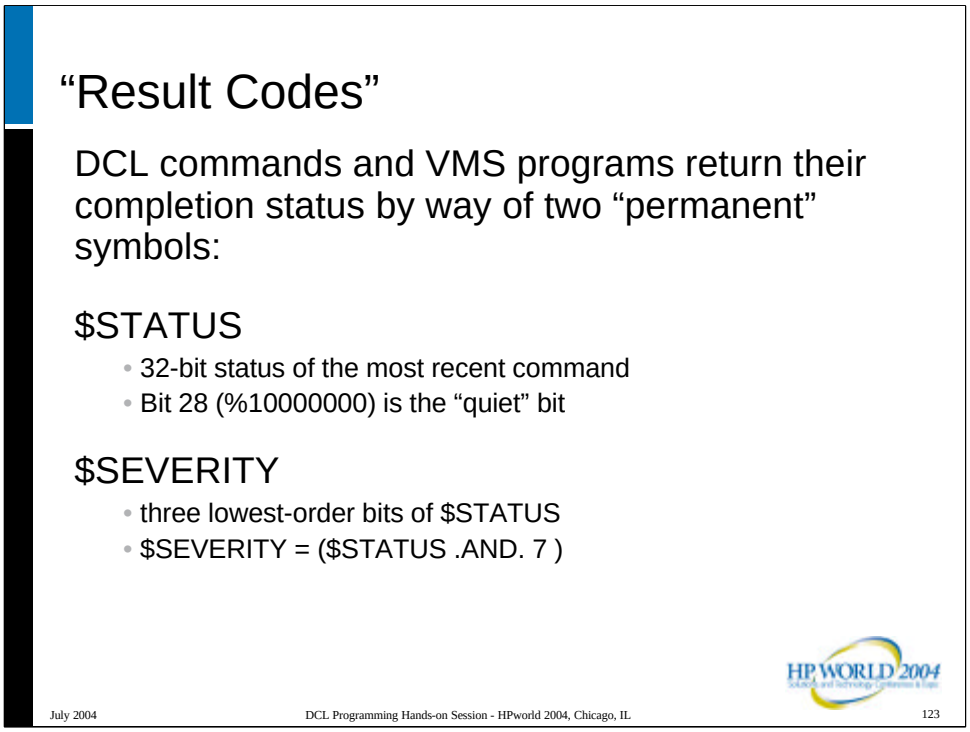
```
$ vbl = (int1 .AND. int2)
$ vbl = (int1 .OR. int2)
$ vbl = (.NOT. int3)
```

Examples:

```
$ STATUS = &$STATUS
$ SEVERITY = (STATUS .AND. 7)
$ FAILURE = (.NOT. (SEVERITY .AND. 1))
$ EXIT_STATUS = (STATUS .OR. %X10000000)
```

DCL provides the logical AND and logical OR boolean operators, and the NOT operator.

The examples show the use of the .AND., .NOT. and .OR. operators. Notice that FAILURE is taken as the logical NOT of a condition that would indicate success.



DCL commands and VMS programs return their completion status by way of two "permanent" symbols: $STATUS and $SEVERITY. These symbols cannot be deleted, nor can their value be changed by using an assignment statement.

$STATUS contains the entire 32-bit status word returned by the command or program. If bit 28 (the "quiet" bit) is not set, an error message will appear after the command or program exits.

$SEVERITY contains the value of the three low-order bits of $STATUS.

“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!

Here are some examples of $STATUS and $SEVERITY.

In the first example, the “quiet” bit is set. As a result, no error message is displayed. The values of $STATUS and $SEVERITY are set according to the value specified on the EXIT statement.

In the second example, the “quiet” bit is NOT set, the error message is displayed.

Note that $STATUS and $SEVERITY are STRING symbols, although their contents are integer 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...

Not all DCL commands and statements change the value of $STATUS and $SEVERITY. The ones that do not are “internal” to DCL.

Listed are most of the DCL commands and statements that do change the value of either $STATUS or $SEVERITY. There are others not listed here that are rarely used, but are documented (DEPOSIT, EXAMINE).

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.

Through the use of the “ON” statement, you can control how DCL behaves when various types of errors or events occur.

A WARNING event occurs when the severity is zero(0) (($STATUS .AND. 7) .EQ. 0).

An ERROR event occurs when the severity is two(2) (($STATUS .AND. 7) .EQ. 2).

A SEVERE ERROR event occurs when the severity is four(4) (($STATUS .AND. 7) .EQ. 4).

Only the most recent “ON” statement takes effect.

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 action becomes CONTINUE)

The "ON" statement's order of precedence is:

ON WARNING effects "warning" events and greater.

ON ERROR effects "error" events and greater. The "ON WARNING" action becomes CONTINUE.

ON SEVERE ERROR effects "severe error" events and greater. The "ON WARNING" and "ON 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.

Through the use of the “ON” statement and the “SET ON” and “SET NOON” statements, you can control how DCL behaves when various type of errors or events occur.

SET NOON allows your DCL code to detect and process the $SEVERITY of the most recent command (not all commands cause a change of $STATUS and $SEVERITY).

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
```

This code segment illustrates how to trap and handle errors without DCL's intervention.

After the "statement" is executed, the value of $STATUS is saved, and the SEVERITY is derived from the saved STATUS.

Different actions are taken depending upon the value of the SEVERITY symbol.

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!

Now, we'll look at some more lexical functions.

F$TRNLNM() is used to get the translation of a logical name, isolate the translation to a specific logical name table, index or mode, or to determine such characteristics about a logical name.

Notice that F$TRNLNM() does **NOT** support wildcarded logical name expressions.

F$TRNLNM() supercedes the older F$LOGICAL() which is deprecated.

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...	

The F$ENVIRONMENT() lexical can be used to get information about the current process environment.

The example keywords shown are some of the more useful keywords. Other keywords are available to determine the CONTROL characters currently enabled, the current procedure depth, the current DCL prompt string, the current default file protection, the current symbol scope, etc.

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
```

This example illustrates one method of suppressing messages selectively. The technique shown is to use the F$ENVIRONMENT lexical to save the MESSAGE display state, change it to what is wanted, then change it back to the original state.

Lexical - F$PARSE

Use to verify or extract portions of a file specification.

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

The F$PARSE() lexical can be used to verify a file specification or to extract portions of a valid file specification.

The “parse_type” keywords are SYNTAX_ONLY and NO_CONCEAL.

SYNTAX_ONLY parses a file spec and returns values whether the specified file and/or path exists or not.

NO_CONCEAL can be used to get the translation of a logical defined with the CONCEALED translation attribute.

Lexical - F$PARSE

A useful example:

```
$ DFSP = F$ENVIRONMENT( “DEFAULT” ) + “.COM”
$ FSP = F$PARSE( “LOGIN”, DFSP )
$ SHOW SYMBOL FSP
  “FSP” = “MYDISK:[MYDIR]LOGIN.COM;”
```


Another use of F$PARSE is to complete a partial file specification, applying default values for those portions not specified.

The example shown illustrates how “LOGIN” can be expanded by applying appropriate default values for the other portions of the file specification.

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
```

This example illustrates how F$PARSE can be used to extract portions of a file specification.

In the example, a new default disk and directory specification is derived from the disk and directory where the currently executing DCL procedure is found, the current default disk and directory are saved, then the new default is applied. A (series of) statement(s) is(are) executed, then the original default disk and directory specification is restored.

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 )
```

This example illustrates how F$PARSE can be used to determine the parent directory of a directory.

In the example, the trailing “]” is reduced from the string and “.-]” is appended indicating that we’re looking for the directory above the last in the path.

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”
```

This example illustrates how F$PARSE can be used to verify that a path exists.

In the example, a file location is passed to F$PARSE(). If the device is MOUNTed and the directory exists, F$PARSE() will return the device and directory specification with “.;” appended; if not, a null string is returned.

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.

The F$GETQUI() lexical function can be used to get information about queues and the jobs on those queues.

F$GETQUI() can be complicated to use; but its usefulness is well worth the effort.

Lexical - F$GETQUI

Functions:

- **CANCEL_OPERATION**
- **DISPLAY_ENTRY**
- **DISPLAY_FILE**
- **DISPLAY_FORM**
- **DISPLAY_JOB**
- **DISPLAY_MANAGER**
- **DISPLAY_QUEUE**
- **TRANSLATE_QUEUE**

This slide lists some of the available function codes. Using these, your procedure can get information about a queue, an entry, a form, a job (where the entry number is not yet known), a queue manager or a logical queue.

The CANCEL_OPERATION function can be used to intentionally destroy the current context. This is useful before creating a new context, to ensure that any previous context has been cleared.

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!

This slide lists just a few of the many items that can be returned about a queue, a job, a form, etc.

The DCL Dictionary, Volume 1 is very useful to have at hand when using F$GETQUI(), as all of the available functions, item codes and other arguments are listed.

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.

When getting information about all the jobs on a queue, first create the queue context using DISPLAY_QUEUE. Then use DISPLAY_JOB repeatedly to loop through all the jobs in the queue.

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”, “*”,)
```

When displaying multiple items about a queue, a job, etc., use the FREEZE_CONTEXT flag on all the items but the last. This prevents the current context from being advanced to the next queue, job, etc. until all the needed items about each queue, job, etc. have been retrieved.

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, “*”,)
```

Using symbols instead of string literals can help shorten DCL statements that use F$GETQUI and other lexicals that require keywords.

For example, here are the same two statements from the previous slide. Notice that they now fit on a line without continuation or wrapping.

This costs some environment space, but it’s usually worth it.

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...
```

This slide and the next illustrate how to use F$GETQUI() in a loop to display the queues on the system and the jobs in them.

This slide shows the symbols set up to replace string literals.Those symbols will be used in the next slide.

Note that for DISPLAY_JOB, standard rules of OpenVMS privileges apply: in order to display jobs that do not belong to your UIC, your process must have OPER privilege.

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
```

This slide and the previous illustrate how to use F$GETQUI() in a loop to display the queues on the system and the jobs in them.

This slide shows the actual loop to do the work. This code was modified from an example in the on-line HELP for the F$GETQUI() lexical function. See HELP Lexicals F$GETQUI, and the Examples subtopic.

Note that for DISPLAY_JOB, standard rules of OpenVMS privileges apply: in order to display jobs that do not belong to your UIC, your process must have OPER privilege.

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.

The previous slide described a loop to display queues and the jobs in them. To do this, we first establish a “queue context” using DISPLAY_QUEUE so we can then use DISPLAY_JOB to get information about the jobs in each queue (if any).

However, OpenVMS provides only one queue context per process. If SHOW QUEUE is used in between F$GETQUI() invocations, the process’s queue context is destroyed and the next invocation of F$GETQUI() may produce an error or unpredictable results.

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" )
```

The F$CVTIME() function returns multiple elements of a date/time expression. It also performs conversion from one format to another, and allows for the addition or subtraction of days, hours, minutes, seconds, etc. from a known date/time or the current date time.

The examples show how to:

1. get the date/time for a week from now.
2. see if tomorrow is the first of the month/year.
3. get an elapsed time based on a starting time and an ending time (may not cross midnight more than once!).

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).

Earlier in this session, we looked at substring replacements. The F$EXTRACT() lexical function allows substring extraction.

The original string is left unchanged; only the contents of the requested substring are returned.

Note that the offset is “zero-relative”. The first character in a string has an offset of zero(0). The offset of the last character in a string is equal to the length of the string minus one(1) .

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…

The F$GETDVI() lexical is used to get information about devices in the system, or to see if a device exists.

Some of the valid items are listed. Many more items are available.

Lexical - F$EDIT

Use to modify strings.

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

Keywords:

- **COLLAPSE**
- **COMPRESS**
- **LOWERCASE**
- **TRIM**
- **UNCOMMENT**
- **UPCASE**

The F$EDIT() lexical function is used to modify strings.

Strings can be COLLAPSEd (all spaces and TABs are removed), COMPRESSed (spaces and TABs between words are reduced to a single space), converted to upper (UPCASE) or LOWERCASE, TRIMmed of leading and trailing spaces and TABs, and comments can be stripped off (UNCOMMENT). The comment delimiter is the exclamation point(“!”).

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” )
```


The F$GETJPI() lexical function is used to get information about the current job or process.

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

The example shows how to retrieve the mode of the current process. This could also be retrieved using the F$MODE() lexical function.

Lexical - F$GETJPI

Some useful items:

- **IMAGNAME**
- **MASTER_PID**
- **MODE**
- **PID**
- **PRCNAM**
- **USERNAME**
- **WSSIZE**

This slide shows just a few of the more useful items than can be retrieved for a process. Many item codes are available.

Refer to the on-line HELP or DCL Dictionary, Volume 1 for a complete listing.

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.

A note of caution about a couple F$GETJPI() items codes:

The AUTHPRIV and CURPRIV items can return strings which are too long to manipulate. Use them with caution.

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	

The F$GETSYI() lexical function can be used to retrieve many useful items of information about the running OpenVMS system.

Any system parameter value can be retrieved, as well as some information about the cluster and the hardware on which the system is running.

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.

When used together, the F$CONTEXT and F$PID functions provide a means to look up processes on the system by a number of selection criteria.

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.

Use the F$CONTEXT function to set up your process selection criteria. This allows a programmatic way of locating processes by name, by mode, by UIC, etc. without the need to use intermediate files or parse the output of a DCL command such as SHOW SYSTEM.

Multiple selection criteria can be specified by issuing multiple invocations of F$CONTEXT. The context constructed this way can be used with the F$PID function in a loop to return the PIDs of all processes matching the selection criteria specified. When there are no more matching processes, F$PID() returns a null String.

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" )
```

Once you have set up your selection context, pass that context symbol to F$PID() and invoke it in a loop until a null string is returned by F$PID().

By default (null context), F$PID() will return the PIDs of all processes in the system (local to a cluster node).

To locate all processes in the cluster matching the other selection criteria, include an F$CONTEXT invocation specifying an item code of NODENAME, a matching string of "*" and a match criterion of "EQL".

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
```

Other lexical functions exist as well as those discussed in this presentation. The slide shows the output of “HELP Lexical” and shows all of the available lexical function names.

Refer to the DCL Dictionary, Volume 1 for complete information on the available lexical functions.

Q & A

Speak now or forever hold your peas.

Let’s pause to answer any questions there may be about what we’ve covered in this section!

Break Time !

We'll continue in a few minutes!

Time for a quick break!

The Advanced portion of the DCL Programming session will follow immediately after a short 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.

In the advanced section, we'll discuss using ampersand's (&) special characteristics to your advantage – assign the entire content of a string to a logical name, preserving case and spacing.

We'll talk about F$TYPE() and some ideas on how to use it.

We'll talk about using F$PARSE() to validate and navigate paths.

We'll talk some more about using F$CONTEXT() and F$PID().

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"

We'll see how to use F$CSID() and F$GETSYI() to return information about cluster nodes, with a practical example.

We'll take a look at the PIPE command, and examine in detail its flexibility and intricacies. We'll see how to use PIPE to get the output of a command into a symbol, and explore other PIPE "magic".

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 Search Order is determined by a search list logical name called LNM$FILE_DEV. It is found in the LNM$SYSTEM_DIRECTORY logical name table. Some DCL features use a logical name called LNM$DCL_LOGICAL which points to LNM$FILE_DEV.

The slide shows how LNM$FILE_DEV is set up in an unmodified OpenVMS system.

Note that LNM$FILE_DEV lists other logical names which point to the actual logical name tables being referenced.

Logical Name Table Search Order

Modifying the search order:

If no CMEXEC privilege:

```
$ DEFINE/TABLE=LNM$PROCESS_DIRECTORY LNM$FILE_DEV -
      LNM$PROCESS,LNM$JOB,LNM$GROUP,LNM_APPL,LNM$SYSTEM
```

With CMEXEC privilege:

```
$ DEFINE/TABLE=LNM$PROCESS_DIRECTORY/EXEC LNM$FILE_DEV -
      LNM$PROCESS,LNM$JOB,LNM$GROUP,LNM_APPL,LNM$SYSTEM
```

Logical Name Search Order can be modified by DEFINE-ing (or ASSIGN-ing) LNM$FILE_DEV in your LNM$PROCECSS_DIRECTORY logical name table.

Executive mode is preferable if you have CMEXEC (Change Mode to Executive) privilege. However, supervisor (the default) mode will suffice for most applications.

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.

It is important to use the F$TRNLNM() lexical function to get the translation of logical names rather than the deprecated F$LOGICAL() lexical function.

The reason is the F$TRNLNM() uses the LNM$FILE_DEV search list. F$LOGICAL() uses a hard-coded search list.

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

The F$TYPE() lexical function returns the datatype of the contents of a symbol - INTEGER or STRING, the symbol type (PROCESS_CONTEXT), or returns a null string if the named symbol does not exist in the environment.

Note that except for PROCESS_CONTEXT symbols, F$TYPE() returns the datatype of the contents of a symbol and not the data type of the symbol itself.

The next slide illustrates this with some examples...

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
```

In the examples, we can see that while a symbol may actually be a character string, F$TYPE() indicates the datatype of the content of the symbol.

Symbol A1 contains the string “1234”, but F$TYPE() reports that it contains an integer value.

Symbol A2 contains the string “01B7”. F$TYPE() reports that it contains a string value because of the presence of the letter “B”. F$TYPE() doesn’t know about hexadecimal!

Symbols A3 and A4 illustrate that integer symbol values can be in decimal or hexadecimal. Octal (%O) is valid, also, but is not shown here.

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:
```


Here is an example of using F$CONTEXT() to find either a specific process by name, or processes that match a wildcarded name.

Some other twists:

If P1 is a hexadecimal expression, P1 is used as the PID of the process to be examined. Otherwise, it is treated as a portion of a process name.

If P2 is not specified, it is assumed that only the local node is to be searched. If P2 is specified as another node name, only that node will be searched. If P2 is specified as a wildcard (“*”), all nodes of the cluster will be searched.

The OUTPUT_PSTAT routine is executed for each process found to display process information.

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
```

This an example from a DCL proc. written by the author of this presentation to get and display information about the cluster and the nodes in it.

This illustrates how to use F$CSID() in a loop to get the Cluster System ID. for each node in the cluster, one node at a time, then use F$GETSYI() to get information about each node.

Mostly useful in system administration duties.

PIPE

PIPE command
- Introduced in OpenVMS V7.2
- Early PIPE has issues:
  - mailbox quotas
  - synchronization.
- O.k. in V7.3 and later

The PIPE command appeared in OpenVMS V7.2, but had some issues initially. The PIPE line could not accept large amounts of data, and synchronization between pipeline elements was troublesome at times. All of these issues are resolved in V7.3 plus the PIPE-related ECOs. In V7.3-1 and later, PIPE is very stable and usable.

PIPE works by executing each (set of) command(s) in a separate subprocess (called a “subshell” in UN*X-land).

PIPE

PIPE command

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

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

The syntax of the PIPE command is essentially the same as it in UN*X-land, except, of course, for the PIPE verb itself.

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

Like UN*X pipelines, the output stream of the first command becomes the input stream of the next, an so on to the end of the pipeline. To do this, PIPE introduces another PPF (Process Permanent File) called SYS$PIPE which is usually the same as SYS$INPUT.

However, sometimes SYS$INPUT and SYS$PIPE behave differently. For example, when a DCL procedure is invoked in a pipeline, SYS$INPUT points to the procedure instead of being the same as SYS$PIPE.

Note that SYS$COMMAND is always unchanged in the pipeline.

PIPE

PIPE command

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

Not all programs and commands expect to read input from the SYS$INPUT stream. Good examples are commands like COPY, SEARCH and PRINT. For SEARCH, SYS$PIPE is useful as the input file.

PRINT is not very useful in a pipeline because it expects a file specification for one or more disk files that can be passed along to a print symbiont. This differs from UN*X where stdout can be piped to LPR. A partial work-around is that a spooled device can be specified as the final output target; however, no print /FORMs can be specified.

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 supports redirection of SYS$INPUT (“stdin”), SYS$OUTPUT (“stdout”) and SYS$ERROR (“stderr”).

Note, however, output redirection is only allowed for the last element of a pipeline, and that PIPE does not provide for an append operator. The work-around is to 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.

The PIPE command supports multiple commands in a pipeline element (“subshell”). This is done by enclosing the commands in parentheses and separating them using semi-colons (“;”). Note that “white space” is expected on either side of the semi-colon to ensure that it is not part of a file specification or other syntax element.

PIPE

PIPE command

Conditional operators:

```
&&
 Continue if previous command returns a success
  status

||
 Continue if previous command returns a failure
  status
```

The PIPE command provides conditional operators to control the sequence of events in a pipeline based on the completion status of the command elements in the pipeline.

Double-ampersand (“&&”) allows that pipeline processing can continue only if the preceding element returns a “success” status.

Double-vertical-bar (“||”, same as the pipe symbol, but two of them) allows that pipeline processing can continue only if the preceding element returns a “failure” status

PIPE

PIPE command

Other operators:

&

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

The PIPE command provides that a portion of a pipeline can be executed asynchronously in a separate (set of) subprocess(es). This is done using the ampersand symbol (single “&”). The pipeline element(s) preceding the ampersand will be executed in a (set of) subprocess(es).

The effect of the ampersand operator is similar to SPAWN/NOWAIT: the subprocess(es) are created, and they process independently of the parent process which proceeds immediately without waiting for the subprocess(es) to complete.

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 …
```


In UN*X-land, usually a command called "tee" is provided so that data flowing through the pipeline can be "tapped" and copied to a separate file.

While OpenVMS does not provide a supported TEE command, the on-line HELP for PIPE includes an example TEE.COM that can be extracted (cut and paste works well) and allows that data flowing through a pipeline can be copied to a disk file for other uses.

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
```

Here's the example TEE.COM from the on-line HELP for PIPE, slightly reformatted to fit the slide. Some comments were also removed.

Very "quick-and-dirty". Essentially, all this does is read data from the pipeline and write it to two different outputs: SYS$OUTPUT and the disk file specified when the procedure was invoked.

Like all DCL procedures, this has the limitations of DCL's string processing capabilities. As long as the input record is 510 bytes or less, this should work fine, except that "WRITE tee_file LINE" may need the /SYMBOL qualifier.

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
```


Adding one line of code:

$ WRITE SYS$COMMAND LINE

…to TEE.COM produces WYE.COM. This could even be used to make two copies of the data in the pipeline, if you needed to.

PIPE

PIPE command

- Sub-process performance
  - Can be improved by using these qualifiers:
    - /NOLOGICAL_NAMES
    - /NOSYMBOLS

VMS's subprocess performance is sometimes berated because of the amount of overhead imposed as compared to the way UN*X does it.

This overhead can be reduced if the pipeline elements do not need to inherit either symbols or logical names from the parent process.

In such case, use the /NOLOGICAL_NAMES and/or /NOSYMBOLS qualifier(s), as needed.

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).

The PIPE command has some restrictions:

•PIPEs cannot be nested. However, if an element of the pipeline invokes a DCL procedure, PIPE in that procedure is legal.

•Because PIPE uses "<" and ">" for redirection of input and output respectively, they cannot be used in file specifications. Use "[" and "]" instead.

•There is no append output operator (">>").

•There is no "2>&1" operator. SYS$OUTPUT and SYS$ERROR are the same unless SYS$ERROR is redirected.

•Complex PIPElines may spawn many subprocesses. Users may need more PRCLM to accommodate this.

PIPE

PIPE command

Some commands make no sense in PIPEs:

- SUBROUTINE - ENDSUBROUTINE
- EXIT, RETURN
- GOTO or GOSUB
- IF-THEN[-ELSE]-ENDIF

Some DCL statements are used to redirect logical flow (EXIT, RETURN, GOTO, GOSUB, IF-THEN[[-ELSE]-ENDIF])or enclose subprogram units (SUBROUTINE, ENDSUBROUTINE).

Thus they make no sense in a pipeline.

PIPE

PIPE command

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

Image data verification is turned off by default in a pipeline. This can make debugging a bit difficult.

To turn image data verification on in a pipeline, specify multiple commands in a pipeline element (subprocess or “subshell”) and use SET VERIFY=IMAGE to turn on image data verification.

Hands-on Exercise

Incorporate PIPE into the DIRECTORY in DCL exercise.

New Elements:
- PIPE
- READ, /END_OF_FILE
- F$PARSE()

Now, let's put we've learned to work.

Let's modify our DCL DIRCETORY procedure to read file specifications from the pipeline instead of using F$SEARCH().

New elements we'll use in this procedure are:
o READ and the /END_OF_FILE qualifier
o The F$PARSE() lexical function.

We'll use the PIPE and DIRECTORY commands to generate the list of file specifications our new procedure will process.

Hands-on Exercise

Incorporate PIPE into the DIRECTORY in DCL exercise.

New Element: PIPE cmd | cmd

Here's a brief summary of the PIPE command. Refer to the on-line HELP if you need to.

Hands-on Exercise

Incorporate PIPE into the DIRECTORY in DCL exercise.

New Element: READ/END=lbl lnm sym

Instead of using F$SEARCH() to retrieve the file specification, we'll use the "real" DIRECTORY command to generate them (hey - this *IS* just a DCL programming exercise, after all!).

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

We'll use F$PARSE() to get those portions of the file specifications that we want to display:

The NAME

The fileTYPE extension

The version number

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.

So, here's the design for the modified program:

Using the code from the previous exercise as a starting point, replace F$SEARCH() with a statement to READ from SYS$PIPE and transfer to control to the end of the loop at /END_OF_FILE. (Hint: Take an example from TEE.COM.)

Use F$PARSE() to get just the name, extension and version number for each file specification; get the other file information (EOF size, ALQ size, Creation DaTe) using F$FILE(), and output everything to SYS$OUTPUT by way of F$FAO().

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.

As a sort of bonus, write a “shell” procedure to invoke the modified DCL DIRECTORY procedure in a PIPE:

Use DIRECTORY/NOHEAD/NOTRAIL to generate a list of file specifications.

Invoke the modified DCL DIRECTORY procedure to display the file information.

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 ))
```

Requests are frequently heard for the ability to get command output into a symbol for further processing.

The example here gets the number of interactive logins seen by VMS into a symbol.

Note that within the pipeline itself, only the logical name in the job table is created. The translation of the logical name is handled in the mainline. This is to keep the length of the PIPE command and its elements within length limits.

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
```

Questions often arise about using symbol substitution in “image data” (data passed to a program within a procedure). Programs do not translate symbols the way DCL does. So, we need to find a work-around.

PIPE affords us this capability. We can write to the SYS$OUTPUT stream of the first pipeline element which becomes the SYS$INPUT (or SYS$PIPE) stream of the next pipeline element.

The example shows one way to do this. The SYSMAN program is used as an example of a program to which one might pass symbol values as part of the “image data”.

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:
```

Sometimes, we need to select files to be archived using selection criteria that are not supported by the compressed achive utility. Using ZIP's "batch" feature, we can utilize DIRECTORY's ability to select files by date, size and other criteria and pass that file list to ZIP in a pipeline.

The example shows one way to do this. Others are possible, of course.

Q & A

Speak now or forever hold your peas.

Let’s pause to answer any questions there may be about what we’ve covered in this section!

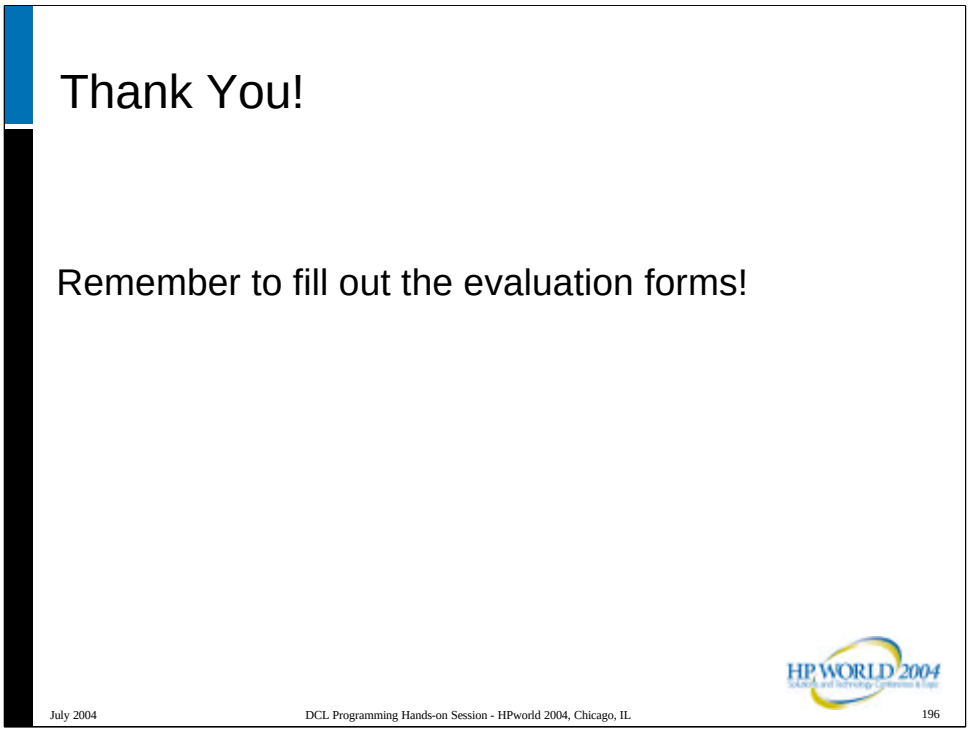
Thank You!

Congratulations!

You survived!

We've barely scratched the surface of what can be done using DCL as a programming language.

Your own imagination, coupled with the documentation and the on-line HELP, will guide you on to many new procedures and useful everyday tools!



If evaluation forms are available, please remember to fill them out and return them to the presenter.

.


HP WORLD 2004
Solutions and Technology Conference & Expo
Co-produced by:
interex
shared knowledge • shared power
encompass
AN HP USER GROUP